CS60092: Introduction to Information Retrieval

Lecture 12: Language Models for IR

Prof. Somak Aditya

Content: Inspired by CS224n Stanford Course

What is a Language Model?

An LM is

- a probability distribution over sequence of words.
- a way to predict the next word

For a sentence S consisting of m words

$$S = w_1 \, w_2 \, w_3 \ \ldots \ldots . . w_m$$

In Language Model, we assume:

$$P(S) = P(w_1 \, w_2 \, w_3 \ \ldots \ldots . . w_m)$$

$$= P(w_1) \times P(w_2|w_1) \times \cdots \times P(w_m|w_{m-1} \ldots w_1)$$

But How it is helpful to us?

What is a Language Model?

Using LM, we can find out

- If a sentence S_1 is more likely than another S_2 (conditioned on q, but ignore for now).

For example:

- S_1: Virat Kohli plays cricket for India.
- S_2: plays Kohli cricket for India Virat.
- S_3: Virat Kohli plays plays for India.

Which is more likely?

Obviously S_1. Hence our LM should say **$P(S_1) > P(S_2)$** and **$P(S_1) > P(S_3)$**.

But, how can LM help us in IR?

Say q is "Kohli" D_1: Virat Kohli plays cricket for India. D_2: Virat Kohli plays plays for India. D_2: Sachin plays for India.

Using LM

- We can compute $P(D_i)$ and $P(q)$. With some assumptions

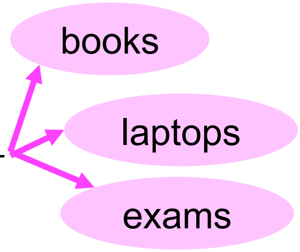
$$P(q|D_i) \propto P(D_i, q) = P(D_i)P(q)$$

- How to compute that?
  - LM helps us learn $v_{D_1}, v_{D_2}, v_{D_3}, v_q \in \mathbb{R}^d$.
  - We can approximate $P(D_i)P(q) \propto \frac{v_{D_i}^T v_q}{\left\|v_{D_i}\right\| \|v_q\|}$

n-gram Language Models

How to compute the Probability of the next word?

the students opened their ___



- ***Question***: *How to learn a language model?*
- ***Answer***: *Learn a n-gram language model.*

Definition: An n-gram is a chunk of n consecutive words.

- unigrams: “the”, “students”, “opened”, ”their”
- bigrams: “the students”, “students opened”, “opened their”
- trigrams: “the students opened”, “students opened their”
- four-grams: “the students opened their”

Idea: Collect statistics about how frequent different n-grams are and use these to predict next word.

n-gram Language Models

Markov Assumption: w_n depends on preceding $n-1$ words.

$$P(w_m|w_{m-1},\dots w_1) = P(w_m|\overbrace{w_{m-1}\dots w_{m-n+2}}^{\text{n-1 words}})$$

$$= \frac{P(w_m, w_{m-1}\dots w_{m-n+2})}{P(w_{m-1}\dots w_{m-n+2})}$$

Prob of n-gram
Prob of n-1 gram

Question: How do we get these n-gram and (n-1)-gram probabilities?

Answer: By counting them in some large corpus of text!

$$\approx \frac{count(w_m, w_{m-1}\dots w_{m-n+2})}{count(w_{m-1}\dots w_{m-n+2})}$$

n-gram LM Model in Practice

You can build a simple trigram Language Model over a 1.7 million word corpus (Reuters) in a few seconds on your laptop*

today the ________

get probability distribution

company	$\boxed{0.153}$
bank	$\boxed{0.153}$
price	0.077
italian	0.039
emirate	0.039
…	

Sparsity problem: not much granularity in the probability distribution

Otherwise, seems reasonable!

*** Try for yourself:** https://nlpforhackers.io/language-models/

Generating text with a n-gram Language Model

You can also use a LM to generate text

today the price of gold per ton , while production of shoe lasts and shoe industry , the bank intervened just after it considered and rejected an imf demand to rebuild depleted european stocks , sept 30 end primary 76 cts a share .

Surprisingly grammatical!

…but **incoherent.** We need to consider more than three words at a time if we want to model language well.

But increasing *n* worsens sparsity problem, and increases model size…

n-gram Language Models

Suppose we are learning a 4-gram Language Model.

~~*as the proctor started the clock, the*~~ *students opened their* ______

discard

condition on this

1. Markov Assumption: Probability of a word depends on previous "n" words. What is the value of this n?
    - If n is small, then it may predict a different word. Eg: Consider S: **In IPL, Virat Kohli plays cricket for ______.** For **n = 5**, then the predicted word may be "**India**" but
    - **n = 7**, then the predicted word may be "**RCB**".
    - If n is very large, computationally extensive.

2. A word may be dependent on next words as well.
    1. The word "United" has very high probability if next 3 words are "__ States of America".

n-gram Language Models

Problem: What if “students opened their w” never occurred in data? Then w has probability 0!

Partial Solution: Add small δ to the count for every $w \in V$. This is called **smoothing**.

$$P(w \mid students\ opened\ their) = \frac{count(students\ opened\ their\ w)}{count(students\ opened\ their)}$$

Problem: What if “students opened their” never occurred in data? Then we can’t calculate the probability of w.

Partial Solution: Just condition on “opened their”. Called backoff.

1. The **numerator** may be zero. We may need to do ***Smoothing***.
2. The **denominator** maybe zero for a given corpus. Say w3, w2 and w1 never cooccur in the corpus.To solve this, we could condition on w2 alone. This is called ***backoff***.

Neural network Language Models

NN-based Language Models solves (some of) these problems related to n-gram Language Models.

$$S = w_1 w_2 w_3 \dots\dots\dots . w_n$$

For the **kth word** w_k, we consider its ***Context*** or surrounding words (w_{-k})

We model the conditional probability:

$$P(w_k \mid \mathit{Context})$$

using a Neural network.

But how?

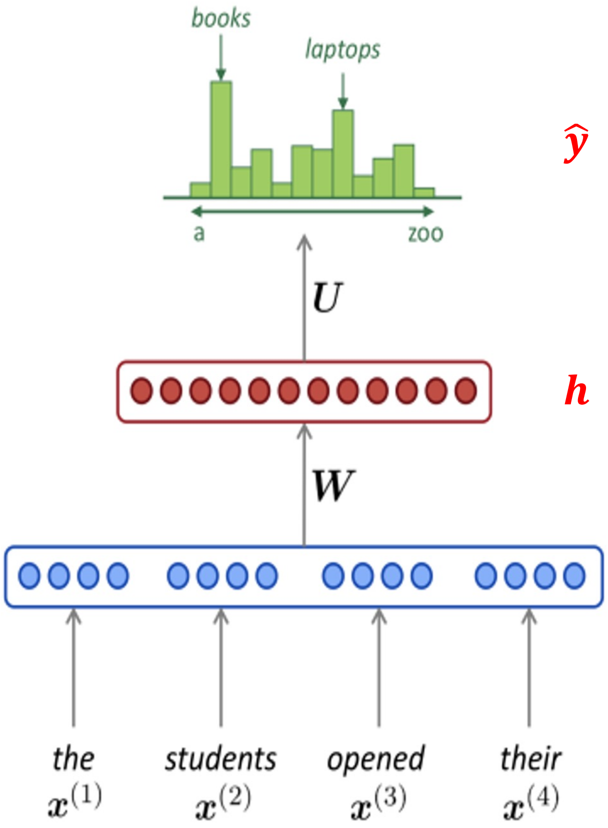
Neural network Language Models

Method 1 (Fixed-Window NN)

1. Word's probability depends on its context (but fixed window)
2. Each word has a fixed "continuous vector representation"
3. How to predict next word for the sentence "the students opened their __"?

   1. Assume you have a vector for each word. Look up vector for each word from a "lookup table"
   2. INPUT: Concatenate vectors $\boldsymbol{e} = [\boldsymbol{e}^{(1)}; \boldsymbol{e}^{(2)}; \boldsymbol{e}^{(3)}; \boldsymbol{e}^{(4)}]$
   3. HIDDEN: $\boldsymbol{h} = \boldsymbol{f}(\boldsymbol{W}\boldsymbol{e} + \boldsymbol{b}_1), \mathbf{W} \in \mathbb{R}^{4n \times d}$
   4. OUTPUT: $\hat{y} = \boldsymbol{softmax}(\boldsymbol{U}\boldsymbol{h} + \boldsymbol{b2}), \boldsymbol{U} \in \mathbb{R}^{d \times |V|}$

$\hat{y}$ is the distribution over words in the vocab.



Neural network Language Models

Method 1 (Fixed-Window NN)

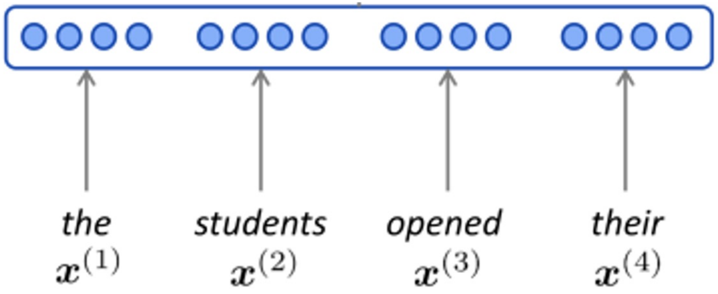
Step 1: Look up the vector representation for each word in the context from the "Look Up Table".

Example: Consider sentence "***the students opened their ____*** "

Index	Word	Continuous Word Representation
1	the	[0.6762, -0.9607, 0.3626, -0.2410, 0.6636]
200	students	[0.1656, -0.1530, 0.0310, -0.3321, -0.1342]
340	opened	[0.5965, 0.9143, 0.0899, 0.7702, -0.6392]
490	their	[-0.0069, 0.7995, 0.6433, 0.2898, 0.6359]

Neural network Language Models

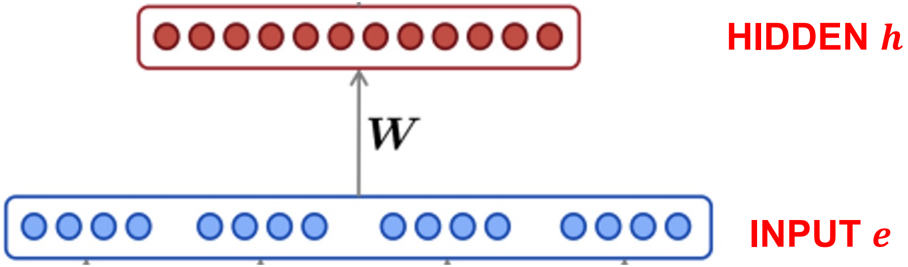
Concatenate the word vectors as shown :



$$\boldsymbol{e} = [\boldsymbol{e}^{(1)}; \boldsymbol{e}^{(2)}; \boldsymbol{e}^{(3)}; \boldsymbol{e}^{(4)}]$$

Concatenated vector **e** is the **INPUT LAYER** to our Neural Network.

