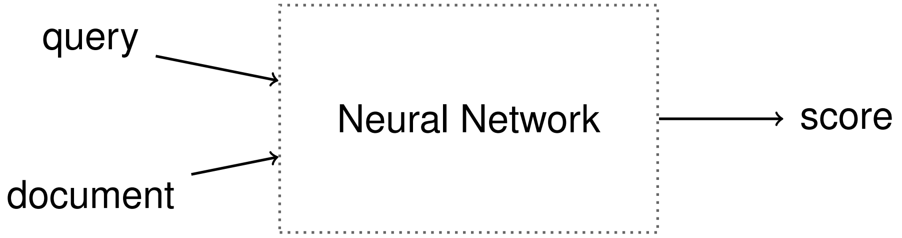
Word Embeddings

Overview

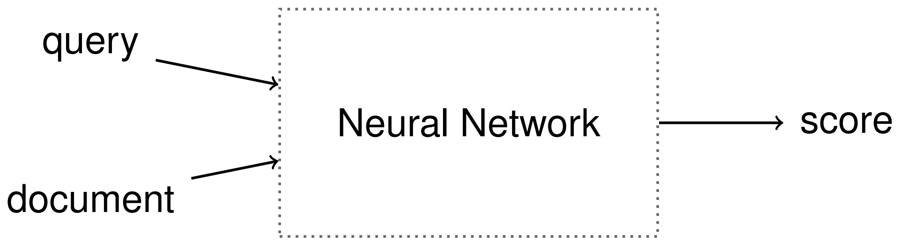
Goal:



Problem: How do we represent text so we can feed it to the neural network?

Word Embeddings
Overview

Goal:



Problem: How do we represent text so we can feed it to the neural network?

Solution: Turn words into numbers.

Word Embeddings

Representing Words

`apples are great`

Word Embeddings
Representing Words

```
apples are great
```

Assign each word a random value.

❑ apples	➜	6.3
❑ are	➜	-3.5
❑ great	➜	4.2

Word Embeddings

Representing Words

```
apples are great

apples are awesome
```

Assign each word a random value.

❑ apples	➜	6.3
❑ are	➜	-3.5
❑ great	➜	4.2
❑ awesome	➜	-32.1

Word Embeddings

Representing Words

`apples are great`

`apples are awesome`

Assign each word a random value.

❑ apples	➜	6.3
❑ are	➜	-3.5
❑ great	➜	4.2
❑ awesome	➜	-32.1

Problems:

- `great` and `awesome` mean similar things and used in similar ways.
- They are likely to have very different values.
- Bad for neural networks, requiring more complexity and training.

Word Embeddings

Developing a Better Representation

How can we let similar words have similar values?

➜ Learning how to use one word helps use the other at the same time.

Word Embeddings

Developing a Better Representation

How can we let similar words have similar values?

➜ Learning how to use one word helps use the other at the same time.

Words can be used in many contexts, pluralised, and so on.

➜ Assign each word multiple values for different contexts.

Word Embeddings

Developing a Better Representation

How can we let similar words have similar values?

➜ Learning how to use one word helps use the other at the same time.

Words can be used in many contexts, pluralised, and so on.

➜ Assign each word multiple values for different contexts.

How to decide which words are similar? How to learn multiple values?

➜ Neural network + clever training.

Word Embeddings

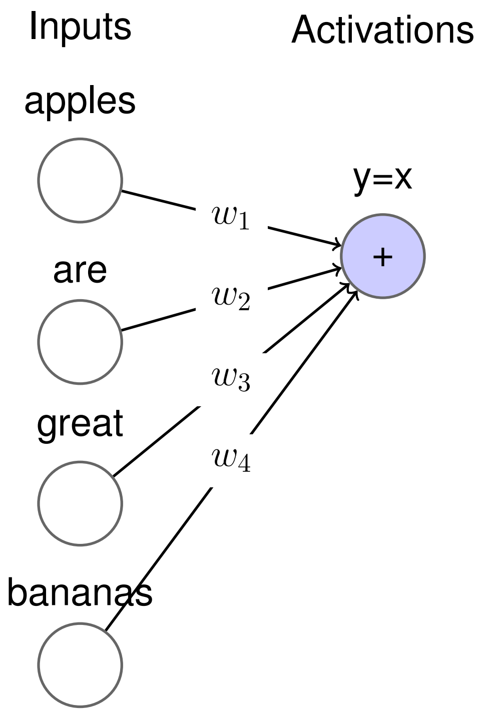
Training a Neural Network

Training data: `apples are great, bananas are great.`

Word Embeddings

Training a Neural Network

Training data: `apples are great, bananas are great.`

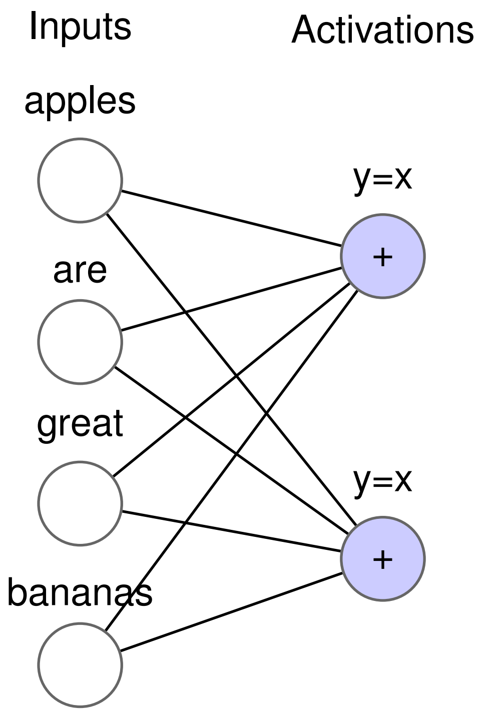


- Four unique inputs, each corresponding to a word.
- Linear activation function does nothing, just a place to do addition.
- Weights randomly initialised and optimised with backpropagation.

Word Embeddings

Training a Neural Network

Training data: `apples are great, bananas are great.`

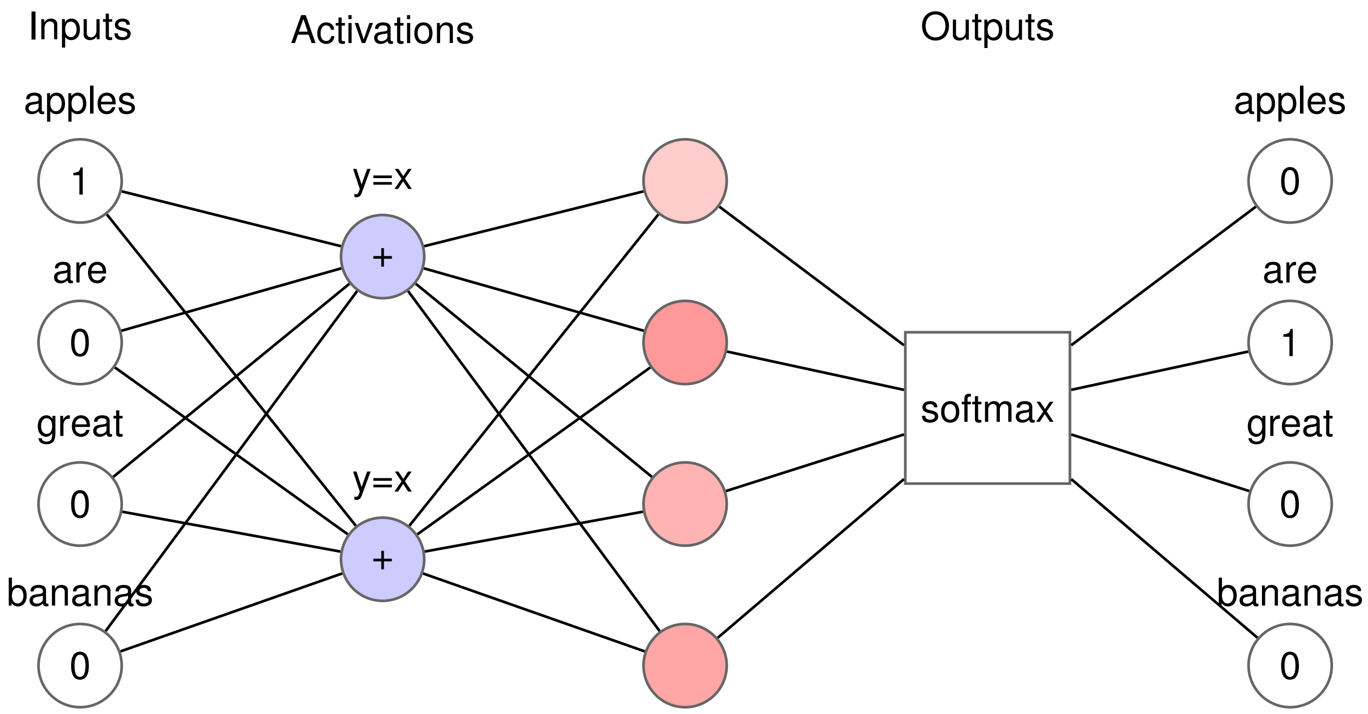


- To represent words with multiple values, add additional activation functions.
- Each activation function is associated with another weight for each word.

Word Embeddings

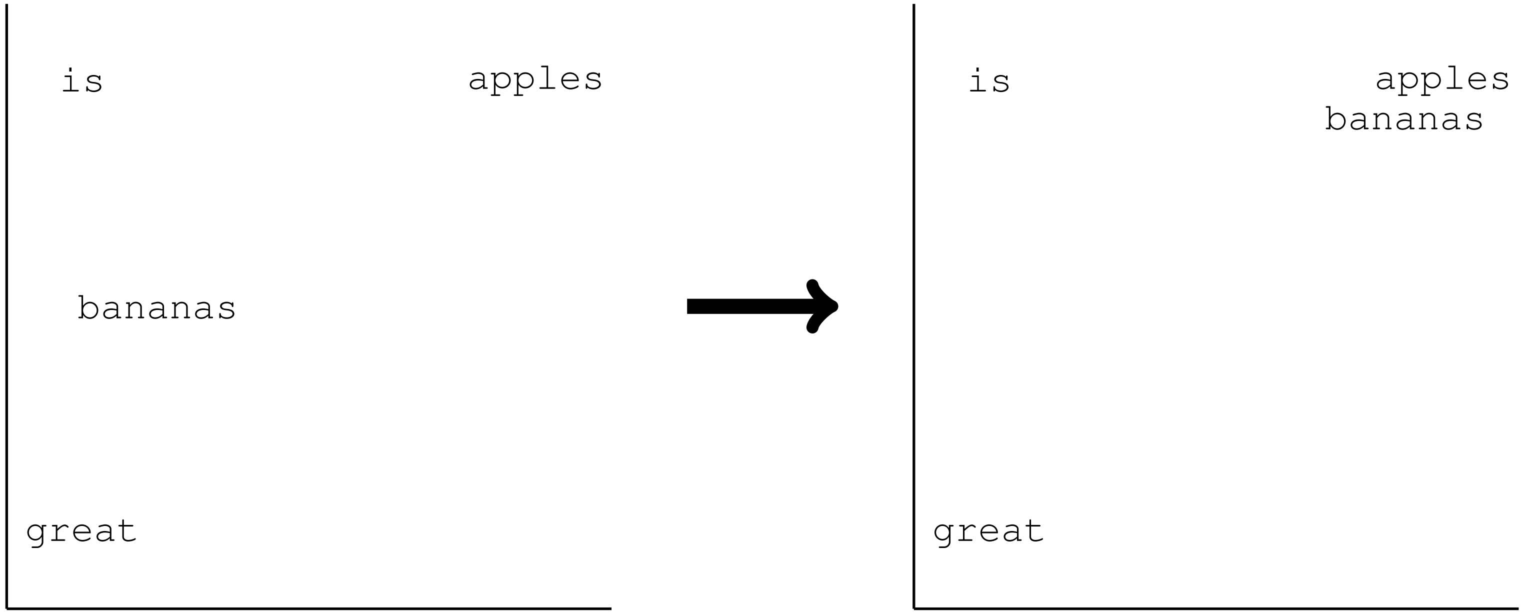
Training a Neural Network

Training data: `apples are great, bananas are great.`



- Use input word to predict next word in phrase ➜ `apples`
- We want the largest output value after softmax to be the target word.
- Cross entropy loss with backpropagation to optimise weights.

Word Embeddings
Visualising Word Embeddings



- Weights going into activation layer are the values associated with each word.
- When words appear in similar contexts, values (weights) become similar.
- All the weights for a given word are called the **word embedding**.

Word Embeddings
Summary

Word embeddings let us represent text as values for machine learning problems.

- ❑ Rather than using random values, use a neural network to learn values.
- ❑ Use context of words in training dataset to optimise weights for embeddings.
- ❑ Similar words get similar embeddings, which helps with training.

