MODEL COMPRESSION FOR CROSS-ENCODER RERANKER IN RETRIEVAL-AUGMENTED GENERATION

Chenglin Zhang [1] **Muyang Xu** [1] **Ziyao Zhao** [1]

ABSTRACT

Transformer-based rerankers have become essential components in Retrieval-Augmented Generation (RAG) systems, but their high inference cost and memory footprint pose deployment challenges. This paper investigates model compression strategies, including unstructured pruning, structured pruning, and post-training quantization, applied to a compact BERT-based cross-encoder reranker. We implement iterative pruning algorithms inspired by the Lottery Ticket Hypothesis and evaluate their impact on ranking accuracy, latency, and memory usage. Using MS MARCO passage reranking data and MRR@10 as the primary evaluation metric, our experiments reveal that compression is achievable with minimal performance degradation even for a smaller distilled reranking model, enabling efficient deployment of rerankers in real-world retrieval systems.

1 INTRODUCTION

Recent advancements in deep learning have led to substantial improvements in information retrieval systems, particularly within the framework of Retrieval-Augmented Generation (RAG) pipelines. In a typical RAG setup, a simple retriever (like BM25) first retrieves a set of candidate passages given a query, and a reranker subsequently refines these candidates by scoring each query-passage pair to produce a more fine-grained ranking result. Cross-encoder rerankers such as TFR-BERT (Han et al., 2020) have demonstrated strong performance by jointly encoding both the query and passage into a single input for fine-grained interaction modeling.

However, while transformer-based rerankers achieve state-of-the-art accuracy, they introduce significant computational overhead. Inference latency and model size can become major bottlenecks, especially in resource-constrained environments, because the original BERT-large model (Devlin et al., 2019) has more than 330M parameters. Even the smaller version, BERT-base, still has 110M parameters. Therefore, the associated computational and storage demands hinder the deployment of cross-encoder rerankers at scale.

[1]Carnegie Mellon University, Pittsburgh, PA, USA. Correspondence to: Chenglin Zhang <chengliz@andrew.cmu.edu>, Muyang Xu <muyangxu@andrew.cmu.edu>, Ziyao Zhao <ziyaoz@andrew.cmu.edu>.

Proceedings of the n^{th} MLSys Conference, Pittsburgh, PA, USA, 2025. GitHub repo link: https://github.com/Aaalan-Zhang/CMU-15642-Final

Problem. To address these challenges, this project investigates the application of model compression techniques to a BERT-based cross-encoder reranker trained on the MS MARCO passage reranking subtask dataset (Bajaj et al., 2018). We explore three specific compression strategies: (1) unstructured pruning to randomly remove model weights, (2) L1 structured pruning to remove neurons, and (3) FP16, BF16, and INT8 post-training quantization to reduce memory consumption and improve inference speed. Furthermore, we explore different compression strategies like iterative compression workflows, such as pruning followed by retraining, which are motivated by the Lottery Ticket Hypothesis paper (Frankle & Carbin, 2019), and different layer-wise pruning configurations. Our objective is to empirically experiment and compare several methods that can produce sparse, efficient sub-networks, preserve the reranking accuracy of the original model, and reduce the model size and inference time.

2 LITERATURE REVIEW

Because the focus of our project is on model compression for reranking models in the context of RAG, this literature review will cover two relevant key areas: (1) the development of reranking models in information retrieval, particularly within the Retrieval-Augmented Generation (RAG) framework, and (2) model compression techniques designed to improve the efficiency of large transformer-based models. In model compression, we will also revisit the Lottery Ticket Hypothesis paper, which inspires some of our compression algorithms.

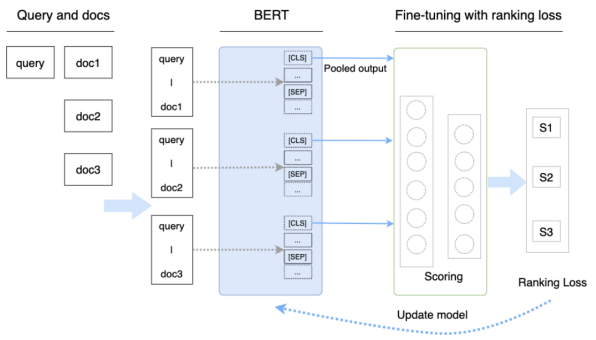

Figure 1. TFR-BERT pipeline. Figure adapted from (Han et al., 2020).

2.1 Reranking Models

As mentioned in Section 1, reranking has emerged as a crucial second-stage retrieval mechanism in modern information retrieval systems. In RAG, after an initial retrieval stage that retrieves a broad candidate set, rerankers apply more sophisticated modeling to reorder these candidates based on fine-grained relevance signals, substantially improving end-to-end retrieval effectiveness.

Early reranking methods primarily relied on manually crafted features such as term frequency (TF), inverse document frequency (IDF), and query-passage overlap features, combined with classical machine learning models like Rank SVM (Joachims, 2002), and RankNet (Burges et al., 2005), and LambdaMART (Burges, 2010). These approaches, while effective to some extent, were constrained by their limited ability to capture deep semantic and interaction patterns between queries and documents.