Neural network Language Models



Step 2: Hidden layer output "h" is calculated as:

$$\boldsymbol{h} = \boldsymbol{f}(\boldsymbol{W}\boldsymbol{e} + \boldsymbol{b_1})$$

$\boldsymbol{W} = ?\ \boldsymbol{b_1} = ?$

W = Weight matrix connecting Input Layer and Hidden Layer

e = Input Layer concatenated vector (see last slide)

$b1$ = bias,

f = tanh or sigmoid

Neural network Language Models

Step 3: Hidden to Output Layer:

$$\boldsymbol{z} = \boldsymbol{Uh} + \boldsymbol{b_2}$$
$$\hat{y} = \boldsymbol{\sigma}(\boldsymbol{z})$$

$\boldsymbol{U} = ?\ \boldsymbol{b_2} = ?$

U = Weight matrix between Hidden Layer and Output Layer.
h = Output of Hidden Layer calculated in the last slide
b_2 = bias

Softmax function: $\hat{y}_i = \sigma(z)_i = \frac{e^{z_i}}{\Sigma_i e^{z_i}}, \ y =< \mathrm{y}_1, \mathrm{y}_2 \ldots, \mathrm{y}_{|V|} >$

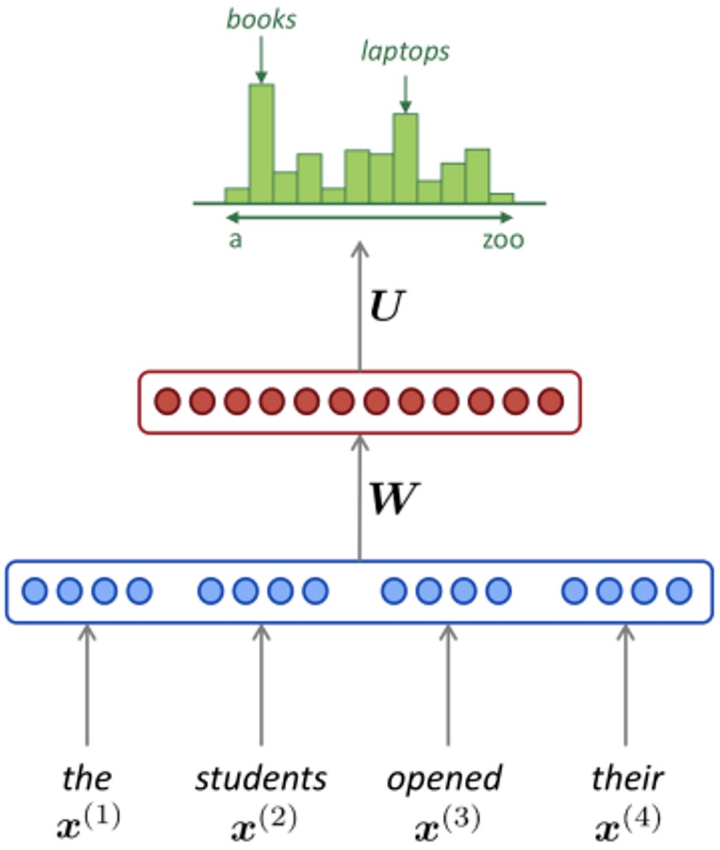
Neural network Language Models

In our example, the word "books" has the highest probability. The word "laptops" has 2nd highest probability.

- $\hat{y}_{books} > \hat{y}_{laptops}$

The final sentence becomes:

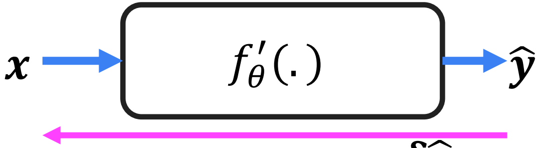
the students opened their books



Neural network Language Models

What did we learn? How do we infer?

- Given set of initial word vectors (lookup table), $\theta =< W, b, U, b2 >$, we can predict next word.
- Hence we can predict $P(S)$. **How?**



$$\boldsymbol{\theta}^{(1)} = \boldsymbol{\theta}^{(0)} - \boldsymbol{\eta}\frac{\boldsymbol{\delta\hat{y}}}{\boldsymbol{\delta\theta}}$$

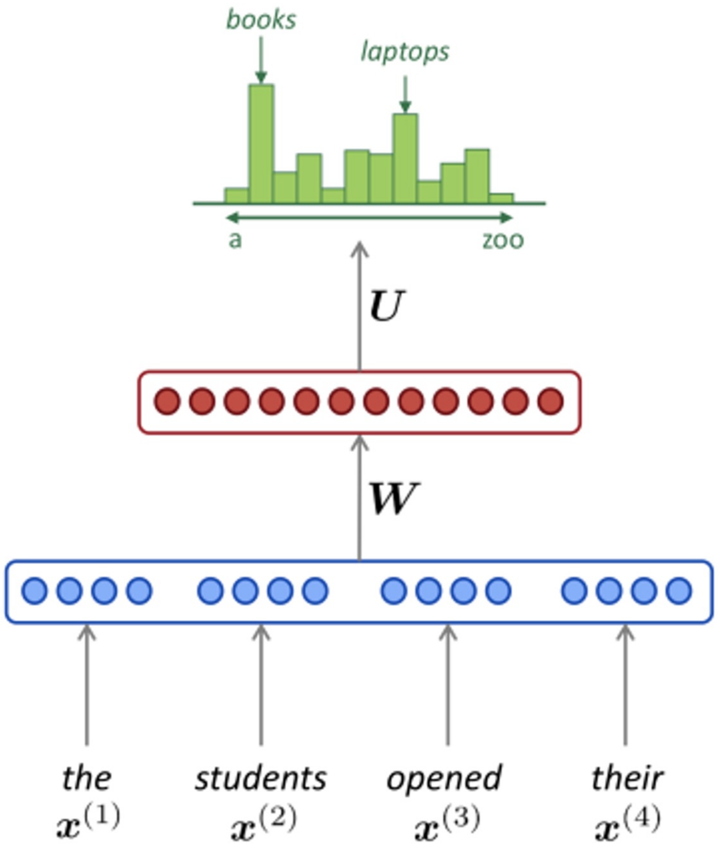
But, how do we train?

- How do we learn parameters $\theta =< W, b, U, b2 >$,?
- Using gradient Descent. What corpus? Labeled or unlabeled? Objective?
- To be covered during the lecture for word2vec.

Neural network Language Models

Points to note:

1. Word's probability depends on the fixed window context (previous or surrounding).
2. A word has a single vector in a table.
   - Even the ones such as "apple", "fall".
3. Estimation is only using a 3-layer NN.

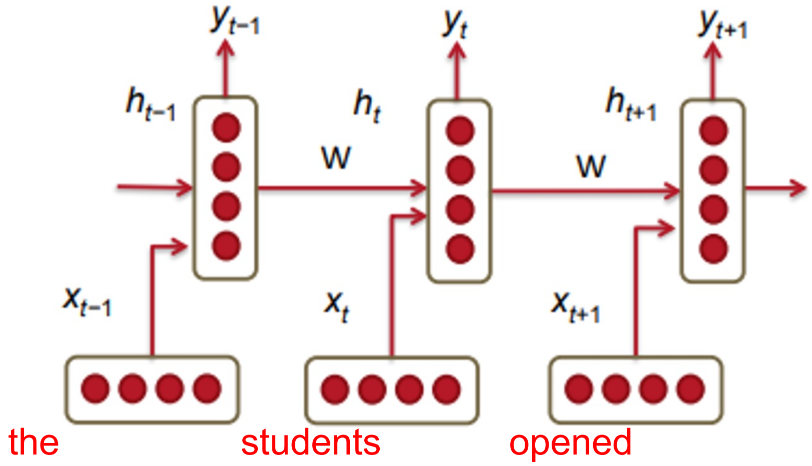


Recurrent Neural Networks (Method 2)

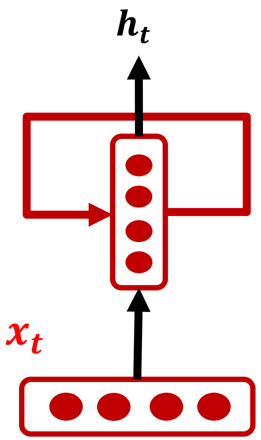
Recurrent Neural Networks (RNN)

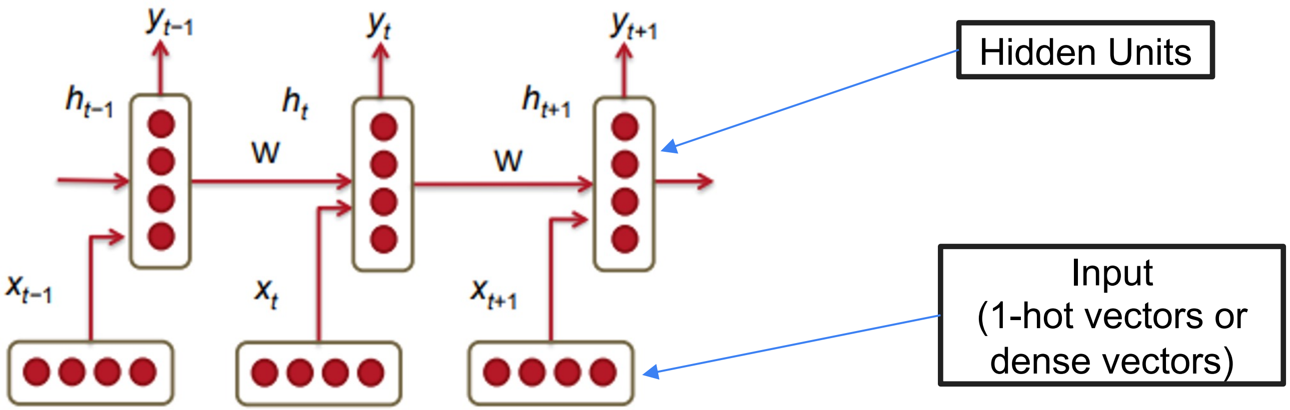
- Each word depends on all previous words in the "sentence/paragraph".
- ***RNNs add the immediate past to the present.***

