Lecture 6: Word Embeddings and Neural Ranking

CS 753/853

Naghmeh Farzi

Sept 24th 2025

Agenda

- **What is neural ranking?**
- **Word embeddings**
  - One-hot
  - Word2Vec / GloVe
- **Simple neural ranking (average embeddings)**
- **Transformers and contextual embeddings**
- **Neural ranking models**
  - Bi-Encoder
  - Cross-Encoder
  - ColBERT

Setting the Stage: What is Neural Ranking?

In **information retrieval (IR)**, we want to **rank a set of documents by their relevance** to a given query.

Traditional Approach:

- **Lexical matching**: Methods like **TF-IDF** and **BM25** rank documents by overlapping **keywords**.
- Good for precision on exact matches, but fails when query and document use different words for the same concept (e.g., *car* vs *automobile*).

Neural Approach:

- Learn **dense vector representations** of queries and documents, so that **semantic similarity** is captured.
- These methods often use **transformers (like BERT)** to capture contextual meaning beyond surface-level word overlap.

Representing meaning as numbers (word embeddings) - One-hot Encoding

Behind the scene, computers only understand **numbers**.

Solution: **word embeddings** = represent each word as a vector (a list of numbers).

The simplest way to do it is one-hot encoding:

Example: **“I ate an apple and played the piano”**

We can begin by indexing each word’s position in the given vocabulary set.

I	ate	an	apple	and	played	the	piano
1	2	3	4	5	6	7	8

Position of each word in the vocabulary

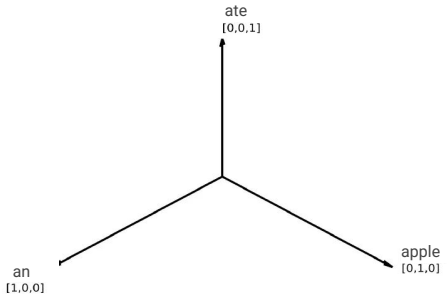
One-hot Encoding (Cont)

“I ate an apple and played the piano”

I	ate	an	apple	and	played	the	piano
1	2	3	4	5	6	7	8

Position of each word in the vocabulary

- dimensions equal to the total number of words
- vectors are orthogonal to each other, implying that there is no semantic relationship between words



One-hot encoding

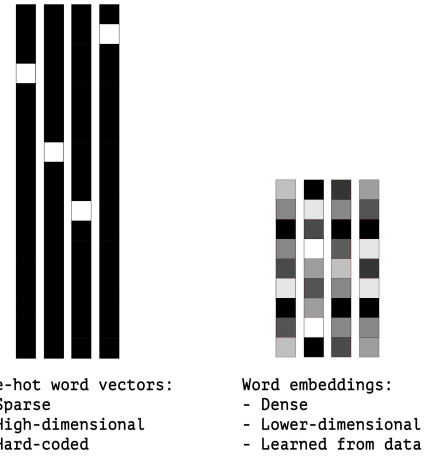
	1	2	3	4	5	6	7	8
I	1	0	0	0	0	0	0	0
ate	0	1	0	0	0	0	0	0
an	0	0	1	0	0	0	0	0
apple	0	0	0	1	0	0	0	0
and	0	0	0	0	1	0	0	0
played	0	0	0	0	0	1	0	0
the	0	0	0	0	0	0	1	0
piano	0	0	0	0	0	0	0	1

Representing meaning as numbers (word embeddings)

Problem with one-hot vectors: they treat all words as equally unrelated. `"cat"` and `"dog"` are no closer to each other than `"cat"` and `"banana"`, even though `"cat"` and `"dog"` are semantically similar.

So how do we teach them that *“cat”* and *“dog”* are similar, but *“banana”* is not?
To capture meaning, we want **dense vectors** where the geometry reflects similarity

- Example:
  - "cat" → [0.21, -0.55, 1.2, ...]
  - "dog" → [0.19, -0.53, 1.1, ...]
  - "banana" → [-0.9, 0.3, -0.8, ...]