The introduction of cross-encoder architectures pre-trained transformer-based language models such as BERT (Devlin et al., 2019), led to a significant shift in reranking research. Cross-encoder architectures, which jointly encode the query and passage together, have demonstrated substantial gains in relevance modeling. Notably, models such as TFR-BERT (Han et al., 2020) fine-tune a pretrained BERT model by concatenating the query and passage as input, enabling rich token-level interactions that are critical for fine-grained relevance prediction. Pipelines like TFR-BERT and what is outlined in the paper *Passage Re-ranking with BERT* (Nogueira & Cho, 2020) have demonstrated good performance on the MS MARCO (Bajaj et al., 2018) passage reranking subtask, which serves as a standard benchmark for training and evaluating rerankers.

2.2 Model Compression

Model compression refers to a variety of techniques aimed at reducing the size and complexity of deep neural networks while preserving their predictive performance. Common approaches include dropout, pruning redundant parameters (Lagunas et al., 2021), quantizing weights to lower precision formats (Zafrir et al., 2019), applying structured sparsity constraints (Li et al., 2017), and distillation (Hinton et al., 2015).

For instance, existing research has demonstrated that structured pruning methods can effectively reduce latency by selectively eliminating less critical parameters. L1-normalization (Li et al., 2017) is commonly applied to penalize and shrink feature weights that contribute minimally to the task, while network slimming (Liu et al., 2017) techniques strategically remove less significant channels during training. In parallel, recent advancements in post-training quantization, such as SmoothQuant (Xiao et al., 2024), have shown that migrating quantization difficulty from activations to weights can enable accurate INT8 quantization of large language models.

It is worthwhile to mention the Lottery Ticket Hypothesis (LTH) (Frankle & Carbin, 2019) theory, which states that within a randomly initialized dense neural network, there exists a sparse subnetwork that, when trained in isolation, can achieve comparable performance to the original network. This insight has inspired pruning-based compression strategies where unimportant weights are iteratively removed, often guided by magnitude-based criteria (e.g., L1 or L2 norms), which we also aim to explore in this project.

2.3 Current Gap and Problem

While a substantial body of foundational work has explored reranking models and various model compression techniques, there remains a notable gap in the literature regarding systematic empirical comparisons of compression methods specifically applied to cross-encoder reranking architectures. In particular, despite the growing adoption of rerankers in retrieval-augmented generation (RAG) systems, few studies provide an evaluation of how different compression methods like structured pruning and post-training quantization affect both retrieval performance and model efficiency in this context.

3 Methodology

3.1 Overall Pipeline

We aim to present an experimental pipeline within the context of reranking and empirically compare compression strategies for a BERT-based reranker, `tomaarsen/reranker-MiniLM-L12-gooaq-bce`,

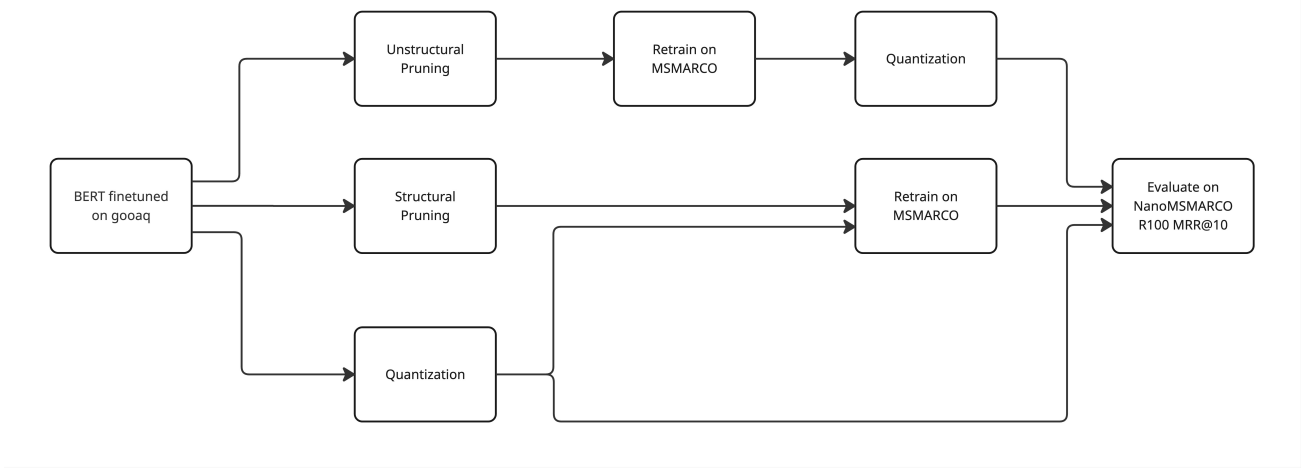

Figure 2. Overall compression and evaluation workflow.

which is a fine-tuned version of `MiniLM-L12-H384-uncased` (Wang et al., 2020). It is an uncased 12-layer model with 384 hidden size distilled from a pre-trained UniLM model (Bao et al., 2020) of the BERT-Base size (110M parameters). The number of parameters of the MiniLM-L12 baseline model is 33.6M and the structure is shown in Appendix A. The overall workflow is illustrated in Figure 2.

3.2 Pretraining Setup and Downstream Transfer

The model `reranker-MiniLM-L12-gooaq-bce` is pretrained on the GooAQ dataset (Khashabi et al., 2021), a large-scale collection of question-answer pairs derived from web search queries. The dataset can be converted to a binary classification format where, for each query, a set of answers is given, some of which are relevant (label 1) and others are irrelevant (label 0).