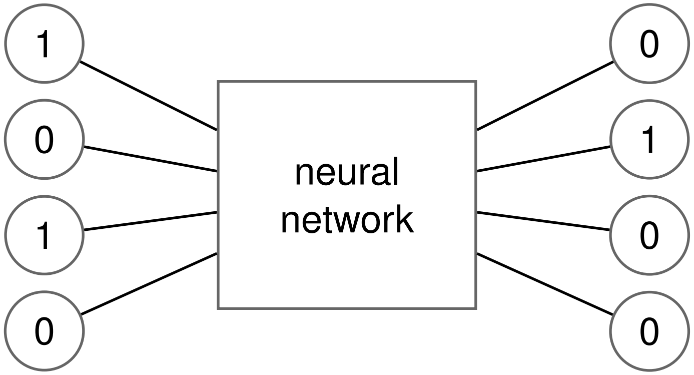
Problem: Just predicting the next word doesn't provide much context.

Word Embeddings

word2vec

Continuous Bag of Words (CBOW)

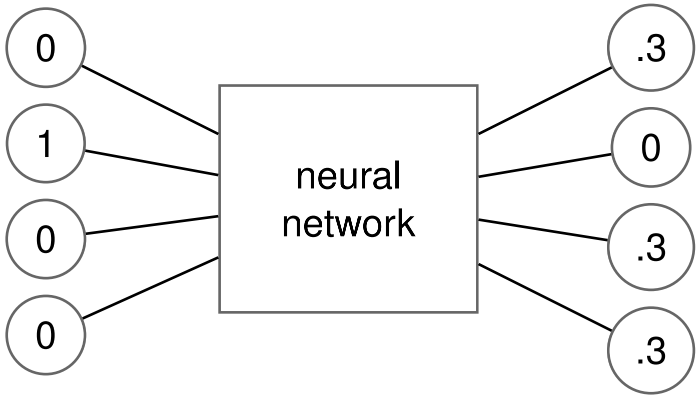
➜ Increase context by using surrounding words to predict what occurs in the middle.



Word Embeddings
word2vec

Skip gram

➜ Increase context by using word in the middle to predict surrounding words.



Word Embeddings

Efficiently Training word2vec

- In practice, there are hundreds of activation functions.
- And significantly more training data (e.g., all of Wikipedia).
- Vocabulary (input size) is much larger, typically 3,000,000 words and phrases.

Total weights to optimise:

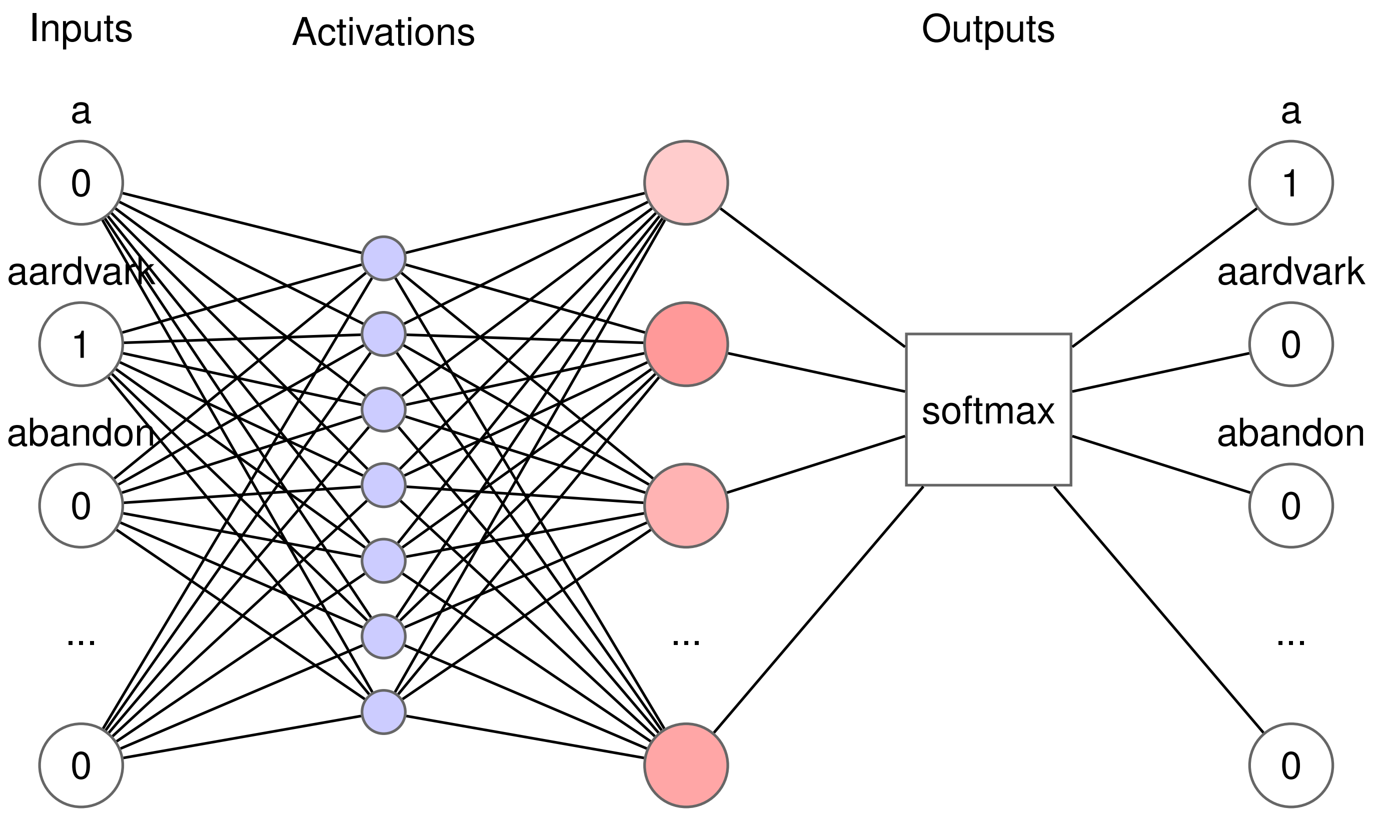
$$3,000,000 \cdot 100 \cdot 2 = 600,000,000$$

3M words, 100 activations (times 2 for input+output).

Solution: negative sampling.

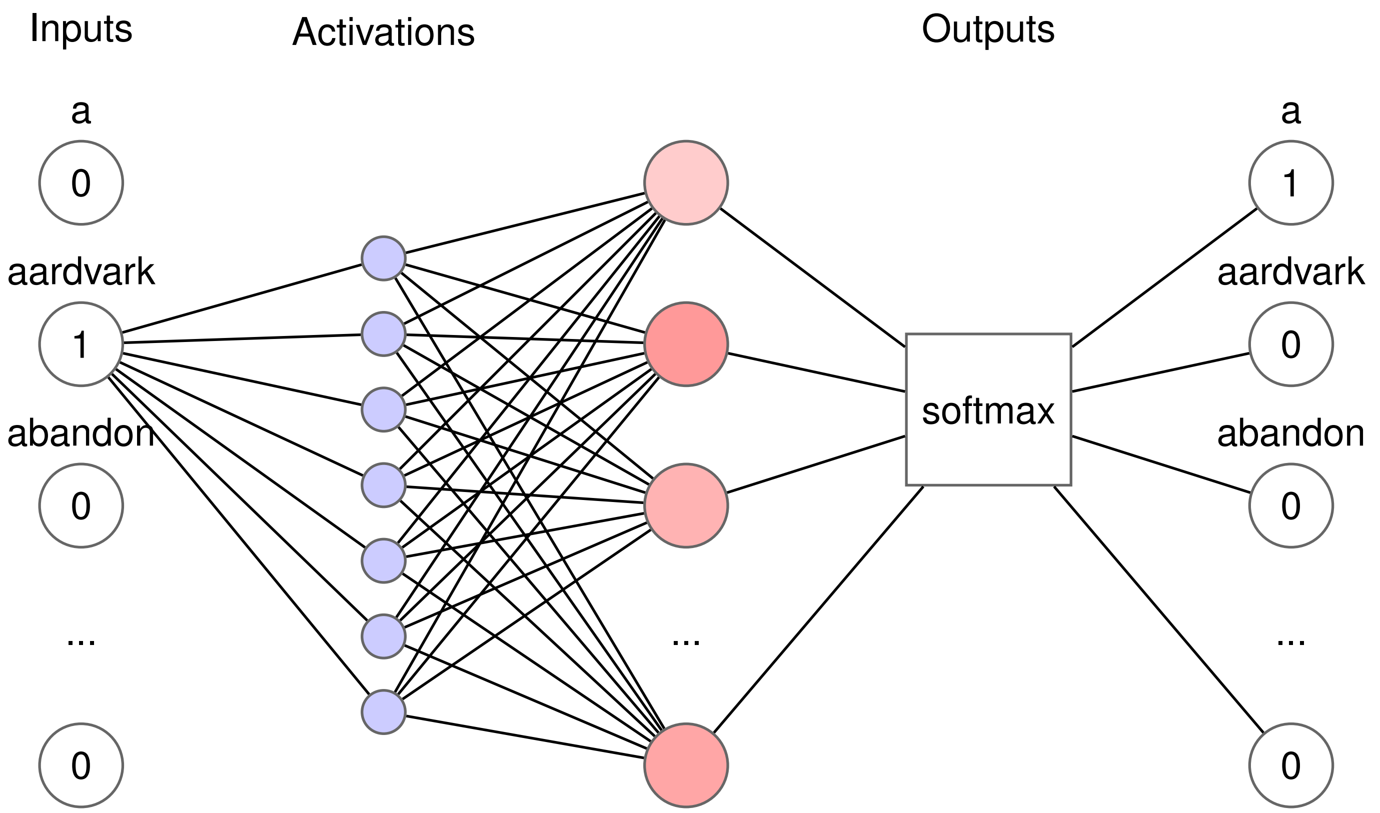
Word Embeddings

Efficiently Training word2vec



Word Embeddings

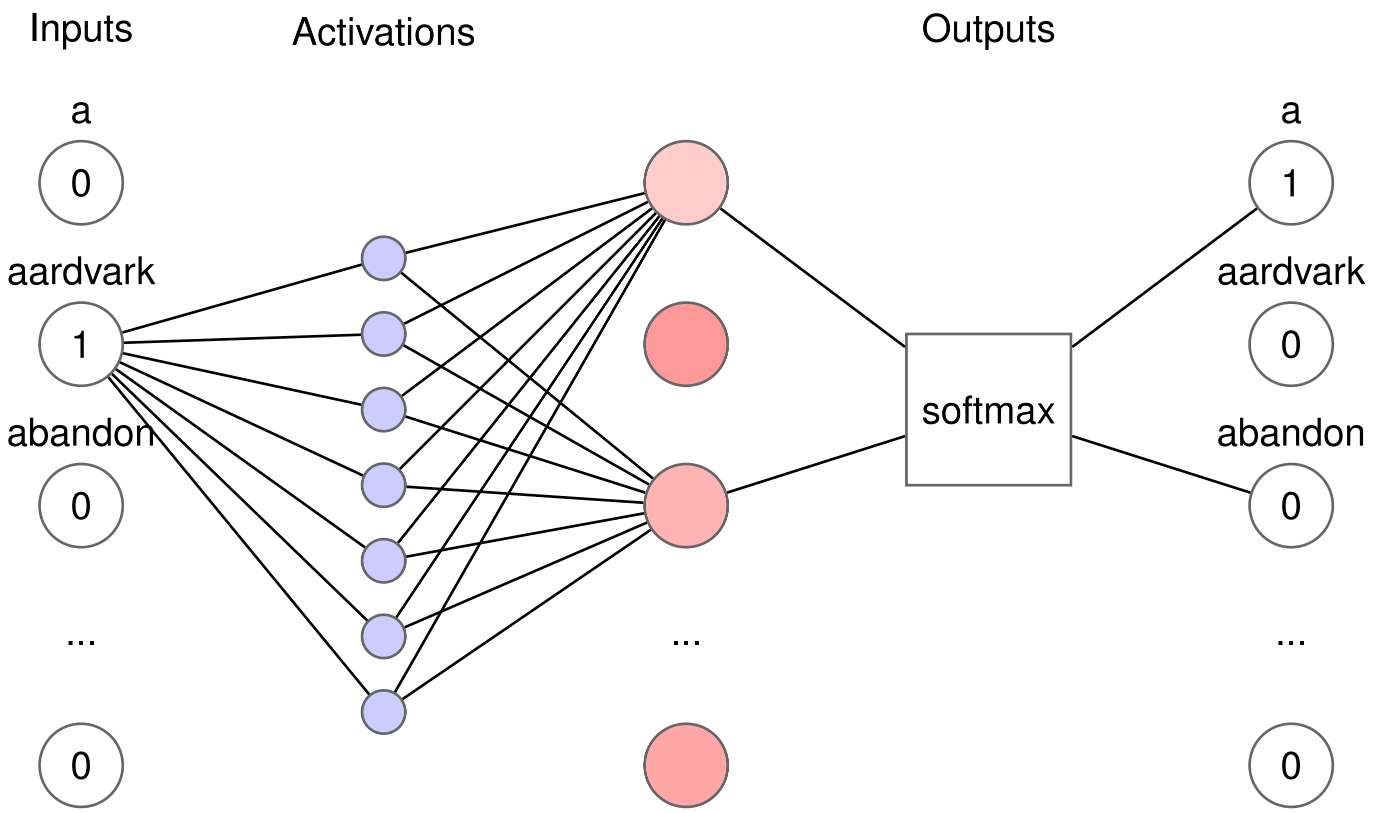
Efficiently Training word2vec



- Drop weights that do not contribute to prediction.
- Still left with over 300,000,000 weights to optimise.

