Introduction to Neural Re-Ranking

Sebastian Hofstätter

sebastian.hofstaetter@tuwien.ac.at
/s_hofstaetter

Today

Introduction to Neural Re-Ranking

1 Re-Ranking Workflow

- Connection to first stage ranker
- Training & evaluation

2 Basic Re-Ranking Models

- Model-stages
- MatchPyramid (inspired by image classification)
- Kernel-pooling (KNRM, CONV-KNRM)
- Influence of vocabulary

Disclaimer: Text Based Neural Models

- Learning-to-rank systems in production can use many features
  - Click count, recency, source quality … (with classical ML: regression, SVM, trees …)
  - Plus, increasingly text-based neural models

- Deploying a re-ranker in production is a huge effort
  - Performance, Quality control, version management, changing indices, …

- This lecture: neural models for **content-based ad-hoc retrieval**
  - We only use the text of the query and document
  - Can be thought of as another signal for a learning-to-rank system
  - Mainly because open test collections are either text-only or features-only

Disclaimer: The Basics are not SOTA Anymore

- In this lecture we look at some basic models
  - Transformer-based models surpassed these models now in terms of quality
  - Still interesting and informative in a historical way (meaning 3 years ago 😂)
  - Many of the intuitions now guide our work with Transformers
  - Much faster and resource efficient to train
    - That’s why we use them in our exercise (so stay tuned)

- The re-ranking workflow is still used today
  - And BERT is trained in the same way

More about Transformers, BERT (SOTA models), and dense retrieval in the coming lectures

Desired Properties of Neural IR Models*

- **Effective**
  - They should work – duh! Otherwise what's the point of using them

- **Fast & Scalable**
  - 10 to 100 ms time budget for the full re-ranking
  - Potentially TB+ indices (this can vary a lot depending on the use case)

- **Interpretable**
  - Search engines filter a lot of information – they act as a gate
    The reasons for including/excluding certain results needs to be explainable

* Pretty much in line with other ML tasks

Beware! Neural Networks Fail Silently …

- Training neural networks is hard
  - And far from the 50-line example on getting started pages of libraries

- Neural IR model training has even more special things to look out for
  - Wild west of a research field in comparison to computer vision or translation

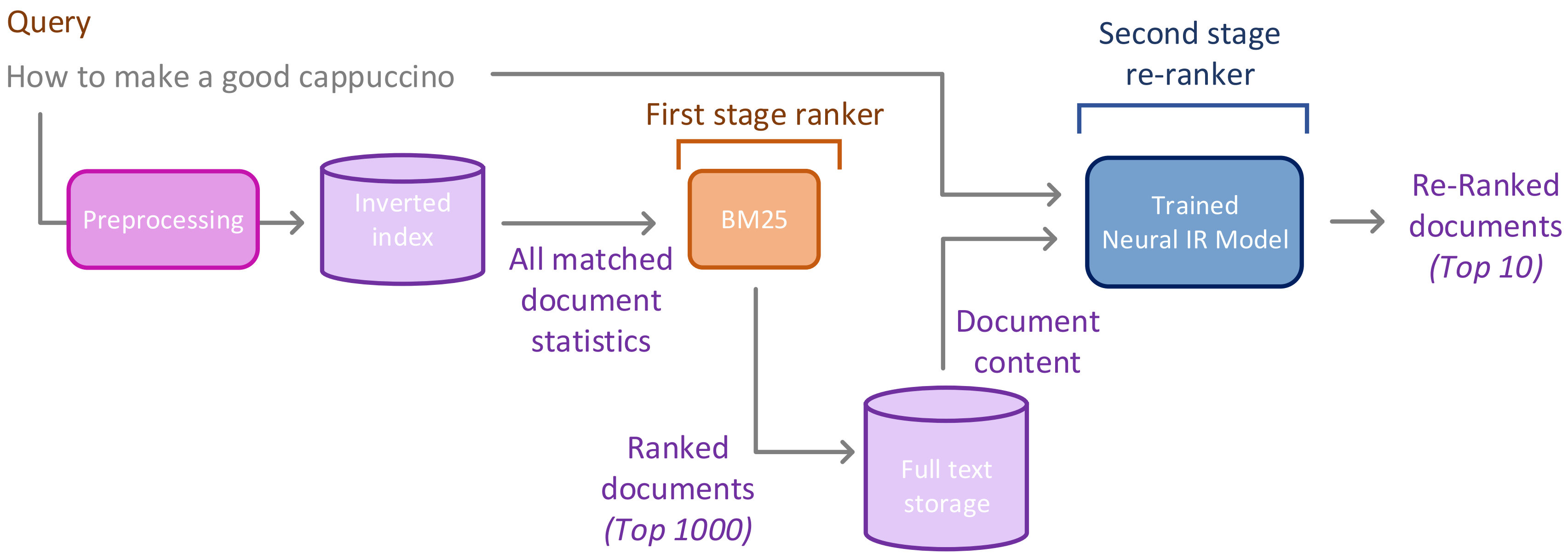
- More information: http://karpathy.github.io/2019/04/25/recipe/
  - Might come in handy for the exercise :)

Re-Ranking Workflow

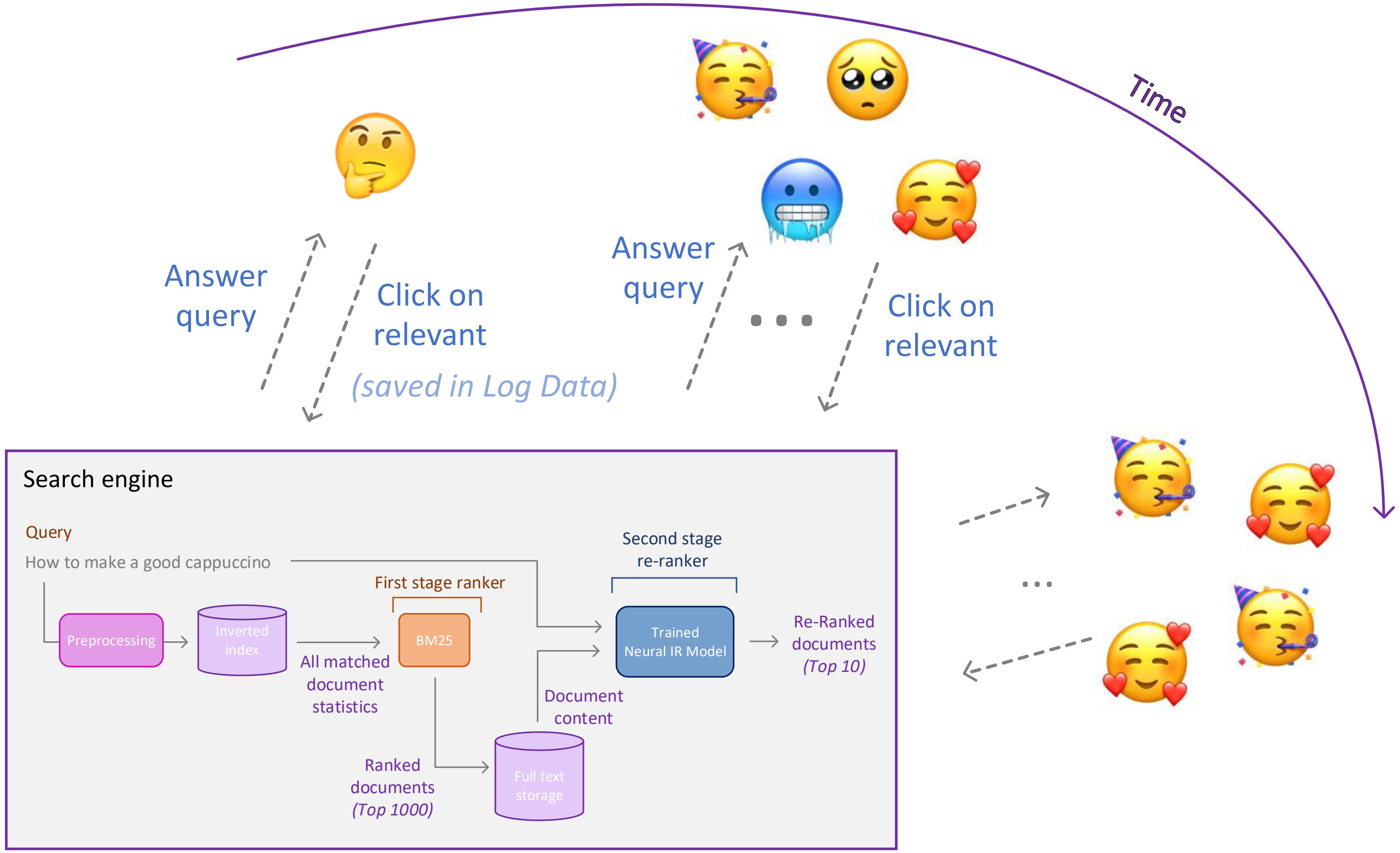
What does it take to run a neural re-ranking pipeline