Importantly, the GooAQ dataset is not the same as our final evaluation set or its parent corpus (MS MARCO passage reranking). This design choice ensures that compression techniques are evaluated independently of the training data used for final evaluation. After applying each compression method—structured pruning, post-training quantization, and dropout-based sparsification, we retrain the resulting model on our target data for additional epochs. This enables us to systematically compare how different compression techniques affect both retrainability and final retrieval performance.

3.3 Compression Methods

We introduced three model compression strategies, applied independently to the BERT model that was first fine-tuned on the GooAQ dataset. These strategies are outlined below.

3.3.1 Unstructured Pruning

Iterative Random Mask-Based Pruning. We implemented a subclass of the `CrossEncoderTrainer` to introduce a mask-based iterative pruning mechanism targeting the `dense` layer weights of the `BertIntermediate` block within all 12 `BertLayer` blocks (see Appendix A for the specific model structure). For each layer, a binary mask is initialized to all ones and maintained throughout training to track the unpruned weights. During every training step, the base training logic is executed first, followed by reapplying the masks to the corresponding weight tensors via element-wise multiplication, ensuring pruned weights remain zeroed out.

Every **50** steps—only for the first **500** steps—an additional fraction (**10%**) of the remaining (i.e., unpruned) weights are randomly dropped. We alternate pruning (for the `BertIntermediate`) between the first six layers (`BertLayer[0-5]`) and the last six layers (`BertLayer[6-11]`) at each pruning round to encourage balanced sparsification. A fresh pruning mask is sampled and element-wise ANDed with the existing mask to generate an updated cumulative pruning mask. This updated mask is then applied to the weights to enforce irreversible pruning.

This strategy is inspired by the Lottery Ticket Hypothesis (LTH) (Frankle & Carbin, 2019), which promotes pruning followed by retraining without regrowth. After a single epoch of retraining (1230 steps), FP16 quantization is applied for further compression. The final model is evaluated using the `CrossEncoderNanoBEIREvaluator`

Algorithm 1 Iterative Random Mask-Based Pruning

Input: model M with layers $L_1, \ldots, L_{12}$; prune ratio p; interval k; max pruning step T_p; total steps T
Initialize masks $\mathcal{M}_i \leftarrow \mathbf{1}$ for all L_i; set `step` $\leftarrow 0$, `round` $\leftarrow 1$
for $t = 1$ to T **do**
  Apply: $W_i \leftarrow W_i \odot \mathcal{M}_i$; then perform one training step
  `step` $\leftarrow$ `step` + 1
  if `step` $\leq T_p$ and `step` mod $k = 0$ **then**
    Select layers $i = 1$ to 6 if `round` is odd; else $i = 7$ to 12
    for each selected L_i **do**
      Sample: $\mathcal{D}_i \sim$ Bernoulli$(1 - p)$ over active weights
      Update: $\mathcal{M}_i \leftarrow \mathcal{M}_i \wedge \mathcal{D}_i$
    end for
    `round` $\leftarrow$ `round` + 1
  end if
end for

on the MS MARCO passage reranking dataset. The pseudocode for this procedure is provided in Algorithm 1.

Iterative Customized Pruning with Schedules. To further extend the Iterative Random Mask-Based Pruning strategy for the `BertIntermediate` layers, we implement a customized iterative pruning approach that applies structured pruning at designated steps, with explicitly specified target layers and L_1-based pruning ratios. Unlike the random selection in the previous method, this strategy allows fine-grained control over which layers are pruned and by how much at each step. The pruning targets can include the `dense` layer weights within either the `BertIntermediate` block (as in the previous method), the `BertSelfOutput` block, or both, within each `BertLayer`. Additionally, both the layer indices and pruning ratios can vary across pruning steps. An example schedule is shown below:

```
self.schedule = {
    200: ("ffn", "all", 0.1),
    250: ("attn", [0, 1], 0.1),
    300: ("ffn", "all", 0.1),
    400: ("attn", [4, 5], 0.1),
}
```

Here, the key denotes the training step at which pruning is applied. The string `"ffn"` refers to the `dense` layer in the `BertIntermediate` block, while `"attn"` refers to the `dense` layer in the `BertSelfOutput` block. The list such as `[0, 1]` or `"all"` specifies the indices of the target `BertLayer` blocks, and the final element, such as `0.1`, indicates the fraction of neurons to prune based on their L_1 norm. This strategy is inspired by Neural Architecture Search (NAS) (Zoph & Le, 2017) and enables a more customized pruning approach. The process is complete after 1 epoch. The pseudo code for this algorithm is shown in Algorithm 2.