Word Embeddings

Efficiently Training word2vec



- Randomly select subset of words will be ‘negative’ samples.
- `a` is still our target word, but now `abandon` is a negative sample.
- Now only need to optimise approximately 300 weights per step.

Transformer Models

Motivation

NLP tasks require good text embeddings, which can be learned using neural networks. Before transformers, recurrent neural networks were primarily used.

RNNs suffer from two problems:

- ❑ Sequence length / long-range dependencies
  - NLP tasks typically require long-distance dependencies between terms
  - Problem: vanishing gradient over long distances in RNNs
  - Solution: reduce the lengths of paths that signals must traverse
- ❑ Training efficiency
  - Parallelized computation within sequences enables large-scale training
  - Problem: recurrent models cannot be parallelized at sequence level (state at timestep $t+1$ depends on state at t), only on batch level
  - Solution: independent computation of each timesteps state

Transformers are efficient (in-sample parallelization) while also handling long-range dependencies (constant path length), enabled by the Attention mechanism.

Transformer Models

Contextual Embeddings

- Goal: allow each embedding to consider the other tokens in the sequence
  - this allows to build contextualized embeddings, i.e., token embeddings that include information about the context around the token
  - starting point is an embedding layer, a simple lookup table that pairs each input token ID with a learned continuous vector (input embedding)
- Idea: represent each token as weighted sum of all tokens in the sequence

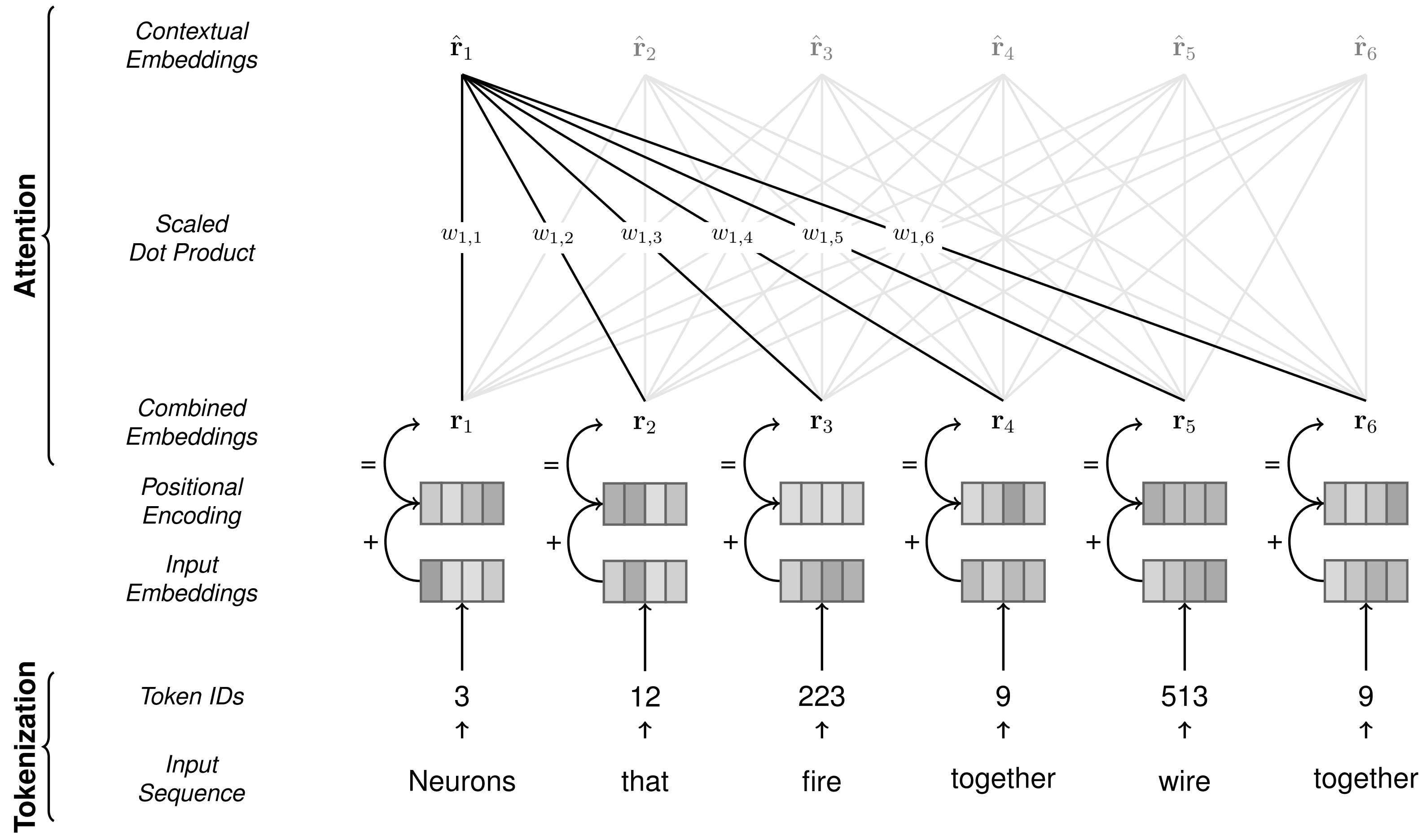
$$\hat{\mathbf{r}}_i = \sum_{j=1}^{t} w_{i,j} \cdot \mathbf{r}_j$$

- This requires four building blocks:

1. a way to decompose strings into tokens	➜	Tokenization
2. an initial representation of each token	➜	Input Embeddings
3. a way of representing the order of tokens	➜	Positional Encoding
4. a way of computing $\hat{\mathbf{r}}_i$ for each token	➜	Attention

Transformer Models

Contextual Embeddings



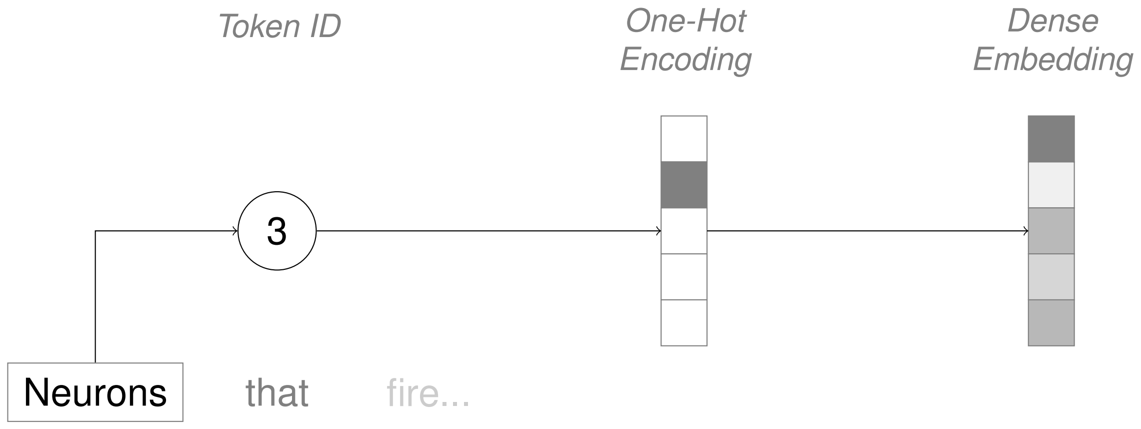
Attention
Tokenization
Contextual Embeddings
Scaled Dot Product
Combined Embeddings
Positional Encoding
Input Embeddings
Token IDs
Input Sequence
$\hat{\mathbf{r}}_1$ $\hat{\mathbf{r}}_2$ $\hat{\mathbf{r}}_3$ $\hat{\mathbf{r}}_4$ $\hat{\mathbf{r}}_5$ $\hat{\mathbf{r}}_6$
$w_{1,1}$ $w_{1,2}$ $w_{1,3}$ $w_{1,4}$ $w_{1,5}$ $w_{1,6}$
$\mathbf{r}_1$ $\mathbf{r}_2$ $\mathbf{r}_3$ $\mathbf{r}_4$ $\mathbf{r}_5$ $\mathbf{r}_6$
= +
3 12 223 9 513 9
Neurons that fire together wire together

Note: weights w are not learned directly, but inferred by the attention mechanism from learned projections $\mathbf{Q}, \mathbf{K}, \mathbf{V}$ of the combined embeddings.

Transformer Models

Tokenization & Input Embeddings

- The tokenizer splits the input string into a sequence of integers which represent each tokens' index in the vocabulary of the tokenizer
- Input embeddings are formed by projecting the (sparse) one-hot encoded token IDs into a (dense) vector space capturing semantic information
- This projection is learned jointly with the rest of the model



Transformer Models

Positional Encoding

- Word order (token position in the sequence) is crucial for language tasks
  - Transformer models lack recurrency, all tokens in the are fed to the network in parallel ➜ position information is lost
  - We need to add some information indicating the word order (position of the word) to the input embeddings ➜ positional encoding
- **Naive approach**: use token indices as positional encodings
  - represent the position of each element in the sequence by its index
  - indices are not bounded and can grow large in magnitude
  - normalized (0-1) indices are incompatible with variable length sequences as these would be normalized differently
- **Better approach**: represent position by a vector where each dimension corresponds to a different sine function evaluated at the current index
  - compatible with long sequences (bounded in magnitude)
  - compatible with variable length sequences (normalized)
  - compatible with arbitrary input embedding dimensionalities

Remarks

Sinusoidal positional encodings represent position by a vector where each dimension corresponds to the output of a function evaluating the current positions' index. Even dimensions are mapped with a sine function, odd positions are mapped with a cosine function, all of differing frequencies.

$$P(k, 2i) = \sin\left(\frac{k}{n^{2i/d}}\right)$$
$$P(k, 2i+1) = \cos\left(\frac{k}{n^{2i/d}}\right)$$

Parameters:

- k ➜ Index in the input sequence
- d ➜ Dimensioniality of positional encoding
- n ➜ Scalar normalization constant (usually $10,000$)
- i ➜ Used for mapping to column indices, $i \in [0 \dots d/2]$

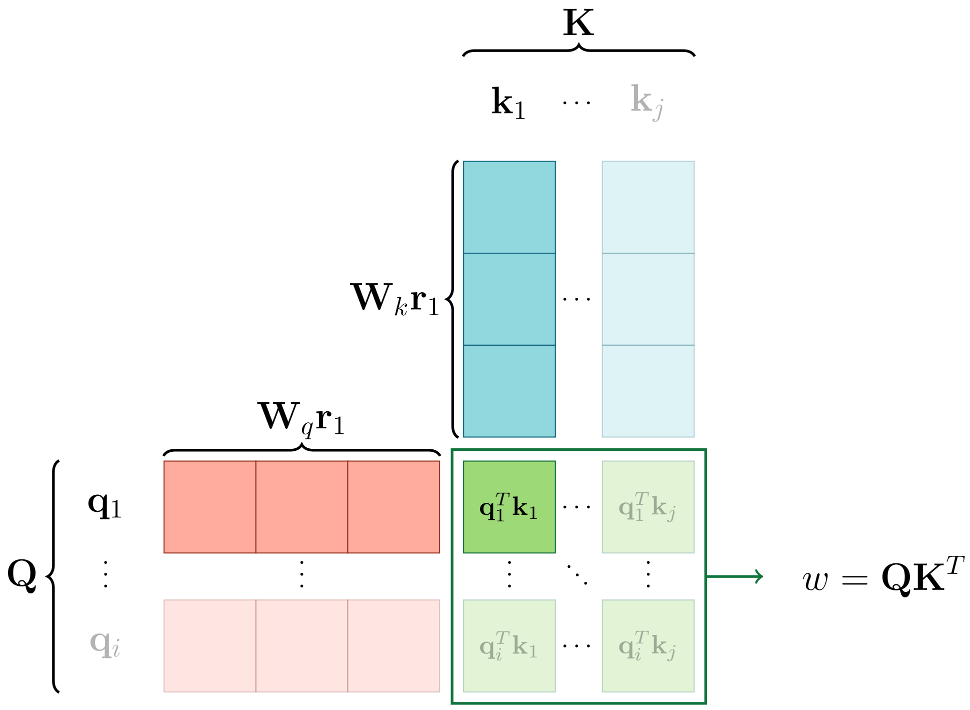
Transformer Models

Scaled Dot Product

To determine $w_{i,j}$ for two token representations $\mathbf{r}_i$ and $\mathbf{r}_j$, we:

- compute a query vector $\mathbf{q}_i = \mathbf{W}_q\mathbf{r}_i$ as linear transformation of $\mathbf{r}_i$
- compute a key vector $\mathbf{k}_j = \mathbf{W}_k\mathbf{r}_j$ as linear transformation of $\mathbf{r}_j$
- compute the dot product $w_{i,j} = \mathbf{q}_i^T\mathbf{k}_j$ that indicates token similarity

This is repeated for every token as key and as query, yielding the weight matrix w.