Here, is a simple architecture of RNN:



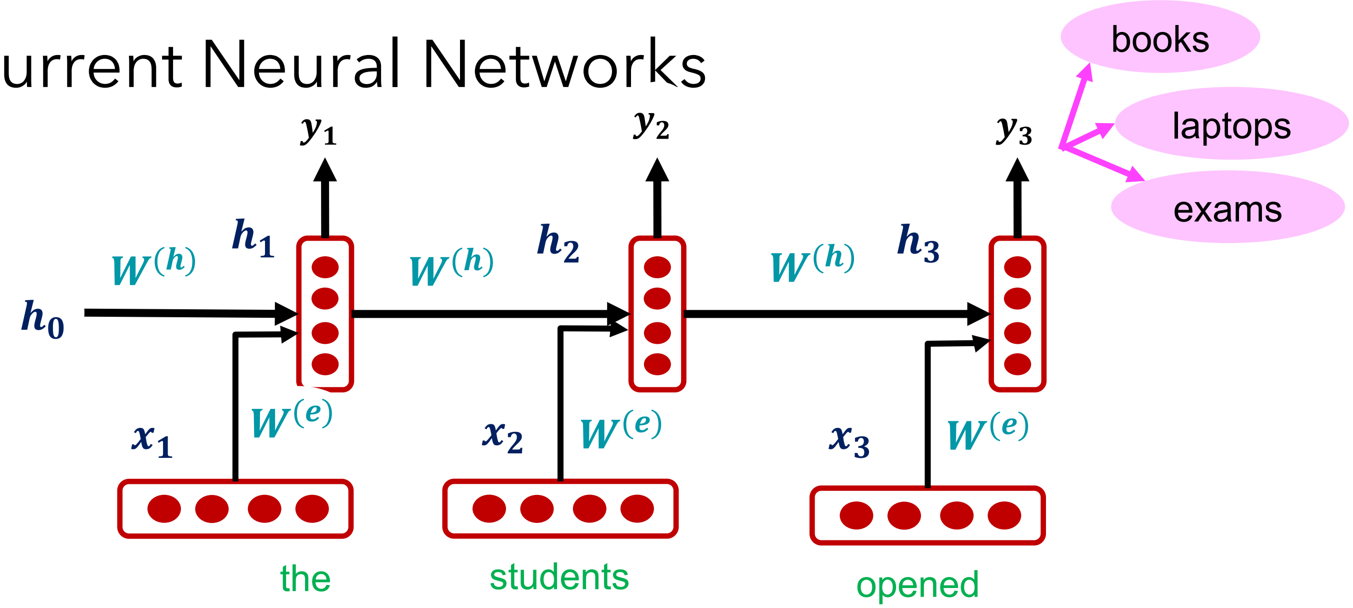
Recurrent Neural Networks





1. INPUT LAYER: $x =< x_1, x_2, \ldots x_n >$ is the input.
2. HIDDEN LAYER
   1. Vertical box is a hidden unit i.e. (h_t = hidden unit at timestep t). There is only one Hidden layer .
   2. The same computation is applied for t timesteps with t different words.
3. The Hidden unit at each step t has two inputs
   1. h_{t-1}: output of the previous timestep and
   2. the input at this timestep x_t.

Recurrent Neural Networks



HIDDEN LAYER COMPUTATION:

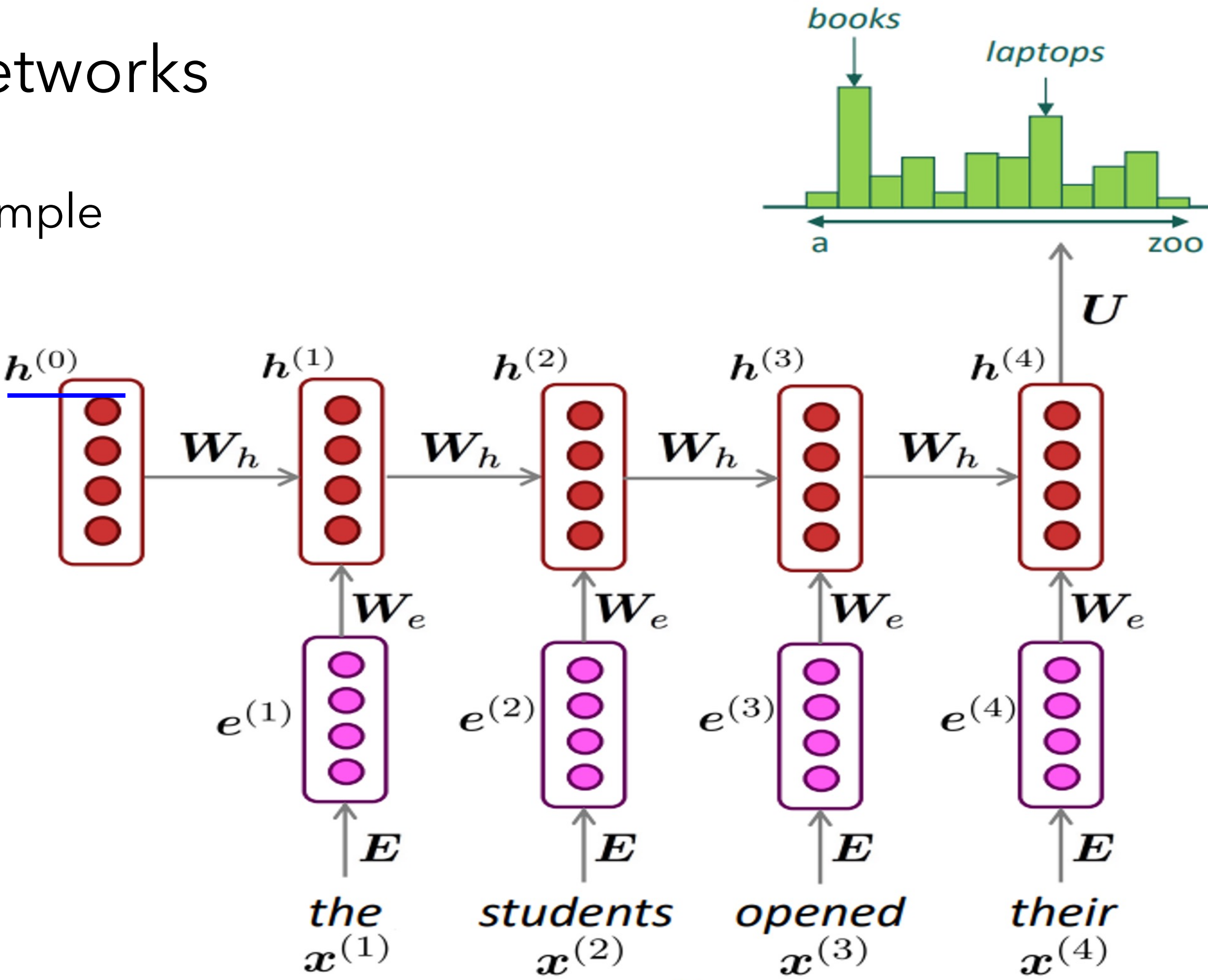
- $\boldsymbol{h_{t-1}}$ and $\boldsymbol{x_t}$ are "scaled" by separate weight matrices to produce $\boldsymbol{h_t}$
- $\boldsymbol{h_t}$ is multiplied with a weight matrix $\boldsymbol{W^{(S)}} \in \mathbb{R}^{d \times |V|}$
- Then a $\boldsymbol{softmax}$() over the vocabulary to get a prediction output $\boldsymbol{y_t}$ of the next word.

$$h_t = \sigma(W^{(h)} h_{t-1} + W^{(e)} x_t)$$
$$y_t = softmax(W^{(S)} h_t)$$

Recurrent Neural Networks

Working of RNN for the example sentence:

the students opened their ____



Recurrent Neural Networks

Advantages of RNNs

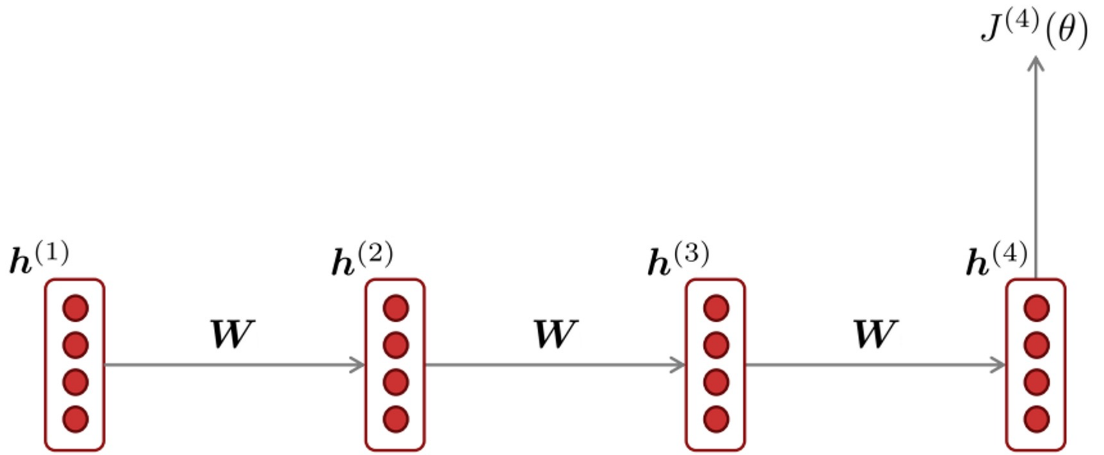
1. They can process input sequences of any length.
2. The model size does not increase for longer input sequence lengths.
3. Computation for step t can (in theory) use information from many steps back.

Disadvantages of RNNs

1. Computation is slow - because it is sequential, it cannot be parallelized.
2. In practice, it is difficult to access information from many steps back due to problems like ***vanishing gradients*** and ***exploding gradients***.