Neural Re-Ranking Models

- Re-rankers: They change the ranking of a pre-selected list of results
  - Same interface as classical ranking methods: *score(q, d)*
- Query Workflow:



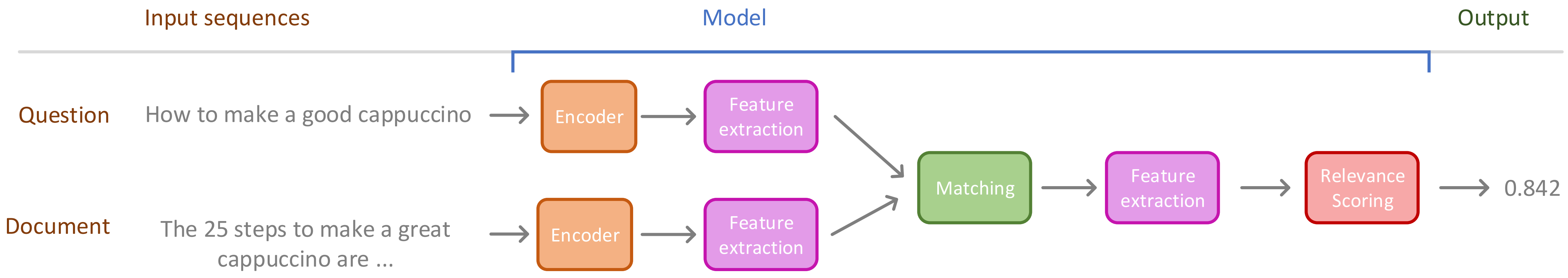
Context



- Search engines have a lot of users
- Interaction is in two-ways:
  - Query results (search engine)
  - Activity data (users)
- Over time logs can be used to improve the search
- Neural IR model trained “offline” and is then swapped

Inside Neural Re-ranking Models

- Core part of re-ranking models is a matching module
  - Operating on a word interaction level



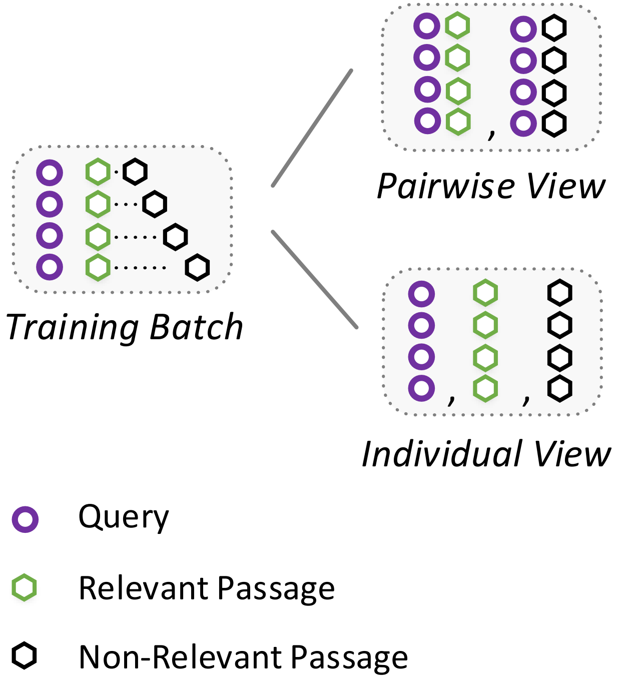
Training

- Training is independent from the rest of the search engine operations
  - But it could be done repeatedly to account for temporal shift in the data

- Neural IR models are typically trained with triples (pairwise +,-)
  - Triple: 1 query, 1 relevant, 1 non-relevant document
  - 2 forward passes: query + relevant doc, query + non-relevant doc
  - Loss function: Maximize margin between rel/non-rel document

- All model components are trained end-to-end
  - Of course we could decide to freeze some parts for more efficient training

Creating Batches



- We form a batch by sampling as many triples as is allowed by the GPU memory
  - Typical batch size: 16-128
  - We mix different queries together
  - Depending on the model we need to create query-passage pairs or run each of the three sequences individually through the model
- We run a backward pass & gradient update per batch
- Sequency inputs come as a single matrix, so we need to pad different length inputs

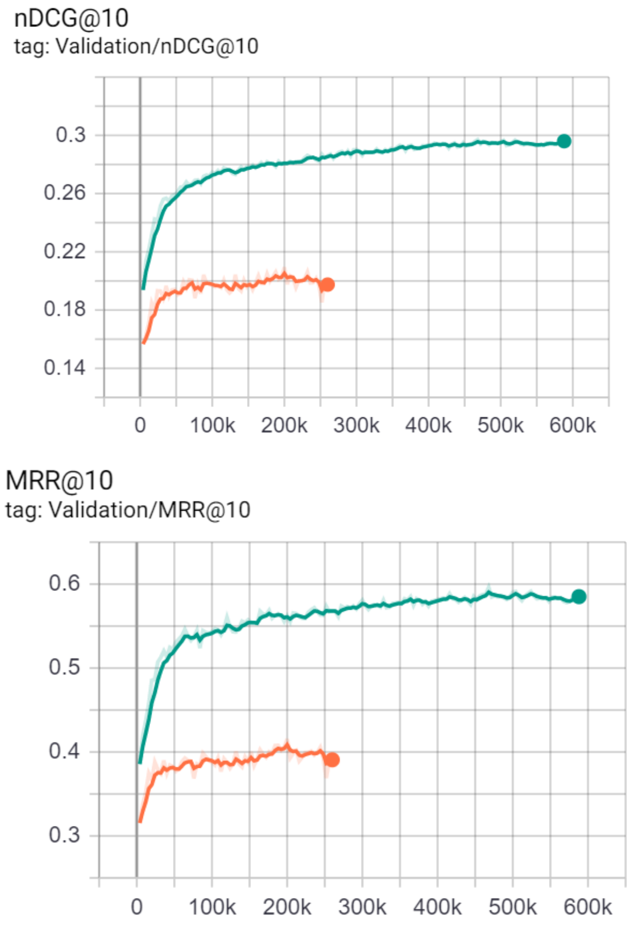
Sampling Non-Relevant Passages

- Most collections only come with judgements of relevant (or false-positive selections from other models) and not truly non-relevant judgements
  - It doesn’t make sense to spend resources annotating random pairs
- We need to tell the model what is non-relevant
  - Simple procedure to sample non-relevant passages:
    - Run BM25 and get the top-1000 results per training query and randomly select a few of those results as non-relevant
    - The non-relevant selections provide some signal
(as there must be at least some lexical overlap)
    - But mostly non-relevant passages -> works pretty good in practice
    - A bit of noise is good (we don’t know the degree of non-relevance, but that’s ok)

More on the composition of batches & sampling in the indexing & dense retrieval lecture

How to Break the Non-Relevant Sampling

- Non-relevant passages need to be truly non-relevant
    - Too many false negatives (actually relevant) *confuse* the model
- If we have click data as relevance signals, we also know about non-clicked passages
    - Sampling them as negatives breaks training (orange)
    - We should not include them as non-relevant in the negative sampling process and only rely on BM25 negatives (green)



Validation results on TripClick-Head(DCTR) using the same training setup with BERT_{CAT}

Loss Functions

- Choice of different methods that aim to maximize the margin between rel & non-rel document
  - Plain Margin Loss:

    `loss = max(0,`s_{nonrel} `–` s_{rel} `+ 1)`

    - Native support in PyTorch: `torch.nn.MarginRankingLoss()`

  - RankNet:

    `loss = BinaryCrossEntropy(`s_{rel} `–` s_{nonrel}`)`

- Both losses assume binary relevance

Deep dive on non-binary label loss function in the knowledge distillation lecture