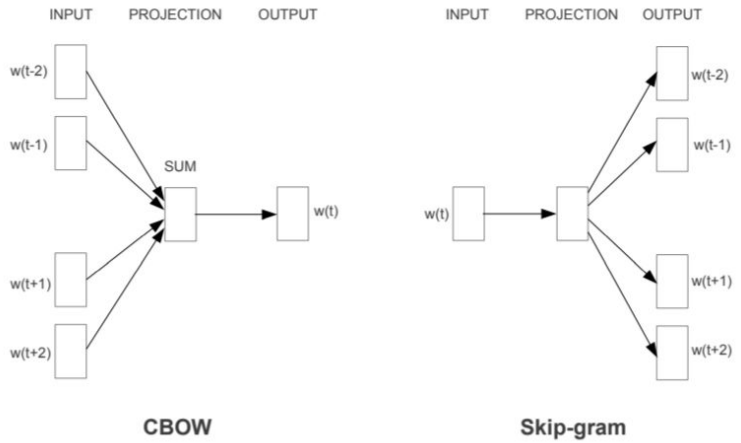
If we measure the distance between vectors, "cat" is close to "dog", but far from "banana".
This is the foundation of **semantic similarity**.

Word2vec

Learns dense word embeddings from **context** using a neural network.

Two main training strategies:

1. **Skip-gram:** Predict surrounding words given a target word.
    - Example: target = “cat”, context = [“the”, “sat”, “on”, “the”, “mat”]
    - Embeddings trained so words appearing in similar contexts are close together
2. **CBOW (Continuous Bag of Words):** Predict target word from surrounding words



GloVe (Global Vectors)

Learns embeddings based on **global co-occurrence statistics** of words in a large corpus.

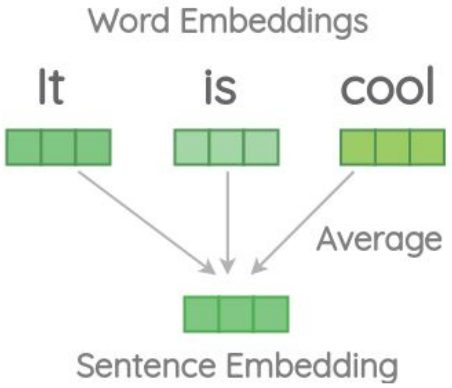
- Intuition: words appearing in similar contexts have similar vectors.
- Difference from Word2Vec:
  - Word2Vec = local context-based
  - GloVe = combines local context with **global statistics** → captures more global relationships

Sentence / Document Representation

Representing Queries and Documents

- Average embeddings → simple sentence representation
- Weighted averaging (TF-IDF) → emphasize important words

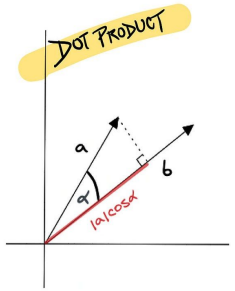
$$\begin{bmatrix} W_{11} \\ W_{12} \\ \\ \\ W_{1n} \end{bmatrix} + \begin{bmatrix} W_{21} \\ W_{22} \\ \\ \\ W_{2n} \end{bmatrix} + \cdots + \begin{bmatrix} W_{n1} \\ W_{n2} \\ \\ \\ W_{nn} \end{bmatrix} = \begin{bmatrix} \frac{W_{11}+W_{21}+\ldots+W_{n1}}{n} \\ \\ \\ \frac{W_{1n}+W_{2n}+\ldots+W_{nn}}{n} \end{bmatrix}$$

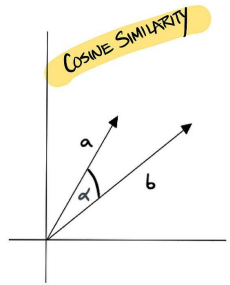


Neural Ranking with Sentence Embeddings Using Word2vec/GloVe

Step 1: Represent query as the **average** of its word embeddings → vector **q**
Step 2: Represent document as the **average** of its word embeddings → vector **d**
Step 3: Compare them with a similarity function:

- **Cosine similarity** (measures angle between q and d)
- **Dot product** (measures how aligned they are)





$$\mathbf{a} \cdot \mathbf{b} = |\mathbf{a}||\mathbf{b}|cos\alpha$$

$$sim(\mathbf{a}, \mathbf{b}) = \frac{\mathbf{a} \cdot \mathbf{b}}{||\mathbf{a}|| \cdot ||\mathbf{b}||}$$

Ranking:

- Do this for *all documents* in the collection.
- Rank documents by similarity score to the query.