Recurrent Neural Networks

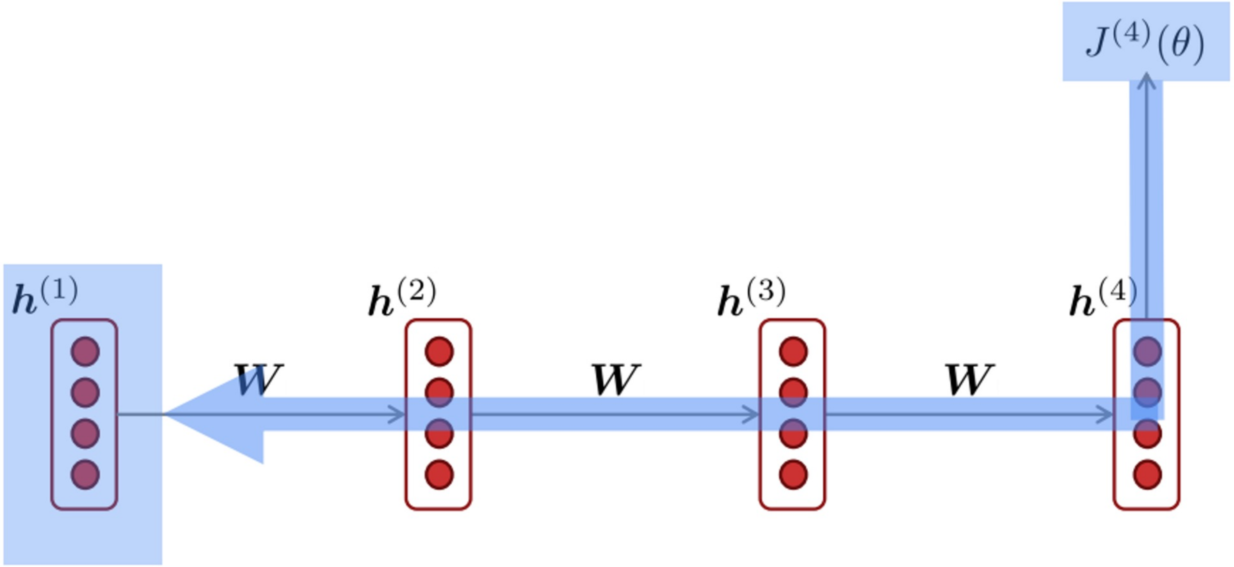
Vanishing and Exploding Gradients



Here, $J^{(4)}(\theta)$ is the final output. We need to calculate the derivative of it w.r.t $h^{(1)}$

Recurrent Neural Networks

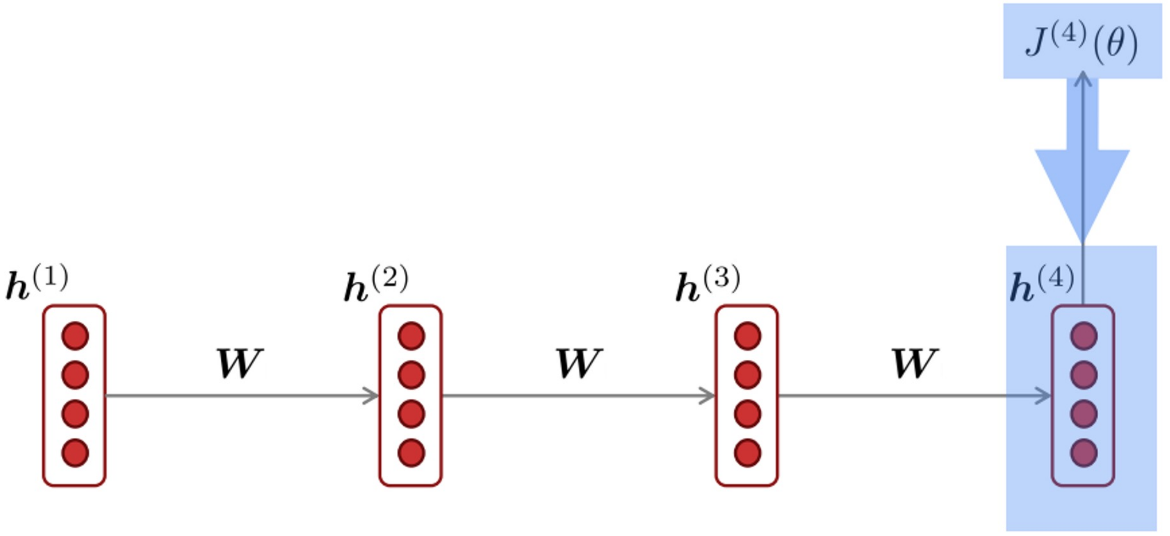
Vanishing Gradient



$$\frac{\partial J^{(4)}}{\partial \boldsymbol{h}^{(1)}} = \ ?$$

Recurrent Neural Networks

Vanishing Gradient

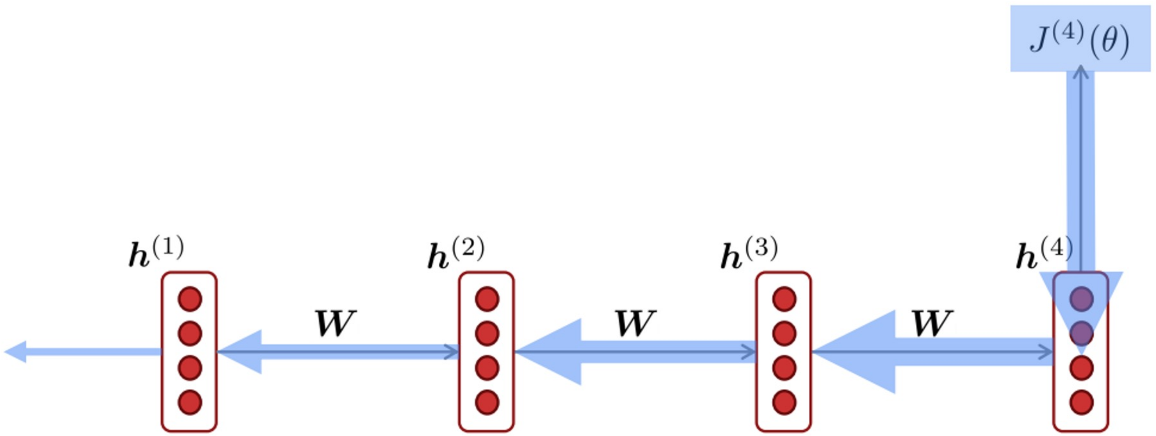


$$\frac{\partial J^{(4)}}{\partial \boldsymbol{h}^{(1)}} = \frac{\partial \boldsymbol{h}^{(2)}}{\partial \boldsymbol{h}^{(1)}} \times \frac{\partial \boldsymbol{h}^{(3)}}{\partial \boldsymbol{h}^{(2)}} \times \frac{\partial \boldsymbol{h}^{(4)}}{\partial \boldsymbol{h}^{(3)}} \times \frac{\partial J^{(4)}}{\partial \boldsymbol{h}^{(4)}}$$

chain rule!

Recurrent Neural Networks

Vanishing Gradient



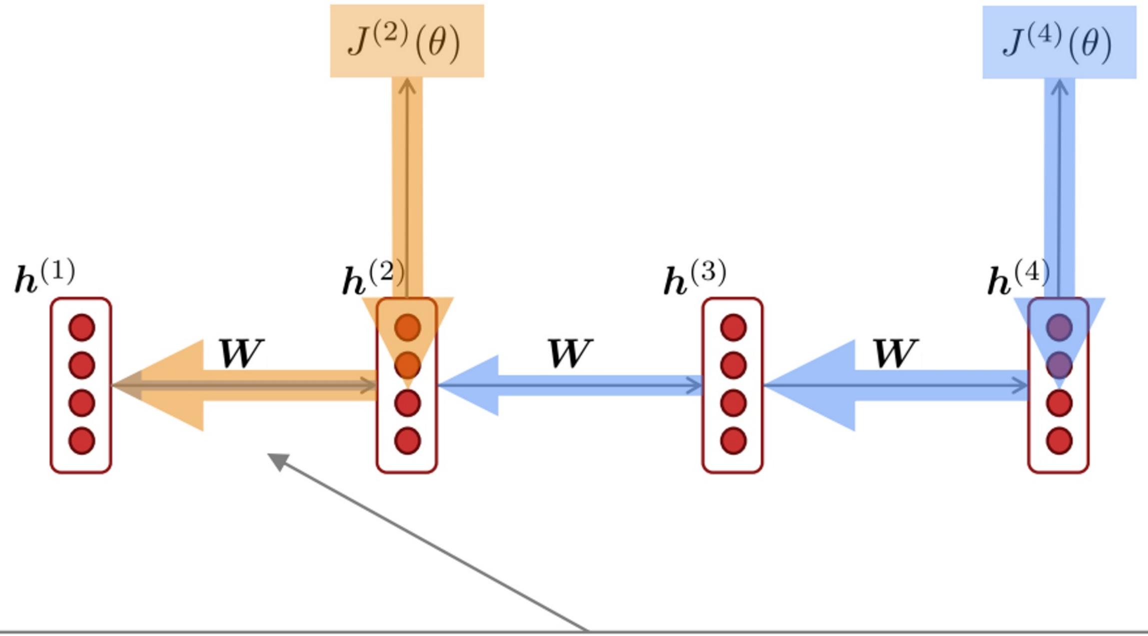
$$\frac{\partial J^{(4)}}{\partial \boldsymbol{h}^{(1)}} = \boxed{\frac{\partial \boldsymbol{h}^{(2)}}{\partial \boldsymbol{h}^{(1)}}} \times \boxed{\frac{\partial \boldsymbol{h}^{(3)}}{\partial \boldsymbol{h}^{(2)}}} \times \boxed{\frac{\partial \boldsymbol{h}^{(4)}}{\partial \boldsymbol{h}^{(3)}}} \times \frac{\partial J^{(4)}}{\partial \boldsymbol{h}^{(4)}}$$

What happens if these are small?

Vanishing gradient problem:
When these are small, the gradient signal gets smaller and smaller as it backpropagates further

Recurrent Neural Networks

Vanishing Gradient



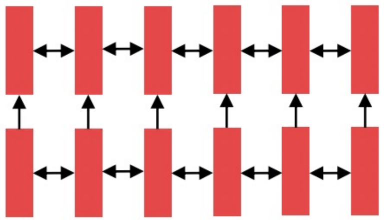
Gradient signal from far away is lost because it's much smaller than gradient signal from close-by.

So, model weights are updated only with respect to near effects, not long-term effects.

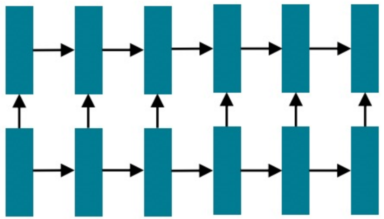
Transformers-based Language Models

RNN – De-facto Standard Till 2017

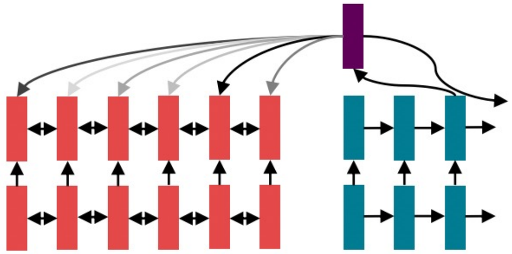
- Circa 2016, de facto in NLP was to encode sentences with a bidirectional LSTM
  - For example, the source sentence in a translation



- Define your output (parse, sentence, summary) as a sequence, and use an LSTM to generate it.



