Projekt Information Retrieval

Bericht zur Implementierung von COIL in LightningIR

Lukas Günther

Samwel Müller

Tom Schmidt

Erstellt am 02 Februar, 2025

Friedrich Schiller Universität Jena
Ferdinand Schlatt

1 Einleitung

1.1 Zielsetzung

Ziel dieser Arbeit war es, das Retrival Model COIL aus dem Paper ”COIL: Revisit Exact Lexical Match in Information Retrieval with Contextualized Inverted List”von Luyu Gao, Zhuyun Dai, Jamie Callan (https://github.com/luyug/COIL) in ”Lightning IR: Straightforward Fine-tuning and Inference of Transformer-based Language Models for Information Retrieval”von Ferdinand Schlatt, Maik Fröbe, Matthias Hagen (https://github.com/webis-de/lightning-ir) zu implementieren.

1.2 COIL

Traditionelle Informationsretrieval-Systeme wie BM25 basieren auf exakten lexikalischen Übereinstimmungen und nutzen invertierte Listen für eine effiziente Suche. Neuere neuronale IR-Modelle tendieren zu weichen semantischen Übereinstimmungen zwischen allen Begriffen in Anfrage und Dokument, verlieren dabei jedoch die rechnerische Effizienz der exakten Übereinstimmungssysteme.

Der im Paper ”COIL: Revisit Exact Lexical Match in Information Retrieval with Contextualized Inverted List”von Luyu Gao, Zhuyun Dai und Jamie Callan vorgestellte Ansatz führt hingegen eine kontextualisierte exakte Übereinstimmungsarchitektur ein, bei der die Bewertung auf den kontextualisierten Repräsentationen von sich überschneidenden Token in Anfrage und Dokument basiert. Diese Architektur speichert die kontextualisierten Token-Repräsentationen in invertierten Listen und vereint so die Effizienz der exakten Übereinstimmung mit der Repräsentationskraft tiefer Sprachmodelle.

Damit liegt der zentrale Unterschied zu traditionellen und neuronalen IR-Methoden liegt in der Art, wie die exakte Wortübereinstimmung verarbeitet wird:

1. Statt nur einfache Termfrequenzen oder BM25-Werte zu speichern, speichert COIL für jedes Token im Dokument eine deep-learning-basierte, kontextualisierte Vektor-Repräsentation. Diese Repräsentationen werden aus einem Transformer-Modell wie zum Beispiel BERT extrahiert, sodass sie die semantische Bedeutung des Tokens im Kontext des Dokuments einfangen.
2. Klassische IR-Methoden speichern für jeden Token nur eine Posting-Liste mit Dokument-IDs und Termfrequenzen. COIL erweitert diese Struktur, indem es kontextualisierte Vektoren für jeden vorkommenden Token speichert. Dadurch bleibt die Effizienz der invertierten Listen erhalten, während gleichzeitig semantische Informationen genutzt werden.

3. COIL führt nur Berechnungen für genau übereinstimmende Tokens (Exact Lexical Match) durch, anstatt eine vollständige semantische Ähnlichkeitsberechnung über alle Token im Dokument durchzuführen. Dies bedeutet, dass es die Effizienz klassischer lexikalischer Methoden mit der Ausdruckskraft tiefer Sprachmodelle kombiniert (Contextualised Exact Lexical Match).
4. Bei der Anfrageverarbeitung sucht COIL nur nach übereinstimmenden Token zwischen Query und Dokument. Die Bewertung erfolgt dann durch die Interaktion der gespeicherten Token-Repräsentationen aus der invertierten Liste mit den Query-Token-Repräsentationen. Dadurch vermeidet COIL eine kostspielige Berechnung über alle Token eines Dokuments, wie es bei dichten neuronalen Repräsentationen der Fall wäre.

