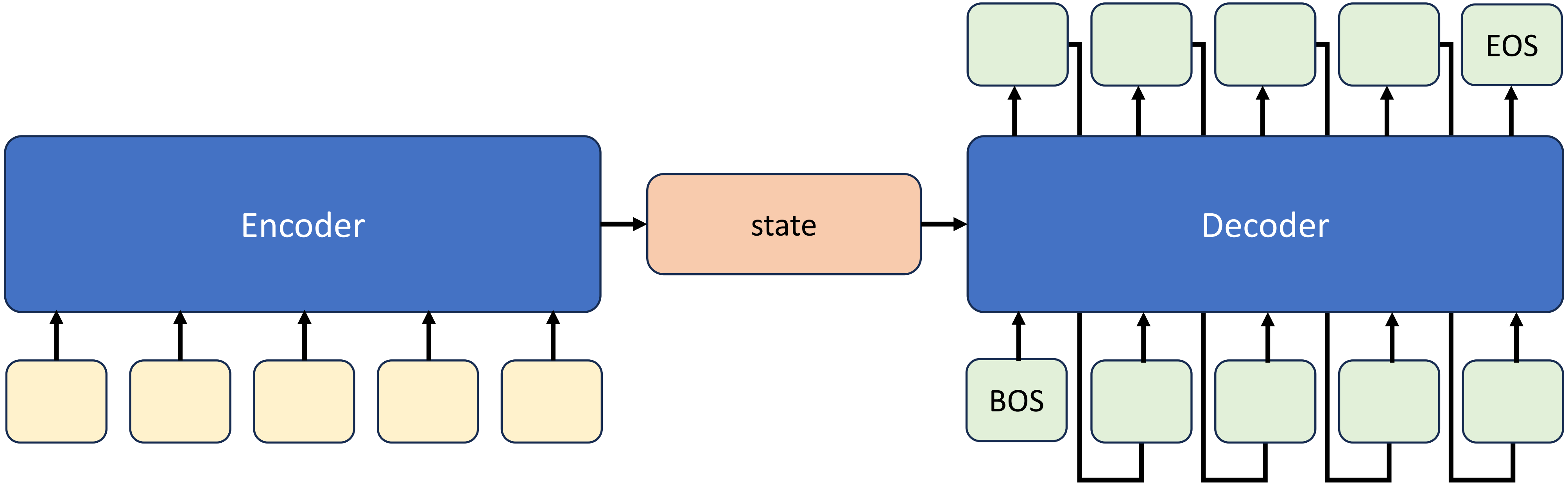
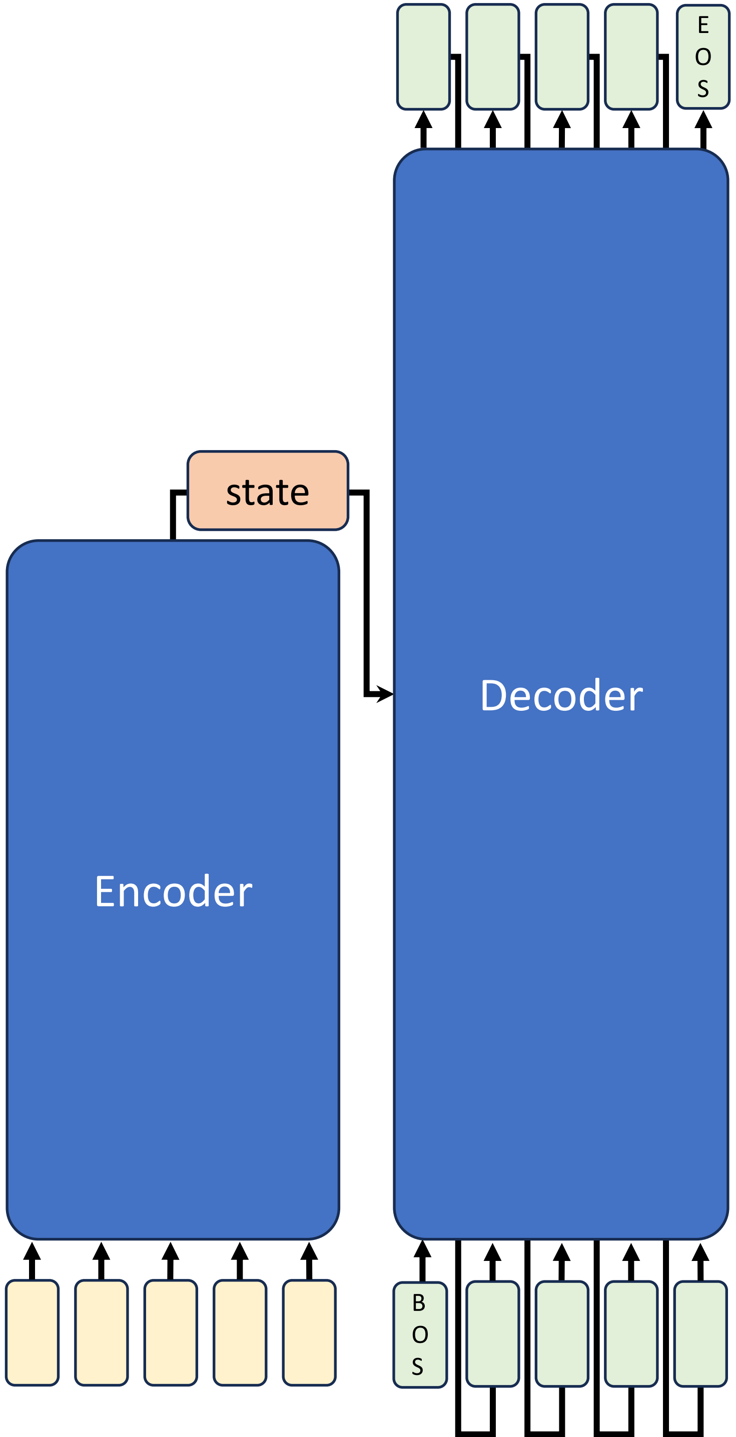
Large Language Models

Transformers

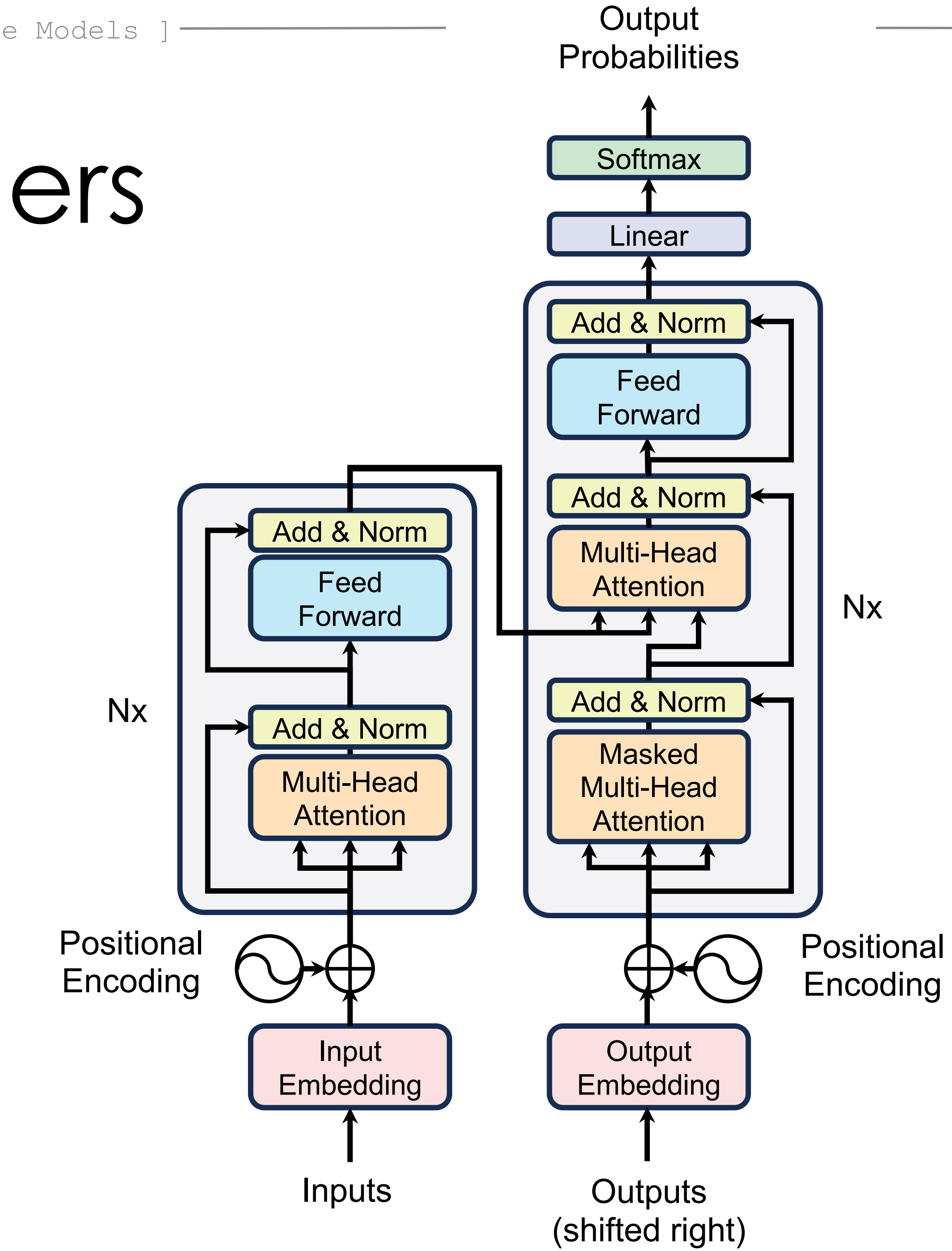
Flavio Giobergia

Encoder-decoder architecture




state
Encoder
Decoder
B
O
S
E
O
S

Transformers



Transformer architecture

- The transformer (Vaswani et al, 2017) is a novel *seq2seq architecture*
    - Originally proposed as an *encode-decoder* architecture
    - (We will see that encoder-only and *decoder-only* are also used)
- No longer based on RNN!
    - Transformers no longer struggle with long-term dependencies
        - However, it becomes *computationally expensive for long sequences* – O(N^2)
    - No need to unroll the model N times (gradients no longer disappear/explode)
    - Parallelizable architecture
- Various aspects of interest we will cover (not all are novel!)
    - Tokenization
    - Positional encoding
    - Attention mechanism
        - (Multi-head) Self-attention & cross-attention

“Attention is all you need”, https://arxiv.org/pdf/1706.03762

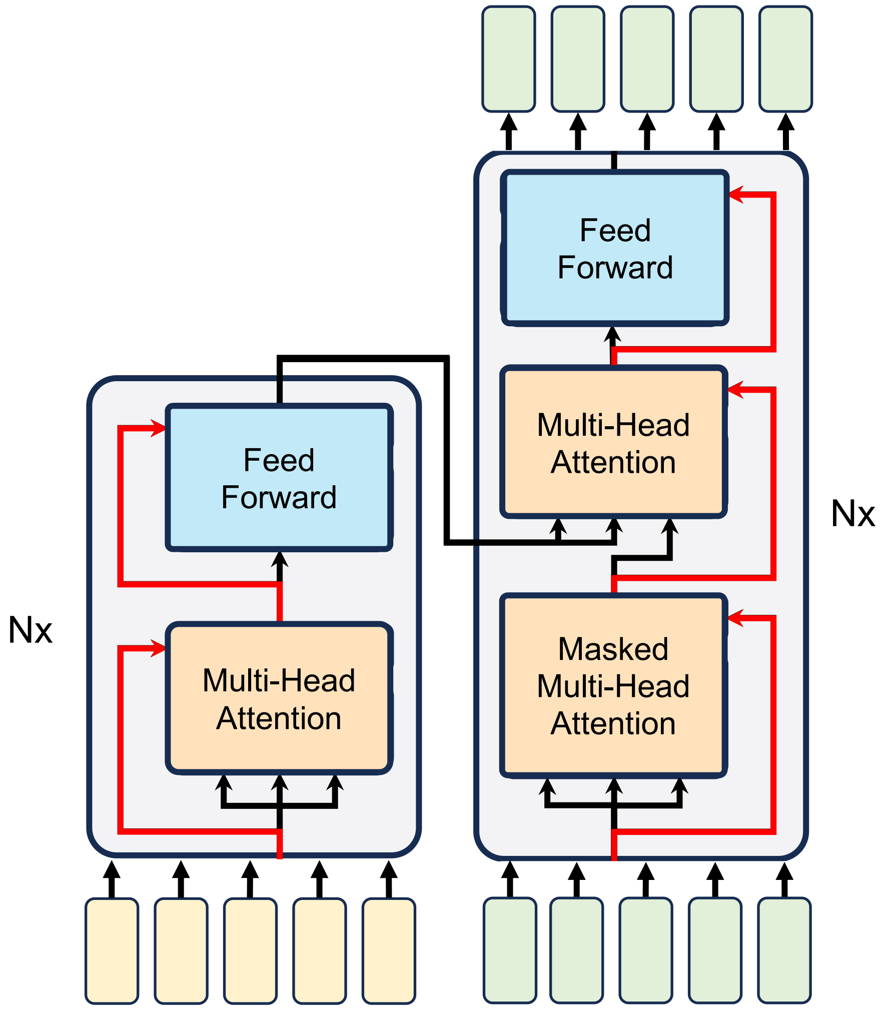
General architecture

Simplified architecture

- Let’s make some simplifying assumptions (we will revisit them later)

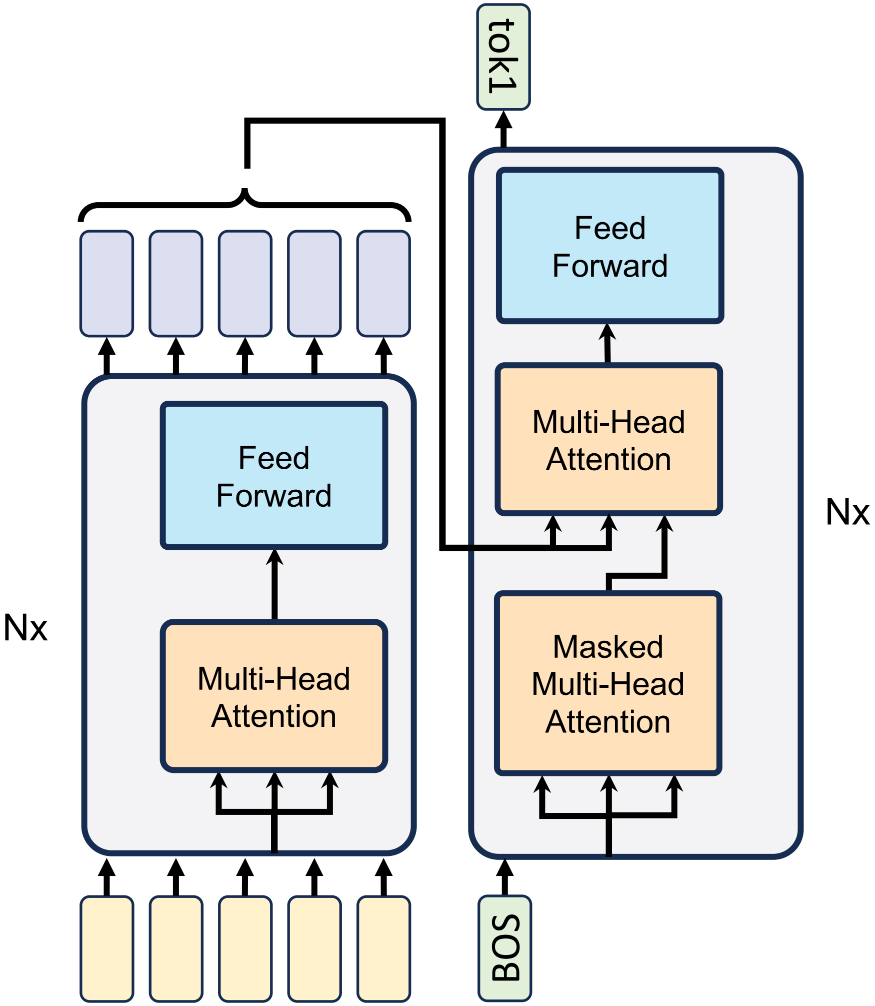
1. Let’s ignore “residual” connections
2. Inputs/outputs are simply (vectors representing) tokens
3. Let’s also ignore Add & Norm layers

- We know what FF networks do
- We are left with these “attention” blocks, having 3 inputs and an output
  - We will explore these modules in more detail!