Re-Ranking Evaluation

- Scoring per tuple (1 query, 1 document)
- List of tuples is then sorted & evaluated with ranking metric per query (for example: MRR@10)
    - MRR@10 = Mean Reciprocal Rank, stop to look at position 10 or first relevant
- Mismatch: You can’t really compare training loss and IR evaluation metric
    - Training loss is only good for checking at the beginning if your network is not completely broken :) – it should go down very quick and then not change

Recall How MS MARCO Looks

- Training triples
  - **Query:** *what fruit is native to Australia*
  - **Relevant:** *Passiflora herbertiana. A rare passion fruit native to Australia. Fruits are green-skinned, …*
  - **Non-Relevant:** *The kola nut is the fruit of the kola tree, a genus (Cola) of trees that are native to the tropical rainforests of Africa.*
- Evaluation tuples
  - **Ids:** 837202 1000252
  - ***Query:*** *what is the nutritional value of oatmeal*
  - ***Document:*** *Oats make an easy, balanced breakfast. One cup of cooked oatmeal contains about 150 calories, four grams of fiber (about half soluble and half insoluble), and six grams of protein. …*

Actual input & output

- 2 Tensors of word ids containing a batch of samples
    - Tensor = multidimensional array abstraction
    - Batch = For efficiency multiple samples are computed in parallel (on the GPU)
- Same dimension for all entries
    - Shorter sequences are padded with 0
- Query tokens:
    - Shape: [batch, query_seq_len]
- Document tokens:
    - [batch, doc_seq_len]

```
[[   58,    83,  5401,   821,    18,     9,     3,   399],
 [   48,    10,     3,  1310,     5,  2654, 16729,     0],
 [   48,    10,  2654,  7545,  3700,     0,     0,     0],
 [  128,    10,  7397,   170,  2652,     0,     0,     0]

…
```

Basic Neural Re-Ranking Models

Before BERT: How are they implemented?

The encoding layer

- Starting point for text processing neural network models
- Word token (id/piece/char based) to dense representation
  - Having word boundaries is important in IR

- Actual implementation or pre-trained-data easily swappable
  - Latest research papers: Evaluating 1 IR model with different encoding layers

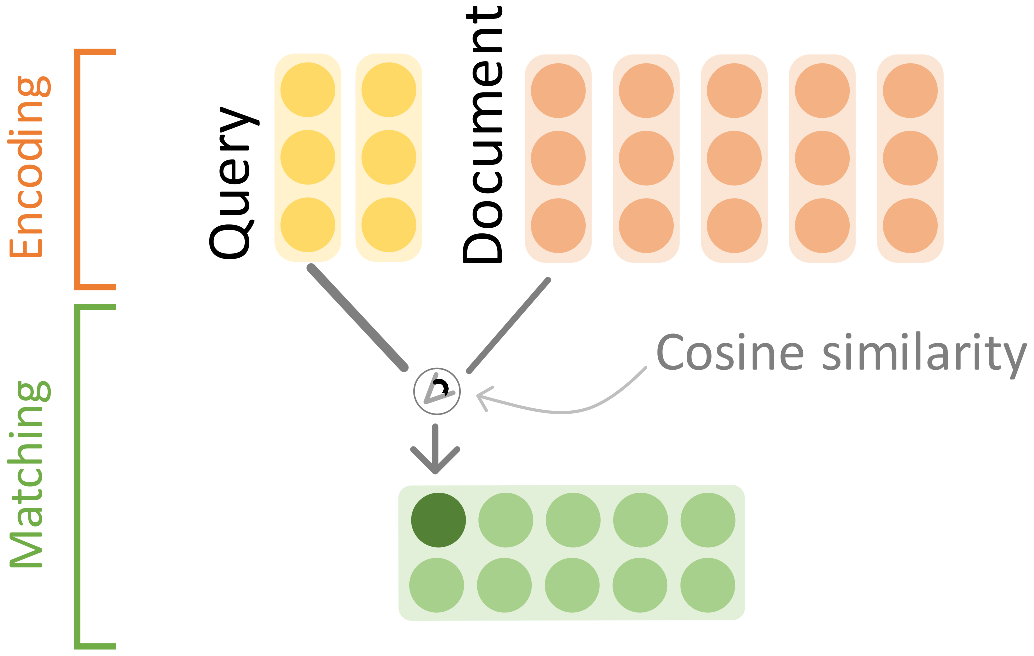
- Usually shared between query & document
  - If the same word appears in the query and the document
    – it gets assigned the same vector*

* Does not hold for contextualized representation, i.e. ELMo/BERT (but keep it simple for a minute)

The encoding layer

- Typically (<2019 🙄) a word embedding
  - Pre-trained Word2vec/Glove
  - Fine-tuned (= trained with the rest of the model)
- 2019: BERT (huge transformer based model, only pre-trained)
  - Shows very strong results
- A simple word embedding has still a strong benefits
  - #1 = Speed (after all it is just an memory address lookup on inference)
  - Somewhat interpretable / easily extracted + analyzed & reused in another task

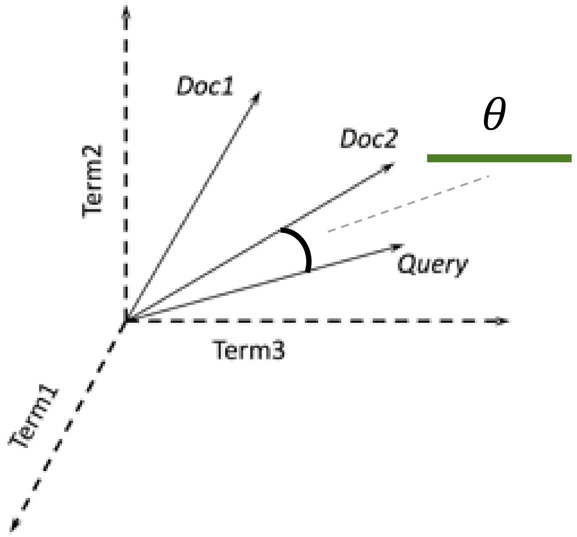
The match matrix



- The core of many early neural IR models*
- Matrix of similarities of individual word combinations
- Only a transformation – not parameterized by itself

* Of course there are other approaches, but we focus on models based on the match matrix

Cosine similarity



$$\mathrm{sim}(d,q) = \cos(\theta) = \frac{d \cdot q}{|d||q|}$$

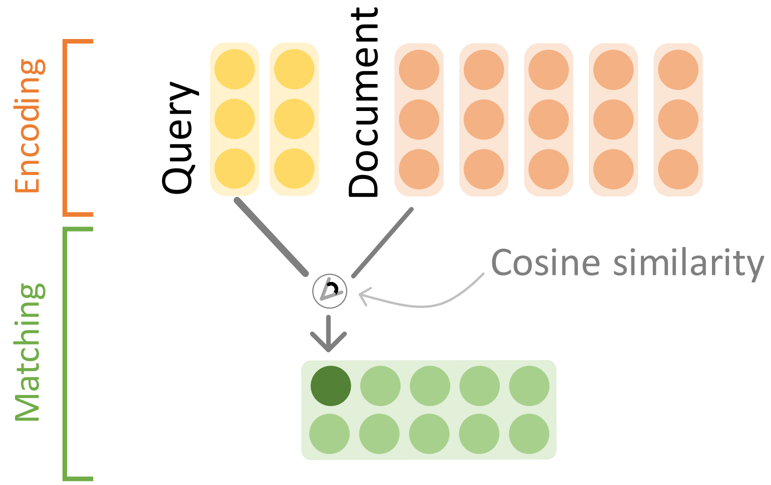
- Cosine similarity measures direction of vectors, but not the magnitude
- Not a distance – but equivalent to Euclidean distance of unit (length=1) vectors

θ	Angle between two vectors
q	Vector of query
d	Vector of document
$d \cdot q$	Dot product
	$= \sum_{i=1}^{dim} d_i * q_i$