Experimentelle Ergebnisse zeigen, dass COIL sowohl klassische lexikalische Retrieval-Methoden als auch moderne neuronale Retrieval-Modelle übertrifft und dabei eine ähnliche oder geringere Latenz aufweist. Damit kombiniert die vorgestellte Architektur die Effizienz klassischer exakter Wortübereinstimmungssysteme mit der Ausdruckskraft tiefer Sprachmodelle.

1.3 Lightning-IR

Das Paper ”Lightning IR: Straightforward Fine-tuning and Inference of Transformer-based Language Models for Information Retrieval”von Ferdinand Schlatt, Maik Fröbe und Matthias Hagen stellt Lightning IR vor, ein Framework zur Anwendung von Transformer-basierten Sprachmodellen im Bereich Information Retrieval. Dieses auf PyTorch Lightning basierende Framework bietet ein modular und erweiterbares Grundgerüst, welches alle Phasen einer Retrieval-Pipeline unterstützt: vom Fine-Tuning und Indexieren bis hin zum Suchen und Re-Ranking. Das Framework ist darauf ausgelegt, benutzerfreundlich, skalierbar und reproduzierbar zu sein und steht als Open-Source-Projekt zur Verfügung.

2 Code

2.1 Overview

Das Python-Document coil.py implementiert das Embedding und Scoring des Coil Retireval Model in Lightning IR. Es enthält die folgenden Klassen:

- CoilEmbedding
- CoilScoringFunction
- CoilConfig
- CoilModel

2.2 CoilEmbedding

Coil Embedding erweitert BiEncoderEmbedding um das CLS Embedding.

```
@dataclass
class CoilEmbedding(BiEncoderEmbedding):
    cls_embeddings: Tensor
```

2.3 CoilScoringFunction

CoilScoringFunction erweitert ScoringFunction.
CoilScoringFunction.forward berechnet die Scores, sowie die CLS Similarity. Die Maske, mit welcher sich die Scores errechnen, wird, wie im Coil Paper, durch Exact Lexical Matching bestimmt. Die Summe der Werte der Arrays Scores und CLS Similarity wird dann ausgegeben.

```python
class CoilScoringFunction(ScoringFunction):

    def forward(
        self,
        query_embeddings: CoilEmbedding,
        doc_embeddings: CoilEmbedding,
        num_docs: Sequence[int] | int | None = None,
    ) -> Tensor:

        num_docs_t = self._parse_num_docs(query_embeddings, doc_embeddings, num_docs)
        query_cls_embeddings = (query_embeddings.cls_embeddings).repeat_interleave(num_docs_t, 0).unsqueeze(2)
        doc_cls_embeddings = doc_embeddings.cls_embeddings.unsqueeze(1)

        query_embeddings = self._expand_query_embeddings(query_embeddings, num_docs_t)
        doc_embeddings = self._expand_doc_embeddings(doc_embeddings, num_docs_t)
        similarity = self._compute_similarity(query_embeddings, doc_embeddings)

        query = query_embeddings.encoding.input_ids.repeat_interleave(num_docs_t, 0)[:, 1:]
        docs = doc_embeddings.encoding.input_ids[:, 1:]

        mask = (query[:, :, None] == docs[:, None, :]).float()

        cls_similarity = self.similarity_function(query_cls_embeddings, doc_cls_embeddings)

        scores = self._aggregate(mask * similarity, doc_embeddings.scoring_mask[:, :, 1:], "max", -1)
        scores = self._aggregate(scores, query_embeddings.scoring_mask[:, 1:, :], self.query_aggregation_function, -2)
        return scores[..., 0, 0] + cls_similarity[..., 0, 0]
```

2.4 CoilConfig

CoilConfig erweitert BiEncoderConfig um die CLS-Embedding-Dimension.

```
1
2  class CoilConfig(BiEncoderConfig):
3      model_type = "coil"
4
5      def __init__(
6          self,
7          query_expansion: bool = False,
8          embedding_dim: int = 32,
9          cls_embedding_dim: int = 768,
10         projection: Literal["linear", "linear_no_bias"] | None = "
       linear_no_bias",
11         **kwargs,
12     ) -> None:
13         kwargs["query_pooling_strategy"] = None
14         kwargs["doc_expansion"] = False
15         kwargs["attend_to_doc_expanded_tokens"] = False
16         kwargs["doc_pooling_strategy"] = None
17         super().__init__(
18             query_expansion=query_expansion,
19             embedding_dim=embedding_dim,
20             projection=projection,
21             **kwargs,
22         )
23         self.cls_embedding_dim = cls_embedding_dim
```