Generating the 1st token

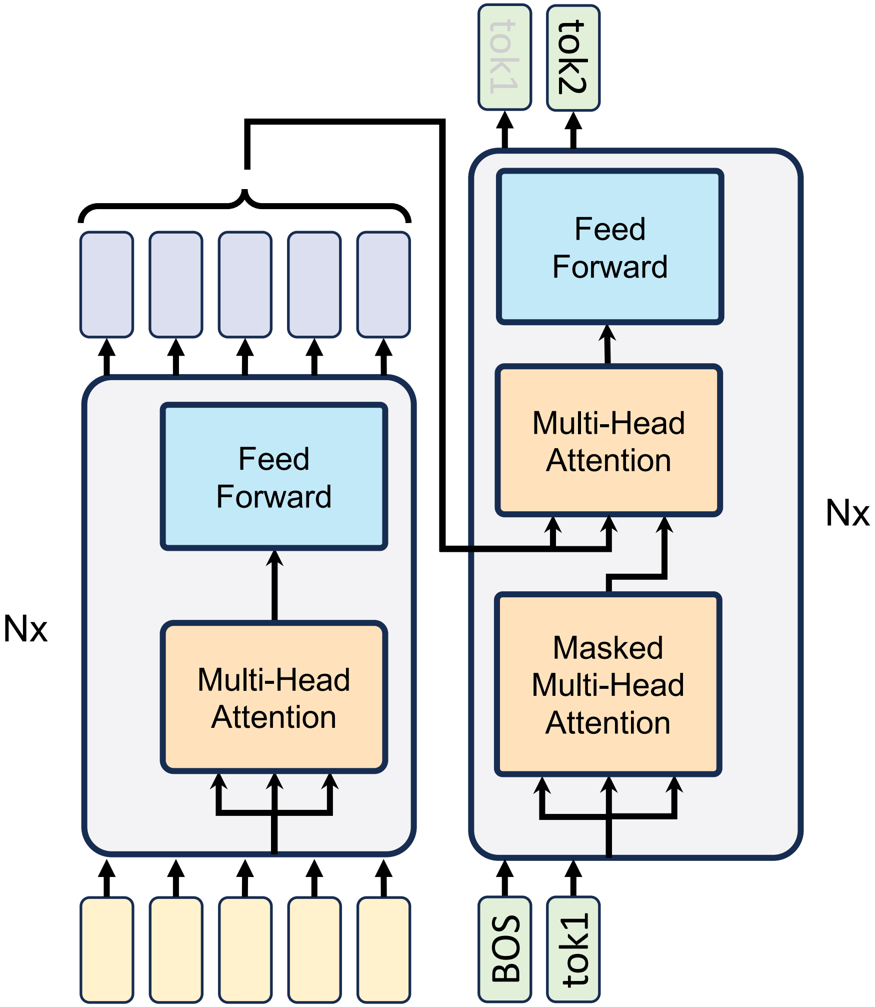
- Encoder
  - The entire *input sequence* is encoded, producing one *code* for each step
- Decoder
  - We feed a “beginning of sequence” (BOS) special token, to:
    - Tell the encoder that we are at the beginning of the sequence
    - Fill the “void” input (the model has not generated anything yet!)
  - The output will be the 1st token generated by the transformer (“tok1”)



Note
The output of the transformer is a probability for each possible token. We can extract a token (e.g., the arg max, or sampling the distribution)

Generating the 2nd token

- Encoder
  - The same *input sequence* is encoded (like before), producing the same *code*
- Decoder
  - We feed BOS + tok1
  - The output will be the 2nd token generated by the transformer
- We continue in this way until the transformer generates the End Of Sequence (EOS) token



Note
The decoder will output a prediction for each input token. At inference time, we are interested the latest generated token. At training time, the loss is computed on all tokens generated at that step!

Tokenization

Tokenization

- Transformers need to process natural language, in the form of sentences
- We need a way to split a sentence into units
  - This process is called *tokenization*, and splits the sequence into *tokens*
- We already considered some tokenization options in past lectures:
  - Character-level ("case correction problem")
  - Word-level (word embeddings, word2vec)
  - Subword-level (word embeddings, fastText)

Sennrich, Rico, Barry Haddow, and Alexandra Birch. "Neural machine translation of rare words with subword units." *arXiv preprint arXiv:1508.07909* (2015).

Character-level tokenization

```
Mrs. Rucastle was downstairs, so I had an admirable opportunity.
Mrs. Rucastle was downstairs, so I had an admirable opportunity.
```

- *Description*
  - Breaks text into individual characters
  - Each character is treated as as token
- *Pros*
  - No out-of-vocabulary issues
  - Robust to misspellings and variations
- *Cons*
  - Longer sequences (we'll see, transformers are O(N^2) in the sequence length)
  - Slower/more complex training
  - Semantic information is harder to capture
  - Inefficient for common words

Word-level tokenization

```
Mrs. Rucastle was downstairs, so I had an admirable opportunity.
Mrs. Rucastle was downstairs, so I had an admirable opportunity.
```

- *Description*
  - Breaks text into individual words.
  - Each word is treated as as token
- *Pros*
  - Captures semantic meaning directly
  - Shorter sequences compared to character-level
  - More intuitive/easier to understand
- *Cons*
  - Out-of-vocabulary (OOV) issues for rare or new words
  - Does not leverage some information shared among words (e.g. prefixes/suffixes)
  - Larger vocabulary needed, leading to memory inefficiency
  - Vectors for rarer words are trained “less”

Subword-level tokenization

Note
Tokenization obtained with BERT base tokenizer (cased).

```
Mrs. Rucastle was downstairs, so I had an admirable opportunity.
Mrs. Rucastle was downstairs, so I had an admirable opportunity.
```

- *Description*
  - Breaks text into subword units (e.g., prefixes, suffixes)
  - fastText uses n-grams contained in each word
  - A more commonly used method is Byte-Pair Encoding (BPE)
- *Pros*
  - Balances between character-level and word-level
  - Handles out-of-vocabulary words effectively
  - Compact vocabulary with better generalization
  - Efficient for both frequent and rare words
- *Cons*
  - Common words are represented by fewer subwords w.r.t. rarer words
  - Requires defining a subword definition policy
  - The subword definition policy may introduce some computational overhead

Subword-level tokenization

- In practice, subword-level is the most commonly adopted tokenization method
  - With very few exceptions, e.g. CharBERT [1] (character-based)
- A simple, yet commonly adopted technique is *Byte-Pair Encoding* (BPE)
- Other approaches/variants exist
  - WordPiece
  - SentencePiece
  - Morfessor

[1] Ma, Wentao, Yiming Cui, Chenglei Si, Ting Liu, Shijin Wang, and Guoping Hu. "CharBERT: Character-aware pre-trained language model." *arXiv preprint arXiv:2011.01513* (2020).

Byte-Pair Encoding – desiderata

- *Idea*
  - Use a data-driven approach to choose the “best” tokens
    - i.e., we don’t manually define the tokens, or the rules to extract them
    - To do so, we need a collection of texts (*corpus*)
  - Common “words” (sequences of characters) to be encoded into a single token
  - Rarer sequences will be split into multiple tokens
  - We define the desired maximum number of tokens

Byte-Pair Encoding

Note
If the corpus contains non-ASCII characters, UTF-8 encoding is used (which can represent all 1M+ Unicode characters with a variable-length encoding – from 1 to 4 bytes). UTF-8 is ASCII-compatible -- we work with a base dictionary of 256 terms (byte-level BPE)

1. The corpus is initially encoded with 1 character (byte) = 1 token
2. The frequency of each pair of subsequent tokens is counted in the corpus
3. The most frequent pair of tokens (F_1,F_2) is assigned to a new token T
4. All occurrences of (F_1,F_2) are replaced with T in the corpus
5. Steps 2-4 are repeated until a target number of tokens is extracted

BPE example

- Corpus
  - would a woodchuck chuck wood
- Step 1: Initial encoding (28 tokens)
  - |w|o|u|l|d|␣|a|␣|w|o|o|d|c|h|u|c|k|␣|c|h|u|c|k|␣|w|o|o|d|

Tokens: {␣,a,c,d,h,k,l,o,u,w}

BPE example

- Corpus
  - `would a woodchuck chuck wood`
- Step 1: Initial encoding
  - |w|o|u|l|d|␣|a|␣|w|o|o|d|c|h|u|c|k|␣|c|h|u|c|k|␣|w|o|o|d|
- Step 2: count frequencies of all pairs of tokens
  - |w|o|u|l|d|␣|a|␣|w|o|o|d|c|h|u|c|k|␣|c|h|u|c|k|␣|w|o|o|d|

w, o	1

BPE example

- Corpus
  - `would a woodchuck chuck wood`
- Step 1: Initial encoding
  - `|w|o|u|l|d|␣|a|␣|w|o|o|d|c|h|u|c|k|␣|c|h|u|c|k|␣|w|o|o|d|`
- Step 2: count frequencies of all pairs of tokens
  - `|w|o|u|l|d|␣|a|␣|w|o|o|d|c|h|u|c|k|␣|c|h|u|c|k|␣|w|o|o|d|`

w, o	1
o, u	1

BPE example

- Corpus
  - `would a woodchuck chuck wood`
- Step 1: Initial encoding
  - `|w|o|u|l|d|␣|a|␣|w|o|o|d|c|h|u|c|k|␣|c|h|u|c|k|␣|w|o|o|d|`
- Step 2: count frequencies of all pairs of tokens
  - `|w|o|u|l|d|␣|a|␣|w|o|o|d|c|h|u|c|k|␣|c|h|u|c|k|␣|w|o|o|d|`

w, o	1
o, u	1
u, l	1

BPE example

- Corpus
  - `would a woodchuck chuck wood`
- Step 1: Initial encoding
  - |w|o|u|l|d|␣|a|␣|w|o|o|d|c|h|u|c|k|␣|c|h|u|c|k|␣|w|o|o|d|
- Step 2: count frequencies of all pairs of tokens
  - |w|o|u|l|d|␣|a|␣|w|o|o|d|c|h|u|c|k|␣|c|h|u|c|k|␣|w|o|o|d|

w, o	1
o, u	1
u, l	1
l, d	1

BPE example

- Corpus
  - `would a woodchuck chuck wood`
- Step 1: Initial encoding
  - |w|o|u|l|d|␣|a|␣|w|o|o|d|c|h|u|c|k|␣|c|h|u|c|k|␣|w|o|o|d|
- Step 2: count frequencies of all pairs of tokens
  - |w|o|u|l|d|␣|a|␣|w|o|o|d|c|h|u|c|k|␣|c|h|u|c|k|␣|w|o|o|d|

w, o	3
o, u	1
u, l	1
l, d	1
d, ␣	1
␣, a	1
a, ␣	1
␣, w	2
o, o	2
o, d	2
d, c	1
c, h	2
h, u	2
u, c	2
c, k	2
k, ␣	2
␣, c	1

BPE example

Tokens: {␣,a,c,d,h,k,l,o,u,w,**wo**}

- Corpus
  - `would a woodchuck chuck wood`
- Step 1: Initial encoding
  - |w|o|u|l|d|␣|a|␣|w|o|o|d|c|h|u|c|k|␣|c|h|u|c|k|␣|w|o|o|d|
- Step 2: count frequencies of all pairs of tokens
  - |w|o|u|l|d|␣|a|␣|w|o|o|d|c|h|u|c|k|␣|c|h|u|c|k|␣|w|o|o|d|
- Step 3: assign a new token to the most frequent pair of tokens
  - The tokens w, o appear 3 times
  - Assign these two tokens a new token (e.g., **wo**)

w, o	3
o, u	1
u, l	1
l, d	1
d, ␣	1
␣, a	1
a, ␣	1
␣, w	2
o, o	2
o, d	2
d, c	1
c, h	2
h, u	2
u, c	2
c, k	2
k, ␣	2
␣, c	1

BPE example

Tokens: {␣,a,c,d,h,k,l,o,u,w,**wo**}

- Corpus
  - `would a woodchuck chuck wood`
- Step 1: Initial encoding
  - |w|o|u|l|d|␣|a|␣|w|o|o|d|c|h|u|c|k|␣|c|h|u|c|k|␣|w|o|o|d|
- Step 2: count frequencies of all pairs of tokens
  - |w|o|u|l|d|␣|a|␣|w|o|o|d|c|h|u|c|k|␣|c|h|u|c|k|␣|w|o|o|d|
- Step 3: assign a new token to the most frequent pair of tokens
  - The tokens w, o appear 3 times
  - Assign these two tokens a new token (e.g., **wo**)
- Step 4: replace the two tokens |w|o| with the new token **|wo|**
  - |w|o|u|l|d|␣|a|␣|w|o|o|d|c|h|u|c|k|␣|c|h|u|c|k|␣|w|o|o|d|
  - |wo|u|l|d|␣|a|␣|wo|o|d|c|h|u|c|k|␣|c|h|u|c|k|␣|wo|o|d|

Note
Multiple pairs can of course have the same frequency. Choose any one among them as the next pair!

BPE example

- Repeat until the desired number of tokens is reached!
  - |w|o|u|l|d|␣|a|␣|w|o|o|d|c|h|u|c|k|␣|c|h|u|c|k|␣|w|o|o|d|

- At the second iteration, | |wo| is merged into | wo|
  - |wo|u|l|d|␣|a|␣|wo|o|d|c|h|u|c|k|␣|c|h|u|c|k|␣|wo|o|d|

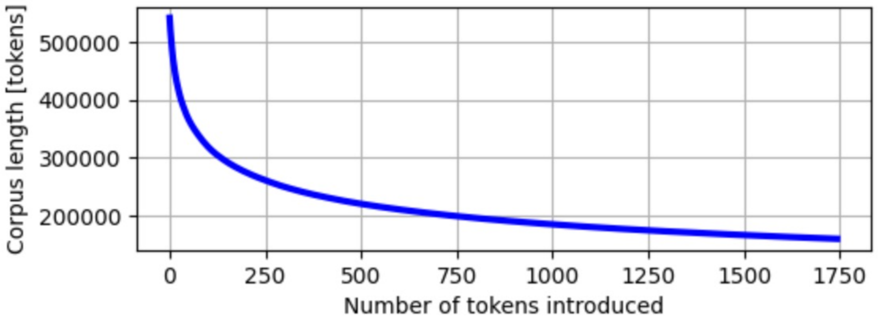
  - |wo|u|l|d|␣|a|␣wo|o|d|c|h|u|c|k|␣|c|h|u|c|k|␣wo|o|d|

Tokens: {␣,a,c,d,h,k,l,o,u,w,wo,␣wo}

BPE example

- |w|o|u|l|d|␣|a|␣|w|o|o|d|c|h|u|c|k|␣|c|h|u|c|k|␣|w|o|o|d|
- |wo|u|l|d|␣|a|␣|wo|o|d|c|h|u|c|k|␣|c|h|u|c|k| |wo|o|d|
- |wo|u|l|d|␣|a|␣wo|o|d|c|h|u|c|k|␣|c|h|u|c|k|␣wo|o|d|
- |wo|u|l|d|␣|a|␣woo|d|c|h|u|c|k|␣|c|h|u|c|k|␣woo|d|
- |wo|u|l|d|␣|a|␣wood|c|h|u|c|k|␣|c|h|u|c|k|␣wood|
- |wo|u|l|d|␣|a|␣wood|ch|u|c|k|␣|ch|u|c|k|␣wood|
- |wo|u|l|d|␣|a|␣wood|chu|c|k|␣|chu|c|k|␣wood|
- |wo|u|l|d|␣|a|␣wood|chuc|k|␣|chuc|k|␣wood|
- |wo|u|l|d|␣|a|␣wood|chuck|␣|chuck|␣wood|
- |wou|l|d|␣|a|␣wood|chuck|␣|chuck|␣wood|
- ...

BPE results



- The number of tokens used to represent the corpus decreases as we increase the number of tokens used
  - (Quite reasonable)
- The trend is shown in the plot above
  - The first few tokens have the most significant reduction effect, as can be expected from Zipf's law
    - i.e., there is a small number of high-frequency items
- Typically, 10's to 100's of thousands of tokens are used
  - Representing a text with fewer tokens is desirable
    - Shorter sequence, semantics better preserved
  - But, tokenizers should avoid being overly for the adopted corpus

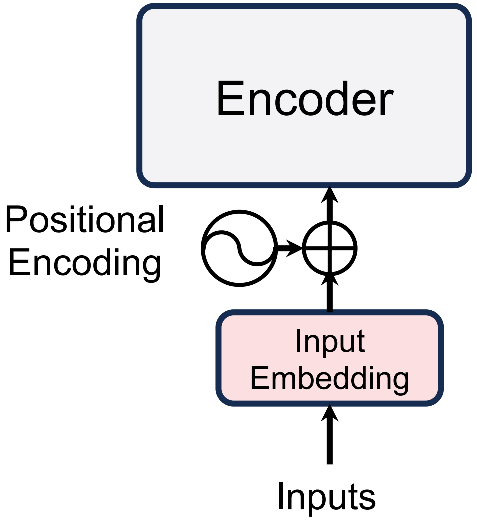
Some extra tokens

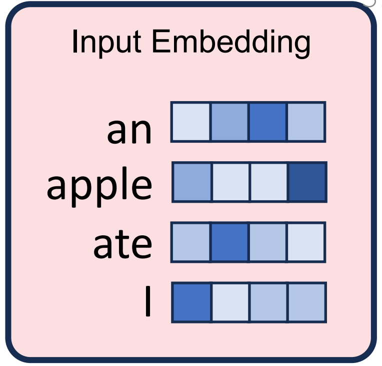
- We introduce some *special* tokens to indicate that special positions within a sequence
- For instance, we can use tokens for beginning/end of sequence:
  - Beginning Of Sequence (*BOS*)
  - End Of Sequence (*EOS*)
- In some tasks (we will see, for BERT), we introduce other tokens:
  - Classification (*CLS*) – with a role similar to BOS (but with a task-specific use)
  - Separator (*SEP*) – to separate sentences within the same input
    - For instance, to separate Question from Answer in Question Answering tasks

Positional encoding

The need for positional encoding (PE)