Algorithm 2 Iterative Customized Pruning with Schedules

Input: model M with layers $L_1, \ldots, L_{12}$; schedule $\mathcal{S}$; total steps T
Initialize masks $\mathcal{M}_i^{\text{ffn}}, \mathcal{M}_i^{\text{attn}} \leftarrow \mathbf{1}$ for all L_i
for $t = 1$ to T **do**
  for each L_i **do**
    Apply: $W_i^{\text{ffn}} \leftarrow W_i^{\text{ffn}} \odot \mathcal{M}_i^{\text{ffn}}$, $W_i^{\text{attn}} \leftarrow W_i^{\text{attn}} \odot \mathcal{M}_i^{\text{attn}}$
  end for
  Perform one training step
  if $t \in \mathcal{S}$ **then**
    $(\text{type}, \text{layers}, \text{rate}) \leftarrow \mathcal{S}[t]$
    targets $\leftarrow \{1, \ldots, 12\}$ if layers = `''all''` else layers
    for $i \in$ targets **do**
      Select: $W \leftarrow W_i^{\text{type}}$, $\mathcal{M} \leftarrow \mathcal{M}_i^{\text{type}}$
      Compute: norms $\leftarrow \|W \odot \mathcal{M}\|_1$ along rows
      Mask inactive rows: norms$[r] \leftarrow \infty$ if $\mathcal{M}[r, :] = 0$
      Let $k \leftarrow \max(1, \lfloor \text{rate} \cdot \text{rows} \rfloor)$
      Identify bottom-k rows; set row_mask$[r] \leftarrow 0$
      Update: $\mathcal{M} \leftarrow \mathcal{M} \wedge \text{Broadcast}(\text{row\_mask})$
      Apply: $W \leftarrow W \odot \mathcal{M}$; update $\mathcal{M}_i^{\text{type}}$
    end for
  end if
end for

3.3.2 *Structured Pruning*

In our structured pruning experiments, we remove entire neurons from the feed-forward block of each BERT layer, rather than masking individual weights. For the pruning and re-training, we follow a similar approach as in the Unstructured Pruning strategies, that we interleave training and pruning in up to 1 epoch, pruning every **50** steps during the first **500** steps. At each pruning interval, we target exactly half of the 12 layers, alternating between layers 1 to 6 on odd rounds and layers 7 to 12 on even rounds, to better preserve representation capacity across the network. After each pruning round, we reconstruct the FFN sublayers (`intermediate.dense` and `output.dense`). This results in a form of structured pruning based on neuron granularity.

We evaluate two distinct criteria for selecting which neurons to remove, which is similar to our previous unstructured pruning: **Iterative Random Neuron Pruning**, and **Iterative ℓ_1-Norm Neuron Pruning.**.

Algorithm 3 Iterative Random Neuron Pruning with Structural Rebuilding

Input: model M with layers $L_1, \ldots, L_{12}$; prune ratio p; pruning interval k; max pruning rounds R; total steps T
Set `round` $\leftarrow 0$
for $t = 1$ to T **do**
  Perform one training step
  if $t \leq R \cdot k$ and $t \bmod k = 0$ **then**
    Select layers $i = 1$ to 6 if `round` even; else $i = 7$ to 12
    for each selected L_i **do**
      Let $W_{\text{int}} \leftarrow$ weights of `intermediate.dense` in L_i
      Let $U_i \leftarrow$ active neuron indices (0 to $d_{\text{ff}}-1$)
      Randomly sample $q = \lceil p \cdot |U_i| \rceil$ indices from U_i to prune
      Let $K_i \leftarrow$ indices in U_i not selected (neurons to keep)
      Rebuild `intermediate.dense` with rows in K_i
      Rebuild `output.dense` with input columns in K_i
    end for
    `round` $\leftarrow$ `round` + 1
  end if
end for

Algorithm 4 Iterative ℓ_1-Norm Neuron Pruning

Input: model M with layers $L_1, \ldots, L_{12}$; keep ratio r; interval k; max pruning rounds R; total steps T
Set counters: `step` $\leftarrow 0$, `round` $\leftarrow 0$
for $t = 1$ to T **do**
  Perform one training step
  if $t \bmod k = 0$ and `round` $< R$ **then**
    Select layers $i = 1$ to 6 if `round` even; else $i = 7$ to 12
    Initialize score list $S \leftarrow [\,]$
    for each selected L_i **do**
      Let $W_i \leftarrow$ weights of `intermediate.dense` in L_i
      Compute row-wise ℓ_1 norms: $n_j = \|W_{i,j:}\|_1$
      Append (i, j, n_j) to S for all j
    end for
    Sort S by n_j and keep top-r fraction as K
    for each L_i being pruned **do**
      Let $K_i \leftarrow$ neuron indices in K for layer L_i
      Rebuild `intermediate.dense` with rows in K_i
      Rebuild `output.dense` with corresponding input cols
    end for
    `round` $\leftarrow$ `round` + 1
  end if
end for

Iterative Random Neuron Pruning. Similar to Iterative Random Mask-Based Pruning in Unstructured Pruning, at fixed intervals, we randomly drop a fixed fraction (5%) of the remaining neurons from the `dense` layer weights of the `BertIntermediate` block in alternating halves of the `BertLayer` blocks of the network. This stochastic strategy tests whether a sparse "lottery ticket" subnetwork can emerge simply by chance (cf. Algorithm 3). The training step is completed after 1 epoch.

Iterative ℓ_1-Norm Neuron Pruning. Instead of random selection, we score each neuron by the ℓ_1 norm of its weight vector, remove the lowest-magnitude 10% from the `dense` layer weights of the `BertIntermediate` block in alternating halves of the `BertLayer` blocks of the network, and rewind the surviving weights to their initial values. This leverages magnitude as a proxy for importance (cf. Algorithm 4). The training step is completed after 1 epoch.

3.3.3 Quantization

To systematically evaluate the impact of model quantization, we adopt two strategies:

Direct Post-Quantization Evaluation In this approach, all the model parameters are quantized directly from FP32 to FP16, BF16, and INT8 respectively without any further modification or retraining. The model is then evaluated with the `CrossEncoderNanoBEIREvaluator` using MS MARCO passage reranking data directly.