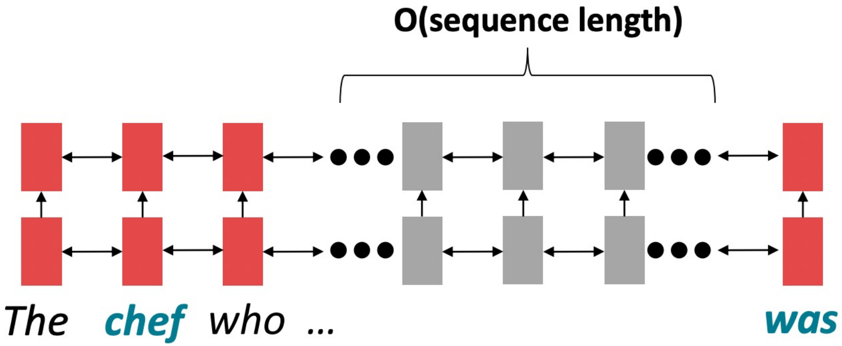
- Use **attention** to allow flexible access to memory



RNN – Linear Interaction Distance/Non-parallelizable

- RNNs are unrolled “left-to-right”.
- Useful: Nearby words often affect each other’s meanings

- Problem: RNNs take O(sequence length) steps for distant word pairs to interact
- Problem: Linear Order is “baked in”. Not sure that is best.
  - Right-to-left
  - Left-to-right
  - Bi-directional RNNs.

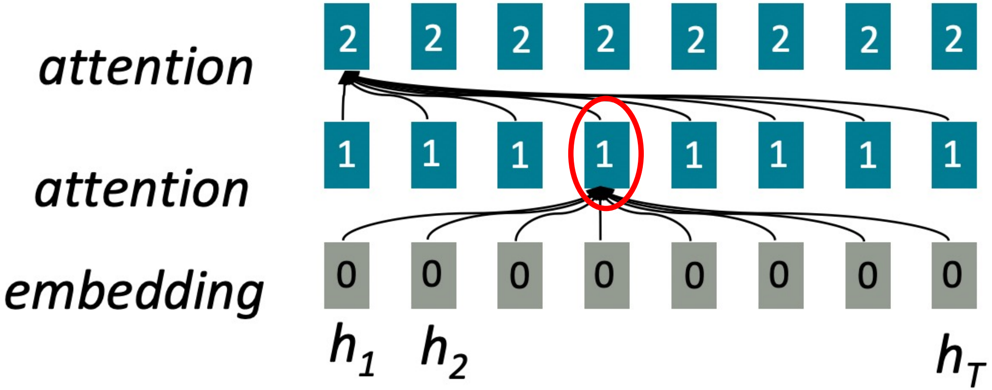


Recurrence to Attention

- Attention treats each word's representation as a query to access and incorporate information from a set of values.
  - For example, Layer 2 each node j computes

$$\sum_{i=1}^{T} \alpha_i w_{ij} h_i, \text{ s.t. } \Sigma_i \alpha_i = 1$$

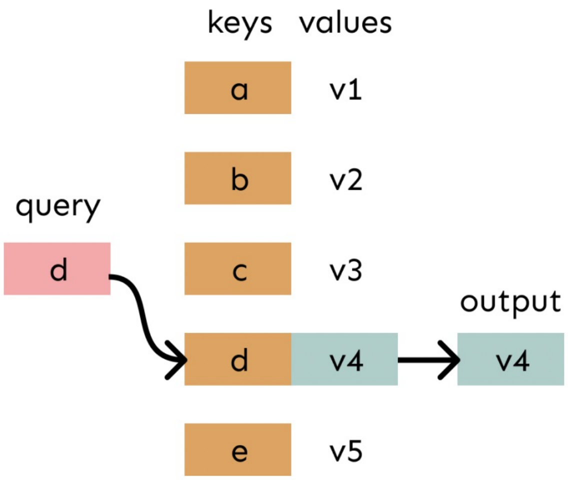
- Max. interaction distance: O(1).



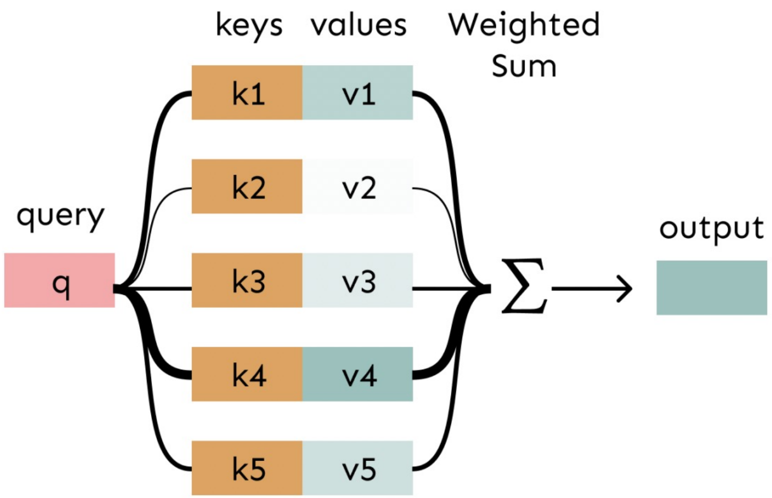
Attention as a soft, averaging lookup table

We can think of attention as performing fuzzy lookup in a key-value store.

In a **lookup table**, we have a table of **keys** that map to **values**. The **query** matches one of the keys, returning its value.



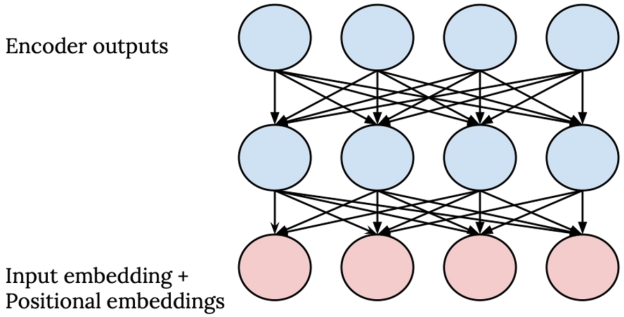
In **attention**, the **query** matches all **keys** *softly*, to a weight between 0 and 1. The keys' **values** are multiplied by the weights and summed.



Transformers - Motivation

How can we speed up the encoding process of sequences?

- Remove the recurrent connection (from RNNs)
- Only use attention

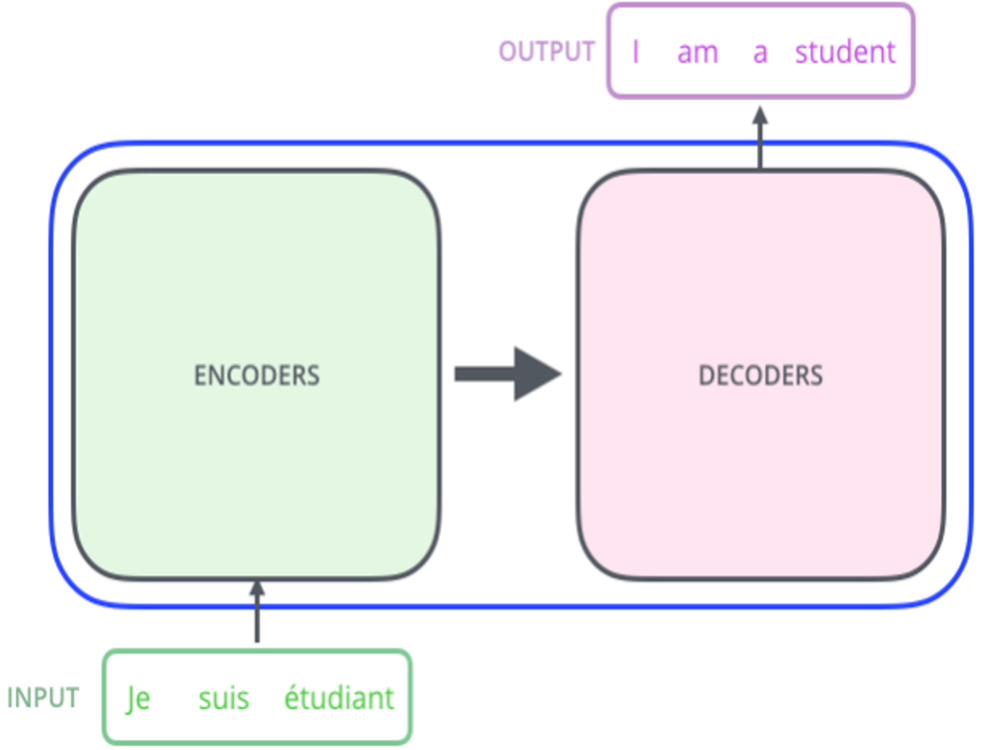


- But No order?
- No nonlinearities. Just weighted average

Transformers – Encoders and Decoders

Types of Transformers

- Encoder-decoder (Machine Translation, most generic)
  - T5, BART
- Decoder only (Most popular, Language Modeling)
  - OpenAI GPT, GPT-3
- Encoder only (mainly for classification)
  - BERT, RoBERTa, ALBERT, ViT, Swin, CLIP



Self-Attention:keys, queries, values from the same sequence

Let $w_{1::n}$ be the words in a vocab V. Like *Zuko made his uncle Tea*.

For a w_i, let $x_i = Ew_i$, where $E \in \mathbb{R}^{d\times|V|}$ is embedding matrix.

1. Transform x_i (word-emb) with weight matrices $Q, K, V \in \mathbb{R}^{d\times d}$

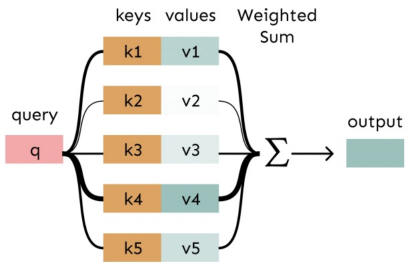
   $q_i = Qx_i$ (queries). $k_i = Kx_i$ (keys). $v_i = Vx_i$ (values).