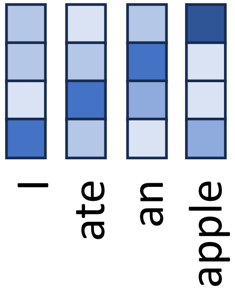
- Each *token* is mapped to a specific vector
- These vectors are *learned*
  - Similarly to word embeddings, we start with random vectors, and adjust them via gradient descent
- The *same token* is mapped to the *same vector*, regardless of position
  - This is not ideal: positions are important in sentences!
- We'll see, *attention* does not really understand sequentiality
  - So our input is essentially a *set* of elements, not a *sequence*!
- We *add* a *positional encoding* to each token vector
  - i.e., a vector that depends on the position of the token itself
  - In this way, some information on the position is passed along



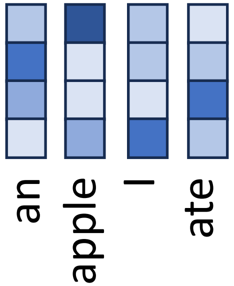


Simple example (no PE)

- Input 1) I ate an apple



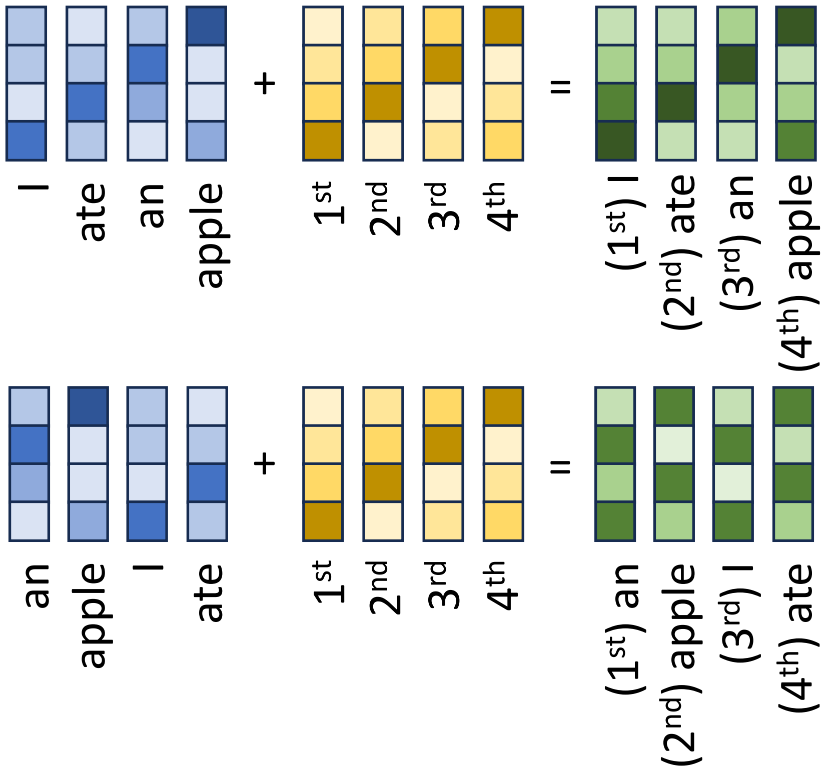
- Input 2) an apple, I ate



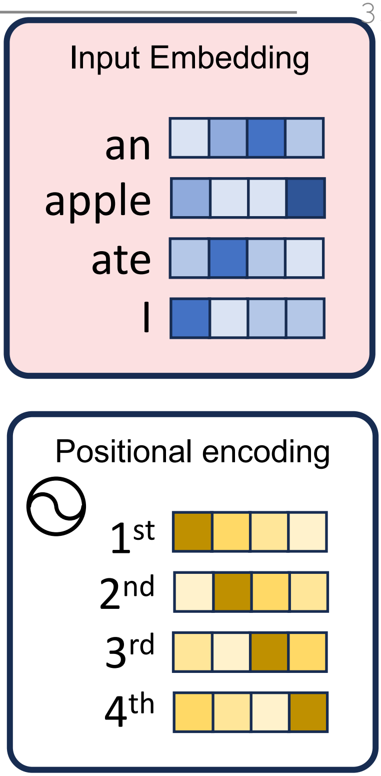
- Both inputs are mapped to the same set of vectors
- No positional encoding used makes the two inputs “identical”
  - (Since the transformer does not handle data in a sequential manner)

Introducing Positional Encoding

- Input 1) I ate an apple
- Input 2) an apple, I ate



- A vector dependent on the position is added to each token vector
- In this way, each vector is altered according to the position in the original sequence



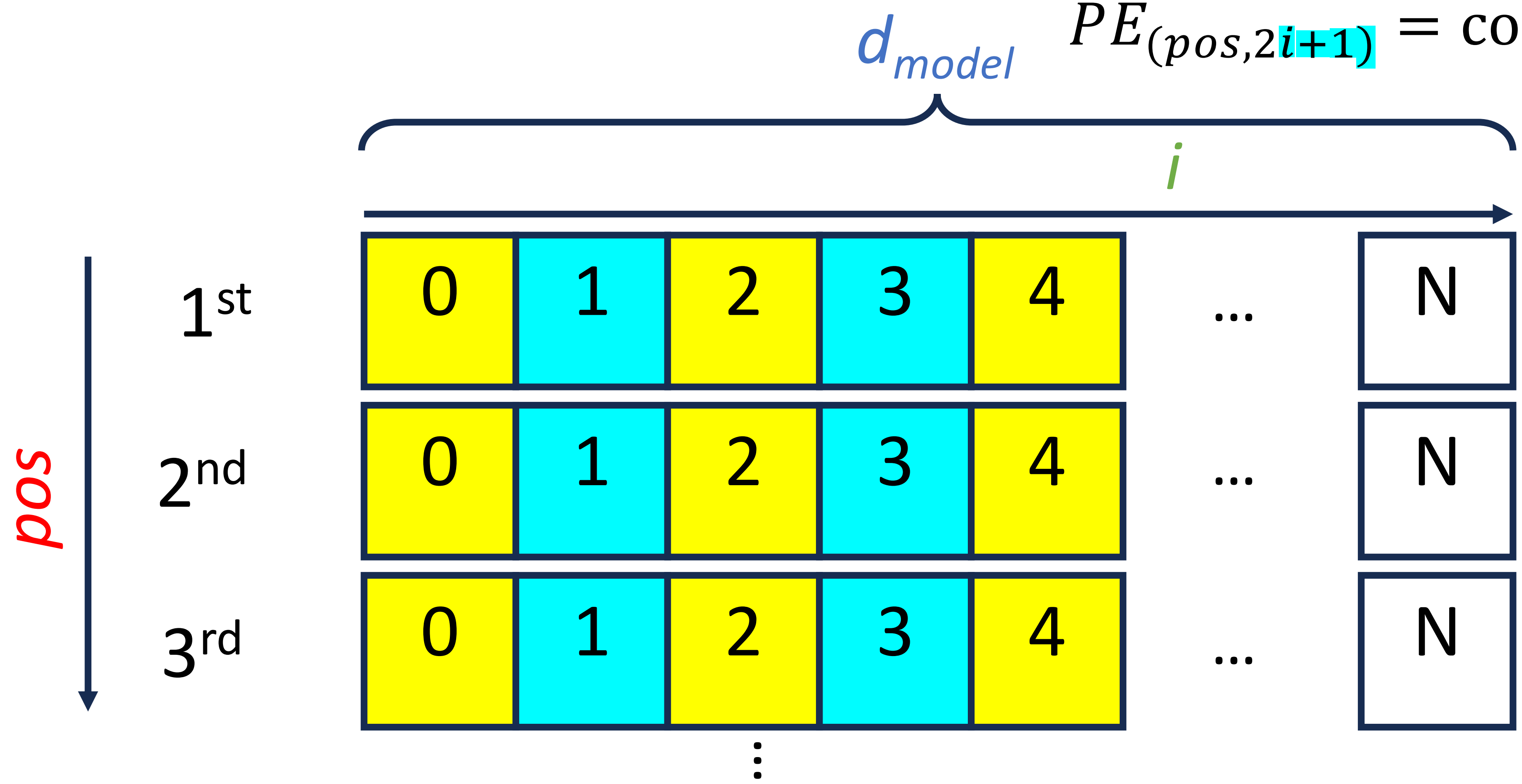
Note
This example uses a simple positional encoding.
Transformers use different approaches (see next slides!)

Sinusoidal Positional encoding

- In the AIAYN paper, the PE vectors are defined as →

$$PE_{(pos,2i)} = \sin\left(\frac{pos}{10000^{2i/d_{model}}}\right)$$

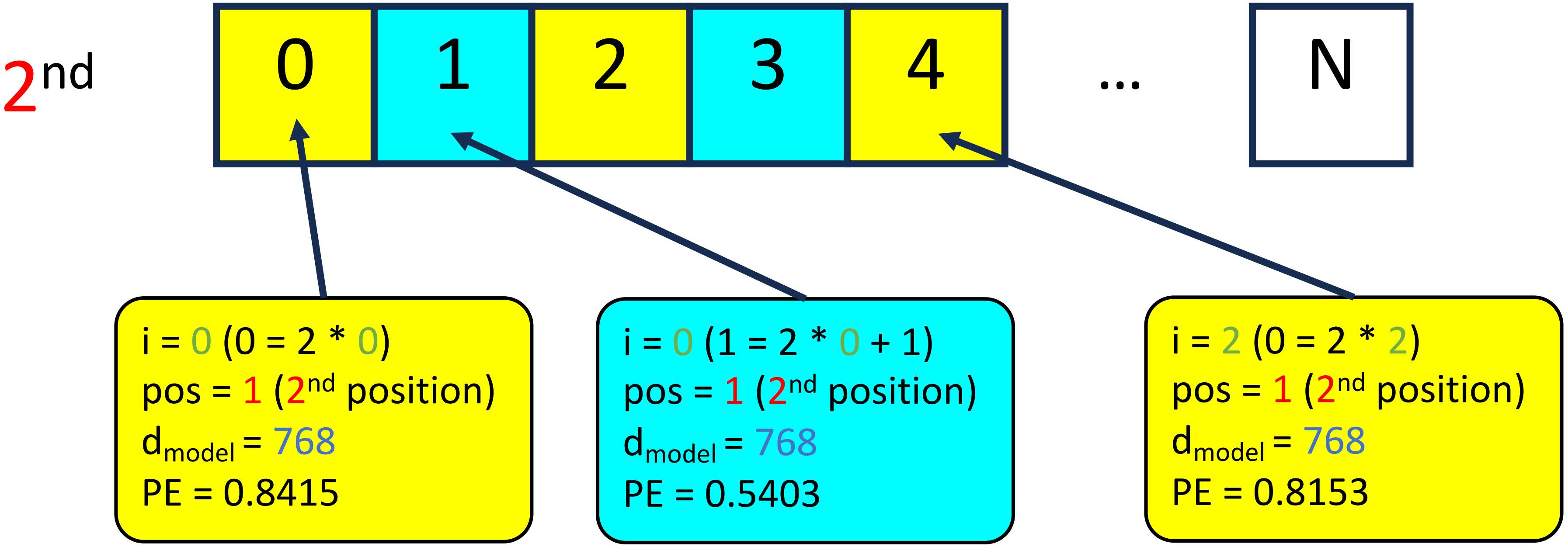
$$PE_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{2i/d_{model}}}\right)$$



1st	0	1	2	3	4	…	N
2nd	0	1	2	3	4	…	N
3rd	0	1	2	3	4	…	N
			⋮				

Sinusoidal Positional encoding

- The positional vectors are defined as follows:
  - $PE_{(pos,2i)} = \sin\left(\frac{pos}{10000^{2i/d_{model}}}\right)$
  - $PE_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{2i/d_{model}}}\right)$



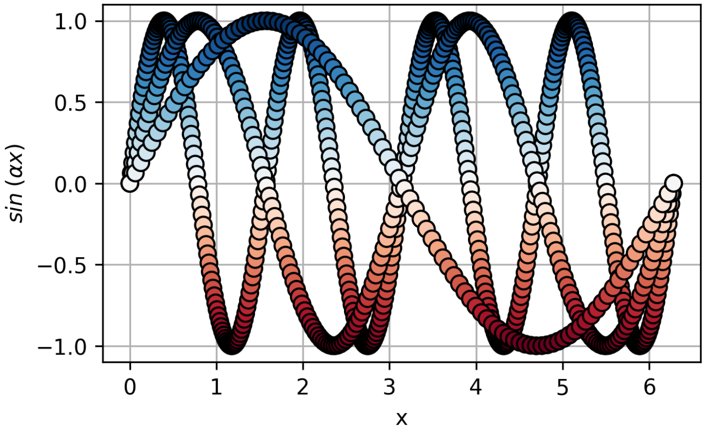
2nd

0	1	2	3	4	…	N

i = 0 (0 = 2 * 0)
pos = 1 (2nd position)
d_{model} = 768
PE = 0.8415

i = 0 (1 = 2 * 0 + 1)
pos = 1 (2nd position)
d_{model} = 768
PE = 0.5403

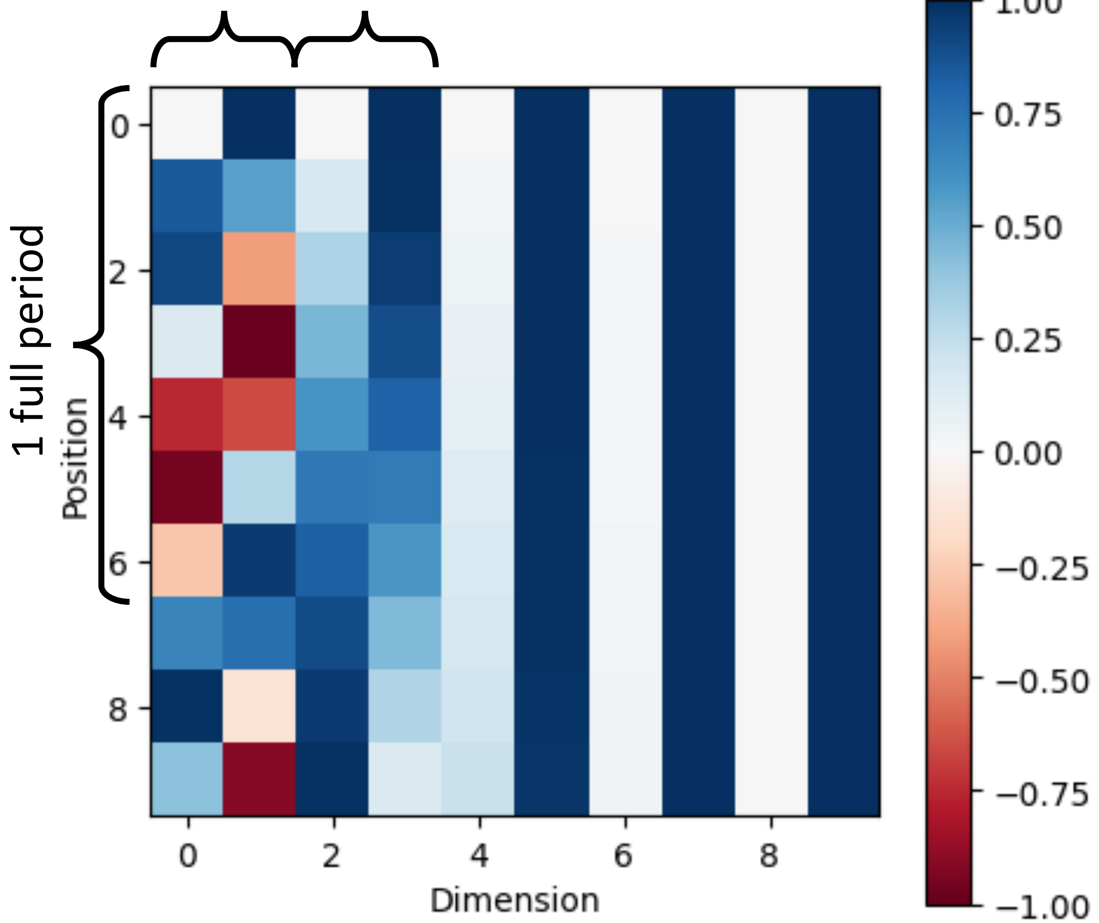
i = 2 (0 = 2 * 2)
pos = 1 (2nd position)
d_{model} = 768
PE = 0.8153