Quantization Followed by Fine-Tuning After the same quantization above, the models then undergo additional fine-tuning on the original training data for **1** epoch. The model is also then evaluated with the `CrossEncoderNanoBEIREvaluator` using MS MARCO passage reranking data.

4 Experiment Setup

4.1 Environment Settings

All experiments were conducted using a single NVIDIA A100 GPU (40.0 GB memory) on Google Colab. The software environment includes PyTorch 2.6.0 and CUDA 12.4. We use `sentence-transformers` version 3.4.1 with `ONNX-gpu` runtime support, enabling INT8 quantization for `CrossEncoder` models.

4.2 Dataset

4.2.1 *Training Dataset*

We use the MS MARCO dataset, loaded via the HuggingFace `load_dataset` API with configuration `v1.1` and the `train` split. The full dataset contains 79,704 examples, each comprising a query and a pre-sorted list of candidate documents. From this, we reserve 1,000 examples for validation and use the remaining for training.

Each example consists of a query, a list of candidate **documents** (typically of length $\sim$10), and corresponding binary **relevance labels** (1 for relevant, 0 for irrelevant). Documents labeled 1 are placed before those labeled 0. Within each relevance group, documents are further sorted by relevance. This order is preserved during training using `respect_input_order=True` in the `ListMLELoss` module.

4.2.2 *Evaluation Dataset*

Evaluation is conducted using the `CrossEncoderNanoBEIREvaluator` on the NanoBEIR reranking benchmark. It comprises 13 topic-specific subsets (including MS MARCO), each containing 50 evaluation queries (totaling 650 queries). The structure closely mirrors the MS MARCO training data.

4.3 Baseline Model

As described in Section 3, our baseline is the `reranker-MiniLM-L12-gooaq-bce` model with 33M parameters (see Appendix A for the specific model structure). On the NanoBEIR benchmark, it achieves an MRR@10 of 0.4205. We further fine-tune this baseline for 1 epoch on the MS MARCO dataset to examine its transfer performance. The model outputs a float score between 0 and 1 for each query-passage pair.

4.4 Loss Function

We train the reranker using the ListMLE loss (Xia et al., 2008), a listwise ranking objective that maximizes the likelihood of the correct permutation of documents. Given a list of n documents with predicted scores $s_1, s_2, \ldots, s_n$ and ground-truth permutation π, where $\pi(i)$ denotes the index of the i-th most relevant document, the loss is defined as:

$$\mathcal{L}_{\text{ListMLE}} = -\sum_{i=1}^{n} \log \frac{\exp(s_{\pi(i)})}{\sum_{j=i}^{n} \exp(s_{\pi(j)})}. \tag{1}$$

We enable `respect_input_order=True`, assuming inputs are sorted by decreasing relevance.

4.5 Hyperparameter Settings

Training is conducted with a batch size of **64** using the AdamW optimizer and a learning rate of 2×10^{-5}. On the hardware described in Section 4.1, one training epoch takes approximately 600–800 seconds.

4.6 Evaluation Metrics

To assess the trade-offs among ranking performance, model compactness, and efficiency, we evaluate all models using the following four metrics.

4.6.1 *MRR@10*

We report Mean Reciprocal Rank at cutoff 10 (MRR@10) as a primary ranking metric. Let Q denote the set of evaluation queries and rank_q the rank of the highest-ranked relevant document for query q. Then:

$$\text{MRR@10} = \frac{1}{|Q|} \sum_{q \in Q} \frac{\mathbf{1}[\text{rank}_q \leq 10]}{\text{rank}_q}. \tag{2}$$

Only queries with relevant documents in the top-10 contribute. All models are evaluated on the NanoMSMARCO benchmark with a candidate pool of 100 documents (R100 setting).

4.6.2 *Sparsity*

Sparsity is the percentage of weights removed during compression. For pruning-based methods, this corresponds to the fraction of zeroed-out parameters. Quantization-only methods do not affect sparsity.

4.6.3 *Model Size*

We report the total size of the model (in MB) after applying compression techniques. This metric reflects model compactness, which is critical for deployment on edge devices.

4.6.4 *GPU Memory Consumption*

This metric captures the peak GPU memory usage during inference on the NanoBEIR benchmark. Lower memory usage is advantageous for scalability and deployment in constrained environments.

4.6.5 *Inference Latency*

We report the average time (in seconds) to rerank all examples in the NanoBEIR evaluation set. Latency is measured on a single A100 GPU under identical conditions to assess runtime efficiency.

Table 1. Comparison of model compression strategies on NanoMSMARCO-R100. Higher MRR@10 indicates better retrieval performance. Latency and GPU memory usage are to be measured.