$\mathbf{K}$
$\mathbf{k}_1$ ⋯ $\mathbf{k}_j$
$\mathbf{W}_k\mathbf{r}_1$
$\mathbf{W}_q\mathbf{r}_1$
$\mathbf{Q}$
$\mathbf{q}_1$
$\mathbf{q}_i$
$\mathbf{q}_1^T\mathbf{k}_1$
$\mathbf{q}_1^T\mathbf{k}_j$
$\mathbf{q}_i^T\mathbf{k}_1$
$\mathbf{q}_i^T\mathbf{k}_j$
$w = \mathbf{QK}^T$

- query vectors are stacked into a query matrix $\mathbf{Q}$
- key vectors are stacked into a key matrix $\mathbf{K}$
- w can be written as dot product of $\mathbf{Q}$ and $\mathbf{K}$

Transformer Models

Attention Mechanism

- The weight matrix w needs to be normalized
  - rescale the values by d (dimensionality of the query and key vectors)
  - normalize the values using softmax to be non-negative and add up to 1
  - this yields the weights to calculate the updated token representations
- The attention mechanism combines the weights with the original embeddings
  - compute values $\mathbf{v}_i = \mathbf{W}_\mathbf{v}\mathbf{r}_i$, stacked into a value matrix $\mathbf{V}$, containing linear projections of the input embeddings
  - updated representations $\hat{\mathbf{r}}_i$ are summation of $\mathbf{V}$ weighted by w

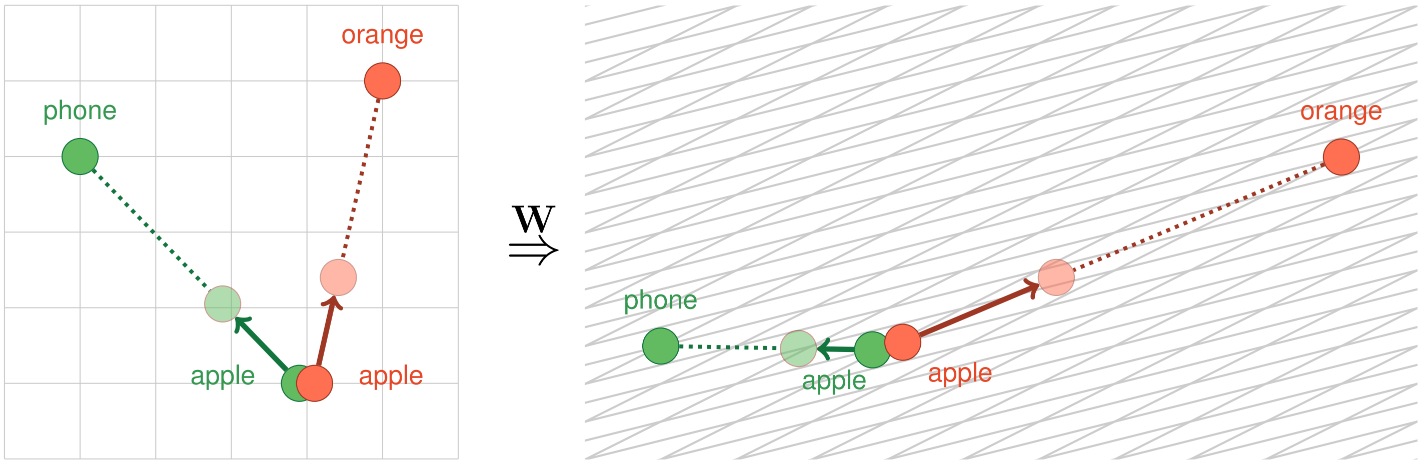
$$\text{Attention}(\mathbf{Q},\mathbf{K},\mathbf{V}) = \underbrace{\text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d}}\right)}_{w} \underbrace{\mathbf{V}}_{\mathbf{W}_v\mathbf{R}^T}$$

- Learnable parameters are the weight matrices $\mathbf{W}_q$, $\mathbf{W}_k$, and $\mathbf{W}_v$ which encode the linear transformations of input embeddings $\mathbf{R}$

Remarks

What does it mean to apply a linear transformation to the input embeddings and why do we do it?

- ❑ Linear transformation ➜ scale/shift of the input space given by a transformation matrix $\mathbf{W}$
- ❑ Consider two sentences ‘apple released their new phone’ and ‘an apple and an orange’
- ❑ Updated embeddings will move closer to their context words during the update step
  - – apple should move closer to phone (its input context is tech-related)
  - – apple should move closer to orange (its input context is fruit-related)
  - – both should move away from each other (increase their discriminative power)
- ❑ Optimum: the transformation matrix $\mathbf{W}$ that maximizes the information gained

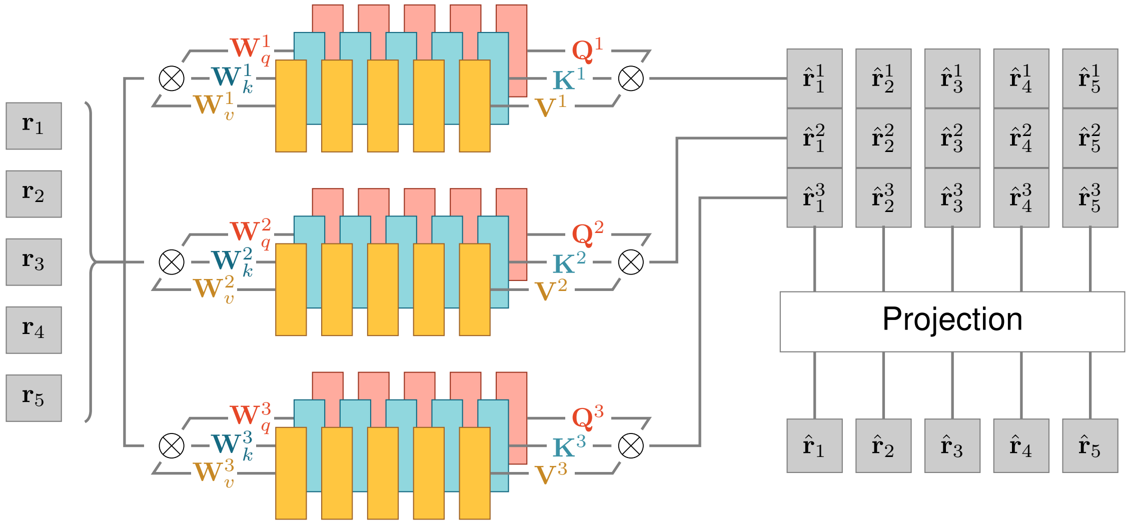


Since the optimal transformation is different for $\mathbf{Q}, \mathbf{K}, \mathbf{V}$, each learns their own matrix $\mathbf{W}$.

Transformer Models

Multi-Head Attention

- Multiple attention mechanisms $1...l$ each with different learned parameters $\mathbf{W}_q^l$, $\mathbf{W}_k^l$,$\mathbf{W}_v^l$ called *heads* can be combined
- each head produces a different output; these are concatenated and passed through a projection to reduce their dimension back to the original
- as each attention head can learn different weightings of representations, they can encode different relationships between tokens in a sequence

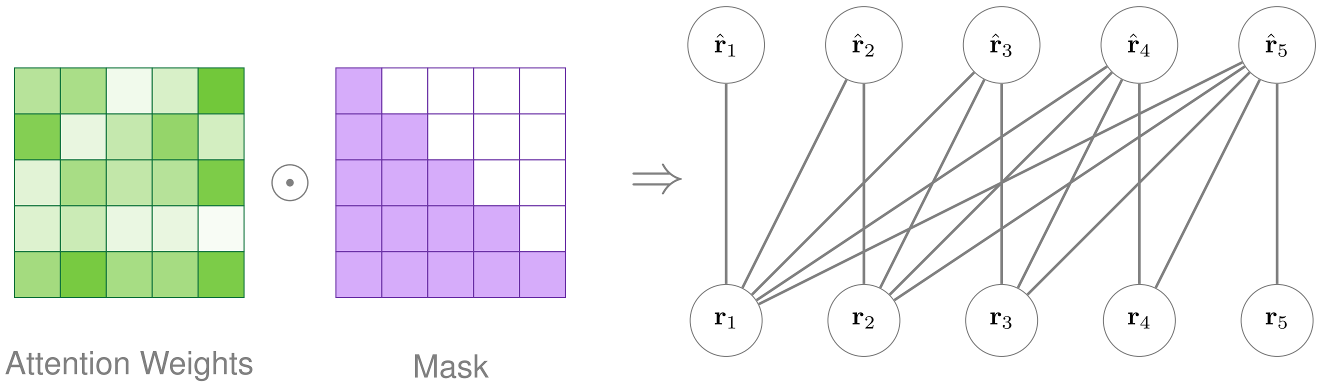


$\mathbf{r}_1$
$\mathbf{r}_2$
$\mathbf{r}_3$
$\mathbf{r}_4$
$\mathbf{r}_5$
$\mathbf{W}_q^1$
$\mathbf{W}_k^1$
$\mathbf{W}_v^1$
$\mathbf{Q}^1$
$\mathbf{K}^1$
$\mathbf{V}^1$
$\mathbf{W}_q^2$
$\mathbf{W}_k^2$
$\mathbf{W}_v^2$
$\mathbf{Q}^2$
$\mathbf{K}^2$
$\mathbf{V}^2$
$\mathbf{W}_q^3$
$\mathbf{W}_k^3$
$\mathbf{W}_v^3$
$\mathbf{Q}^3$
$\mathbf{K}^3$
$\mathbf{V}^3$
$\hat{\mathbf{r}}_1^1$ $\hat{\mathbf{r}}_2^1$ $\hat{\mathbf{r}}_3^1$ $\hat{\mathbf{r}}_4^1$ $\hat{\mathbf{r}}_5^1$
$\hat{\mathbf{r}}_1^2$ $\hat{\mathbf{r}}_2^2$ $\hat{\mathbf{r}}_3^2$ $\hat{\mathbf{r}}_4^2$ $\hat{\mathbf{r}}_5^2$
$\hat{\mathbf{r}}_1^3$ $\hat{\mathbf{r}}_2^3$ $\hat{\mathbf{r}}_3^3$ $\hat{\mathbf{r}}_4^3$ $\hat{\mathbf{r}}_5^3$
Projection
$\hat{\mathbf{r}}_1$ $\hat{\mathbf{r}}_2$ $\hat{\mathbf{r}}_3$ $\hat{\mathbf{r}}_4$ $\hat{\mathbf{r}}_5$

Transformer Models

Masked Attention

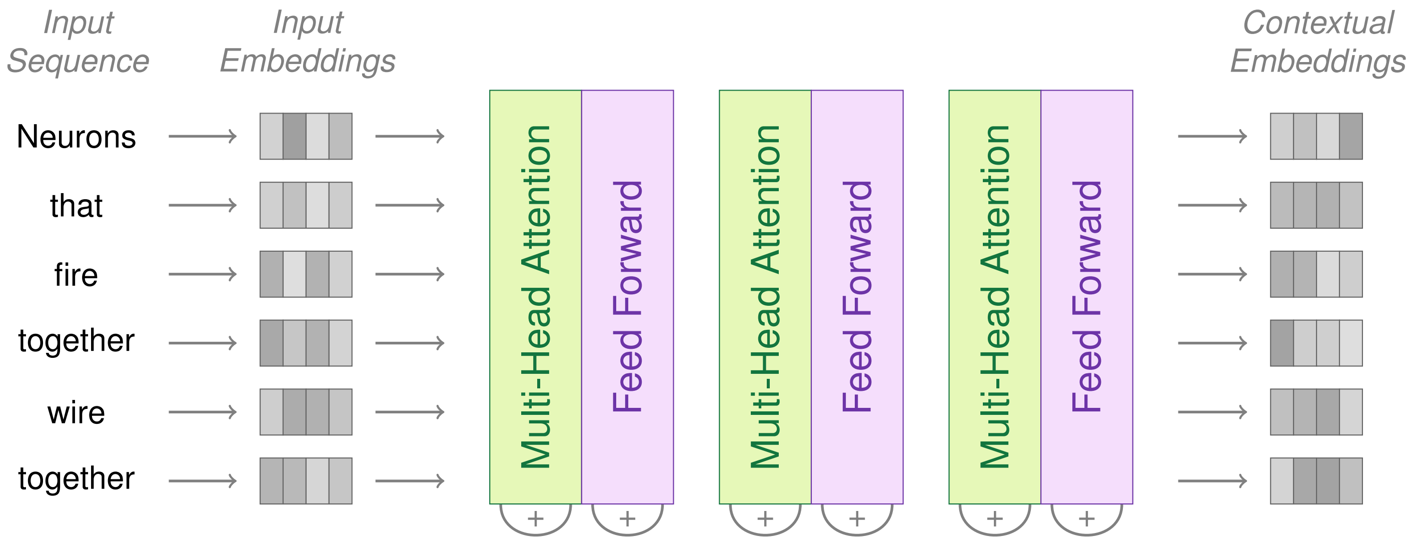
- Masking is used to prevent attention to certain tokens
- A binary mask is applied to the weight matrix
  - masking has to commence before normalization to not influence scores
  - masked scores are set to $-\infty$, thus resulting in a 0 after the $\mathrm{softmax}$
- For example, the mask below can be used to have every token attend only to tokens before it (causal language modeling)