Positional encoding example

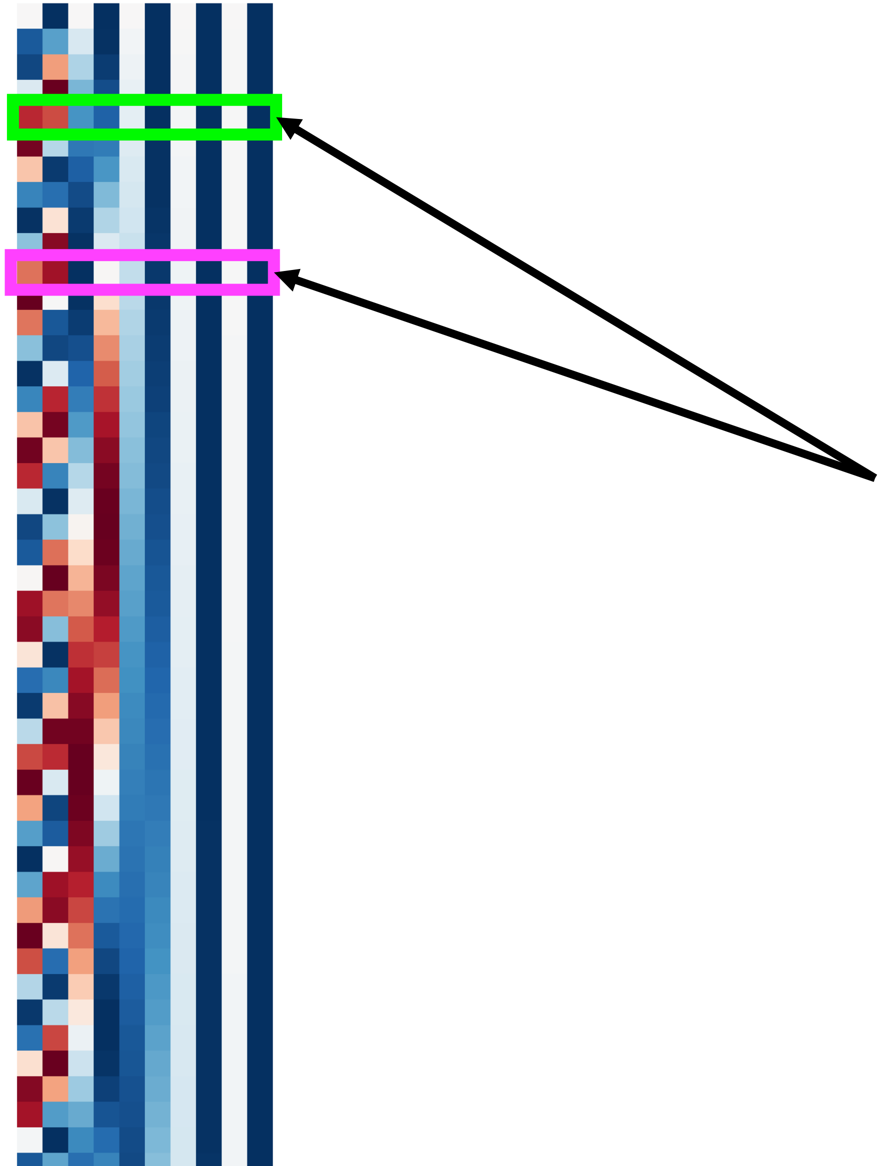
i = 1 (lower frequency – slower change)

i = 0 (high frequency)



1 full period

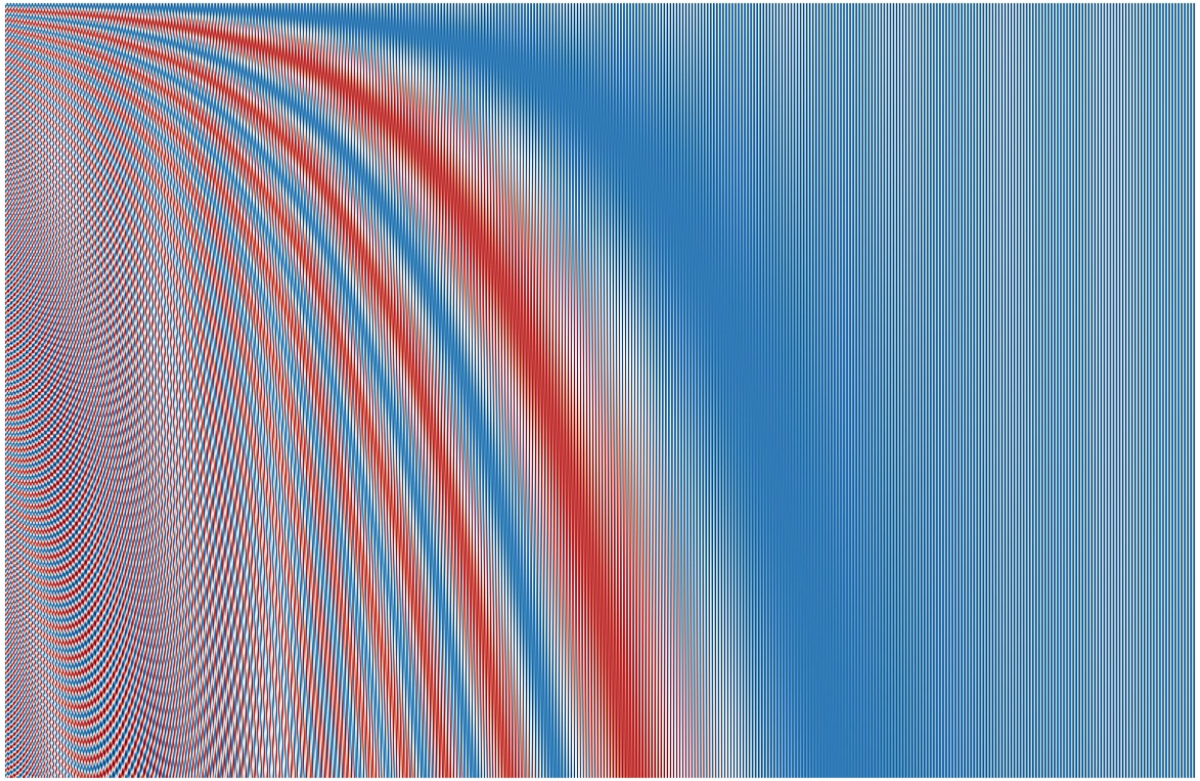
Positional encoding uniqueness



The vector for each position is unique… At least for the first ~60,000 positions ($2\pi \cdot 10{,}000$), then they start repeating (for longer sequences, we can just change the 10,000 constant!)

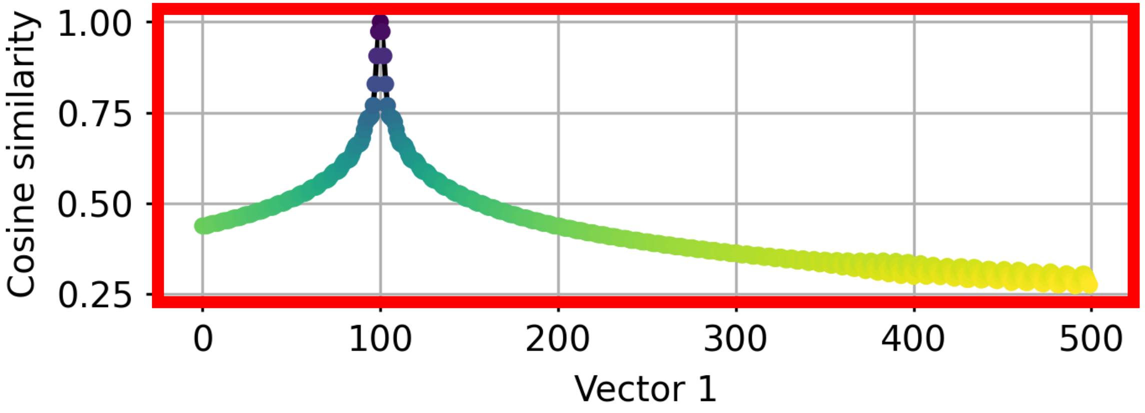
Positional encoding preserves similarity

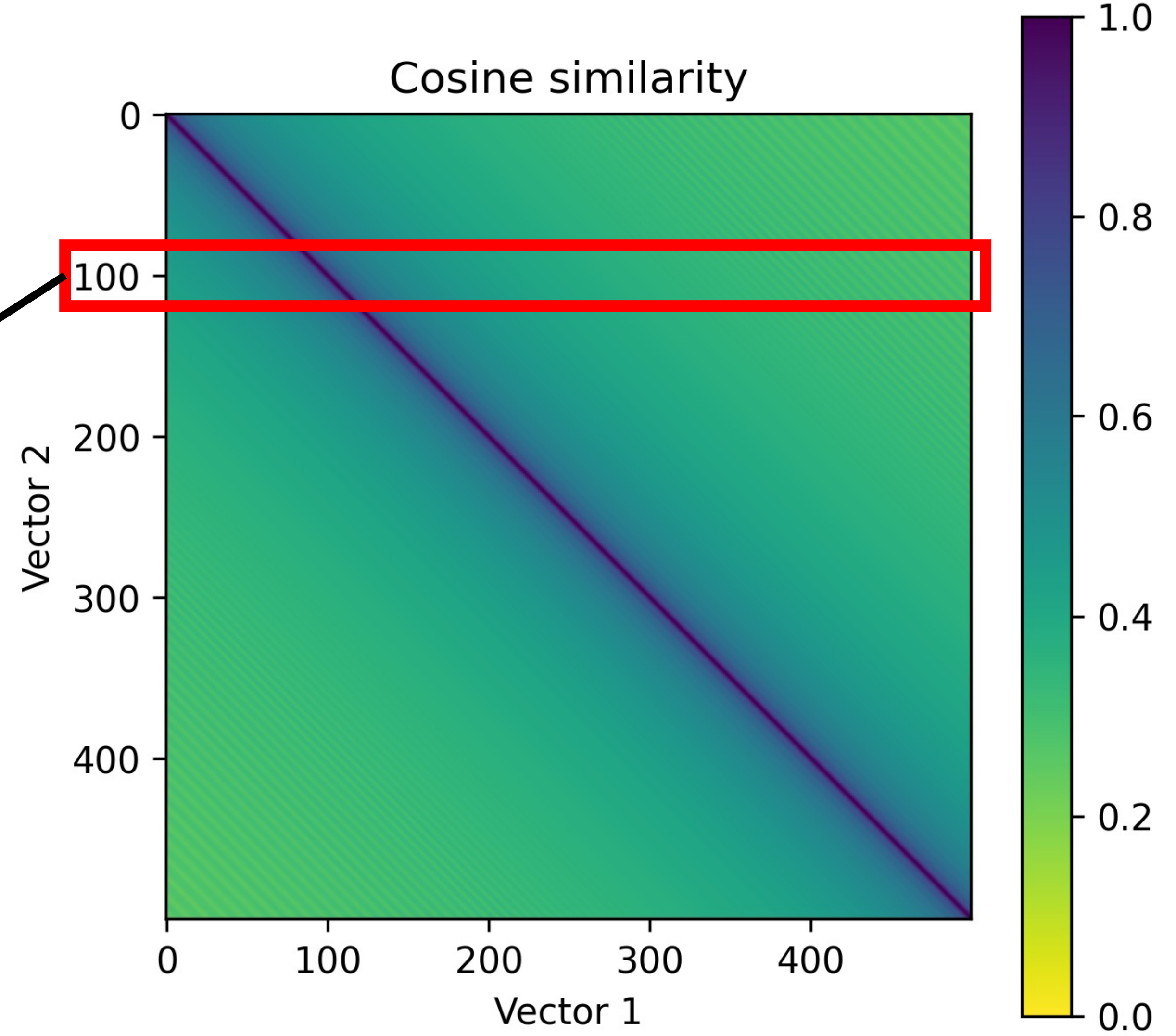
- If position 1 is close to position 2 (e.g., the third vs the fourth words in a sentence) we want positional vectors to also be close!
- We can easily verify that using trigonometric functions ensures that this is the case
- (Example for max position = 1,000 and d_{model} = 768)



Positional encoding preserves similarity

- Cosine similarity of each pair of the previous positional vectors
- Maximum similarity along the diagonal (of course)





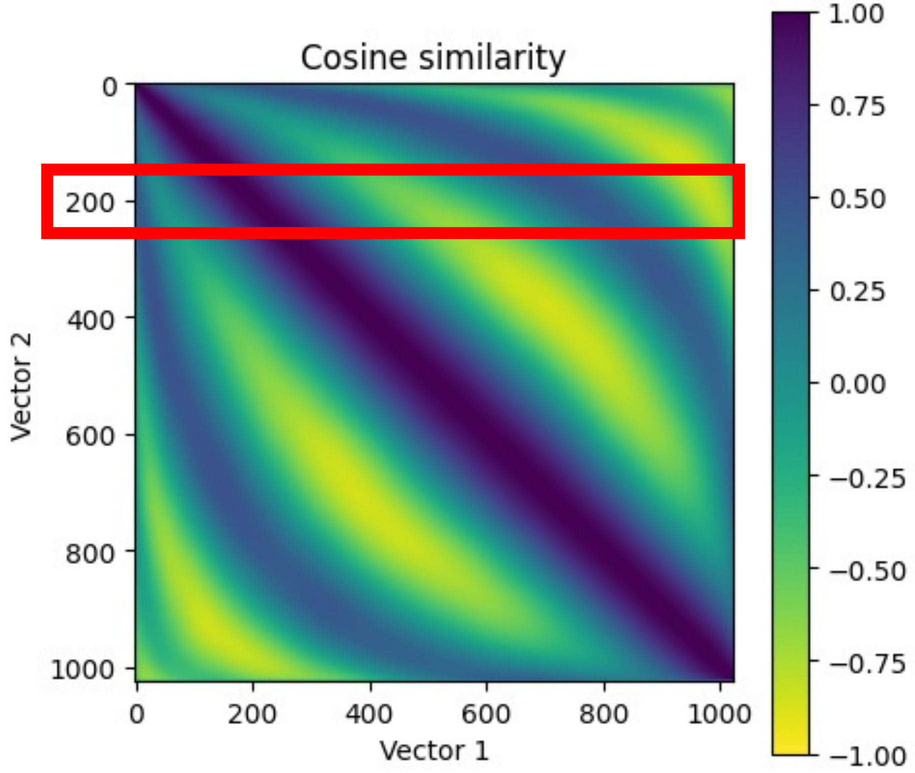
Learned positional embeddings

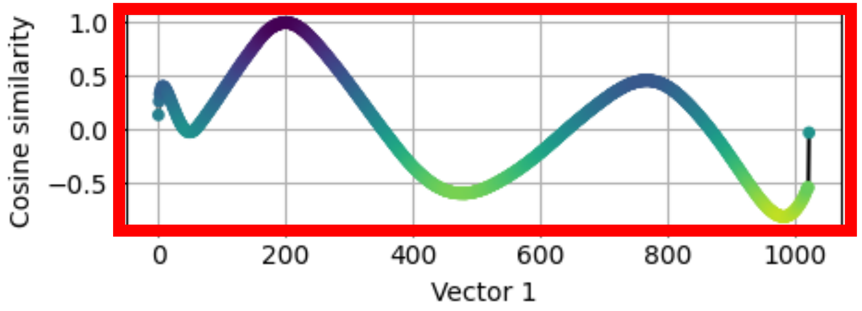
- Some models (e.g., GPT family) do not use sinusoidal PEs.

- Instead, they *learn* *positional embeddings* along with the other weights

- This approach was also considered in AIAYN, but discarded as it did not produce better results

- GPT-2 learned positional embeddings ➔





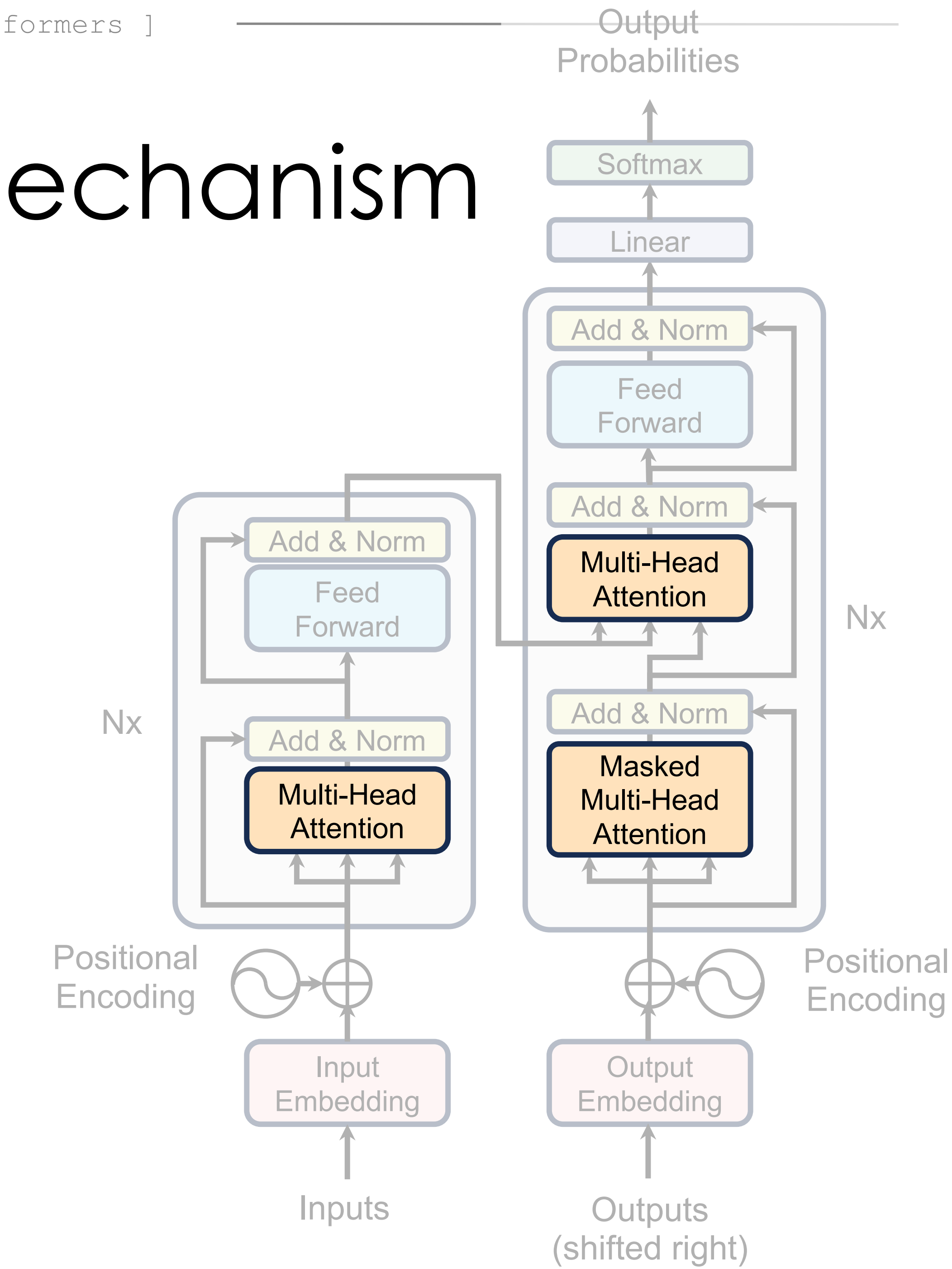
Attention

(Simplified) attention mechanism

- The *attention* mechanism is used in many situations in the Transformer's architecture