Strengths:

- Simple, fast, and easy to implement.

Weaknesses:

- Reduces text to just an average meaning → loses word order and nuance.
- Cannot distinguish nuanced cases (e.g., “flights from Paris to New York” vs. “flights from New York to Paris”).
- Same word with different meanings has the same embedding (e.g., bank as “river bank” vs. bank as “financial institution”).

What is a Transformer?

- Instead of just averaging, a **transformer** builds **contextualized embeddings**.
- Core idea: **Attention mechanism**
  - Each word can “pay attention” to other words in the sequence.
  - Example: *“bank”* in *“river bank”* vs. *“savings bank”* → different meanings.

- **Sequence** is preserved:
  - *“Paris to New York” ≠ “New York to Paris”*.

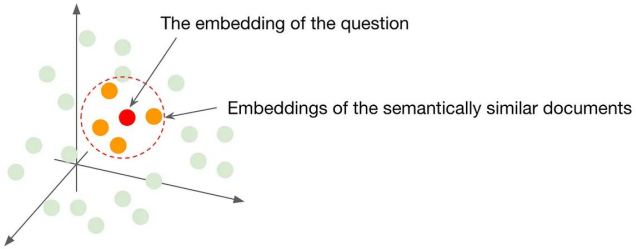
- Output: A set of embeddings that encode both **meaning** and **context**.

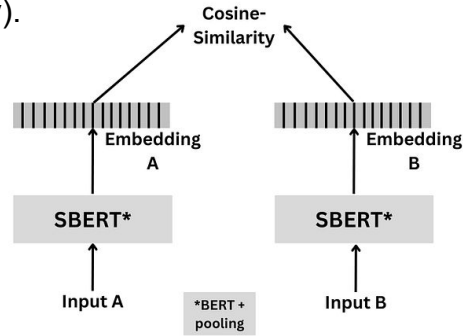
This is why transformers (like **BERT**) became the backbone of modern neural ranking.

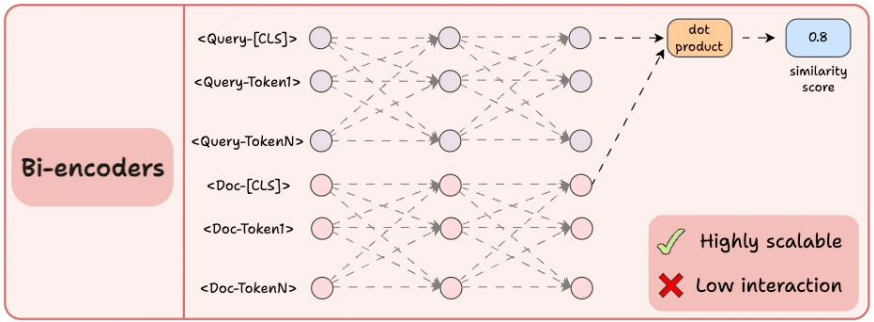
Bi-Encoder

Now we can introduce our first real neural ranking model.

1. Take the query. Pass it through a neural network (e.g., BERT) → get an embedding vector.
2. Take the document. Pass it through (possibly the same) neural network → get another embedding vector.
3. Compare them with a similarity function (dot product, cosine similarity).







Strengths and weaknesses of the Bi-Encoder

Strengths:

- Pre-compute document embeddings once → efficient for large-scale search.
- At query time, just encode the query and compare.

Weaknesses:

- **Does not perform word-to-word similarity.**
- Only compares query and document vectors as wholes.
- Even though BERT understands sequence/context, encoding separately means signal can get drowned in long documents.