Cosine similarity in PyTorch

- Using efficient batched-matrix multiplication

- Input shape:
  - Query: [batch, query_seq_len, emb_dim]
  - Document: [batch, doc_seq_len, emb_dim]
- Output shape: [batch, query_seq_len, doc_seq_len]



```
a_norm = matrix_1 / (matrix_1.norm(p=2, dim=-1, keepdim=True) + 1e-13)
b_norm = matrix_2 / (matrix_2.norm(p=2, dim=-1, keepdim=True) + 1e-13)
result = torch.bmm(a_norm, b_norm.transpose(-1, -2))
```

From: https://github.com/allenai/allennlp/blob/master/allennlp/modules/matrix_attention/cosine_matrix_attention.py

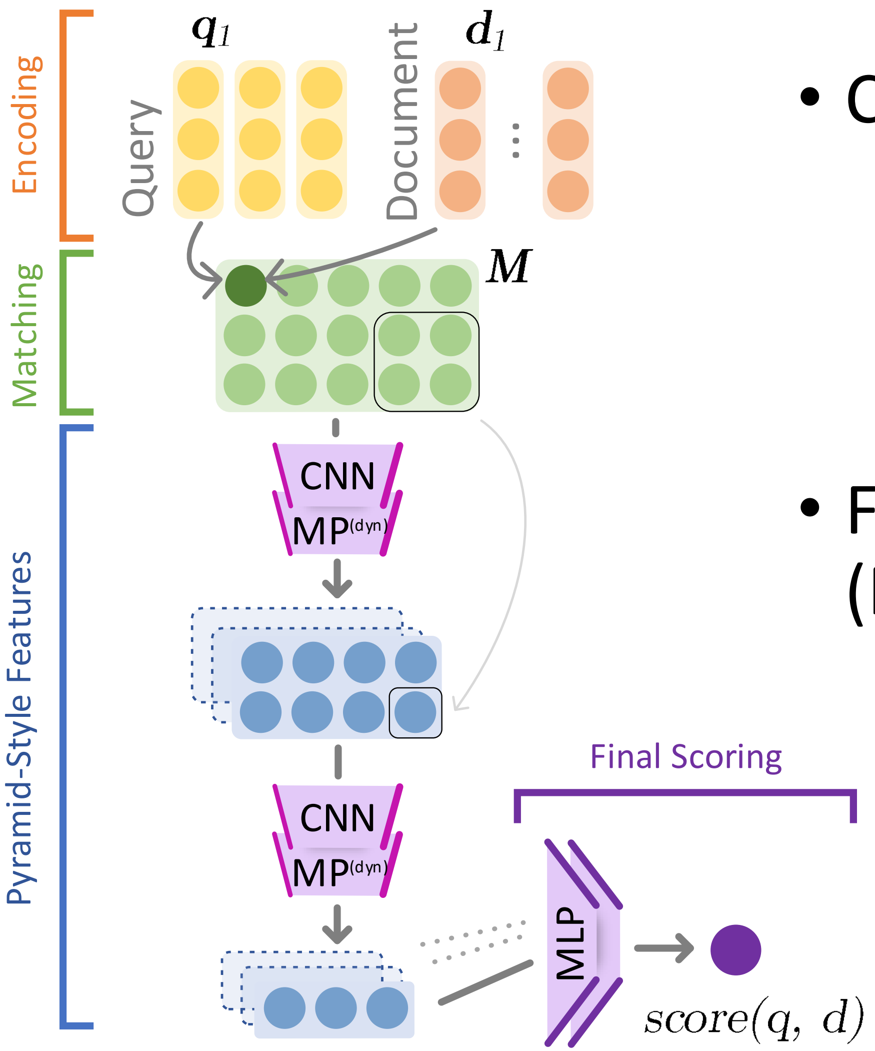
MatchPyramid

- Apply a set of 2D convolutional layers on top of the match matrix
    - Inspired by computer vision: match matrix = image
- Each Conv Layer: 2D-CNN & 2D-Dynamic-Pooling
    - Dynamic Pooling takes care of variable length input to fixed output

- Architecture & effectiveness strongly depends on configuration
    - How many layers
    - Which CNN & pooling kernel sizes
    - Generally: Pooling output becomes gradually smaller (like the Pyramid)

Liang Pang, Yanyan Lan, Jiafeng Guo, Jun Xu, Shengxian Wan, and Xueqi Cheng.
2016. Text Matching as Image Recognition. In Proc. of AAAI.

MatchPyramid



- Conv-layers extract local interaction features
  - Max pooling only keeps strong interaction signals (better matches)
  - Different channels can learn different interaction patterns
- Finally a multi-layered feed forward module (MLP) scores the extracted feature vectors

Simplified architecture illustration: Omitted zero-padding (at the start of the pyramid), more layers & channels in practice, last layer can be >1D (nD can be flattened to 1D)

MatchPyramid

Matching

$$M_{ij} = \mathbf{cos}(q_i, d_j\ ) = \frac{d_j \cdot q_i}{|d_j||q_i|}$$

Pyramid-Style Features

$$z_{ij}^{(1,c)} = \boldsymbol{2D\_Conv}(M_{ij})$$

$$= ReLU\left(\sum_{s=0}^{r_c-1}\sum_{t=0}^{r_c-1} w_{s,t}^{(1,c)} * M_{i+s,j+t} + b^{(1,c)}\right)$$

$$z_{ij}^{(2,c)} = \boldsymbol{dyn\_max\_pool}\left(z_{ij}^{(1,c)}\right) = \max_{0\le s<d_c}\ \max_{0\le t<d_c}\ z_{i*d_c+s,j*d_c+t}^{(1,c)}$$

…

$$z_{ij}^{(l,c)} = \boldsymbol{max\_pool}\left(\boldsymbol{2D\_Conv}(z_{ij}^{(l-1)})\right)$$

Scoring

$$s = \boldsymbol{MLP}(z^l) = W_2 * ReLU(W_1 * z^l + b_1) + b_2$$

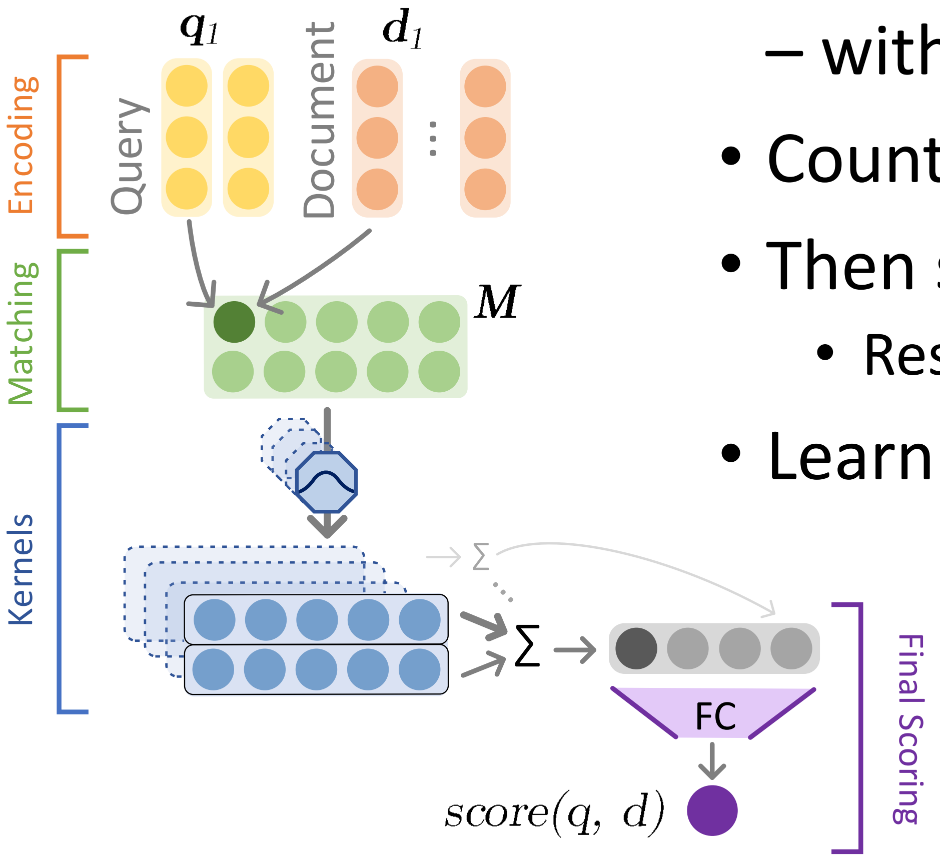
q_i	Vector of *i*-th query token
d_j	Vector of *j*-th document token
M	Match-matrix
$z^{(n,\_)}$	Sequential variable (n layers)
c	Channels
r_c	Size of channel c
d_c	Dynamic pooling kernel size
W_*, b_*	Weights & biases
s	Output score

KNRM

- KNRM: **K**ernel based **N**eural **R**anking **M**odel
- Counts the amount of different similarities between the query and document
- Very few learnable parameters (other than the embedding)
- Very fast – there is no complicated architecture increasing the runtime
- Roughly the same effectiveness as MatchPyramid

Chenyan Xiong, Zhuyun Dai, Jamie Callan, Zhiyuan Liu, and Russell Power. 2017.
End-to-End Neural Ad-hoc Ranking with Kernel Pooling. In Proc. of SIGIR.