2. Compute key-query similarities, and normalize

$$e_{ij} = q_i^T k_j \qquad \alpha_{ij} = \frac{\exp(e_{ij})}{\sum_{j'} \exp(e_{ij'})}$$

3. Compute output for each word as weighted sum of values

$$\boldsymbol{o}_i = \sum_j \boldsymbol{\alpha}_{ij}\, v_i$$



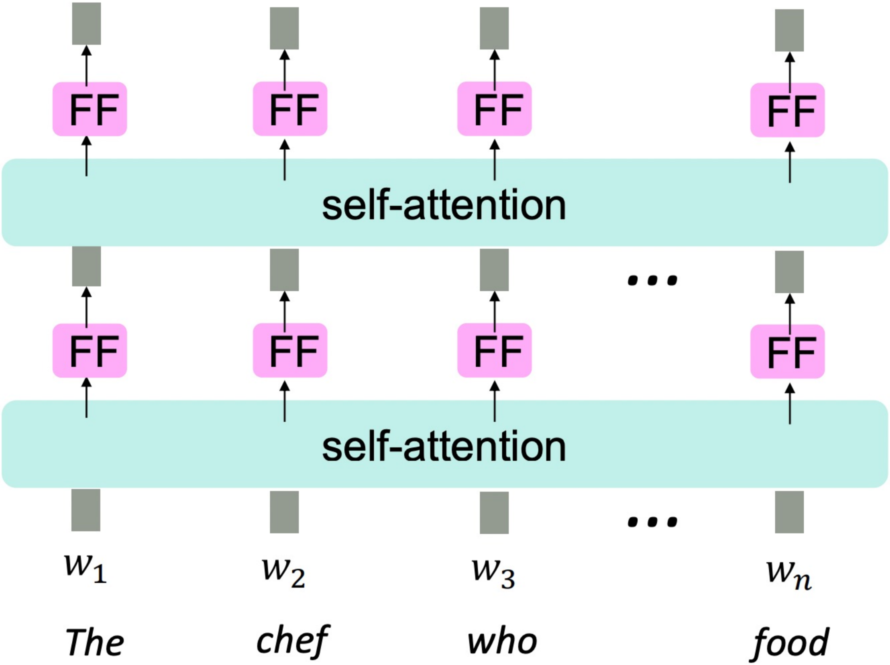
Transformers - Motivation

How can we speed up the encoding process of sequences?

- Only use attention

- But No order → Encode positions as embeddings
- No nonlinearities. Just weighted average
  - Add Feed-forward network with non-linearity process each vector.

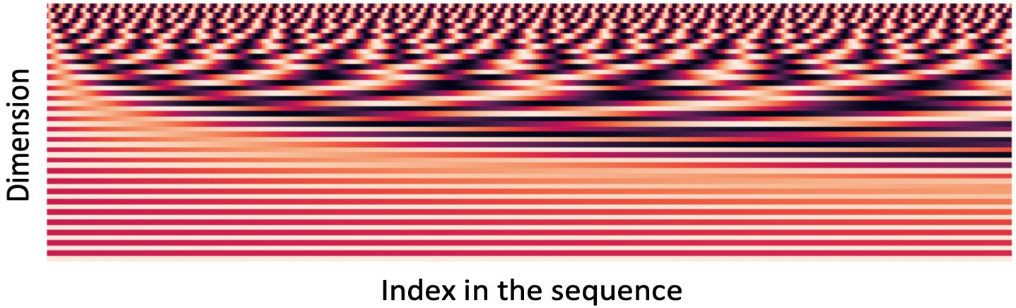
$$m_i = MLP(output_i)$$
$$= W_2 * ReLU(W_1 output_i + b_1) + b_2$$



Transformers Position Encoding

- Encode as $\tilde{x}_i = x_i + p_i$, where $p_i \in \mathbb{R}^d$
- Sinusoidal position representations: concatenate sinusoidal functions of varying periods:

$$\boldsymbol{p}_i = \begin{pmatrix} \sin(i/10000^{2*1/d}) \\ \cos(i/10000^{2*1/d}) \\ \vdots \\ \sin(i/10000^{2*\frac{d}{2}/d}) \\ \cos(i/10000^{2*\frac{d}{2}/d}) \end{pmatrix}$$



- Pros:
  - Periodicity indicates that maybe "absolute position" isn't as important
  - Maybe can extrapolate to longer sequences as periods restart!
- Cons: Not learnable; also the extrapolation doesn't really work!

Barriers and solutions for Self-Attention as a building block

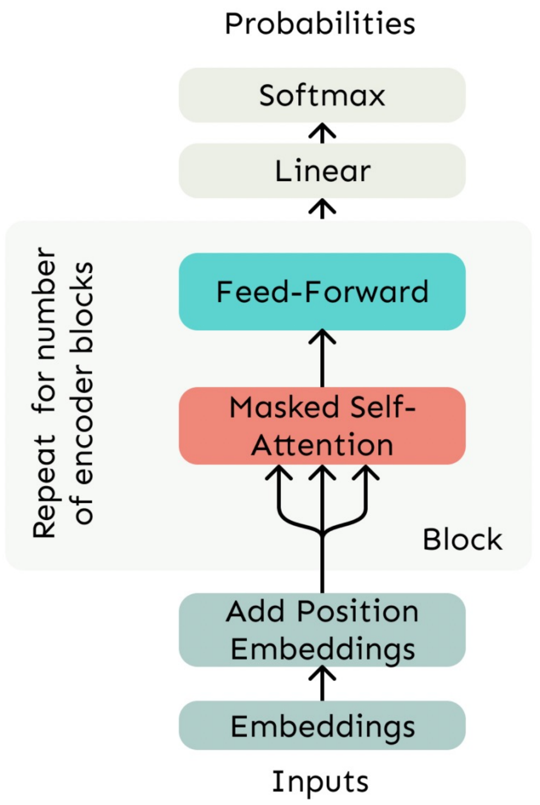
- Doesn't have an inherent notion of order!
- Add position representations to the inputs

- No nonlinearities for deep learning magic! It's all just weighted averages
- Easy fix: apply the same feedforward network to each self-attention output.

- Need to ensure we don't "look at the future" when predicting a sequence
  - Like in machine translation Or language modeling
- Mask out the future by artificially setting attention weights to 0!

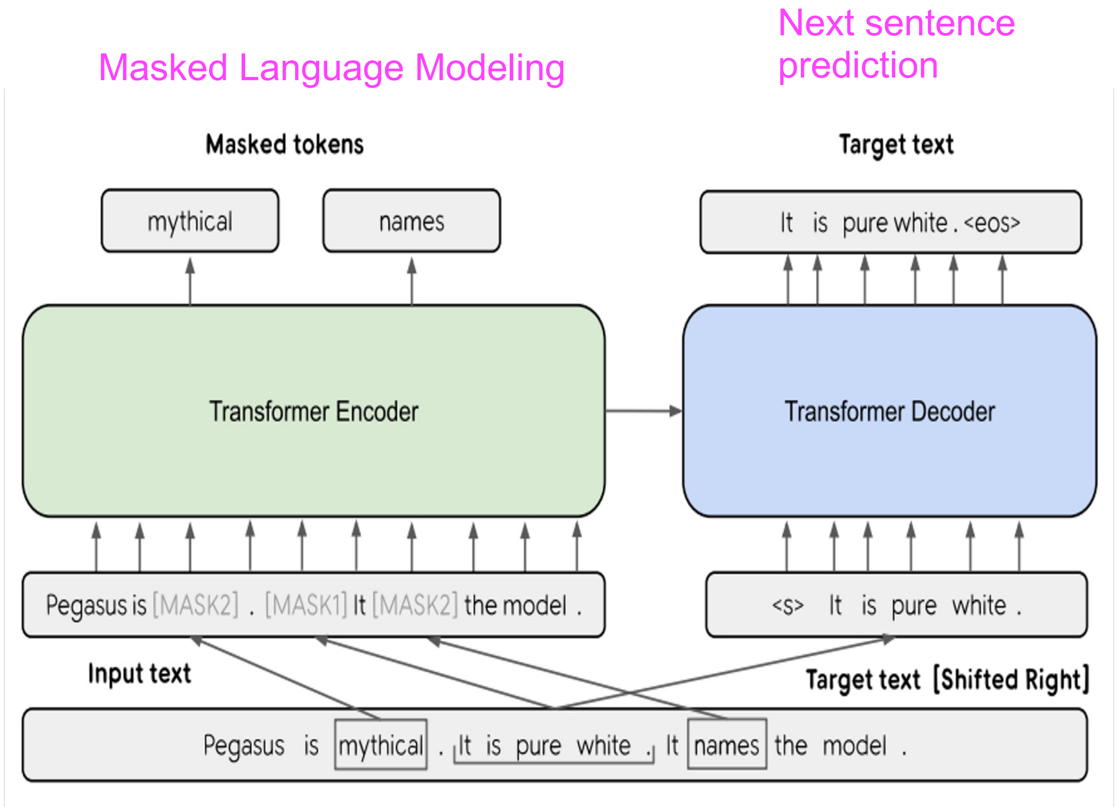
Necessities for a self-attention building block

- Self-attention
- Position representations:
  - Specify the sequence order, since self-attention is an unordered function of its inputs.
- Nonlinearities:
  - At the output of the self-attention block
  - Frequently implemented as a simple feedforward network.
- Masking
  - In order to parallelize operations while not looking at the future.
  - Keeps information about the future from "leaking" to the past.



Encoder-Decoder Models

- Encoder-Decoder
  - BART, T5, Pegasus
- Trained using Unsupervised Objectives
  - Mask some tokens and predict
  - Predict the next sentence.