2.5 CoilModel

CoilModel erweitert BiEncoderModel.
CoilModel.encode encoded die Token-Sequencen für die Queries und Documents. Zurückgegeben wird ein CoilEmbedding, welches die CLS-Embeddings und die Tokenembeddings von Query und Document separat enthält.

```python
class CoilModel(BiEncoderModel):
    config_class = CoilConfig

    def __init__(self, config: BiEncoderConfig, *args, **kwargs) -> None:
        super().__init__(config, *args, **kwargs)
        self.scoring_function = CoilScoringFunction(self.config)

        if self.config.projection is not None:
            if "linear" in self.config.projection:
                self.cls_projection = Linear(
                    self.config.hidden_size,
                    self.config.cls_embedding_dim,
                    bias="no_bias" not in self.config.projection,
                )
            else:
                raise ValueError(f"Projection for COIL is not supported")
        else:
            if self.config.embedding_dim != self.config.hidden_size:
                warnings.warn(
                    "No projection is used (embedding_dim != hidden_size)"
                    "Output embeddings will not have embedding_size dim."
                )
    def encode(
        self,
        encoding: BatchEncoding,
        expansion: bool = False,
        pooling_strategy: Literal["first", "mean", "max", "sum"] | None = None,
        mask_scoring_input_ids: Tensor | None = None,
    ) -> BiEncoderEmbedding:
        embeddings = self._backbone_forward(**encoding).last_hidden_state

        if self.projection is not None:
            cls_embeddings = self.cls_projection(embeddings[:, [0]])
            embeddings = self.projection(embeddings[:, 1:])

        embeddings = self._sparsification(embeddings, self.config.sparsification)
        embeddings = self._pooling(embeddings, encoding["attention_mask"], pooling_strategy)

        if self.config.normalize:
            embeddings = normalize(embeddings, dim=-1)
        scoring_mask = self.scoring_mask(encoding, expansion, pooling_strategy, mask_scoring_input_ids)
        return CoilEmbedding(embeddings, scoring_mask, encoding, cls_embeddings)
```

3 Modell Checkpoint auf Hugging Face

Die Entwickler des COIL Models stellen vortrainierte Modelle bereit und unterscheiden dabei zwischen zwei Systemen: eines verwendet harte Negative (HN) und das andere nicht. COIL ohne HN wird mit BM25-Negativen trainiert, während COIL mit HN zusätzlich mit harten Negativen trainiert wird, die mit einem anderen trainierten COIL-Modell gewonnen wurden.

Für die Implementierung in COIL wurde der, sich in der von COIL bereitgestellten GitHub Repository befindende, Checkpoint *hn-checkpoint.tar.gz* genutzt. An diesem Checkpoint wurde für das im Paper beschriebene CLS Token noch ein neuer Projectionslayer hinzugefügt und dann mittels *module.save_pretrained()* ein neues Model erstellt. Das Modell wurde dann auf Hugginface unter `https://huggingface.co/inforettom/InfoRet-Coil` bereitgestellt.

4 Evaluation durch ReRanking

Die Entwickler des COIL-Models haben für ihr Model schon eine ausreichende Auswertung gemacht. Trotzdem wäre es interessant zu testen, ob unsere Implementierung funktioniert.