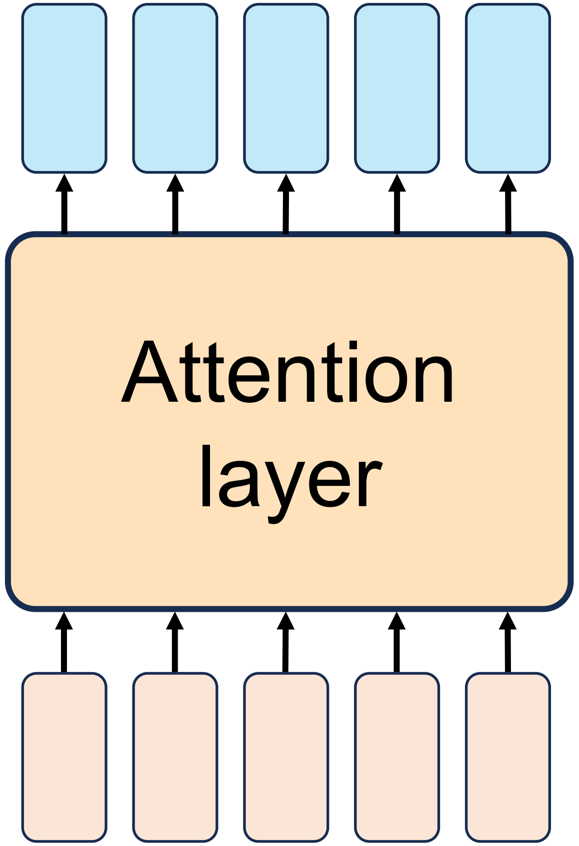
- In short, it takes *independent* input vectors, mixes them up and returns a *contextualized* version of those inputs

- The contextualization is done by summing fractions of the vectors of the sequence to each vector



(Simplified) attention mechanism

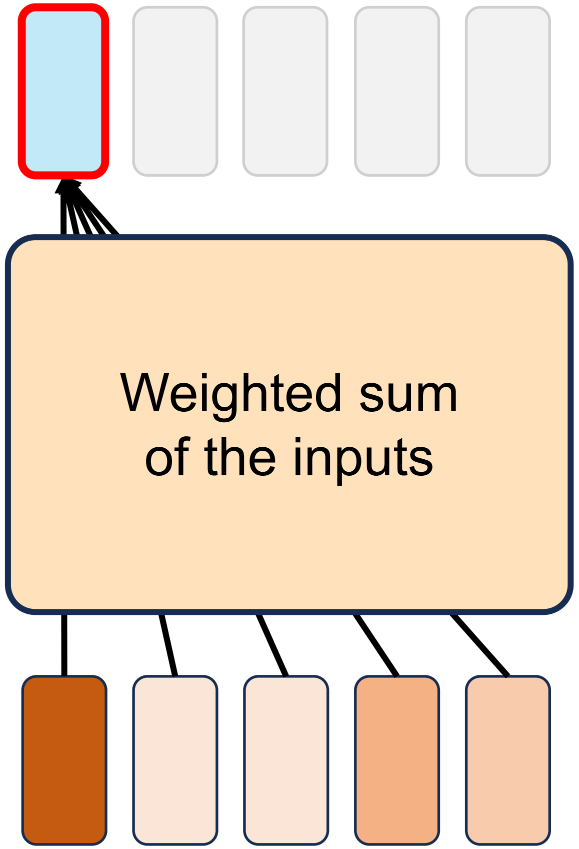
- “How much” of each other vector is added is defined by the *attention* paid to them
  - Ideally, useful tokens will be weighted a lot
  - Unuseful tokens will be weighted less
- The transformer *learns* to choose these weights based on the input sequence
  - Weight = Attention



(Simplified) attention mechanism

- “How much” of each other vector is added is defined by the *attention* paid to them
  - Ideally, useful tokens will be weighted a lot
  - Unuseful tokens will be weighted less

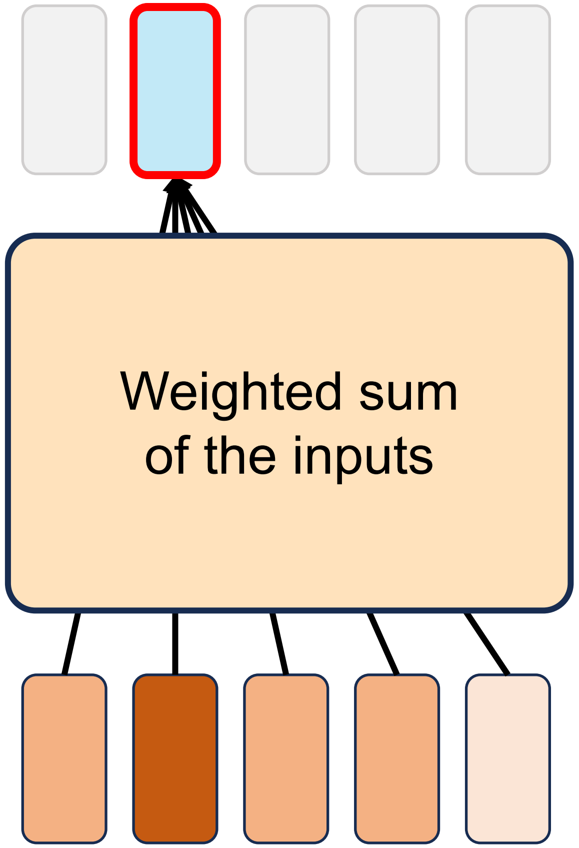
- The transformer *learns* to choose these weights based on the input sequence
  - Weight = Attention



(Simplified) attention mechanism

- “How much” of each other vector is added is defined by the *attention* paid to them
  - Ideally, useful tokens will be weighted a lot
  - Unuseful tokens will be weighted less

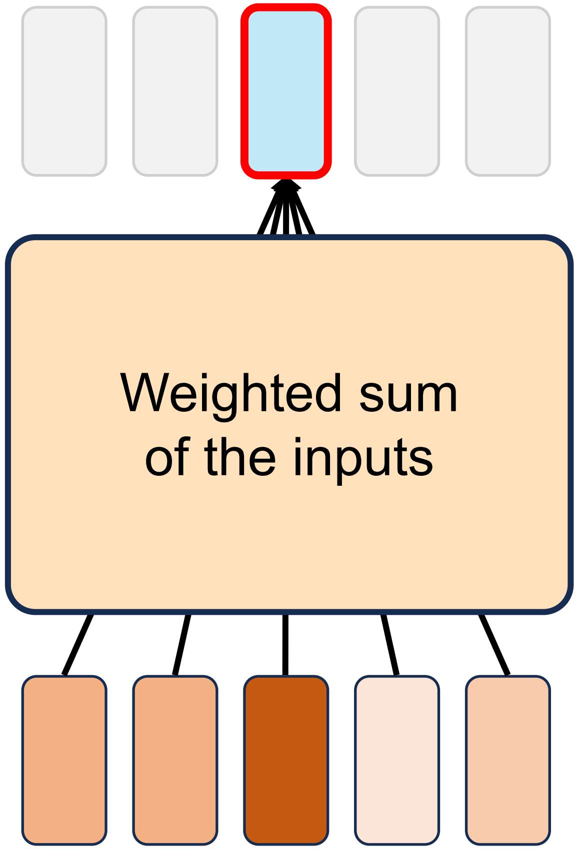
- The transformer *learns* to choose these weights based on the input sequence
  - Weight = Attention



(Simplified) attention mechanism

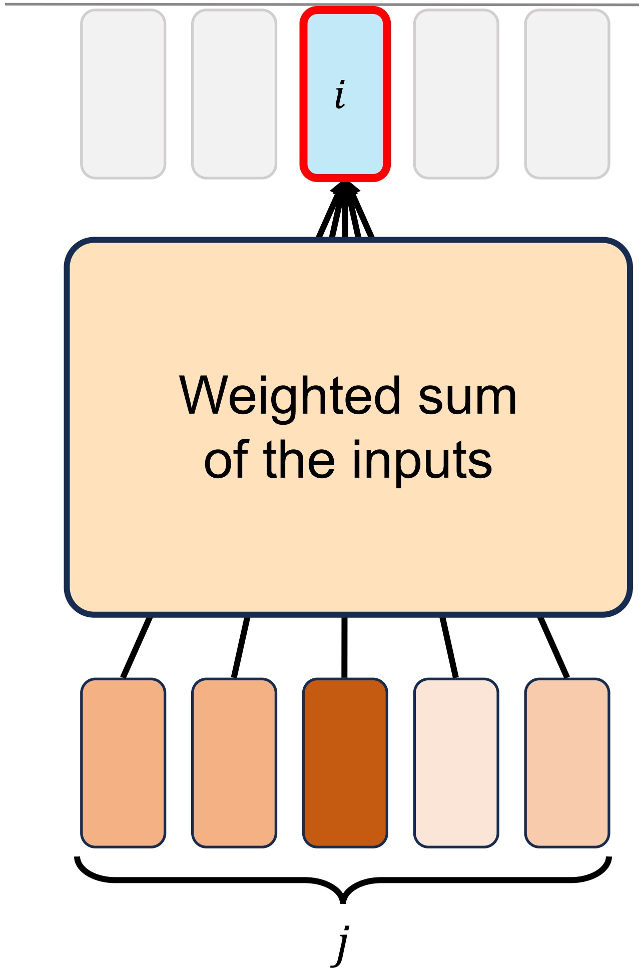
- “How much” of each other vector is added is defined by the *attention* paid to them
  - Ideally, useful tokens will be weighted a lot
  - Unuseful tokens will be weighted less

- The transformer *learns* to choose these weights based on the input sequence
  - Weight = Attention



Attention definition

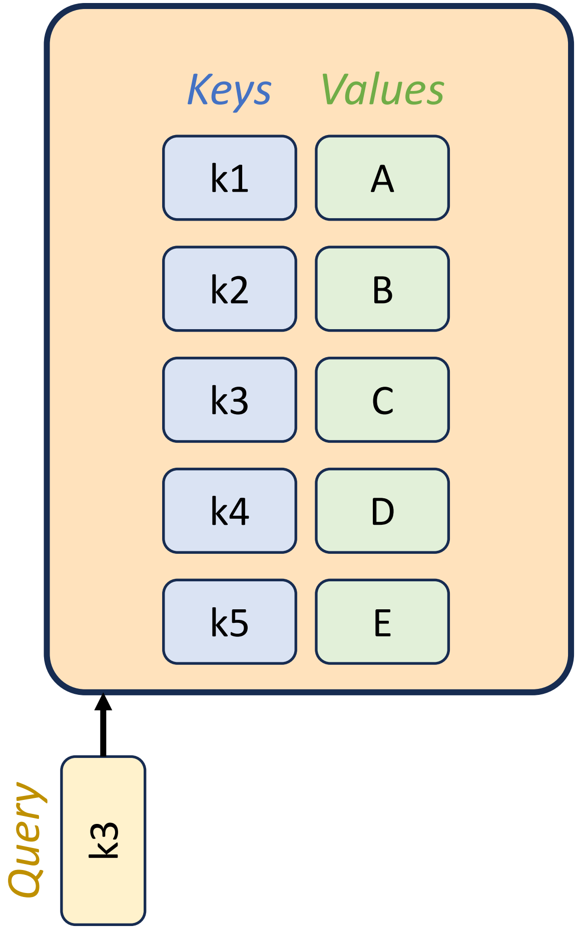
- The attention module requires:
  - An *input* (to be *contextualized*)
  - A way to define the *attention weights*
- The i-th *output* is defined as:
  - $out_i = \sum_j Attn\_weight(i,j) f(in_j)$



- The attention mechanism needs to be able to figure out the attention weights on its own

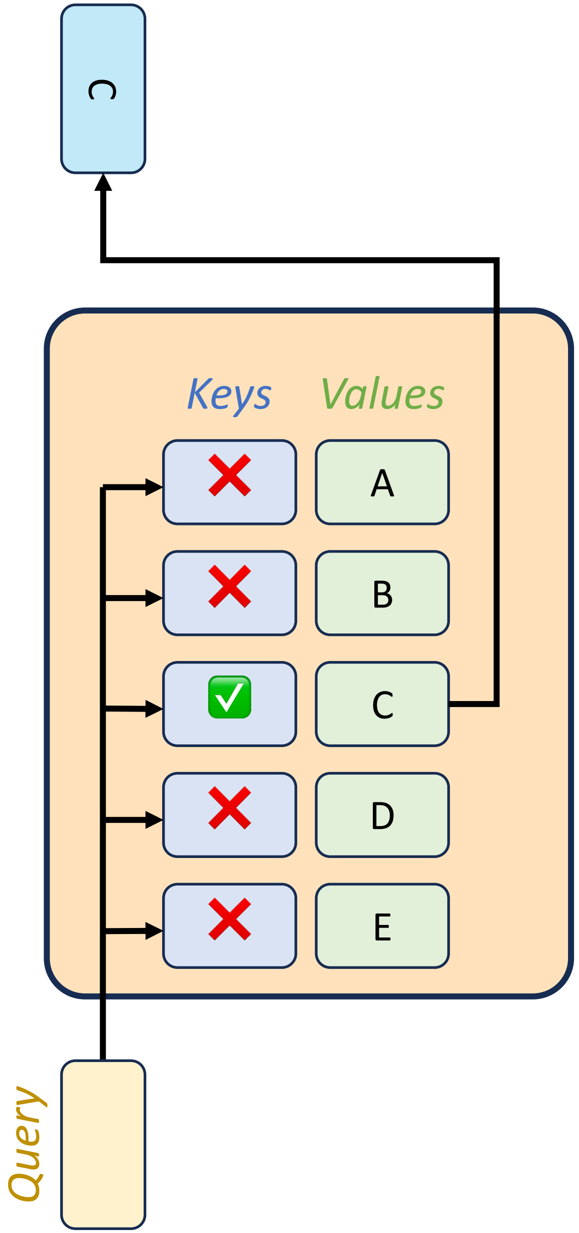
Classic (dictionary) lookup

- (A small digression…)
- In a *classic dictionary* (map, hash) we have *key*-*value* pairs and a *query*
  - The result is the *value* associated with the *key* that matches the *query*
- In this case, there is a single match, so we return the value for a single dictionary entry
- What if instead of a 0/1 match, we used a continuous match?



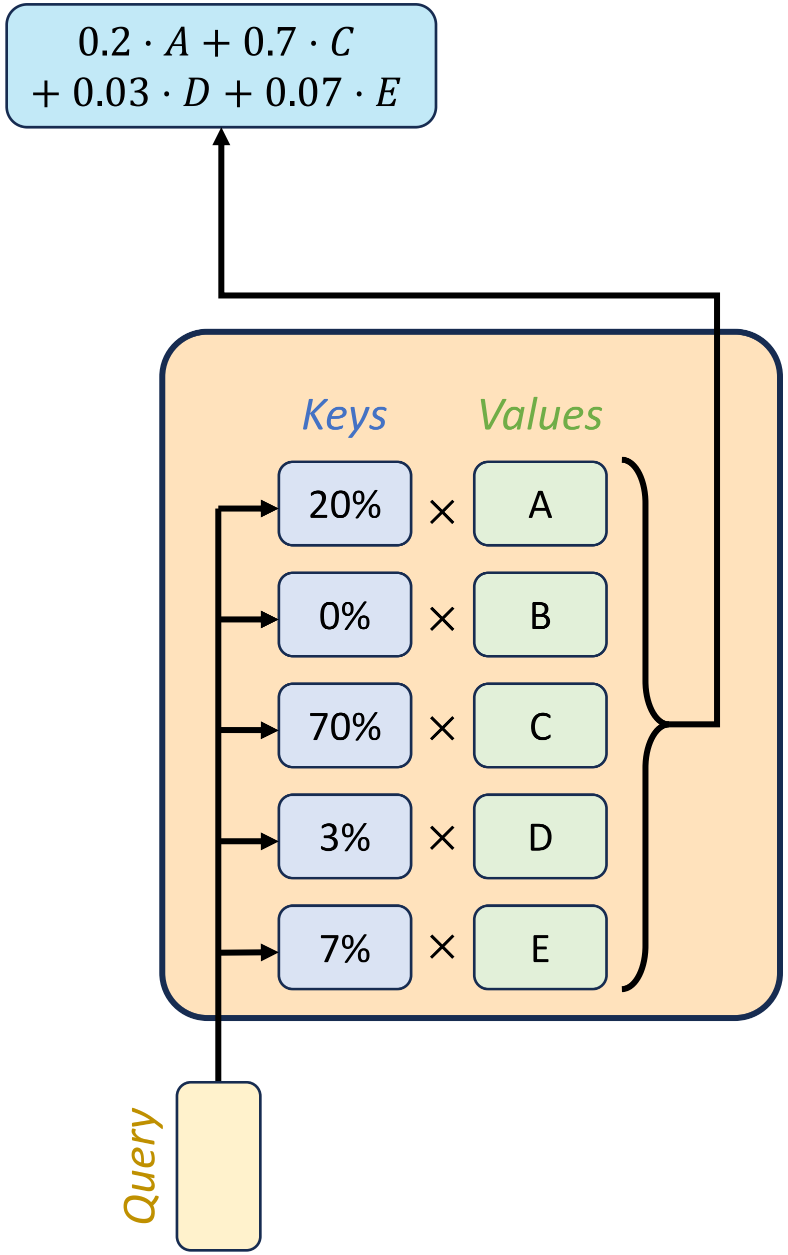
Classic (dictionary) lookup

- (A small digression…)
- In a *classic dictionary* (map, hash) we have *key*-*value* pairs and a *query*
    - The result is the *value* associated with the *key* that matches the *query*
- In this case, there is a single match, so we return the value for a single dictionary entry
- What if instead of a 0/1 match, we used a continuous match?