KNRM



- Transform the similarity values of the match-matrix – with a “RBF-kernel” function
- Count the matches in different similarity bins
- Then sum up on doc and query dimensions
  - Result: 1 value per bin
- Learn bin value combination weights

Simplified architecture illustration: Σ represents 2x summation (1x on the query dim & 1x on the doc dim), more kernels in practice

KNRM

Matching

$$M_{ij} = \mathbf{cos}(q_i, d_j) = \frac{d_j \cdot q_i}{|d_j||q_i|}$$

Kernels

$$K_k\ (M_i) = \underline{\sum_j \underline{\exp(-\frac{(M_{ij} - \mu_k)^2}{2\sigma_k^2})}}$$

"RBF-Kernel"
(applied on a single match)

Sum alongside document dimension

$$K_k\ (M) = \underline{\sum_{i=1}^{n} \log(K_k(M_i))}$$

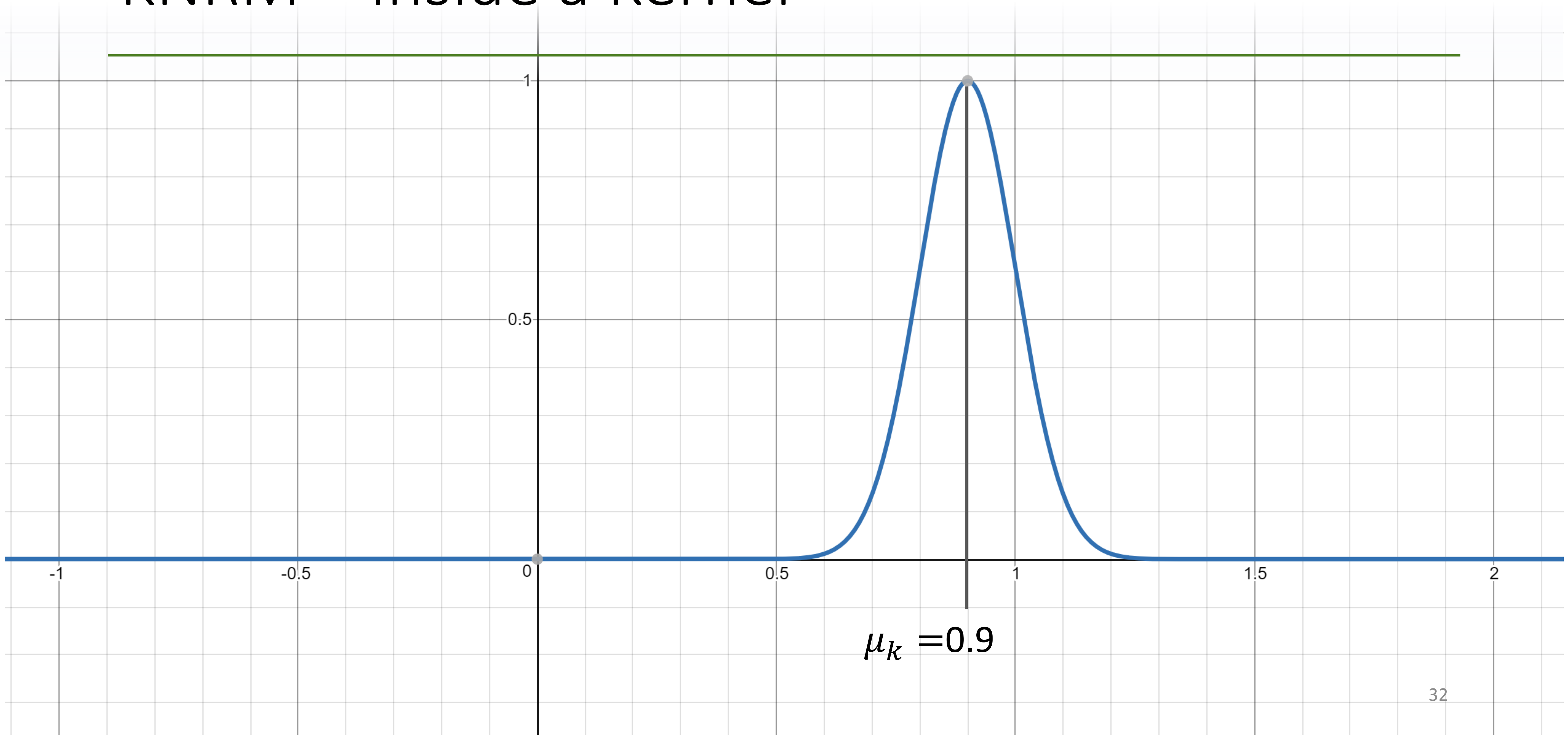
Sum alongside query dimension

Scoring

$$s = \boldsymbol{FC}(K) = W * K + b$$

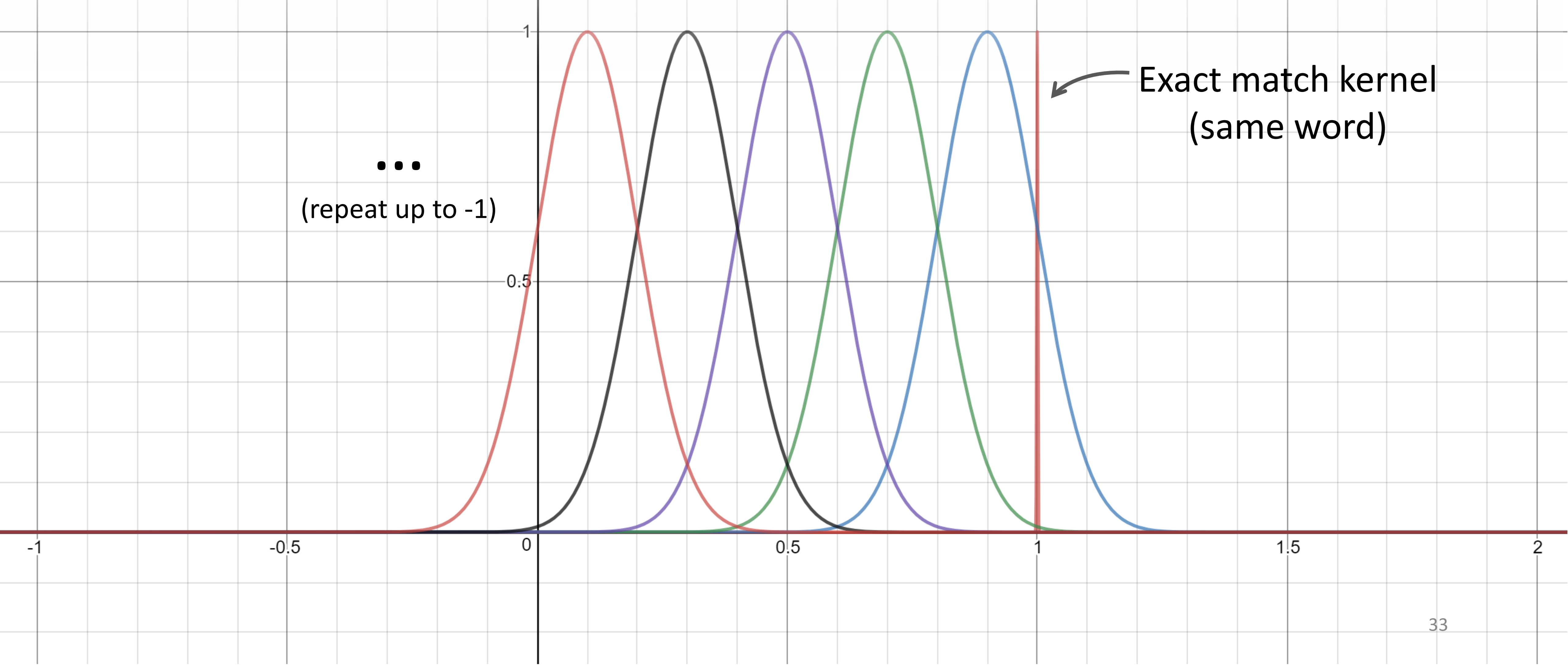
q_i	Vector of *i*-th query token
d_j	Vector of *j*-th document token
M	Match-matrix
K	All Kernels
K_k	*k*-th kernel
μ_k	Similarity level
σ_k	Kernel-width/range
W, b	Weights & biases
s	Output score

KNRM – Inside a Kernel



μ_k =0.9

KNRM – Inside all Kernels



Exact match kernel (same word)

...