Für die Auswertung wird ein Re-Ranking zwischen COIL und ColBERT durchgeführt. Das ursprüngliche Ranking stammt vom Modell Anserini (BM25) und basiert auf den Datensätzen *msmarco-passage-trec-dl-2019-judged* und *msmarco-passage-trec- dl-2020-judged*. Wir nutzen den nDCG@10 (normalized Discounted Cumulative Gain at rank 10) als Metrik für unsere Ausertung. nDCG@10 basiert auf dem DCG (Discounted Cumulative Gain), der die Relevanz von Dokumenten unter Berücksichtigung ihrer Position in der Ergebnisliste gewichtet, indem höher gerankte Dokumente stärker gewichtet werden. Der DCG wird anschließend normalisiert (nDCG), indem er durch den idealen DCG (IDCG) geteilt wird, um Werte zwischen 0 und 1 zu erhalten. Ein nDCG@10 von 1 bedeutet, dass die zehn relevantesten Dokumente in perfekter Reihenfolge präsentiert wurden.

4.1 Anserini (BM25)

Die Shell-Befehle für die Auswertung des gegebenen Rankings:

```
$ ir_measures "msmarco-passage/trec-dl-2019" "msmarco-passage-trec-dl-2019-judged.run" nDCG@10

$ ir_measures "msmarco-passage/trec-dl-2020" "msmarco-passage-trec-dl-2020-judged.run" nDCG@10
```

Das Ausgangs-Ranking erzielt folgende nDCG@10-Werte:

Test Metric	nDCG@10
msmarco-passage/trec-dl-2019/judged	0.5058
msmarco-passage/trec-dl-2020/judged	0.4796

Tabelle 1: nDCG@10-Werte - Anserini (BM25)

4.2 COIL

Das Skript für das Re-Ranking mittels COIL:

```
trainer
  logger false
model
  class_path lightning_ir.BiEncoderModule
  init_args
    model_name_or_path ./hn-checkpoint
    config coil.CoilConfig
    evaluation_metrics
    - nDCG@10
data
  class_path lightning_ir.LightningIRDataModule
  init_args
    inference_datasets
    - class_path: lightning_ir.RunDataset
      init_args
        run_path_or_id ./msmarco-passage-trec-dl-2019-judged.run
        depth 100
    - class_path: lightning_ir.RunDataset
      init_args
        run_path_or_id ./msmarco-passage-trec-dl-2020-judged.run
        depth 100
    inference_batch_size 4
```

Das Re-Ranking mit COIL erzielt folgende nDCG@10-Werte:

Test Metric	nDCG@10
msmarco-passage/trec-dl-2019/judged	0.6837
msmarco-passage/trec-dl-2020/judged	0.6512

Tabelle 2: nDCG@10-Werte - COIL

4.3 ColBERT

Das Skript für das Re-Ranking mittels ColBERT:

```
trainer
  logger false
model
  class_path lightning_ir.BiEncoderModule
  init_args
    model_name_or_path webis/colbert
    evaluation_metrics
    - nDCG@10
data
  class_path lightning_ir.LightningIRDataModule
  init_args
    inference_datasets
    - class_path: lightning_ir.RunDataset
      init_args
        run_path_or_id ./msmarco-passage-trec-dl-2019-judged.
   run
        depth 100
    - class_path: lightning_ir.RunDataset
      init_args
        run_path_or_id ./msmarco-passage-trec-dl-2020-judged.
   run
        depth 100
    inference_batch_size 4
```

Das Re-Ranking mit ColBERT ergibt folgende nDCG@10-Werte:

Test Metric	nDCG@10
msmarco-passage/trec-dl-2019/judged	0.3980
msmarco-passage/trec-dl-2020/judged	0.3161

Tabelle 3: nDCG@10-Werte - ColBERT

4.4 Auswertung

Das COIL-Re-Ranking verbessert den nDCG@10-Wert des ursprünglichen Rankings deutlich. ColBERT hingegen schneidet sogar schlechter als das Originalranking ab. Dies könnte dran liegen, dass wir für das Re-Ranking eine Tiefe von 100 nehmen und

nicht das volle Ranking. **Das war jedoch Aufgrund von Hardwarerestriktionen und dem Umfang des Moduls zum jetzigen Zeitpunkt nicht möglich.**