METHOD	PRUNE RATE (%)	MRR@10	LATENCY (MS)	GPU MEM (MB)
BASELINE	0.00	0.420	398	132.96
BASELINE + TUNING	0.00	0.530	422	132.96
UNSTRUCTURED PRUNING				
RANDOM PRUNING (ALG. 1)	9.47	**0.488**	396	132.96
CUSTOMIZED PRUNING (ALG. 2, SCHED. B.1)	12.94	0.106	396	132.96
CUSTOMIZED PRUNING (SCHED. B.2)	**11.80**	0.316	396	132.96
CUSTOMIZED PRUNING + FP16 (SCHED. B.1)	—	0.263	**349**	**65.57**
CUSTOMIZED PRUNING + BF16 (SCHED. B.1)	—	0.153	**320**	**65.57**
STRUCTURED PRUNING				
RANDOM NEURON PRUNING (ALG. 3)	4.16	0.482	396	**122.17**
ℓ_1-NORM NEURON PRUNING (ALG. 4)	3.85	0.172	**392**	128.96
QUANTIZATION				
FP16 (NO FINE-TUNE)	0.00	0.427	323	65.57
BF16 (NO FINE-TUNE)	0.00	0.381	314	65.57
INT8 (NO FINE-TUNE)	0.00	0.398	9996	32.7

5 RESULTS

Table 1 presents the effectiveness of different model compression strategies evaluated on the NanoMSMARCO-R100 benchmark. We report mean reciprocal rank at 10 (MRR@10), average per-query latency (in milliseconds), and peak GPU memory usage (in megabytes). Only queries with relevant documents in the top-10 are included in the metric, and all retrieval is performed with a candidate pool size of 100 documents (R100 setting).

5.1 Baseline and Fine-Tuning

The untuned baseline model achieves an MRR@10 of 0.420 with an average latency of 398 ms and a GPU memory footprint of 133 MB. After one epoch of task-specific fine-tuning, the model's MRR@10 improves to 0.530. This result establishes a strong reference point for evaluating the trade-offs introduced by subsequent compression methods.

5.2 Unstructured Pruning

Applying iterative random pruning (Algorithm 1) removes 9.47% of the model's parameters and results in an MRR@10 of 0.488. This performance is competitive with the fine-tuned baseline while maintaining the same memory usage and slightly reducing latency to 396 ms.

In contrast, customized schedule-based pruning targeting only the feed-forward layers with higher sparsity (11.8%) leads to a drop in accuracy (MRR@10 = 0.316). If the target includes attention layers, the MRR@10 will drop significantly. When mixed-precision quantization is applied on top of this variant, inference speed improves significantly (349 ms for FP16 and 320 ms for BF16), and GPU memory is nearly halved (to approximately 66 MB). However, this comes at a large cost of further degradation in retrieval quality, with MRR@10 dropping to 0.263 (FP16) and 0.153 (BF16). Further analysis needs to be done to investigate this issue.

5.3 Structured Pruning

Structured pruning exhibits varied performance depending on the selection strategy. The random neuron pruning method (Algorithm 3), which prunes 4.16% of neurons, maintains strong retrieval quality with an MRR@10 of 0.482. It matches the performance of unstructured pruning while reducing memory usage by approximately 8 MB (to 122 MB). However, the pruning rate is relatively trivial.

On the other hand, ℓ_1-norm–based neuron pruning (Algorithm 4), which prunes a similar proportion (3.85%), achieves only 0.172 MRR@10. Although this method slightly improves latency (392 ms) and memory (128.96 MB), it severely compromises accuracy, suggesting that magnitude-based structural pruning may require additional mechanisms, such as more structure-aware or layer-wise adaptation, to be effective in reranking tasks.

5.4 Quantization

Post-training quantization offers substantial improvements in efficiency with minimal complexity. FP16 quantization reduces memory consumption by 50% and latency by over 18%, achieving an MRR@10 of 0.427—only 0.009 below the untuned baseline. BF16 quantization delivers even faster inference (314 ms), though with a moderate accuracy drop to 0.381. This outcome, however, is somewhat atypical, as BF16 usually preserves accuracy better than FP16 due to its larger dynamic range. The observed drop may be attributed to insufficient calibration or numerical instability caused by

reduced mantissa precision.

INT8 quantization, despite offering the smallest memory footprint (32.7 MB), suffers from a significant latency increase (9996 ms per query) due to kernel fallback on unsupported hardware. Additional research needs to be done to look into this issue.

5.5 Overall Trade-Offs

Among the evaluated strategies, iterative random unstructured pruning (following the mechanism of LTH) offers the best balance of retrieval quality and efficiency. It achieves near-baseline accuracy without increasing memory consumption and with slightly improved latency. Structured pruning yields modest memory savings but underperforms in accuracy unless carefully tuned. Finally, INT8 quantization demonstrates that extreme compression is viable in theory but requires hardware-aware execution pipelines to be practical.

6 Limitations and Discussions

We evaluate our compression pipeline using the widely adopted state-of-the-art reranker, `cross-encoder/ms-marco-MiniLM-L6-v2` (see Appendix A for architectural details), a distilled and fine-tuned variant of `MiniLM-L12-H384-uncased`. This model contains 22.7M parameters (with a pruning rate of 32% if using our evaluation metrics), is pre-trained on the MS MARCO dataset, and achieves an MRR@10 of 0.59 on the NanoBEIR evaluation set. While serving as a strong baseline, our results highlight several limitations of the current compression pipeline.

Limited Sparsity under Fine-Grained Evaluation. To preserve near-baseline retrieval performance (as measured by MRR@10), we constrained both unstructured and structured pruning to relatively conservative levels—removing no more than 12% of weights or 5% of neurons. These modest sparsity ratios limit the achievable model size reduction. To enable more aggressive pruning without degrading performance, future work could explore adaptive pruning schedules, iterative rewinding, or the integration of complementary strategies such as knowledge distillation.