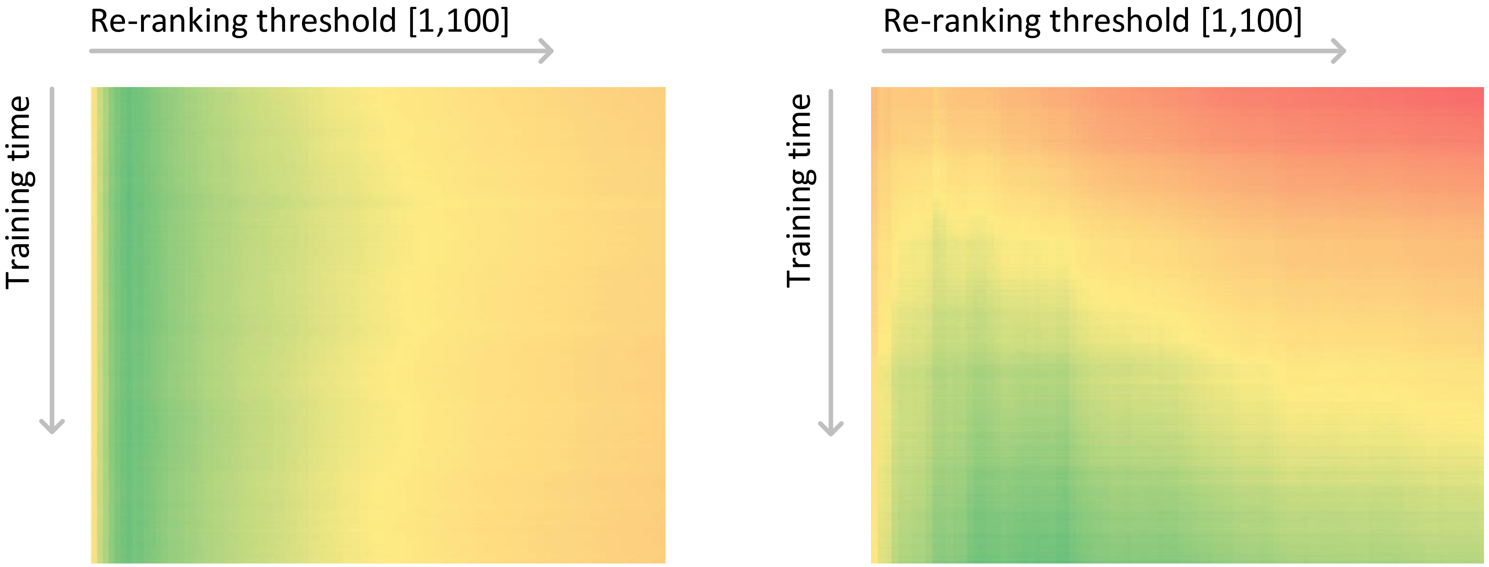
(repeat up to -1)

And now … A short tale about reproducibility

Illustration From: https://undraw.co/illustrations

KNRM – Implementation Details Matter

- What is different between these two output heatmaps?

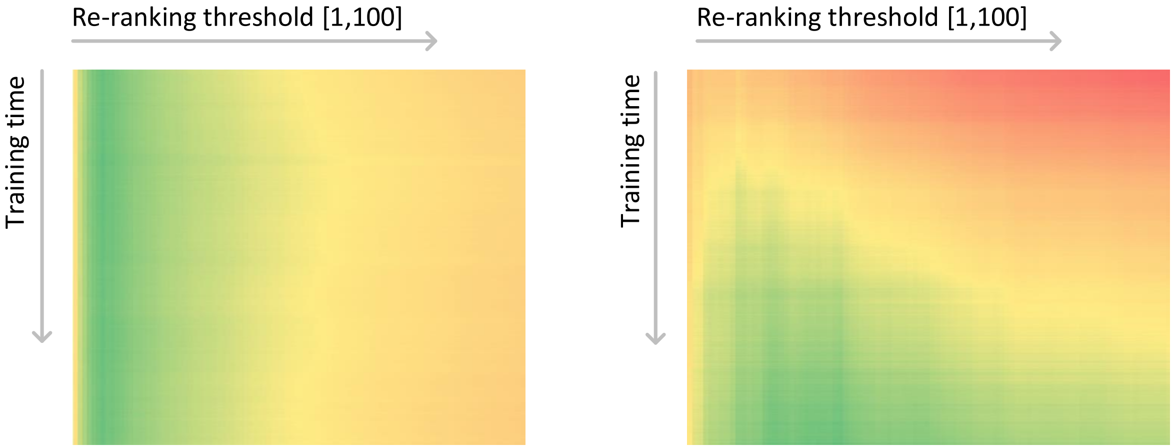


(a) Not working
Best MRR@10 = **0.186**
(same as BM25)

(b) Training properly
Best MRR@10 = **0.221**

** Note: the color scales are min (red, yellow) to max (green) individual per run*

KNRM – Implementation Details Matter



(a) Not working
Best MRR@10 = **0.186**
(same as BM25)

(b) Training properly
Best MRR@10 = **0.221**

** Note: the color scales are min (red, yellow) to max (green) individual per run*

- Difference: (a) = log(1 + soft_tf)
  (b) = log(soft_tf)

- There is open-source code for both

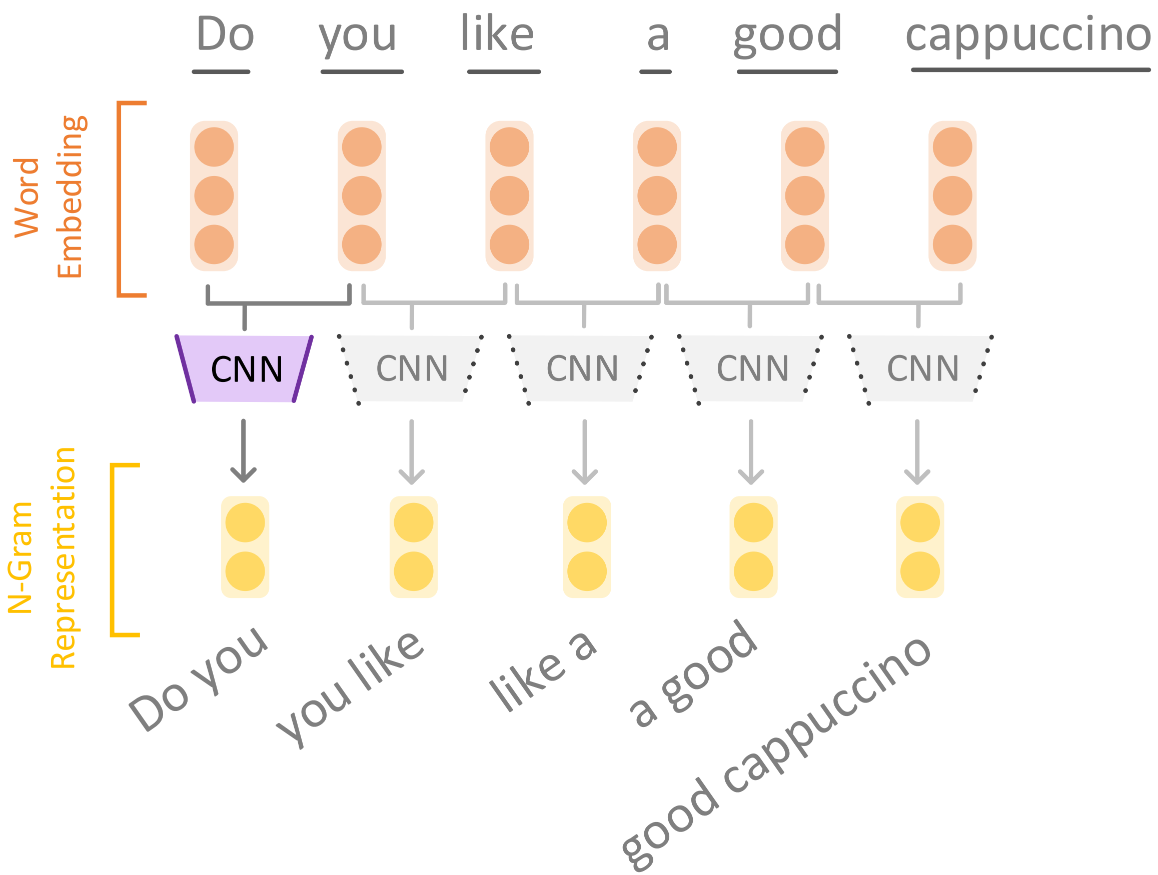
- At first it seems counterintuitive, our best educated guess:
  - log(soft_tf) acts as regularization, meaning it basically ignores single occurrences as noise