Transformer Models

Transformer Architecture

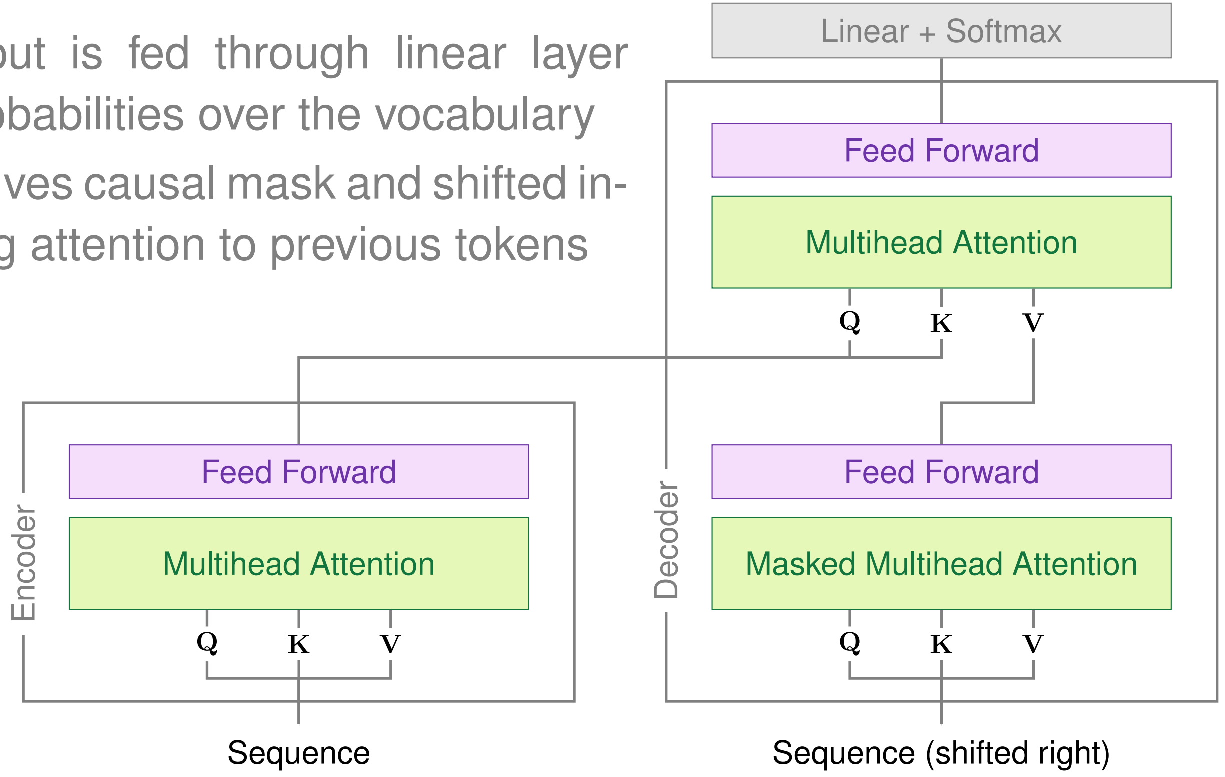
- Multiple attention blocks can be stacked to form a Transformer model
- A fully connected feed-forward layer is applied to each embedding separately and identically after each attention block
  - this allows the Transformer to learn complex relationships (non-linear)
  - repeated attention without would compute only weighted averages (linear)
- Residual connections ('+') are added to add a portion of the input back to the output of each layer (yields stabler gradients due to shorter signal path)



Transformer Models

Encoder-Decoder Models

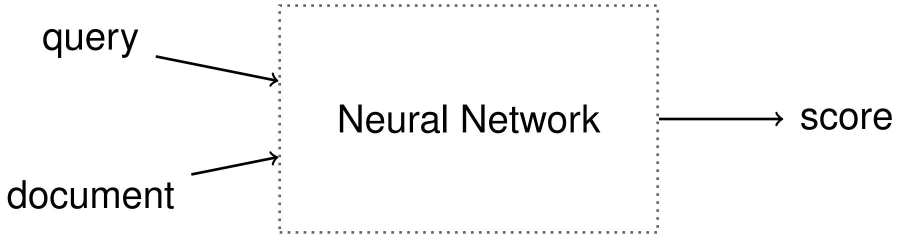
- The stack from the previous slide is commonly called an Encoder
  ➜ produces contextualized embeddings for an input sequence
- It can be coupled with a Decoder which adds cross-attention
  ➜ applies encoder embeddings as query and keys to own values
  - decoder output is fed through linear layer predicting probabilities over the vocabulary
  - decoder receives causal mask and shifted input, restricting attention to previous tokens



Transformer Retrieval Models

Overview

Goal:

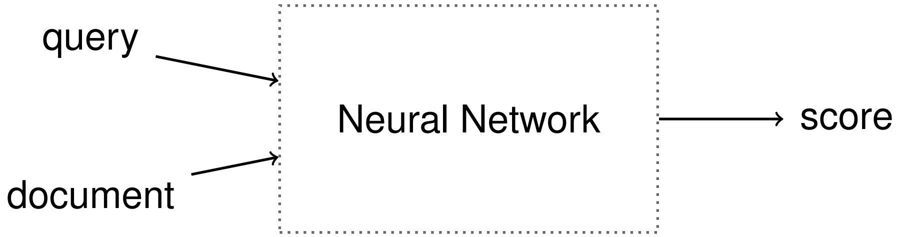


Problem: How do we represent text so we can feed it to the neural network?

Transformer Retrieval Models

Overview

Goal:



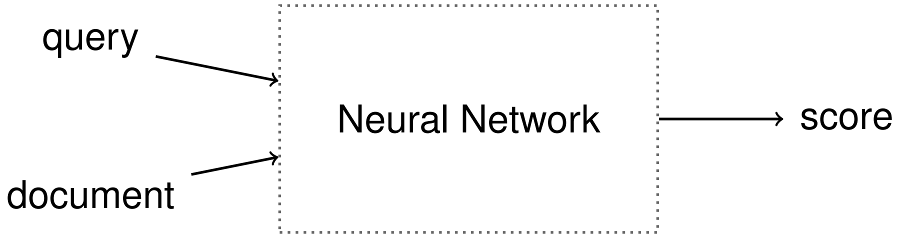
Problem: How do we represent text so we can feed it to the neural network?

Solution: text embeddings

Transformer Retrieval Models

Overview

Goal:



Problem: How do we represent text so we can feed it to the neural network?

Solution: text embeddings

New problem: How do we train the neural network to score documents?

➜ How can we architecture the transformer model to score documents?

Transformer Retrieval Models

Model Architecture: Encoder-Decoders, Encoders, Decoders

Encoder-Decoder

- Feed query+document into encoder.
- Decode token(s) representing score/use token logits as score.
- Possible models: T5 [Raffel et al. 2020], FLAN-T5 [Chung et al. 2022]

Encoder Only

- Feed query+document into encoder.
- Use contextualised embeddings to calculate score.
- Possible models: BERT [Devlin et al. 2019]

Decoder Only

- Feed query+document+prompt into decoder.
- Decode token(s) representing score/use logits as score.
- Possible models: GPT-2 [Radford et al. 2019], LLaMA [Touvron et al. 2023]

Transformer Retrieval Models

Model Architecture: Encoder-Decoders, Encoders, Decoders

Encoder-Decoder

- Feed query+document into encoder.
- Decode token(s) representing score/use token logits as score.
- Possible models: T5 [Raffel et al. 2020], FLAN-T5 [Chung et al. 2022]

Encoder Only

- Feed query+document into encoder.
- Use contextualised embeddings to calculate score.
- Possible models: BERT [Devlin et al. 2019]

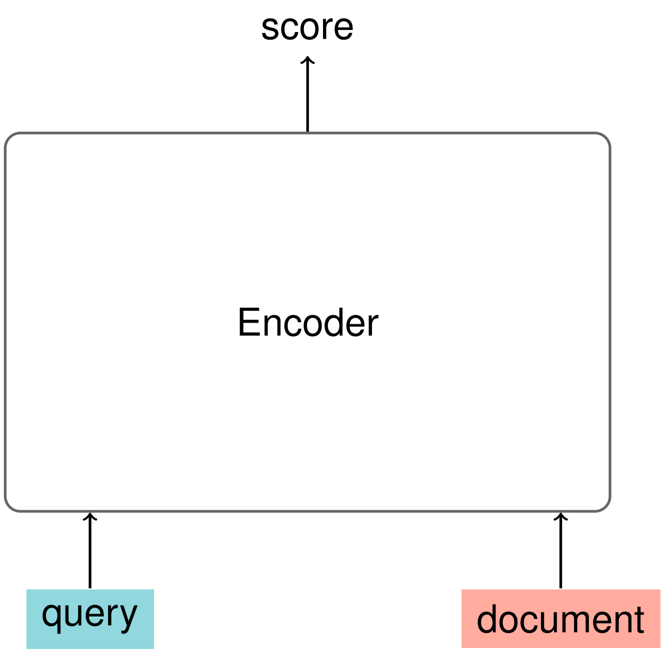
Decoder Only

- Feed query+document+prompt into decoder.
- Decode token(s) representing score/use logits as score.
- Possible models: GPT-2 [Radford et al. 2019], LLaMA [Touvron et al. 2023]

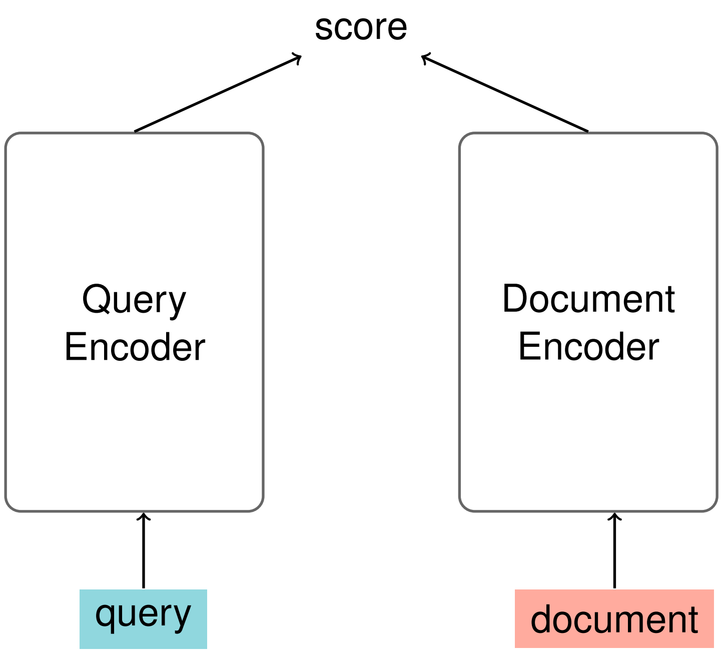
Can you already see some potential effectiveness/efficiency trade-offs?