Cross-Encoder

What if we want the model to look **inside both texts at once**?

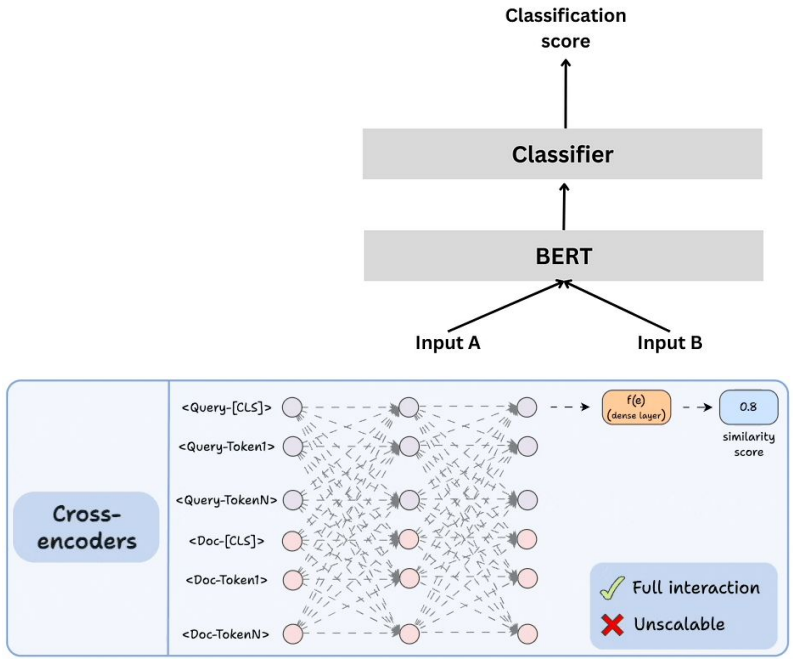
1. Concatenate the query and document into one sequence:

   `[CLS] query tokens [SEP] document tokens [SEP]`

2. Feed this into a transformer (like BERT).

3. The transformer allows **every query word to directly attend to every document word**.

4. At the end, use the `[CLS]` embedding to predict a **relevance score** (a single number).



Strengths and weaknesses of the Cross-Encoder

Strengths:

- Models **fine-grained** query-document interactions.
- **Higher accuracy f**or ranking **small** candidate sets.

Weaknesses:

- Very expensive at inference: must re-run the transformer for each query-document pair.
- Not feasible for large retrieval (millions of documents) but suitable for **re-ranking** small candidate sets.

ColBERT (Contextualized Late Interaction Over BERT)

Attention is expensive, specially when dealing with large document collection.

- **Step 1.** Preprocess the documents (offline, once).
  - Tokenize every document.
  - Pass through BERT → get a contextual embedding for each token.
    Store all token embeddings in an Approximate Nearest Neighbor (ANN) index (like FAISS)
- **Step 2.** At query time (online, for each query). → This immediately gives you a set of candidate documents.
  - Tokenize query → get embeddings for each query token with BERT.
  - For each query token embedding:
    Look up its nearest document token embeddings in the ANN index.

  Candidate pruning: Instead of considering all documents, you only keep the ones that had **at least one token close to a query token.**

- **Step 3.** Late interaction scoring (on candidates).
  - For each candidate doc:
  - For each query token, compute similarity with all tokens of that document.
  - Take the maximum similarity (MaxSim).
  - Add them up → final document score.

ColBERT (Cont.)

Strengths	Weaknesses
Token-level matching: each query token compares to all document tokens (MaxSim).	Higher storage: need embeddings for every document token.
Preserves fine-grained matches → distinguishes subtle query differences.	Slower than bi-encoder: MaxSim computed for candidate documents.
Efficient for large collections: documents pre-encoded and ANN prunes candidates.	ANN is approximate → may miss some token matches.
Scalable middle-ground between fast bi-encoder and accurate cross-encoder.	Might be slightly less accurate than full cross-encoder (doesn't model full query-document attention).

Link to google colab notebook on the same topics.