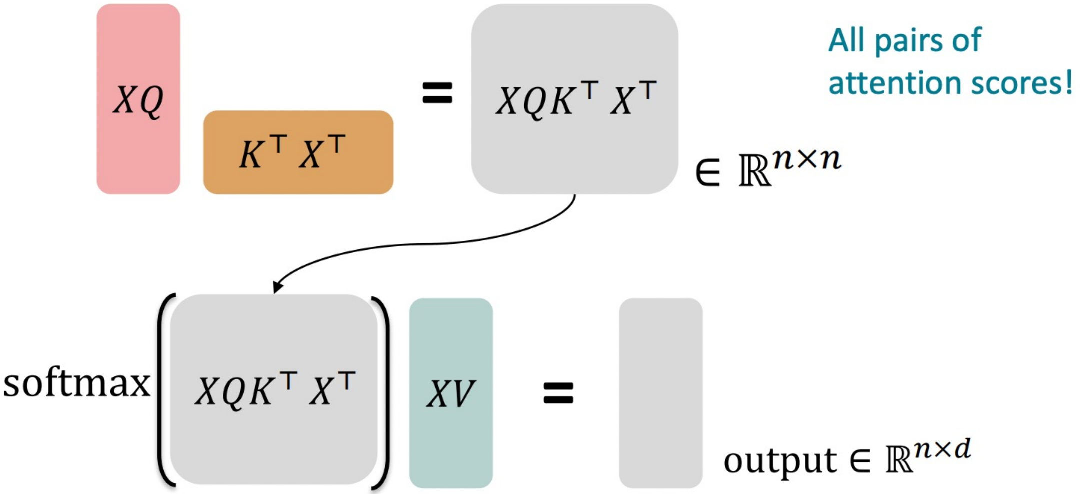


Multi-Head Attention (Sequence Stacked)

Key-query-value attention in matrix format

- $X = [x_1; \dots; x_n] \in \mathbb{R}^{n \times d}$
- Note $XK \in \mathbb{R}^{n \times d}$, $XQ \in \mathbb{R}^{n \times d}$, $XV \in \mathbb{R}^{n \times d}$

$$output = softmax(XQ(KX)^T) * XV$$



Transformer - Decoder

<u>Trick 1:</u> Multiple Attention heads.

- A single attention "head" learns to concentrate on a single property.
- One for logically related, another for subject-objects
- We need multiple heads.
- $Q_l, K_l, V_l \in \mathbb{R}^{d \times \frac{d}{h}}$, h is #attention-heads, l ranges from 1 to h.
- Each head

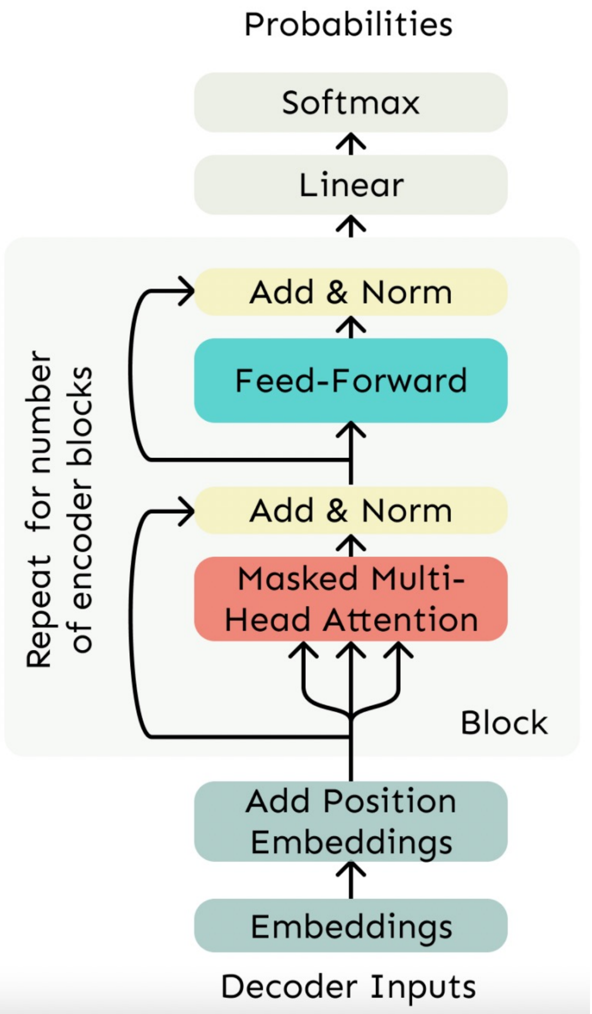
$$\mathrm{output_l} = \ softmax(XQ_lK_l^TX^T) * XV_l,$$

- Final output: $output = [output_1; \ldots, output_h]$,

Transformer - Decoder

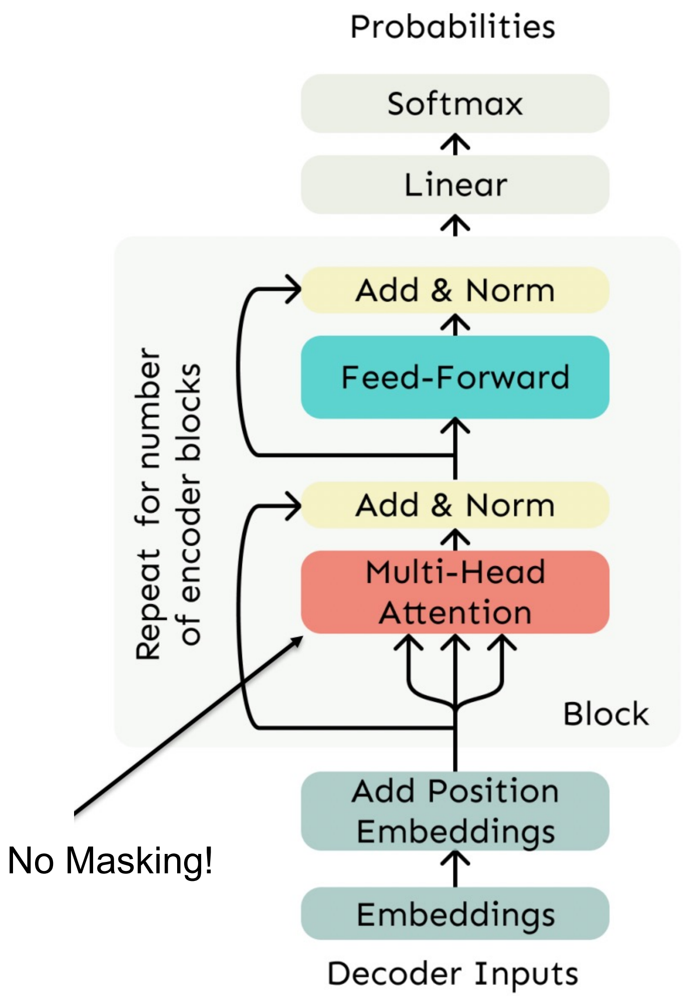
<u>Trick 2 & 3:</u> Optimization Tricks

- Residual Normalization (add the input back)
  - $x + f(x)$
- Layer Normalization
  - Gradient descent
- In most Transformer diagrams, these are often written together as “Add & Norm”



Transformer - Encoder

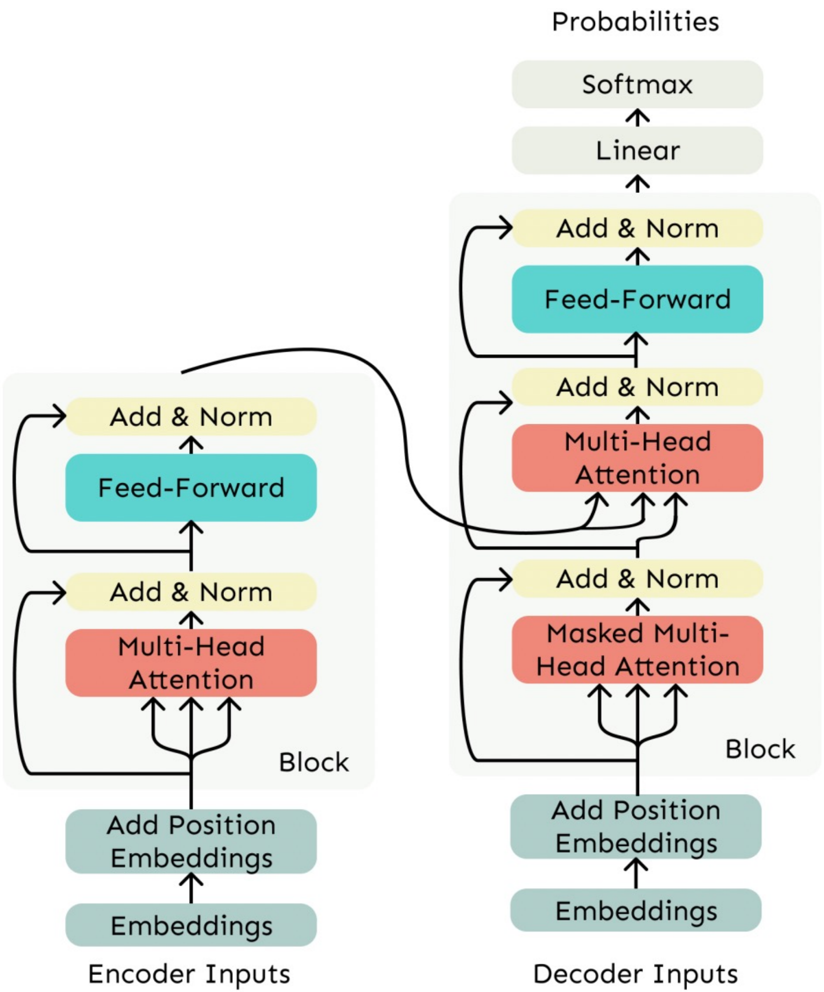
- The Transformer Decoder constrains to unidirectional context, as for language models.
- What if we want bidirectional context?
- This is the Transformer Encoder. The only difference is that we remove the masking in the self-attention.



Transformers Encoder-Decoder

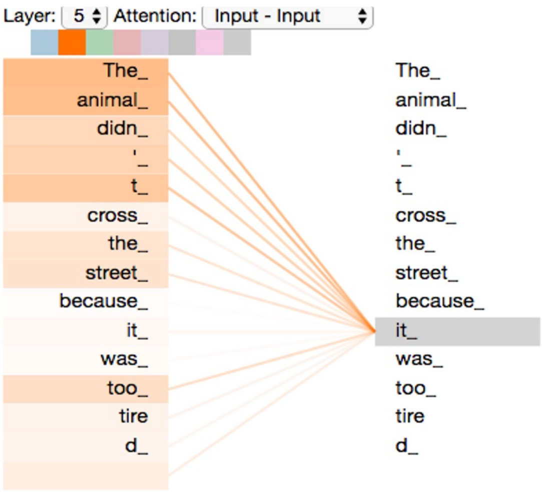
- In Translation, we processed the source sentence with a bidirectional model and generated the target with a unidirectional model.
- For this kind of seq2seq format, we use a Transformer Encoder-Decoder.