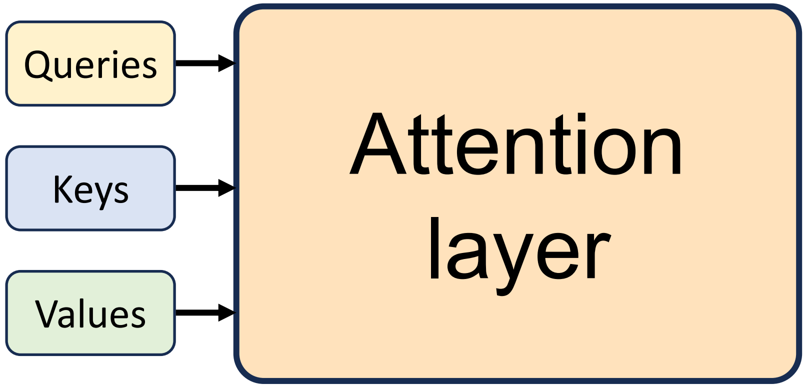
Dot-product attention ("weighted lookup")

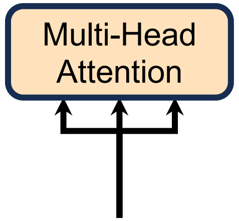
- We extend the "classic" lookup by:
  - Converting *key*, *values*, *queries* into vectors
  - Now the match is not 0/1 (match/no match), but rather *continuous*
    - E.g., dot-product between *query* and each *key* to quantify how similary each *query* is to each *key*
  - We weigh each *value* based on how similar the associated *key* is to the *query*



Attention as Keys/Values/Queries

- We can consider the attention as having 3 separate “inputs”:
    - *Queries*, i.e. the values to be looked up (we want 1 output for each query)
    - *Keys*, i.e. the values to be matched against the queries
    - *Values*, i.e. the values associated to each key
- (This explains the 3-inputs block in the transformers’ architecture)



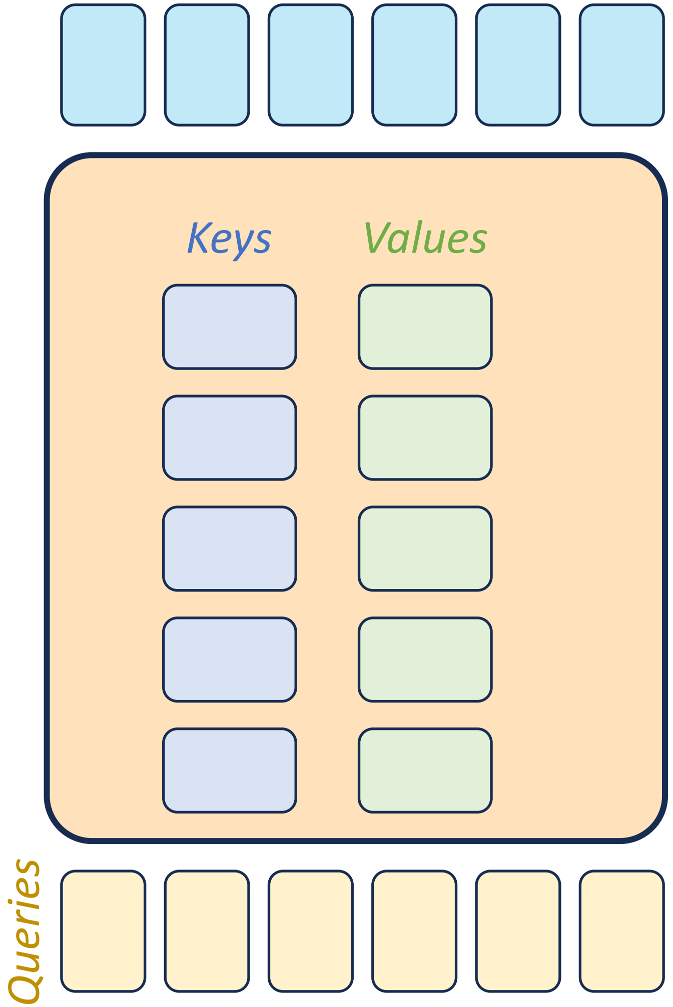


From dot product to weights

- We can use the dot product to quantify the similarity between a *query* and each *key*
- This provides un unbounded similarity
  - 0 if the vectors are orthogonal
  - No upper bound (depends on the magnitude of the vectors)
- When all dot products are computed between a *query* and all *keys*, we obtain K similarity values
  - With a softmax, we guarantee that the values are in [0,1] and sum to 1
  - (i.e., “fractions of contributions” to be applied to the sum of *values*)

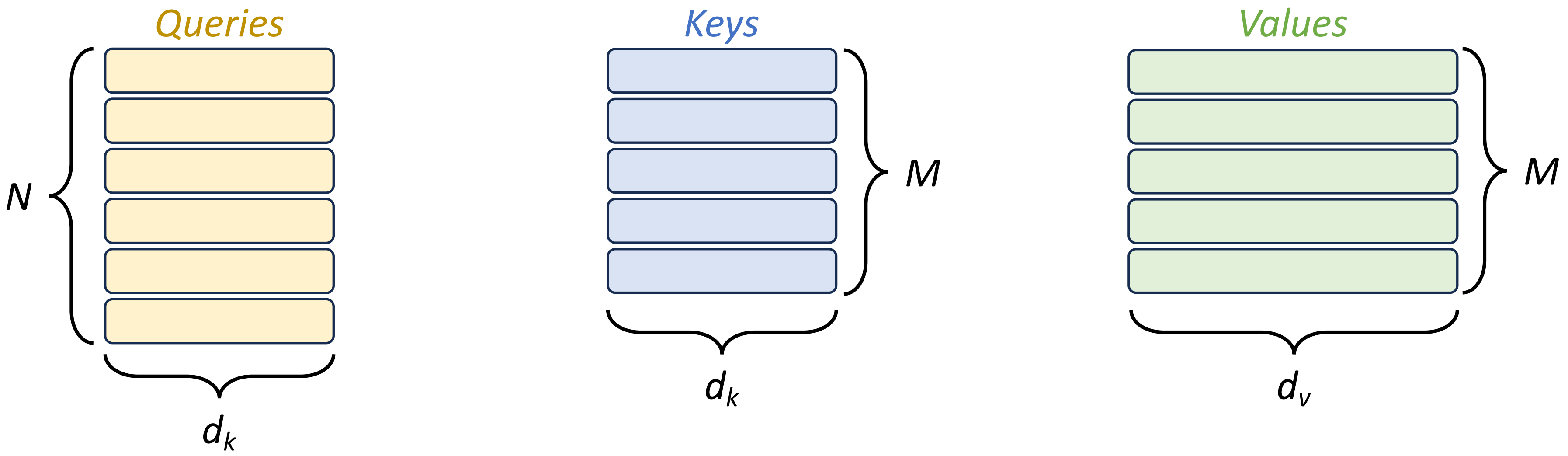
Packing everything into matrices

- We discussed the situation having a single query.
- Of course, we can have multiple queries, each treated in the same way
- Note that, for multiple queries, the keys and values will stay the same
  - (However, different queries will produce different weighted sums of the values)
- We can pack all queries, keys and values into three matrices, Q, K, V



Packing everything into matrices

- For convenience, we can summarize the set of all queries, keys and values as three matrices, *Q*, *K*, *V*.
    - *Q* contains ***N*** queries, each being a $\boldsymbol{d_k}$ dimensional vector
    - *K* contains ***M*** entries, each being a $\boldsymbol{d_k}$ dimensional vector
    - *V* contains ***M*** entries (one value for each key), each being a $\boldsymbol{d_v}$ dimensional vector

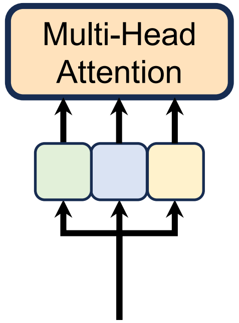


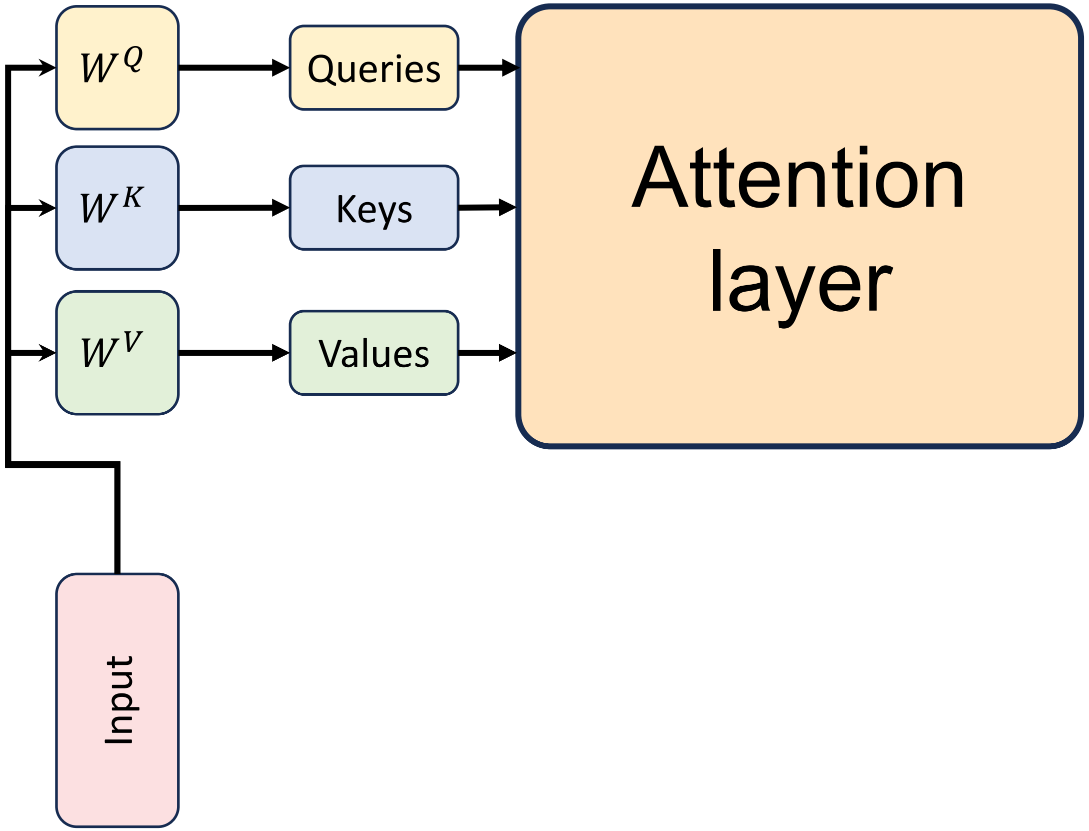
Scaled Dot-product attention

- In the original transformers, the final attention is defined as:
  - $Attention(Q, K, V) = softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)V$
- Notice the various steps:
  - QK^T efficiently computes the dot product between each query and each key. This produces an $N \times M$ matrix
  - $\sqrt{d_k}$ is a scaling factor that limits the growth of the variance of QK^T
    - You can easily verify that it scales the variance to 1, if we assume $Q, K \sim N(0,1)$
  - $softmax(\cdot)$ rescales the values so that each row QK^T sums to 1
    - In other words, the i^{th} row represents the weights to be used for the i^{th} query
  - Multiplying by V creates a weighted sum of entries in V, i.e. an $N \times d_v$ matrix

Where do Q, K, V come from?

- So far, we assumed Q, K, V given
- In practice, the Attention input(s) are transformed with a linear layer (matrix) to produce the values.
  - W^Q, W^K, W^V
- (Note: there will be two sources of inputs for cross-attention, as shown later)





Attention example

$$Attention(Q, K, V) = softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

$$W^Q \quad \begin{bmatrix} 0.9 & 0.9 \\ 0.5 & 0.3 \end{bmatrix}$$

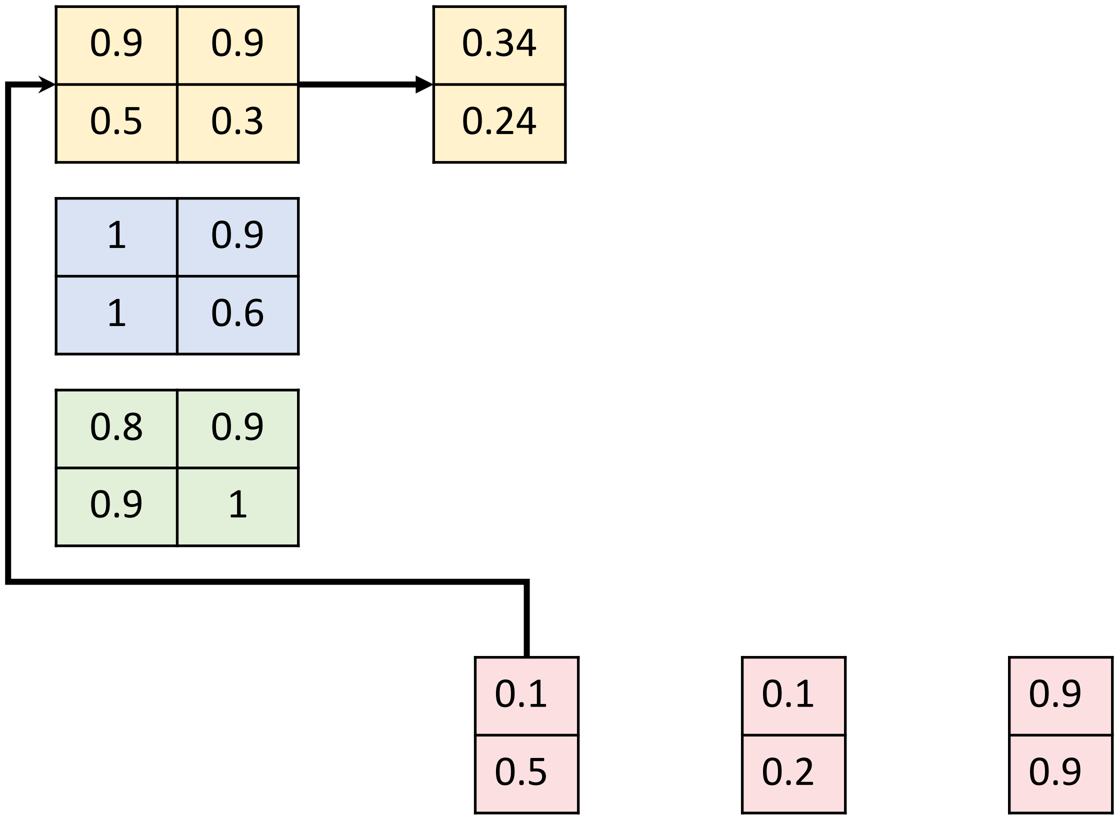
$$W^K \quad \begin{bmatrix} 1 & 0.9 \\ 1 & 0.6 \end{bmatrix}$$

$$W^V \quad \begin{bmatrix} 0.8 & 0.9 \\ 0.9 & 1 \end{bmatrix}$$

$$\underbrace{\begin{bmatrix} 0.1 \\ 0.5 \end{bmatrix} \quad \begin{bmatrix} 0.1 \\ 0.2 \end{bmatrix} \quad \begin{bmatrix} 0.9 \\ 0.9 \end{bmatrix}}_{\text{Input tokens}}$$

Attention example

$$Attention(Q, K, V) = softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$