Transformer Retrieval Models

Model Architecture: Cross-Encoders, Bi-Encoders



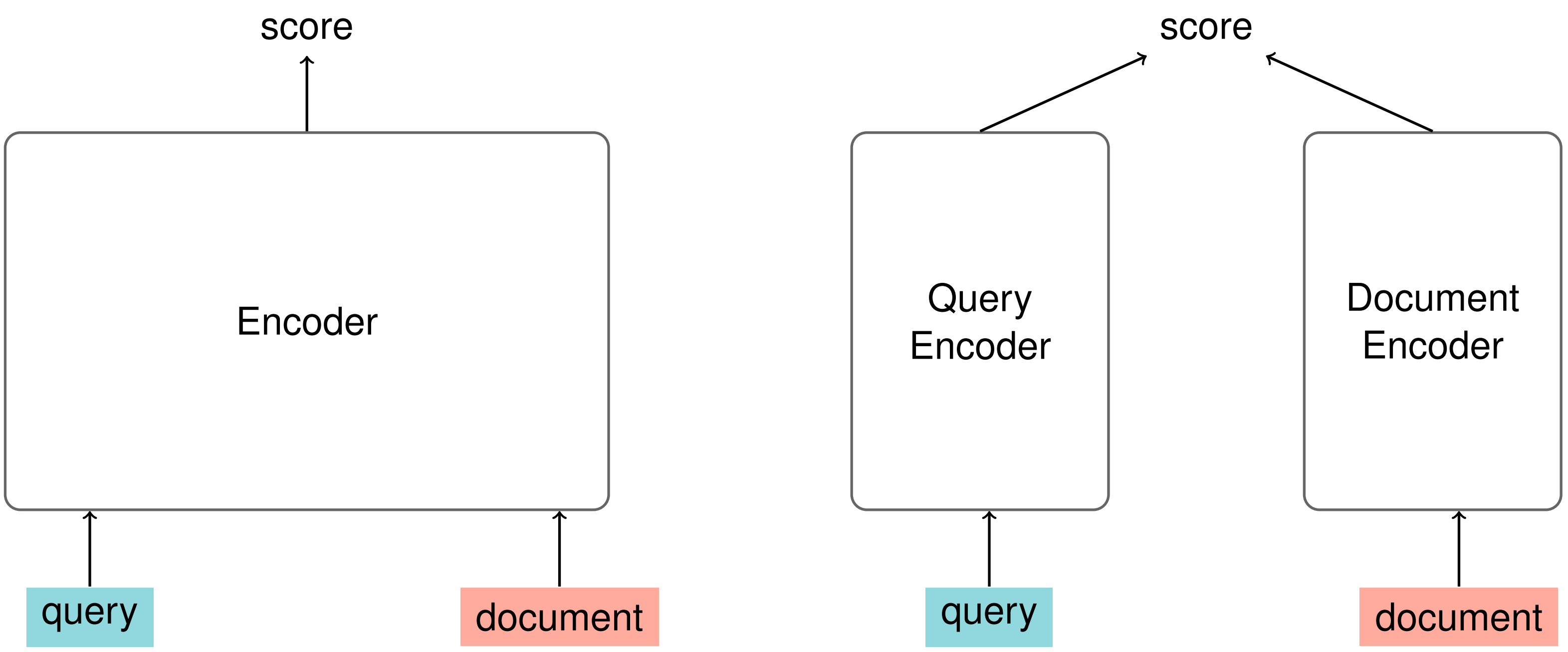
a.k.a Interaction Retrieval Models



a.k.a Dense Retrieval Models

Transformer Retrieval Models

Model Architecture: Cross-Encoders, Bi-Encoders



a.k.a Interaction Retrieval Models

a.k.a Dense Retrieval Models

Why do you think these two model architectures exist?

Transformer Retrieval Models

Model Architecture: Pointwise, Pairwise, Listwise

Pointwise

- cross-encoder

  monoBERT [Nogueira et al. 2019]

- bi-encoders

  DPR [Karpukhin et al. 2020], TILDE [Zhuang and Zuccon 2021], ColBERT [Khattab and Zaharia 2020]

Pairwise

- cross-encoder

  duoBERT [Nogueira et al. 2019]

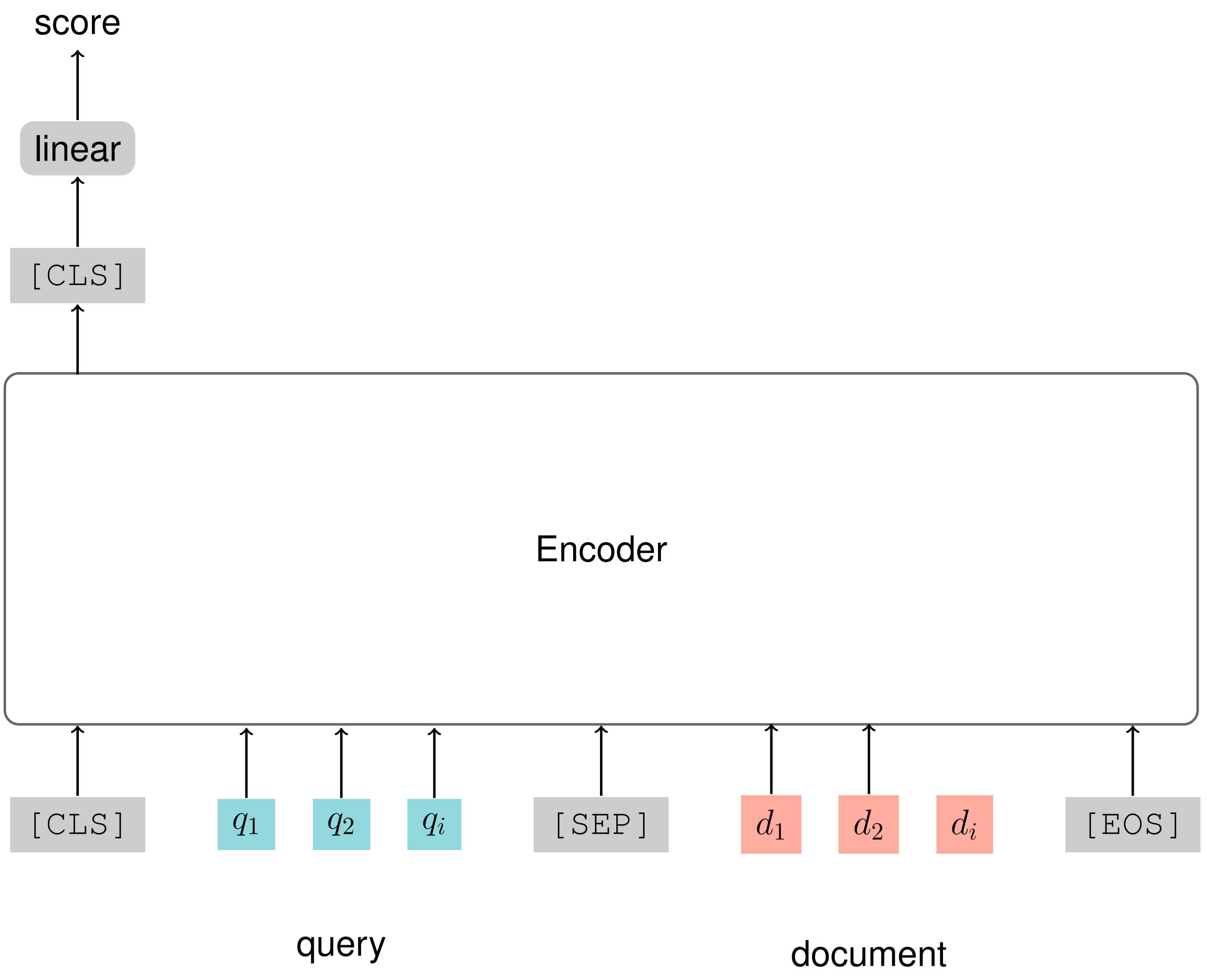
Listwise

- cross-decoder

  LRL [Ma et al. 2023]

Transformer Retrieval Models

Model Architecture: monoBERT



Transformer Retrieval Models

Model Architecture: monoBERT

Scoring a document:

- ❑ Estimate a score s quantifying how relevant d is to a q.

$$\rho(q, d) = s(q, d)$$

- ❑ Use `[CLS]` embedding as input to single layer neural network to predict s.

Training the model:

- ❑ Cross-entropy loss:

$$\mathcal{L} = -\sum_{i \in D^+} \log(s(q, d_i)) - \sum_{j \in D^-} \log(1 - s(q, d_j))$$

- ❑ D^+ and D^- are the sets of relevant and non-relevant documents.
  - In other words, positive and negative samples.
- ❑ We want the model to assign relevant documents higher scores than non-relevant documents.

Transformer Retrieval Models

Model Architecture: monoBERT

Pros:

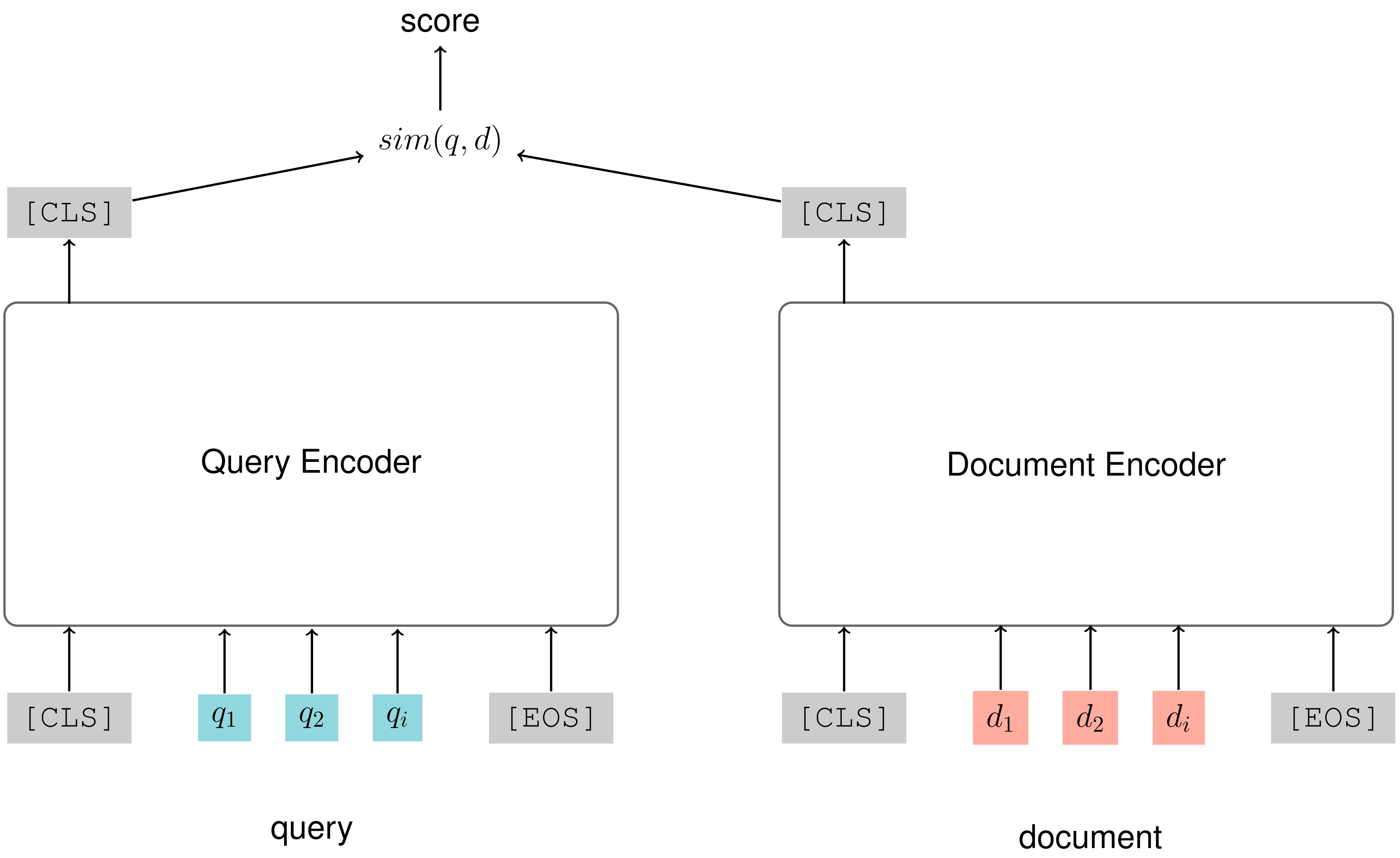
- Very straightforward model to train.
- Interaction between query and document within the model makes it effective.

Cons:

- Fixed input size means long queries/documents must be truncated.
  - Query and document must fit into same input.
- Expensive at inference time, less effective than monoT5 [Pradeep et al. 2021].
  - Longer input sequences make inference slower.
  - Attention scales quadratically.
  - Not possible to pre-compute or index documents.

Transformer Retrieval Models

Model Architecture: DPR



Transformer Retrieval Models

Model Architecture: DPR

Scoring a document:

- Quantify relevance as similarity between q and d,

$$\rho(q,d) = sim(q,d) = E_Q(q)^\top E_D(d)$$

- Intuitively, DPR is equivalent to vector space model.

Training the model:

- Negative log likelihood loss for a single relevant document:

$$\mathcal{L} = -\log \frac{e^{sim(q,d^+)}}{e^{sim(q,d^+) + \sum_{j=1}^{n} e^{sim(q,d_i^-)}}}$$

- Where d^+ is the relevant document and d_i^- is a non-relevant document.
- We want both encoder models to learn representations where queries and relevant documents are similar, while pushing away non-relevant documents.
- Many negative examples allow the model to learn much faster.