Conv-KNRM

- Extends KNRM before the match-matrix
- Crossmatches n-gram representations, and then applies KNRM
  - Allows to match: “convolutional neural networks” with “deep learning”
- N-grams (and term proximity) are very important in retrieval
  - Not feasible to create a vocabulary with all possible n-grams
- Most effective model highlighted today

Zhuyun Dai, Chenyan Xiong, Jamie Callan, and Zhiyuan Liu. 2018. Convolutional Neural Networks for Soft-Matching N-Grams in Ad-hoc Search. In Proc. of WSDM

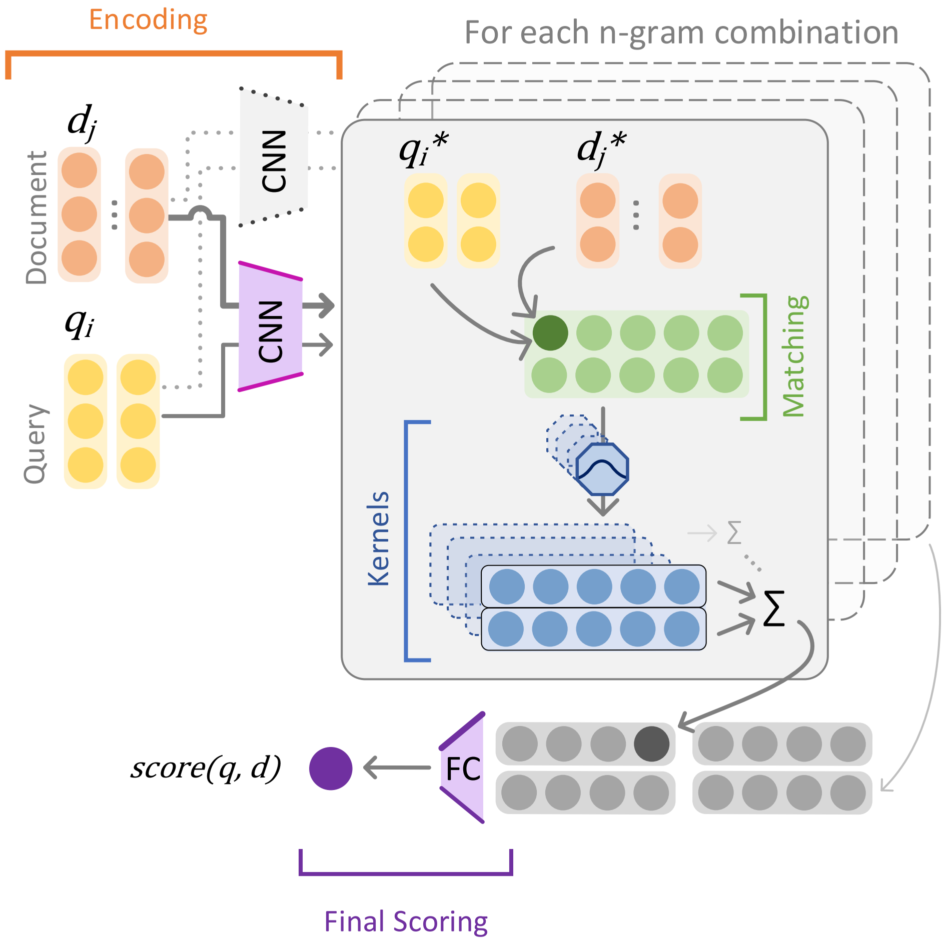
Recall: Word N-Grams with 1D CNNs



- Apply a 1D CNN on a sequence of word vectors
- N of N-grams = filter size
  - In this example N=2
- Output is a sequence of N-gram representations
  - Further used in other network components
- WE & CNN can be trained end-to-end

Kalchbrenner, N., Grefenstette, E. and Blunsom, P., 2014. A Convolutional Neural Network for Modelling Sentences. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics*

Conv-KNRM



- Creates n-gram representations for query and doc sequences
- Matches the n-gram vectors
  - 1 match matrix for every n x n: 1,1 – 1,2 – 1,3 – 2,2, …
- Then KNRM and concatenation of result vectors

Conv-KNRM

Encoding

$$q_{1..n}^{h} = 1D\_CNN^{h}(q_{1..n}) \qquad d_{1..m}^{h} = 1D\_CNN^{h}(d_{1..m})$$

Matching

$$M_{ij}^{h_q h_d} = \mathbf{cos}\left(q_i^{h_q}, d_j^{h_d}\right) \Big|$$

For each $h_q h_d$ combination

Kernels

$$K_k^{h_q h_d}(M) = \underline{\sum_{i=1}^{n} \log(\sum_{j} \exp(-\frac{\left(M_{ij}^{h_q h_d} - \mu_k\right)^2}{2\sigma_k^2}))}$$

Same as KNRM

Scoring

$$s = \boldsymbol{FC}(K) = W * K + b$$

$q_{1..n}$	Query token vecs
$d_{1..m}$	Doc. token vecs
M	Match-matrix
h, h_q, h_d	n-gram size
K	All Kernels
K_k	*k*-th kernel
μ_k	Similarity level
σ_k	Kernel-width/range
W, b	Weights & biases
s	Output score

Other models

- PACCR, Hui et al. 2017
  - Applies multiple 2D-Conv layers on top of the match matrix
  - N-gram matching after the single match matrix
- DUET, Mitra et al. 2017
  - Individual word matches + single vector per doc & query matching
- DRMM, Guo et al. 2016
  - Does not use a match matrix, rather histogram of similarities
  - Embedding is not updated

Bonus paper 💯

On the Effect of Low-Frequency Terms on Neural IR models

On the Effect of Low-Frequency Terms

- Infrequent terms carry a lot of relevance information in IR
- But: removed with a fixed vocabulary
  - Mainly as a concession to efficiency demands
  - Neural IR model doesn’t “see” removed terms

- Fixed vocabulary for all terms doesn’t scale
  - Missing or very little training data – useless representations
  - Unseen query terms are again OOV

Sebastian Hofstätter, Navid Rekabsaz, Carsten Eickhoff, and Allan Hanbury. 2019.
On the Effect of Low-Frequency Terms on Neural-IR Models. In SIGIR.