0.9	0.9
0.5	0.3

0.34
0.24

1	0.9
1	0.6

0.8	0.9
0.9	1

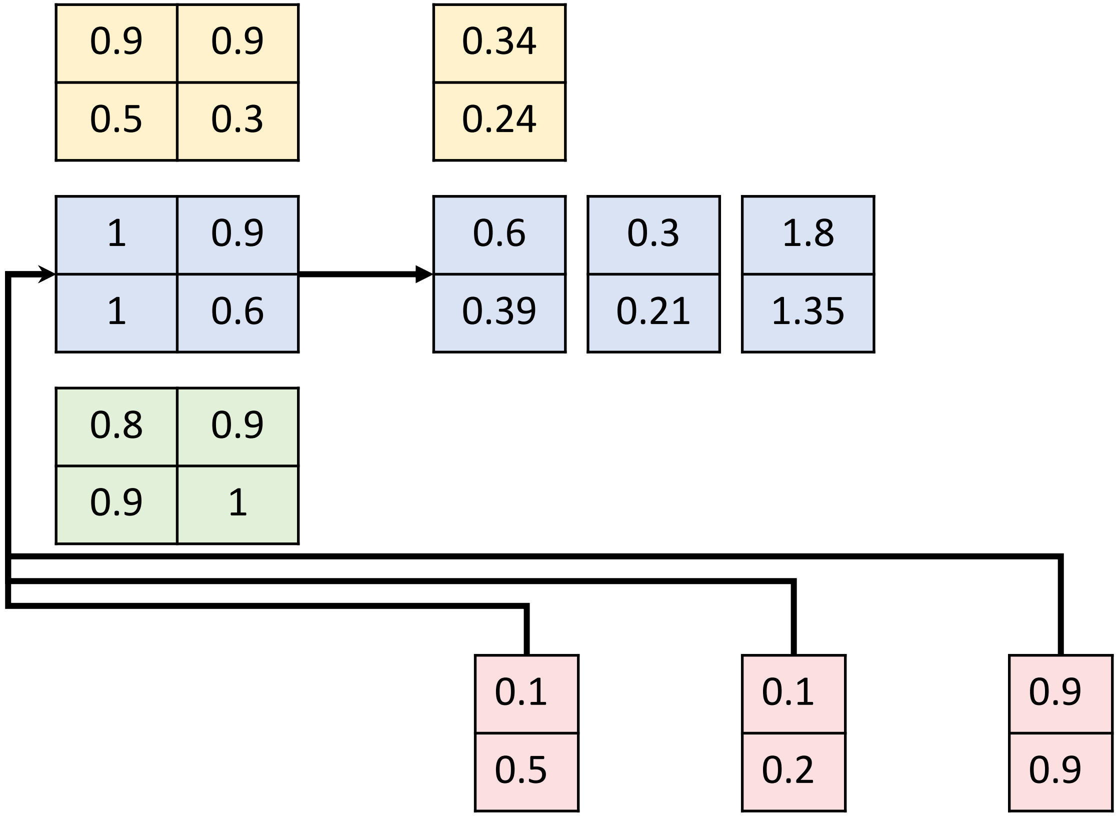
0.1
0.5

0.1
0.2

0.9
0.9

Attention example

$$Attention(Q, K, V) = softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$



0.9	0.9
0.5	0.3

0.34
0.24

1	0.9
1	0.6

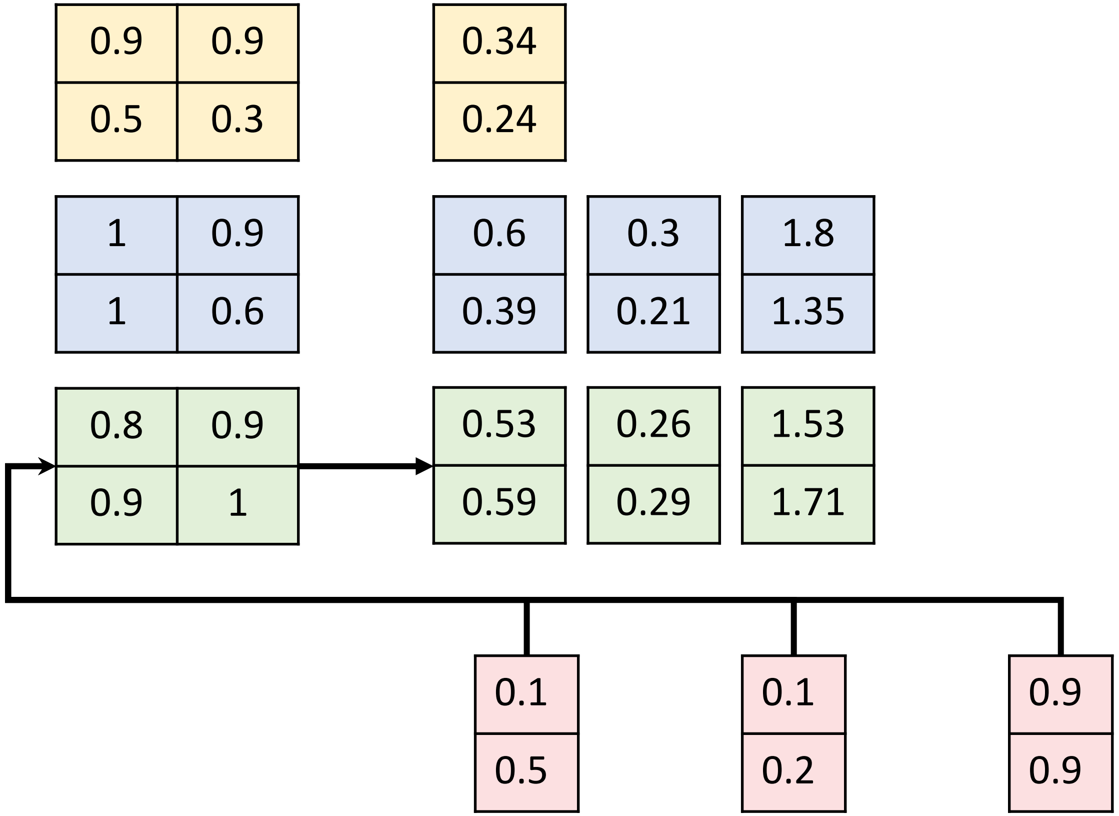
0.6	0.3	1.8
0.39	0.21	1.35

0.8	0.9
0.9	1

0.1	0.1	0.9
0.5	0.2	0.9

Attention example

$$Attention(Q, K, V) = softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$



$$\begin{bmatrix} 0.9 & 0.9 \\ 0.5 & 0.3 \end{bmatrix} \quad \begin{bmatrix} 0.34 \\ 0.24 \end{bmatrix}$$

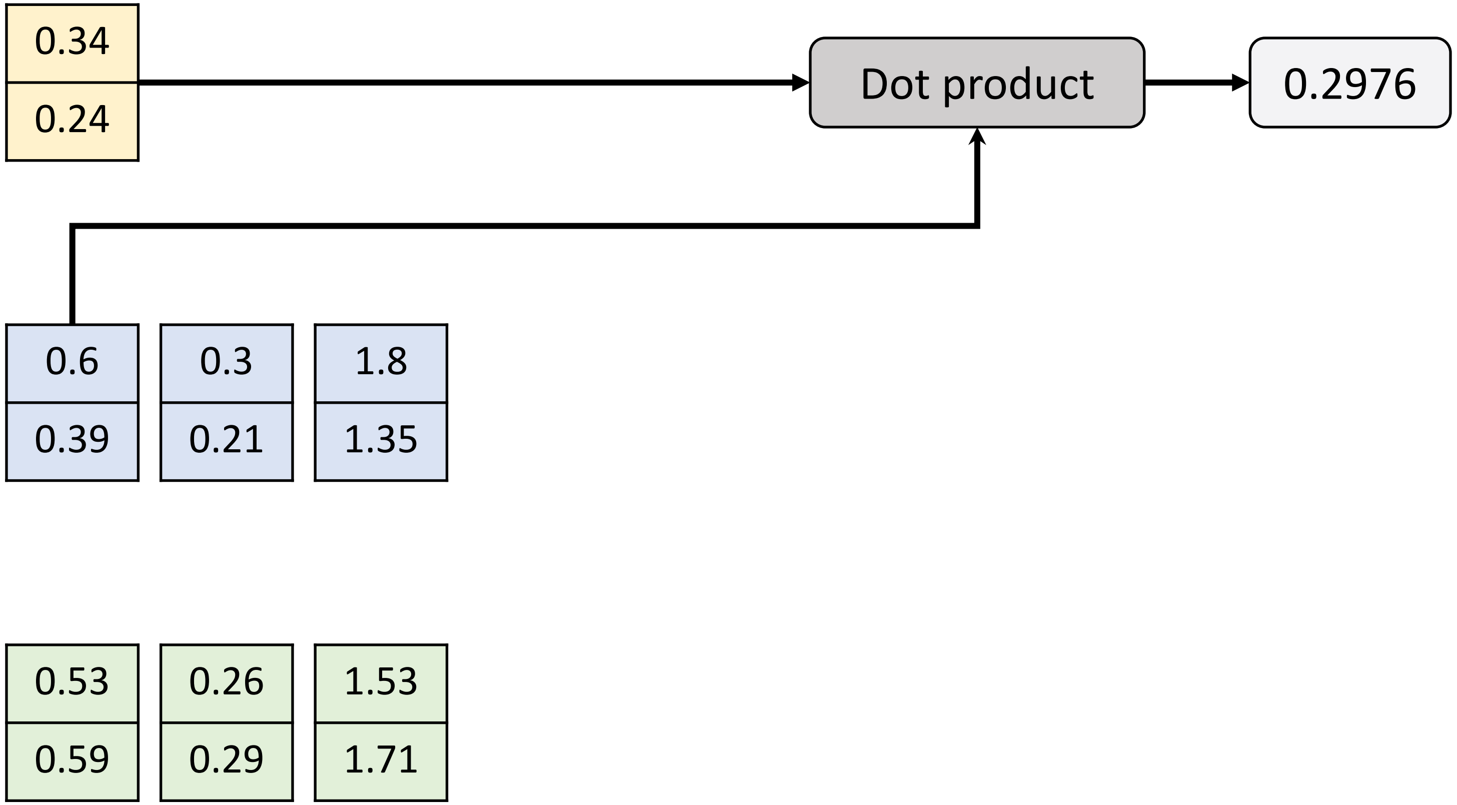
$$\begin{bmatrix} 1 & 0.9 \\ 1 & 0.6 \end{bmatrix} \quad \begin{bmatrix} 0.6 \\ 0.39 \end{bmatrix} \begin{bmatrix} 0.3 \\ 0.21 \end{bmatrix} \begin{bmatrix} 1.8 \\ 1.35 \end{bmatrix}$$

$$\begin{bmatrix} 0.8 & 0.9 \\ 0.9 & 1 \end{bmatrix} \rightarrow \begin{bmatrix} 0.53 \\ 0.59 \end{bmatrix} \begin{bmatrix} 0.26 \\ 0.29 \end{bmatrix} \begin{bmatrix} 1.53 \\ 1.71 \end{bmatrix}$$

$$\begin{bmatrix} 0.1 \\ 0.5 \end{bmatrix} \quad \begin{bmatrix} 0.1 \\ 0.2 \end{bmatrix} \quad \begin{bmatrix} 0.9 \\ 0.9 \end{bmatrix}$$

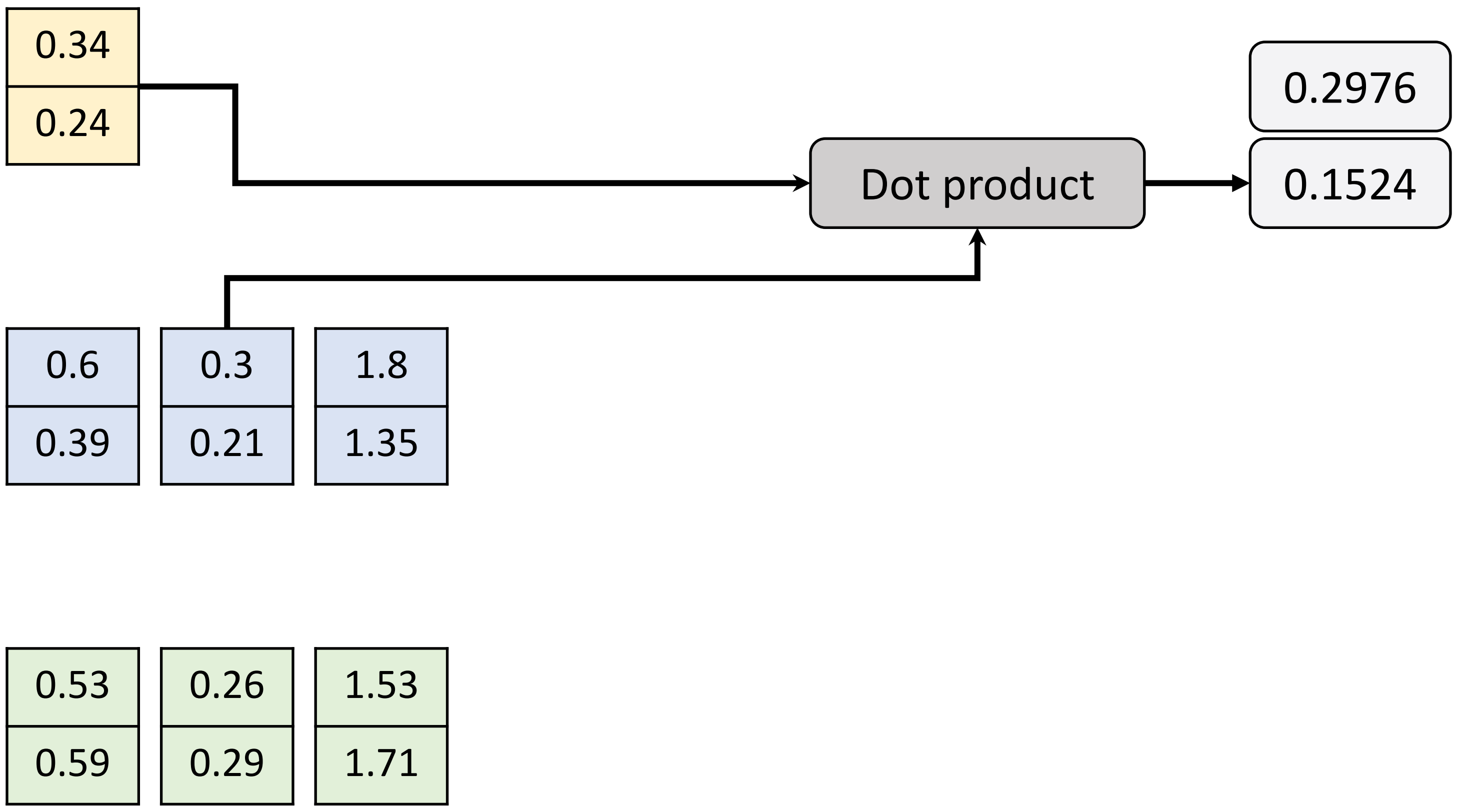
Attention example

$$Attention(Q, K, V) = softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$



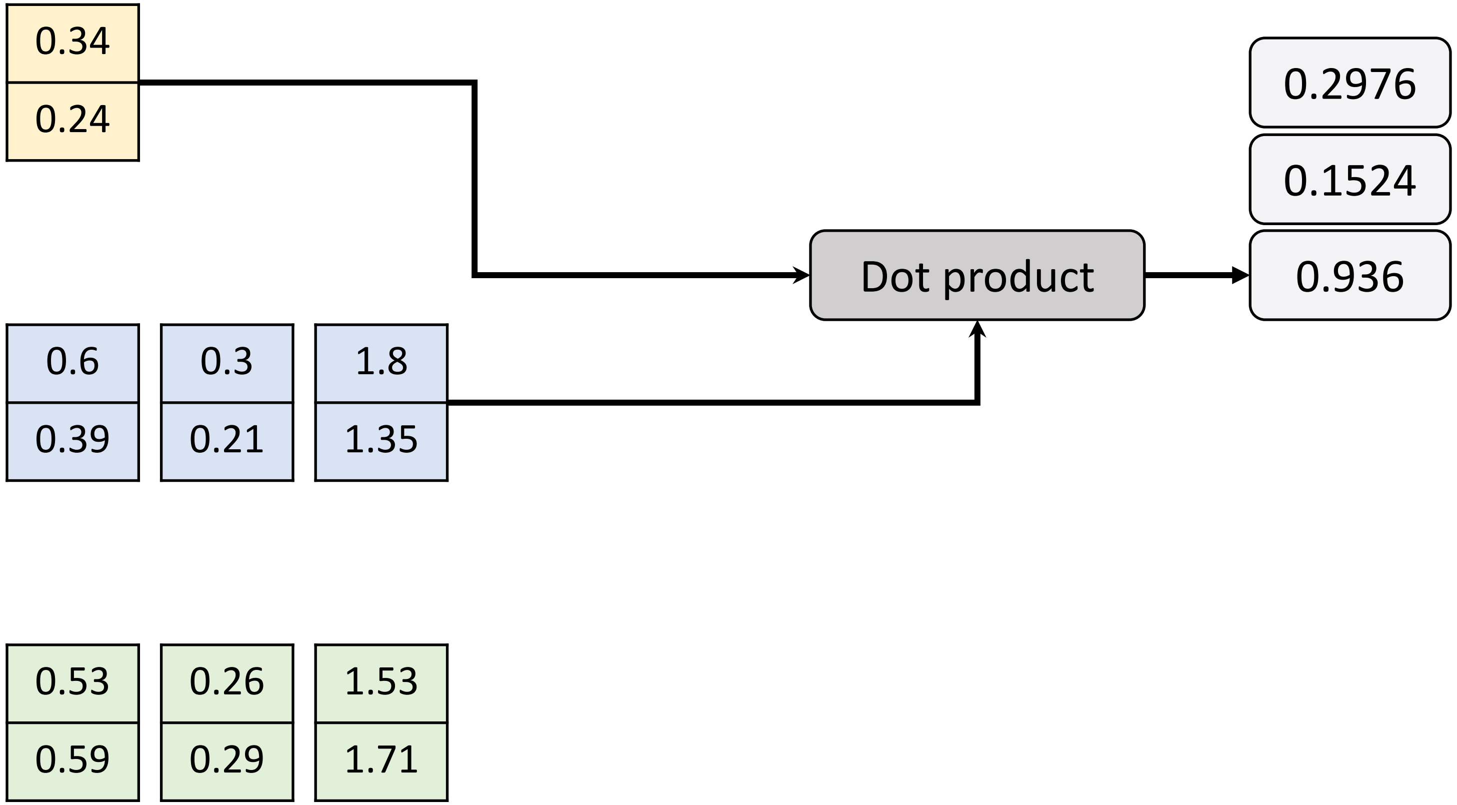
Attention example

$$Attention(Q, K, V) = softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$