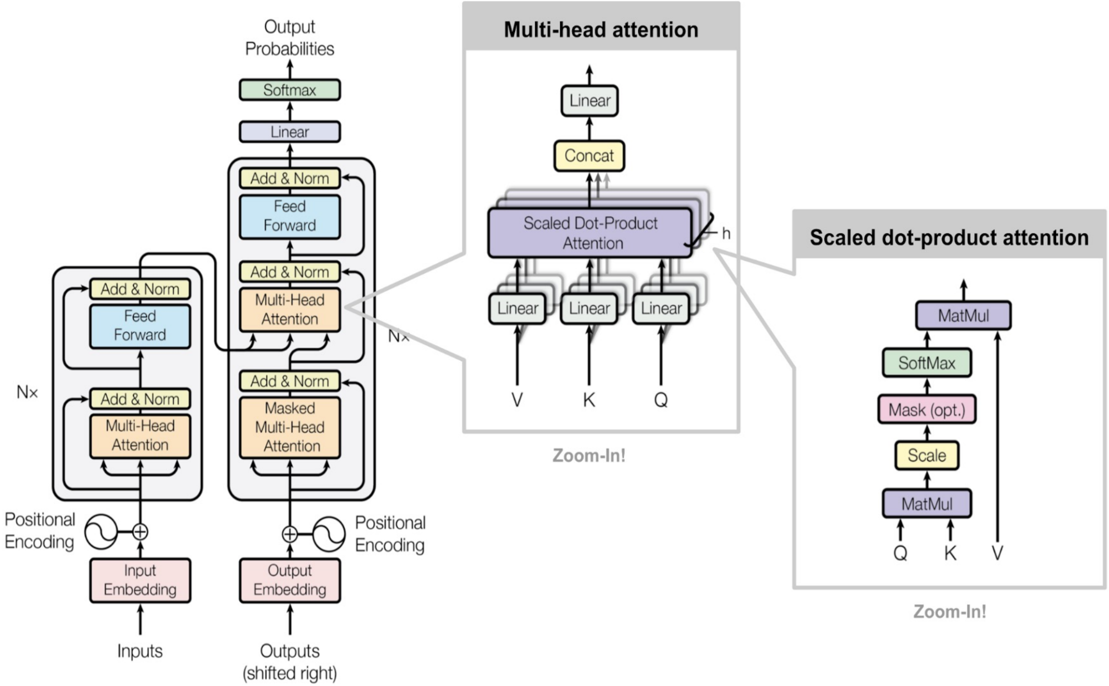
- We use a normal Transformer Encoder.
- Our Transformer Decoder is modified to perform cross-attention to the output of the Encoder



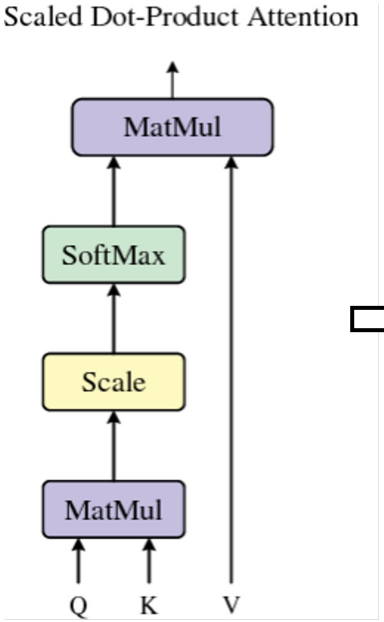
The Self-Attention Process (diagrammatic)

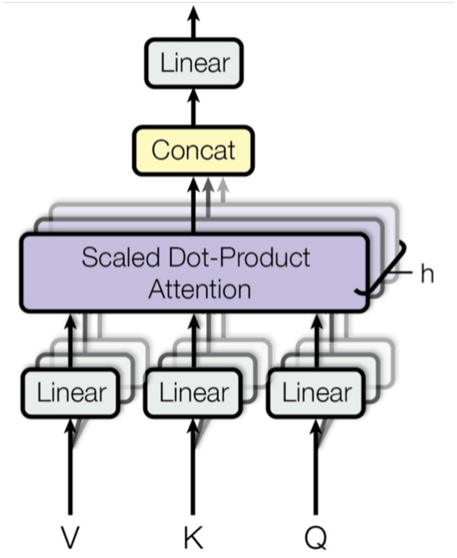
$$\text{Attention}(\mathbf{Q},\mathbf{K},\mathbf{V}) = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{n}}\right)\mathbf{V}$$





Multi-Head Attention (diagrammatic)



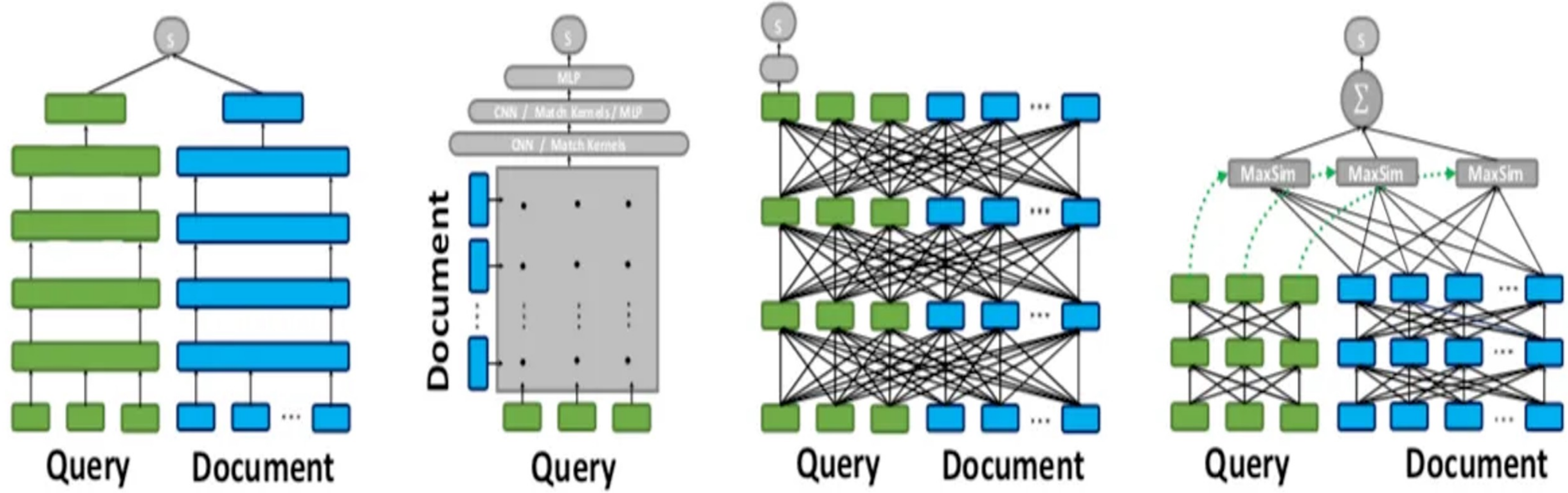


$$\mathrm{MultiHead}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = [\mathrm{head}_1; \ldots; \mathrm{head}_h]\mathbf{W}^O$$
$$\text{where } \mathrm{head}_i = \mathrm{Attention}(\mathbf{Q}\mathbf{W}_i^Q, \mathbf{K}\mathbf{W}_i^K, \mathbf{V}\mathbf{W}_i^V)$$

Retrieval × LMs

- Document-Query Interaction
- Retrieval-augmented LMs

Retrieval LMs (Multi-Vector Representations)



(a) Representation-based Similarity
(e.g., DSSM, SNRM)

(b) Query-Document Interaction
(e.g., DRMM, KNRM, Conv-KNRM)

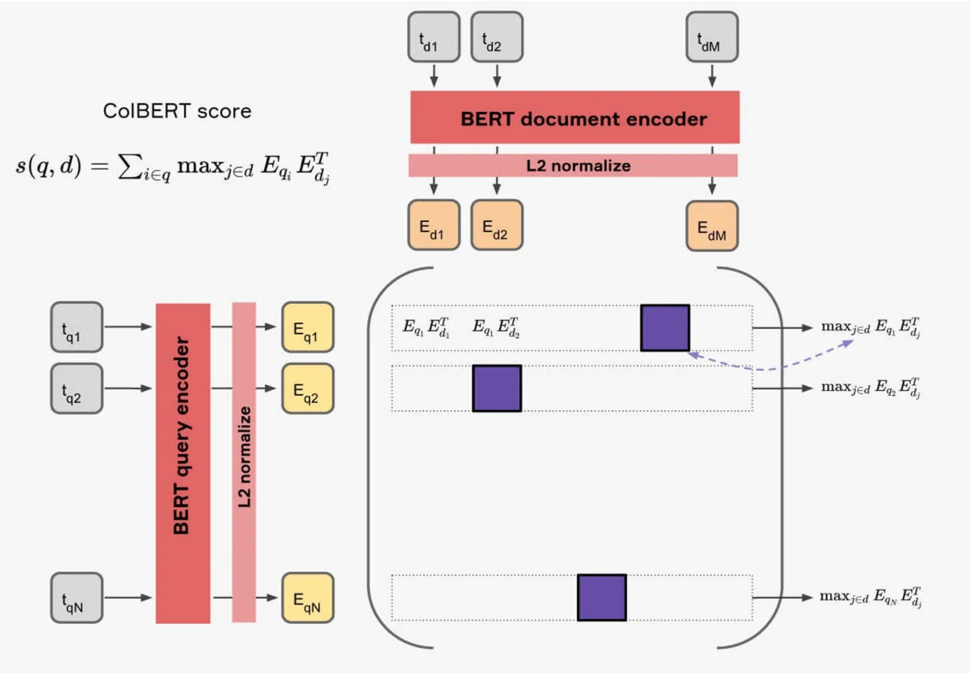
(c) All-to-all Interaction
(e.g., BERT)

(d) Late Interaction
(i.e., the proposed ColBERT)

Retrieval Augmented LLMs

ColBERT (Contextualized Late interaction over BERT)

- ColBERT uses a late interaction architecture
- Encodes the query and document independently
- Compute similarity later
    - Use cached contextual document embeddings



Retrieval Augmented LMs (for QA)

- **REALM** is a language model pre-training paradigm
- Novelty: It also incorporates a knowledge retriever to retrieve textual world knowledge
- REALM models avoid relying solely on model parameters, which can lead to memorizing all knowledge

$$p(y \mid x) = \sum_{z \in \mathcal{Z}} p(y \mid z, x)\, p(z \mid x).$$

Generate

Retrieve

Knowledge Retriever The retriever is defined using a dense inner product model:

$$p(z \mid x) = \frac{\exp f(x, z)}{\sum_{z'} \exp f(x, z')},$$

$$f(x, z) = \texttt{Embed}_{\texttt{input}}(x)^\top \texttt{Embed}_{\texttt{doc}}(z),$$

REALM Performance