Quantization Bottlenecks and Hardware Constraints. Quantization is a promising compression strategy (Zafrir et al., 2019). However, our post-training INT8 quantization experiments led to significantly increased inference latency, likely due to the lack of optimized low-precision kernels on general-purpose hardware. Furthermore, applying FP16 or BF16 quantization to pruned models during unstructured pruning resulted in a substantial drop in accuracy. These findings suggest that effective deployment of quantized models requires either quantization-aware training or hardware-specific optimizations. Future research should systematically benchmark performance across diverse accelerators and toolchains. Additionally, embedding quantization may offer further opportunities for latency reduction.

Narrow Architectural and Task Scope. All experiments were conducted on a single architecture—a distilled MiniLM cross-encoder—and evaluated solely on the MS MARCO passage reranking task. The observed compression-performance trade-offs may not generalize to larger models (e.g., long-context transformers) or to domain-specific retrieval tasks (e.g., biomedical or legal corpora). Broader evaluations across architectures and benchmarks are necessary to assess the robustness and generality of our findings.

7 Conclusion

Our experiments show that even an already distilled MiniLM-based reranker can be further compressed—via unstructured and structured pruning or post-training quantization—while preserving strong MS MARCO MRR@10. Random and L1-norm pruning remove up to $\sim 10\%$ of parameters with minimal accuracy loss, and FP16/BF16 quantization (with fine-tuning) halves memory and speeds up inference. Against larger state-of-the-art compact rerankers, our methods deliver competitive performance from a much smaller starting point. These results point to a viable strategy for ultra-compact, high-quality rerankers in latency- and memory-constrained settings. Future work will explore compression-aware training, distillation–pruning hybrids, and hardware-optimized quantization to further enhance efficiency.

REFERENCES

Bajaj, P., Campos, D., Craswell, N., Deng, L., Gao, J., Liu, X., Majumder, R., McNamara, A., Mitra, B., Nguyen, T., Rosenberg, M., Song, X., Stoica, A., Tiwary, S., and Wang, T. Ms marco: A human generated machine reading comprehension dataset, 2018. URL https://arxiv.org/abs/1611.09268.

Bao, H., Dong, L., Wei, F., Wang, W., Yang, N., Liu, X., Wang, Y., Piao, S., Gao, J., Zhou, M., and Hon, H.-W. Unilmv2: Pseudo-masked language models for unified language model pre-training, 2020. URL https://arxiv.org/abs/2002.12804.

Burges, C., Shaked, T., Renshaw, E., Lazier, A., Deeds, M., Hamilton, N., and Hullender, G. Learning to rank using gradient descent. In *Proceedings of the 22nd international conference on Machine learning*, pp. 89–96, 2005.

Burges, C. J. From ranknet to lambdarank to lambdamart: An overview. *Learning*, 11(23-581):81, 2010.

Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. Bert: Pre-training of deep bidirectional transformers for language understanding, 2019. URL https://arxiv.org/abs/1810.04805.

Frankle, J. and Carbin, M. The lottery ticket hypothesis: Finding sparse, trainable neural networks, 2019. URL https://arxiv.org/abs/1803.03635.

Han, S., Wang, X., Bendersky, M., and Najork, M. Learning-to-rank with bert in tf-ranking, 2020. URL https://arxiv.org/abs/2004.08476.

Hinton, G., Vinyals, O., and Dean, J. Distilling the knowledge in a neural network, 2015. URL https://arxiv.org/abs/1503.02531.

Joachims, T. Optimizing search engines using clickthrough data. In *Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 133–142, 2002.

Khashabi, D., Ng, A., Khot, T., Sabharwal, A., Hajishirzi, H., and Callison-Burch, C. Gooaq: Open question answering with diverse answer types, 2021. URL https://arxiv.org/abs/2104.08727.

Lagunas, F., Charlaix, E., Sanh, V., and Rush, A. M. Block pruning for faster transformers, 2021. URL https://arxiv.org/abs/2109.04838.

Li, H., Kadav, A., Durdanovic, I., Samet, H., and Graf, H. P. Pruning filters for efficient convnets, 2017. URL https://arxiv.org/abs/1608.08710.

Liu, Z., Li, J., Shen, Z., Huang, G., Yan, S., and Zhang, C. Learning efficient convolutional networks through network slimming, 2017. URL https://arxiv.org/abs/1708.06519.

Nogueira, R. and Cho, K. Passage re-ranking with bert, 2020. URL https://arxiv.org/abs/1901.04085.

Wang, W., Wei, F., Dong, L., Bao, H., Yang, N., and Zhou, M. Minilm: Deep self-attention distillation for task-agnostic compression of pre-trained transformers, 2020. URL https://arxiv.org/abs/2002.10957.

Xia, F., Liu, T.-Y., Wang, J., Zhang, W., and Li, H. Listwise approach to learning to rank: theory and algorithm. In *Proceedings of the 25th international conference on Machine learning*, pp. 1192–1199, 2008.

Xiao, G., Lin, J., Seznec, M., Wu, H., Demouth, J., and Han, S. Smoothquant: Accurate and efficient post-training quantization for large language models, 2024. URL https://arxiv.org/abs/2211.10438.

Zafrir, O., Boudoukh, G., Izsak, P., and Wasserblat, M. Q8bert: Quantized 8bit bert. In *2019 Fifth Workshop on Energy Efficient Machine Learning and Cognitive Computing - NeurIPS Edition (EMC2-NIPS)*, pp. 36–39. IEEE, December 2019. doi: 10.1109/emc2-nips53020.2019.00016. URL http://dx.doi.org/10.1109/EMC2-NIPS53020.2019.00016.