On the Effect of Low-Frequency Terms

❶ We show the importance of covering all terms of the collection

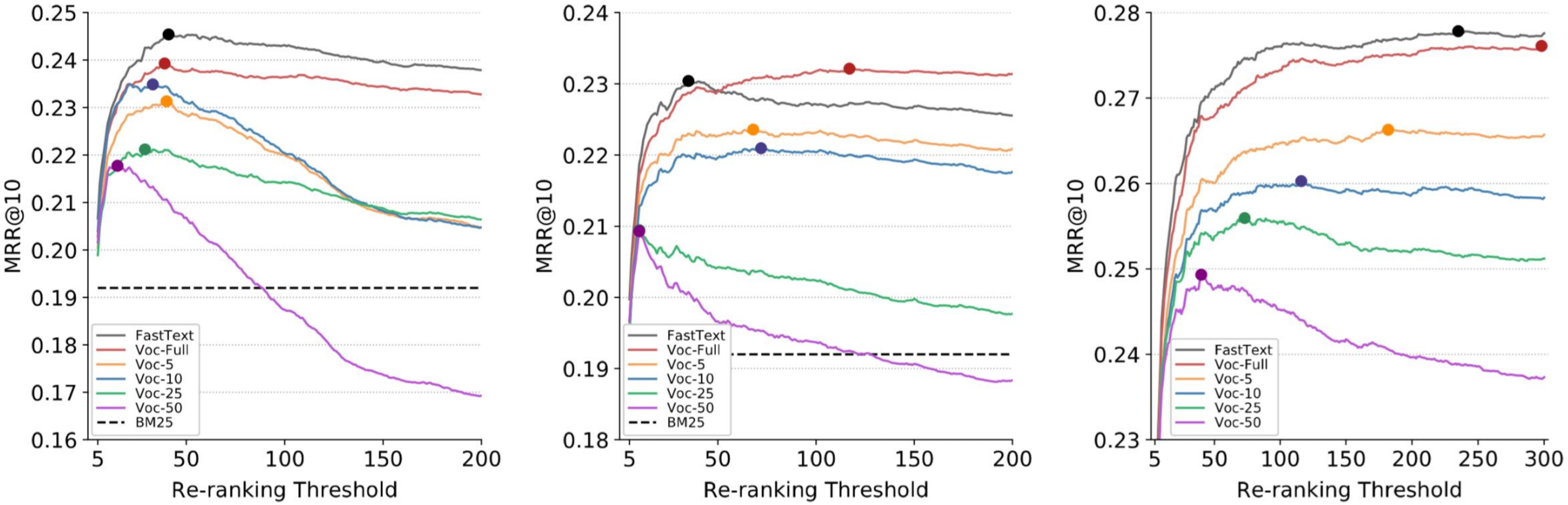
- First to analyze the re-ranking threshold – great tool for diagnostics
- If the model doesn’t perform well: more re-ranking docs decrease effectiveness

❷ Using the FastText strongly increases the effectiveness on queries with low-frequency terms

- FastText: Sub-word embedding
- Character n-gram composition for low frequency terms
- Keeps good inference performance

Sebastian Hofstätter, Navid Rekabsaz, Carsten Eickhoff, and Allan Hanbury. 2019.
On the Effect of Low-Frequency Terms on Neural-IR Models. In SIGIR.

Effect of the Fixed Vocabulary Size



(a) ***MatchPyramid*** (b) ***KNRM*** (c) ***CONV-KNRM***

Handling of Low-Frequency Terms

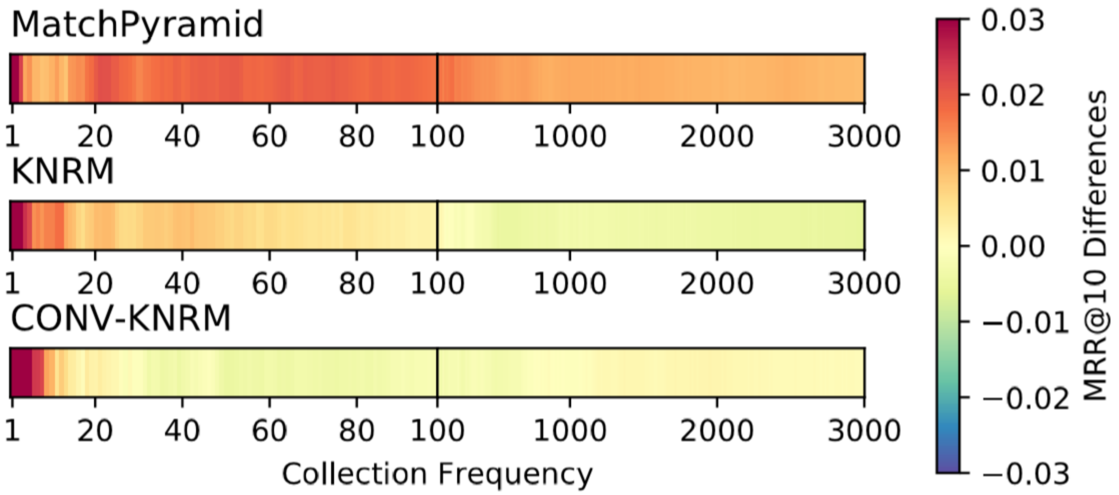


Figure 2: MRR differences of the models, trained on the FastText embeddings and the embeddings with full vocabularies, over the queries with minimum collection frequency of their terms smaller or equal to the X-axis
(red = FastText is better, blue=Vocab-Full is better)

- Focus on difference between full vocab <–> FastText
- FastText is better on queries with <10 occurrence terms
- Differences become less with higher frequency terms
  - Fundamental the same concept of encoding (not contextualized etc..)

Handling of Low-Frequency Terms

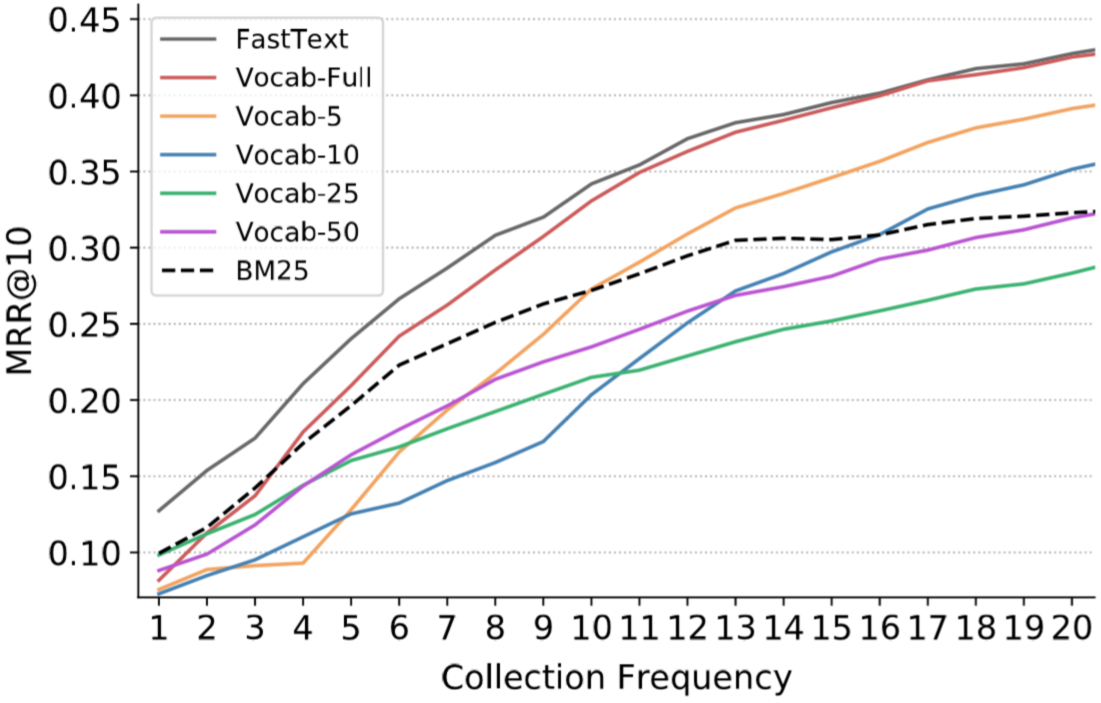


Figure 3: MRR results of *CONV-KNRM* over the queries with at least one term with collection frequency smaller than or equal to the values on the X-axis

- Focus on difference between all vocabs <–> FastText

- Keep in mind the huge scale of the y-axis
- Secondary insight: Neural IR models strongly bounded by BM25 first stage ranking
  - Future Work!

Summary: Neural Networks for IR

1. Pair (1 query, 1 doc) scoring – triples training – listwise evaluation

2. Word level match matrix: core building block of Neural IR models

3. Environment (vocabulary, re-ranking depth) matters a lot

1. Pair (1 query, 1 doc) scoring – triples training – listwise evaluation
2. Word level match matrix: core building block of Neural IR models
3. Environment (vocabulary, re-ranking depth) matters a lot

Thank You