Transformer Retrieval Models

Model Architecture: DPR

Pros:

- ❑ Query and document lengths can be full input size ➜ less need for truncation.
  - – Separate encoders for query and document.
  - – Not necessarily required to use two separate encoders.
- ❑ Once trained, document embeddings can be pre-computed.
  - – Fewer computations at inference time, faster than monoBERT.
  - – Index document embeddings using approximate nearest neighbour (ANN) index, e.g., `faiss` [Johnson et al. 2017]

Cons:

- ❑ No interaction between query and document within the model.
  - – Not as effective as monoBERT.
- ❑ Need to re-compute document embeddings if document encoder is changed.
- ❑ Negative sampling for fine-tuning can have a big impact on effectiveness.

Transformer Retrieval Models

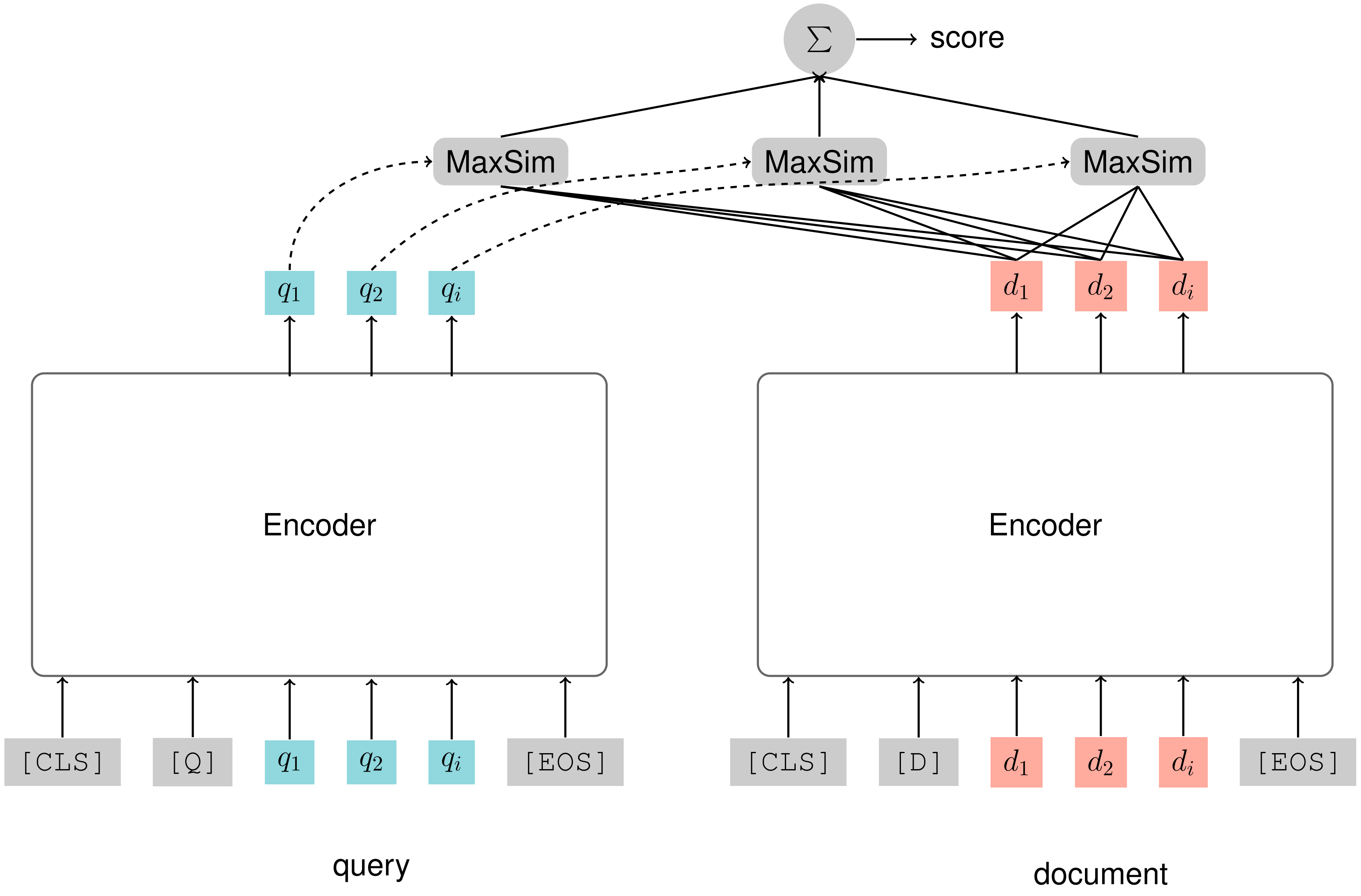
Model Architecture: DPR

Related models:

- ❑ ANCE [Xiong et al. 2020]
  - – Same architecture as DPR, but does ‘hard negative mining’ globally in the collection ➜ negative contrastive estimation.
  - – Intuition is that if BM25 is used for negative samples, model may favour lexical matching.
- ❑ RepBERT [Zhan et al. 2020]
  - – Same architecture as DPR, but uses shared encoder to create token embeddings.
  - – Similarity between mean query/document token embeddings used to score documents.

Transformer Retrieval Models

Model Architecture: ColBERT



Transformer Retrieval Models

Model Architecture: ColBERT

Scoring a document:

- Estimate ρ via "late interaction", the summation of maximum similarity between query tokens and document tokens.

$$\rho(q, d) = \sum_{i \in [|E(q)|]} \max_{j \in [|E(d)|]} E(q_i)^\top E(d_j)$$

Training the model:

- Pairwise softmax cross-entropy loss

$$\mathcal{L} = -(d^+ - d^-) + \log(1 + \exp(d^+ - d^-))$$

- Where d^+ is a relevant document and d^- is a non-relevant document.
- We want the score predicted for the relevant document to be as high as possible compared to the non-relevant document.

Transformer Retrieval Models

Model Architecture: ColBERT

Pros:

- Like DPR, document embeddings can be pre-computed for efficiency.
- Unlike DPR, the same encoder learns representations for queries *and* documents.
  - Although, it is possible to use the same encoder with DPR.
- Late interaction mechanism makes ColBERT effective and efficient.
  - Approximately effective as monoBERT; approximately efficient as BM25.
  - Possible to perform end to end retrieval.

Cons:

- Like DPR, need to re-compute documents if model is re-trained.
- Index size is significantly larger than DPR.
  - For each document, need to index document token embeddings.

Transformer Retrieval Models

Model Architecture: ColBERT

Efficient retrieval with ColBERT:

- Indexing
  1. Index document token embeddings into ANN index.
  2. Create mapping of document token embedding to document.
- Querying
  1. Issue N_q vector similarity queries to ANN index.
  2. Retrieve top-k document token embeddings.
  3. Map each document token embedding producing $k \cdot N_q$ document IDs.
  4. Remove duplicate document IDs, forming K documents.
  5. Map document IDs to document embeddings within ANN index.
  6. Proceed to score K documents normally.

Transformer Retrieval Models

Model Architecture: ColBERT

ANN Index	Token Positions	Document Offsets	Summed Offsets
$d_1 t_1$	1		
$d_1 t_2$	2		
$d_1 t_3$	3	3	3$\rightarrow d_1$
$d_2 t_1$	4		
$d_2 t_2$	5	2	5$\rightarrow d_2$
$d_3 t_1$	6		
$d_3 t_2$	7		
$d_3 t_3$	8	3	8$\rightarrow d_3$

Query token top-1: $d_2 t_1$

Map $d_2 t_1$ to d_2: token position, 3(d_1)>4($d_2 t_1$)>8(d_3)

Collect d_2 document tokens for scoring: last token=5, first token=5-2-1

Transformer Retrieval Models

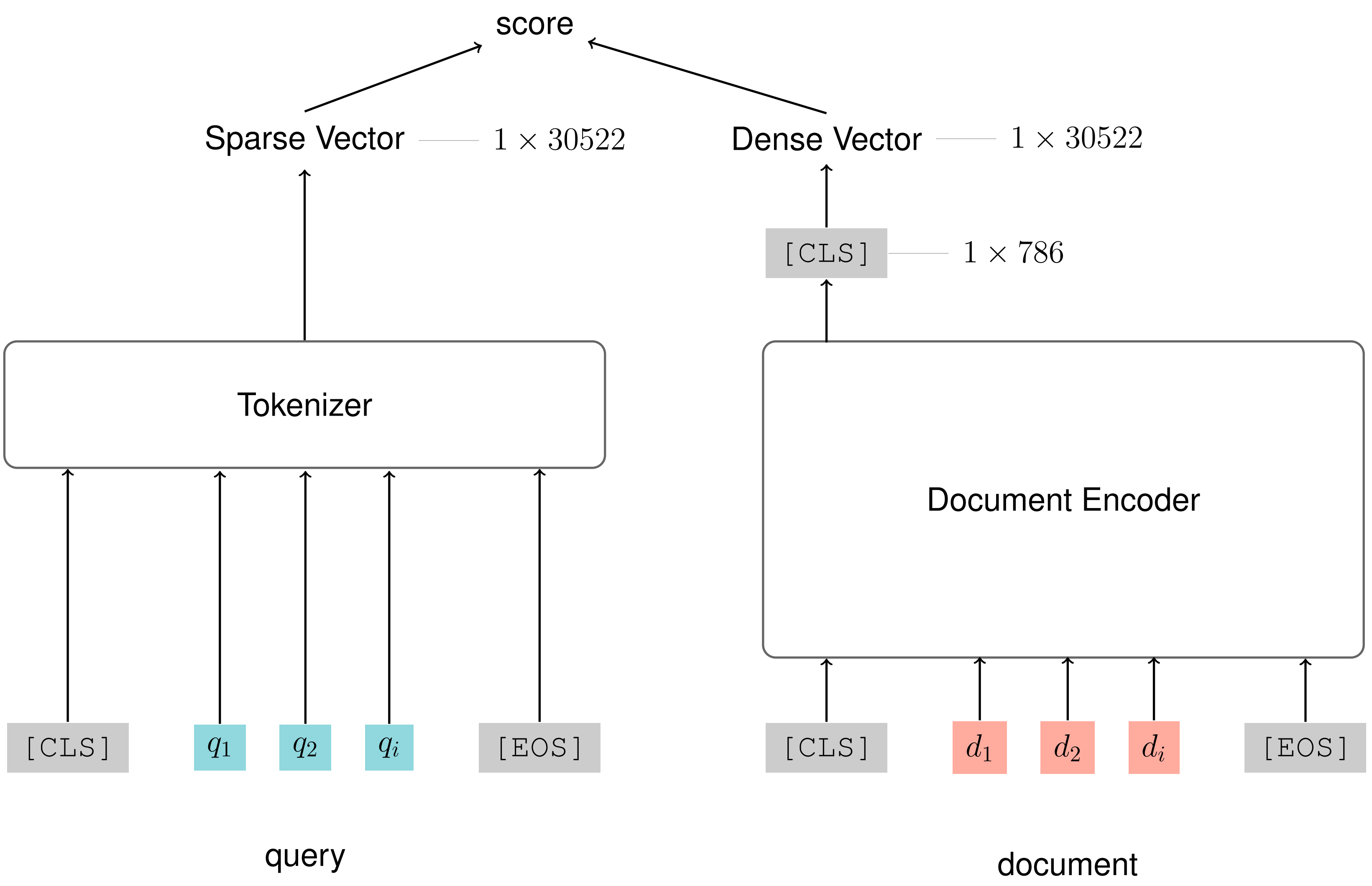
Model Architecture: ColBERT

Related models:

- ColBERTv2 [Santhanam et al. 2022]
  - Implements aggressive compression over ANN index to reduce space footprint.

Transformer Retrieval Models

Model Architecture: TILDE



Transformer Retrieval Models

Model Architecture: TILDE

Scoring a document:

- Term independent query-document likelihood:

$$\rho(q,d) = \alpha \cdot \sum_{i}^{|q|} \log(P(q_i|d)) + (1-\alpha) \cdot \frac{1}{|d|} \sum_{i}^{|d|} \log(P(d_i|q))$$

- Query likelihood (QL) is the summed log probabilities for each query term appearing in the document.
- Document likelihood (DL) is the summed log probabilities for each document term appearing in the query.
  - Must divide by $|d|$ since longer documents have lower log probabilities.
- α parameter controls weight associated with QL or DL components.

Transformer Retrieval Models

Model Architecture: TILDE

Training the model:

- Assume independence between tokens, use traditional QL loss function:

$$\mathcal{L}_{QL} = -\sum_{(q,d)\in J} \frac{1}{|V|} \sum_{i}^{V} y \log(P(t_i|d)) + (1-y)\log(1-P(t_i|d)),$$

$$\mathcal{L}_{DL} = -\sum_{(q,d)\in J} \frac{1}{|V|} \sum_{i}^{V} y \log(P(t_i|q)) + (1-y)\log(1-P(t_i|q)),$$

$$\mathcal{L} = \frac{\mathcal{L}_{QL} + \mathcal{L}_{DL}}{2}$$

- Where y is a binary variable which equals 1 if $t_i \in q$ else 0.
- For $\mathcal{L}_{QL}$ d is encoded for a dense representation to compute log probabilities and q is tokenized for a sparse representation; the opposite is done for $\mathcal{L}_{DL}$.
- We want the model to increase the probability of generating a given token if the document is relevant, otherwise decrease the probability.