Zoph, B. and Le, Q. V. Neural architecture search with reinforcement learning, 2017. URL https://arxiv.org/abs/1611.01578.

A MODEL ARCHITECTURE

A.1 tomaarsen/reranker-MiniLM-L12-gooaq-bce

```
CrossEncoder(
  (model): BertForSequenceClassification(
    (bert): BertModel(
      (embeddings): BertEmbeddings(
        (word_embeddings): Embedding(30522, 384, padding_idx=0)
        (position_embeddings): Embedding(512, 384)
        (token_type_embeddings): Embedding(2, 384)
        (LayerNorm): LayerNorm((384,), eps=1e-12, elementwise_affine=True)
        (dropout): Dropout(p=0.1, inplace=False)
      )
      (encoder): BertEncoder(
        (layer): ModuleList(
          (0-11): 12 x BertLayer(
            (attention): BertAttention(
              (self): BertSdpaSelfAttention(
                (query): Linear(in_features=384, out_features=384, bias=True)
                (key): Linear(in_features=384, out_features=384, bias=True)
                (value): Linear(in_features=384, out_features=384, bias=True)
                (dropout): Dropout(p=0.1, inplace=False)
              )
              (output): BertSelfOutput(
                (dense): Linear(in_features=384, out_features=384, bias=True)
                (LayerNorm): LayerNorm((384,), eps=1e-12, elementwise_affine=True)
                (dropout): Dropout(p=0.1, inplace=False)
              )
            )
            (intermediate): BertIntermediate(
              (dense): Linear(in_features=384, out_features=1536, bias=True)
              (intermediate_act_fn): GELUActivation()
            )
            (output): BertOutput(
              (dense): Linear(in_features=1536, out_features=384, bias=True)
              (LayerNorm): LayerNorm((384,), eps=1e-12, elementwise_affine=True)
              (dropout): Dropout(p=0.1, inplace=False)
            )
          )
        )
      )
      (pooler): BertPooler(
        (dense): Linear(in_features=384, out_features=384, bias=True)
        (activation): Tanh()
      )
    )
    (dropout): Dropout(p=0.1, inplace=False)
    (classifier): Linear(in_features=384, out_features=1, bias=True)
  )
  (activation_fn): Sigmoid()
)
```

A.2 cross-encoder/ms-marco-MiniLM-L6-v2

```
BertForSequenceClassification(
  (bert): BertModel(
    (embeddings): BertEmbeddings(
      (word_embeddings): Embedding(30522, 384, padding_idx=0)
      (position_embeddings): Embedding(512, 384)
      (token_type_embeddings): Embedding(2, 384)
      (LayerNorm): LayerNorm((384,), eps=1e-12, elementwise_affine=True)
      (dropout): Dropout(p=0.1, inplace=False)
    )
    (encoder): BertEncoder(
```

```
      (layer): ModuleList(
        (0-5): 6 x BertLayer(
          (attention): BertAttention(
            (self): BertSdpaSelfAttention(
              (query): Linear(in_features=384, out_features=384, bias=True)
              (key): Linear(in_features=384, out_features=384, bias=True)
              (value): Linear(in_features=384, out_features=384, bias=True)
              (dropout): Dropout(p=0.1, inplace=False)
            )
            (output): BertSelfOutput(
              (dense): Linear(in_features=384, out_features=384, bias=True)
              (LayerNorm): LayerNorm((384,), eps=1e-12, elementwise_affine=True)
              (dropout): Dropout(p=0.1, inplace=False)
            )
          )
          (intermediate): BertIntermediate(
            (dense): Linear(in_features=384, out_features=1536, bias=True)
            (intermediate_act_fn): GELUActivation()
          )
          (output): BertOutput(
            (dense): Linear(in_features=1536, out_features=384, bias=True)
            (LayerNorm): LayerNorm((384,), eps=1e-12, elementwise_affine=True)
            (dropout): Dropout(p=0.1, inplace=False)
          )
        )
      )
    )
    (pooler): BertPooler(
      (dense): Linear(in_features=384, out_features=384, bias=True)
      (activation): Tanh()
    )
  )
  (dropout): Dropout(p=0.1, inplace=False)
  (classifier): Linear(in_features=384, out_features=1, bias=True)
)
```

B ITERATIVE CUSTOMIZED PRUNING SCHEDULES

B.1 Schedule 1

```
self.schedule = {
    150: ("ffn", "all", 0.1),
    200: ("ffn", "all", 0.1),
    250: ("attn", [0,1], 0.1),
    300: ("ffn", "all", 0.1),
    350: ("ffn", "all", 0.1),
    400: ("attn", [4,5], 0.1),
    450: ("ffn", "all", 0.1),
    500: ("ffn", "all", 0.1),
    550: ("attn", [8,9], 0.1),
}
```

B.2 Schedule 2

```
self.schedule = {
    150:  ("ffn", "all", 0.12),
    200:  ("ffn", [0,1,2], 0.2),
    300:  ("ffn", [3,4,5], 0.2),
    350:  ("ffn", [6,7,8], 0.2),
    450:  ("ffn", [9,10,11], 0.2),
    550: ("ffn", "all", 0.12),
    650: ("ffn", "all", 0.12),
}
```