Transformer Retrieval Models

Model Architecture: TILDE

Pros:

- When $\alpha = 1$, only QL is computed, which only requires tokenization of q.
  - Pre-compute log probabilities for all documents, store in lookup table.
  - Inference can be done on CPU very efficiently (faster than ColBERT).
- Even when $\alpha < 1$, TILDE is still much more efficient than monoBERT, while only slightly less effective.

Cons:

- α parameter requires tuning.
- Indexing is much slower and indexes are much larger compared to DPR.
  - Pruning under-representative tokens can reduce index size.

Transformer Retrieval Models

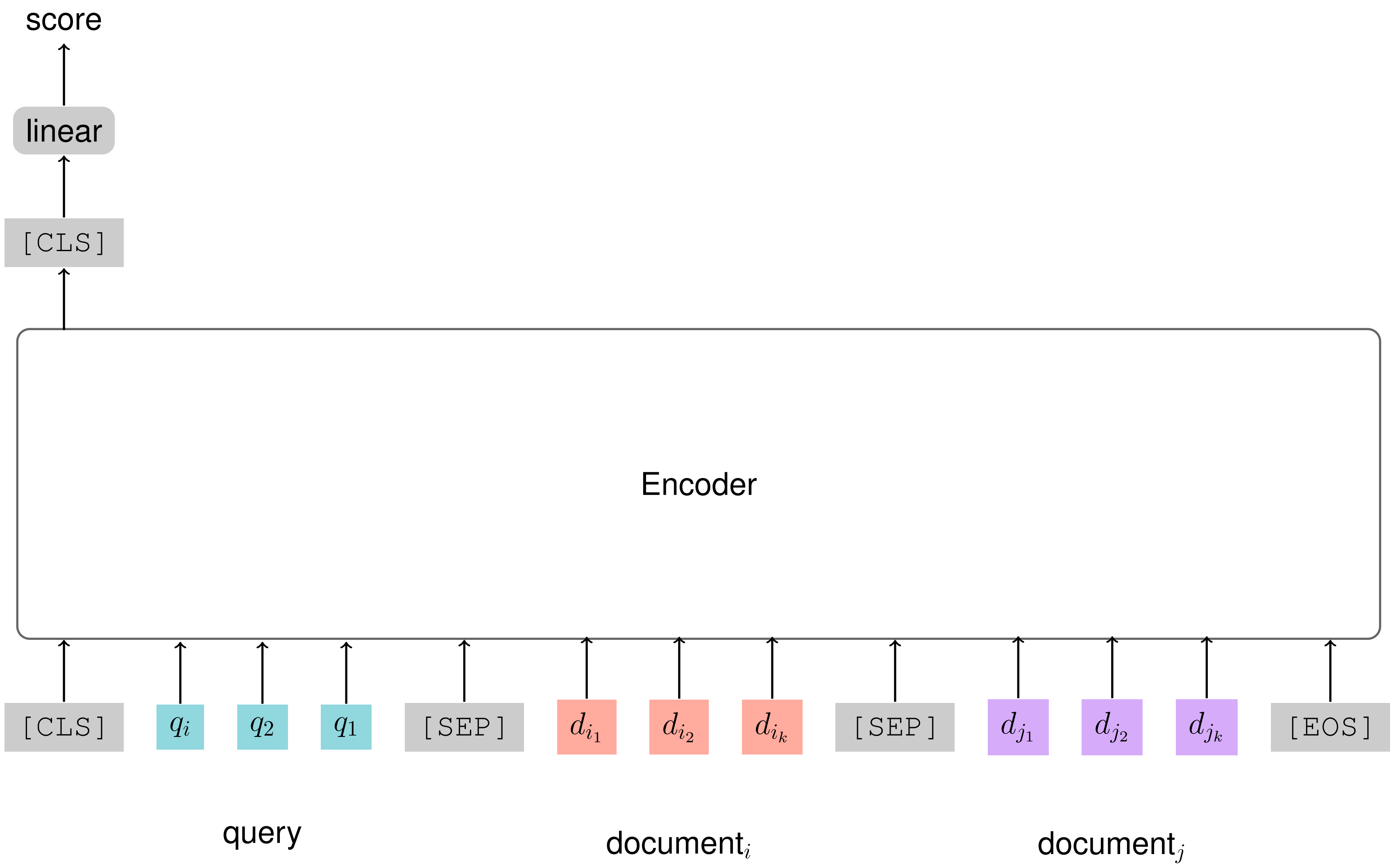
Model Architecture: TILDE

Related models:

- TILDEv2 [Zhuang and Zuccon 2021]
  - Prune document index by only storing scores of tokens inside a document (rather than a dense vector of size $|V|$).
  - Overcome exact term matching limitation now introduced with document expansion.
    - Token likelihood distribution from TILDE to choose expansion terms.
  - 99% lower index size; 25% better ranking effectiveness.

Transformer Retrieval Models

Model Architecture: duoBERT



Transformer Retrieval Models

Model Architecture: duoBERT

Scoring a document:

- Aggregate the pairwise scores between all document pairs:

$$\rho(q, D) = \sum_{d_i \in D} \sum_{d_j \in \{\forall\ d \in D\ :\ d \neq d_i\}} s(q, d_i, d_j)$$

- Other aggregation methods exist, sum shown for simplicity.

Training the model:

- Same strategy as monoBERT, but with two documents at a time.
- Cross-entropy loss:

$$\mathcal{L} = -\sum_{i \in J^+, j \in J^-} \log(s(q, d_i, d_j)) - \sum_{i \in J^+, j \in J^-} \log(1 - s(q, d_i, d_j))$$

- We want the scores of relevant+non-relevant documents to be higher than scores of non-relevant+non-relevant.

Transformer Retrieval Models

Model Architecture: duoBERT

Pros:

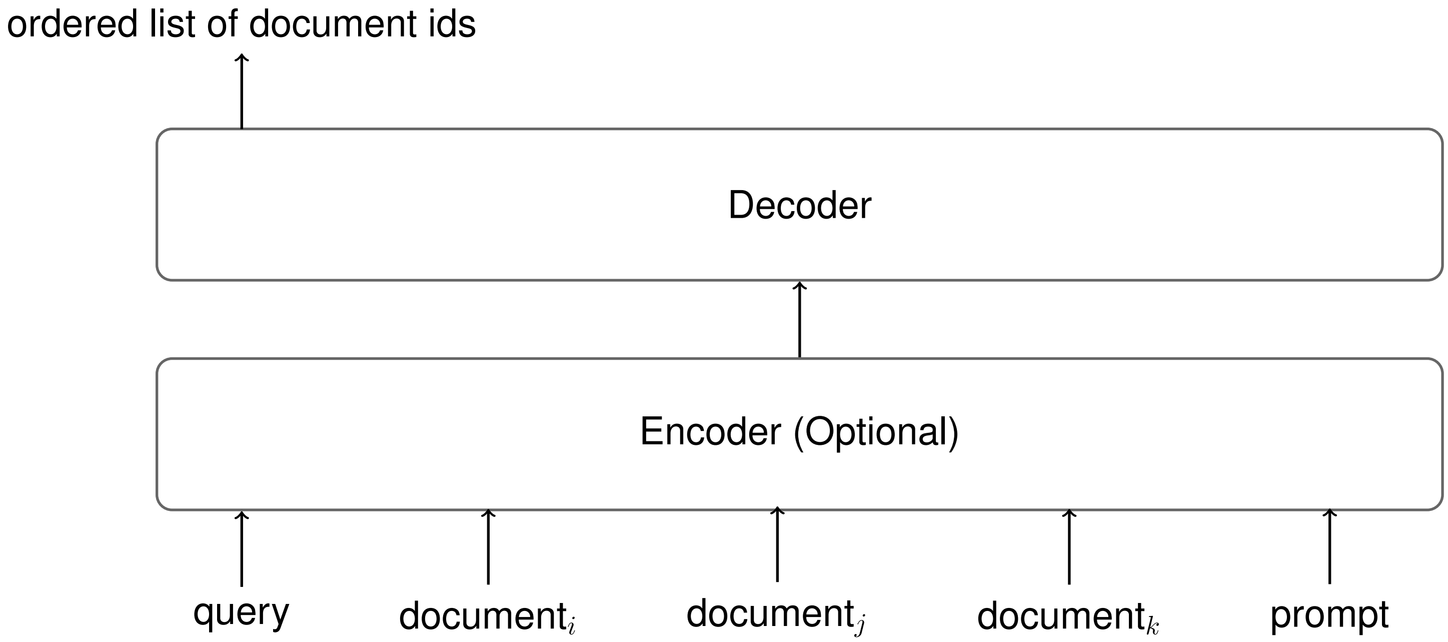
- Much higher effectiveness compared to monoBERT.

Cons:

- Document length can only be half as long as monoBERT.
  - Must fit two documents into input sequence, plus query.
- Document scoring requires $k(k-1)/2$ comparisons, compared to k comparisons for monoBERT.
  - Can only reasonably score significantly fewer documents at inference.

Transformer Retrieval Models

Model Architecture: LRL



Transformer Retrieval Models

Model Architecture: LRL

Scoring a document:

- Design a prompt template to prime the decoder:

```
Document1 = {document_1}
...
Document10 = {document_10}
Query = {query}
Documents = {Document1,..Document10}
Sort the Documents by their relevance to the Query.
Sorted Documents = [
```

Training the model:

- No training (i.e., fine-tuning) necessary.
  - LRL is “zero-shot”; we don’t show it any query-document pairs for training.

Transformer Retrieval Models

Model Architecture: LRL

Pros:

- Comparable to monoBERT and ColBERT effectiveness, no training needed.

Cons:

- Input size is still limited, often several thousand tokens.
  - Can only rank a handful of documents at a time.
  - To rank long lists, start at the bottom of list and slide window up (progressive ranking).
- Decoders require a full forward pass through the transformer for each token.
  - Inference time scales linearly with number of documents to rank.
  - Still, more efficient at inference time than duoBERT.
- Difficult to constrain the output.
  - Model is just predicting next token, output will possibly be ill-formed.
- Initial document ordering may bias the model.

Transformer Retrieval Models

Model Architecture: LRL

Related models:

- Setwise [Zhuang et al. 2023]
  - Use logits to predict document order, only requires a single forward pass.
- RankZephyr [Pradeep et al. 2023]
  - Use knowledge distillation from GPT-{3,4} to fine-tune decoder model.
- *Rank-without-GPT* [Zhang et al. 2023]
  - Use instruction fine-tuning to improve consistency and effectiveness.

Note:

- Several other decoder models that use the same intuition also for pointwise and pairwise ranking.

Transformer Retrieval Models

Retrieval Pipelines

First-stage retrieval➜ retrieving documents from an index to rank.

Re-ranking ➜ ranking documents from first-stage retrieval.

Generally, transformer retrieval models are not used as first stage rankers

- Although DPR and ColBERT can be used as both.

Retrieval Pipelines

- Usually start with cheap ranking method like BM25, exploit inverted index.
- First top-k documents (usually 1000) re-ranked by pointwise model.
- Next top-k documents (usually 100) re-ranked by pairwise model.
- Final top-k documents (usually 10) re-ranked by listwise model.

Transformer Retrieval Models

Retrieval Pipelines

First-stage retrieval➜ retrieving documents from an index to rank.

Re-ranking ➜ ranking documents from first-stage retrieval.

Generally, transformer retrieval models are not used as first stage rankers

- ❑ Although DPR and ColBERT can be used as both.

Retrieval Pipelines

- ❑ Usually start with cheap ranking method like BM25, exploit inverted index.
- ❑ First top-k documents (usually 1000) re-ranked by pointwise model.
- ❑ Next top-k documents (usually 100) re-ranked by pairwise model.
- ❑ Final top-k documents (usually 10) re-ranked by listwise model.

What are some potential advantages/disadvantages with such a retrieval pipeline?
Think about efficiency (time/space) vs effectiveness trade-offs.

Transformer Retrieval Models

Takeaway Questions and Further Reading

Questions

- What benefit do contextualised word embeddings provide over word embeddings (e.g., word2vec)?
- How would you design a decoder-only model for pointwise and pairwise ranking?
- Why don't we see many listwise methods using encoders? In other words, what are the reasons listwise methods can be more easily implemented in decoder models than encoder models?

Further Reading

- Lossy compression for ANN indexes [Mackenzie and Moffat 2023]
- Learning sparse representations of queries+documents (SPLADE) [Formal et al. 2021], an exploration of SPLADE [Mackenzie et al. 2023]
- Energy [Scells et al. 2022] and water [Zuccon et al. 2023] measurements of transformer retrieval models.