Attention example

$$Attention(Q,K,V) = softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$



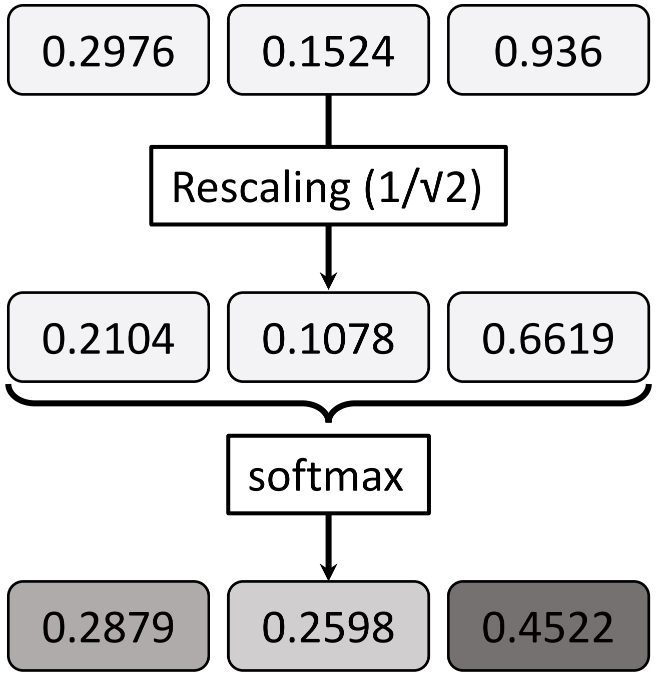
Attention example

$$Attention(Q,K,V) = softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

0.34
0.24

0.6	0.3	1.8
0.39	0.21	1.35

0.53	0.26	1.53
0.59	0.29	1.71



0.2976 0.1524 0.936

Rescaling (1/√2)

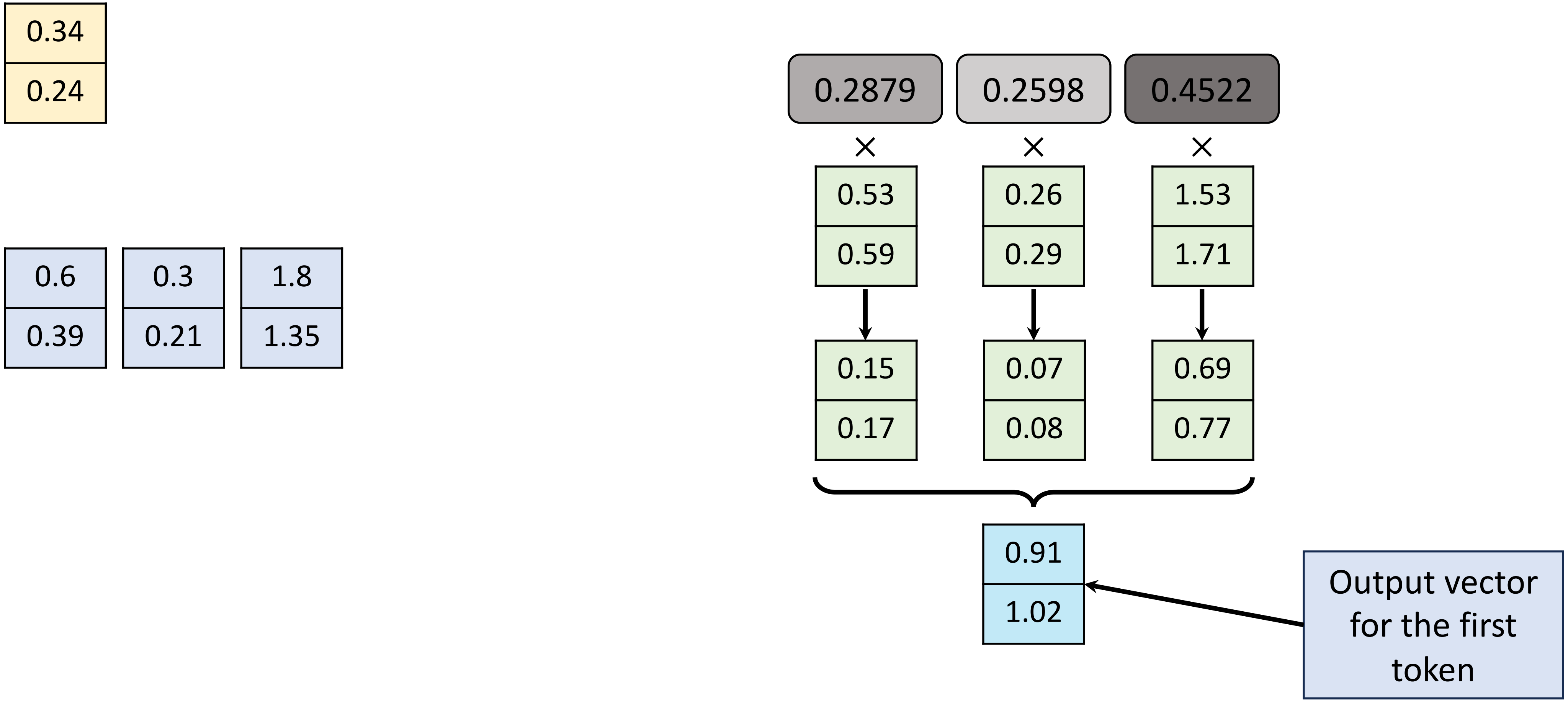
0.2104 0.1078 0.6619

softmax

0.2879 0.2598 0.4522

Attention example

$$Attention(Q,K,V) = softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$



0.34
0.24

0.6 | 0.3 | 1.8
0.39 | 0.21 | 1.35

0.2879 | 0.2598 | 0.4522

× | × | ×

0.53 | 0.26 | 1.53
0.59 | 0.29 | 1.71

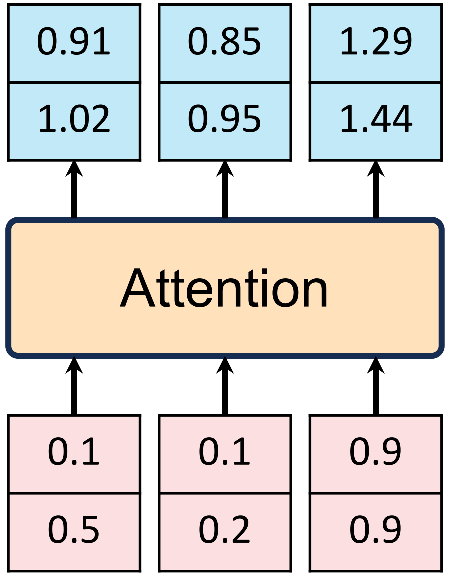
0.15 | 0.07 | 0.69
0.17 | 0.08 | 0.77

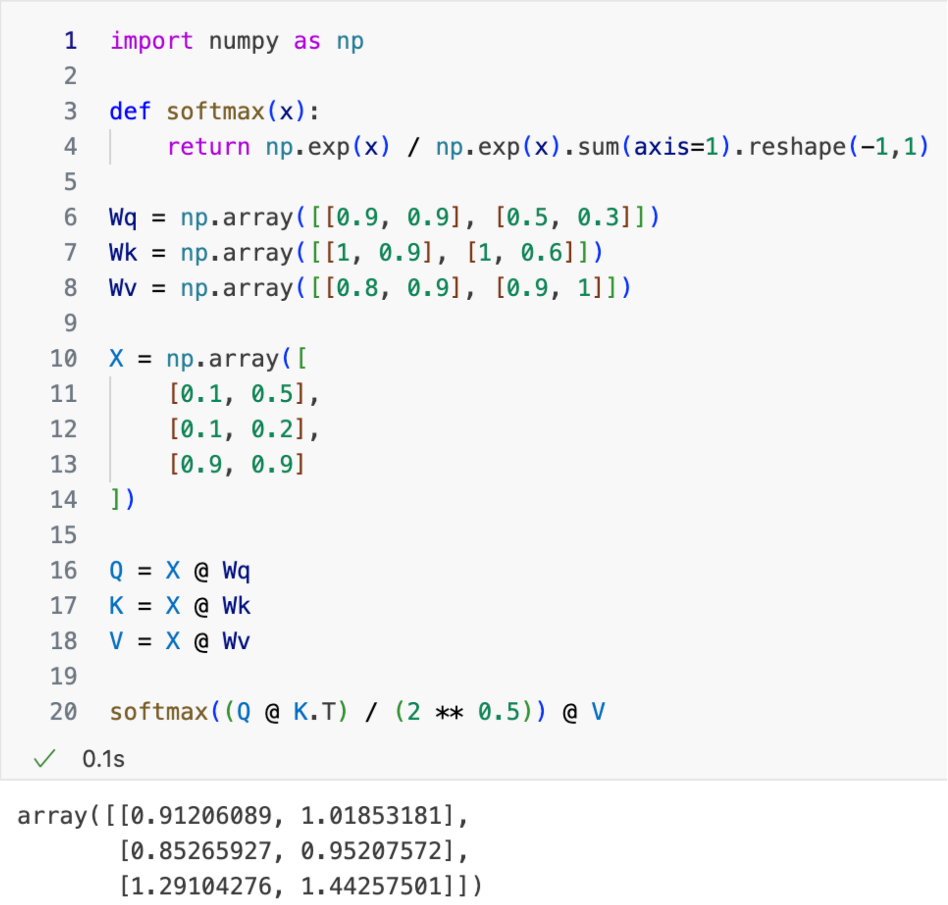
0.91
1.02

Output vector for the first token

Attention example

- We can repeat the above process for the second and third inputs
    - (or let Python do the work)



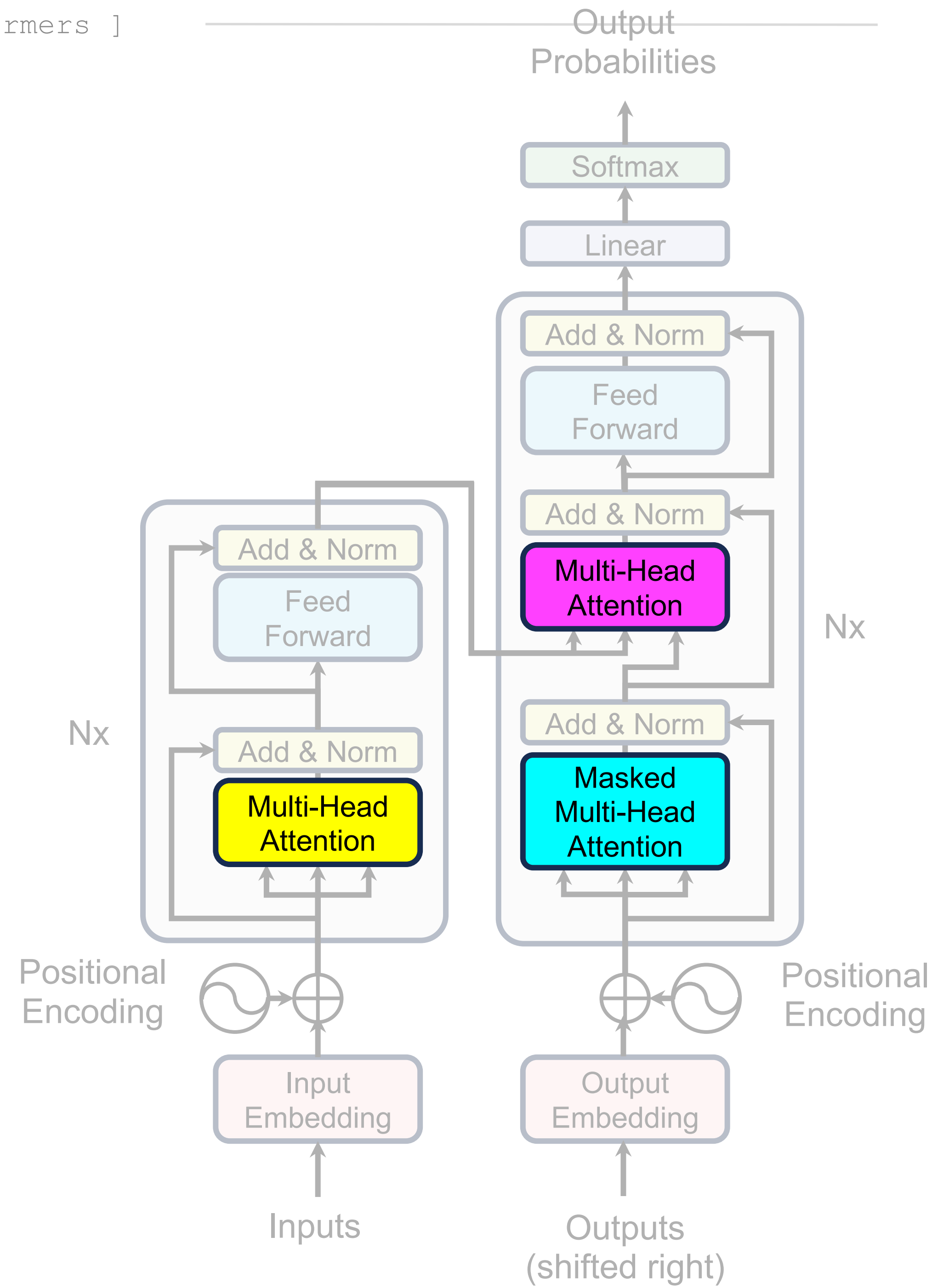


```python
import numpy as np

def softmax(x):
    return np.exp(x) / np.exp(x).sum(axis=1).reshape(-1,1)

Wq = np.array([[0.9, 0.9], [0.5, 0.3]])
Wk = np.array([[1, 0.9], [1, 0.6]])
Wv = np.array([[0.8, 0.9], [0.9, 1]])

X = np.array([
    [0.1, 0.5],
    [0.1, 0.2],
    [0.9, 0.9]
])

Q = X @ Wq
K = X @ Wk
V = X @ Wv

softmax((Q @ K.T) / (2 ** 0.5)) @ V
```

```
array([[0.91206089, 1.01853181],
       [0.85265927, 0.95207572],
       [1.29104276, 1.44257501]])
```

Types of attention

- Attention is used in three situations in the classic transformer architecture
  - Encoder self-attention
  - Decoder (masked) self-attention
  - Encoder-decoder cross-attention



Encoder self-attention

- This is the “classic” attention mechanism, as described in the previous slides

- Each token ”sees” (*attends*) all other tokens in the input sequence

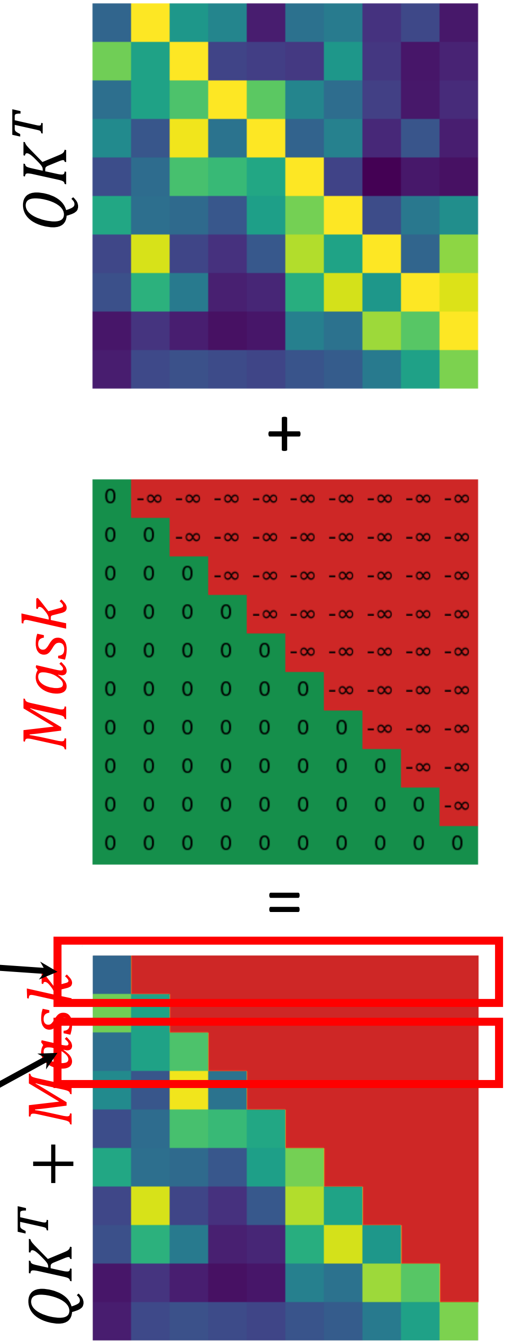
- The same input sequence generates query, keys, values
  - Hence the name, *self*-attention

Decoder (masked) self-attention

- This is one of the attention blocks used by the decoder
- Used on the output tokens
- We introduce the property of *causality*
  - “Each token can only see the past”
    - i.e., previously generated tokens
  - The model cannot know the future yet!

Decoder (masked) self-attention

- This constraint is applied by ”masking” all invalid attention weights
  - Values in the attention matrix are set to $-\infty$ (to have 0 attention)
- $Masked\ Attention(Q, K, V) = softmax\left(\frac{QK^T}{\sqrt{d_k}} + Mask\right)V$



0	-∞	-∞	-∞	-∞	-∞	-∞	-∞	-∞	-∞
0	0	-∞	-∞	-∞	-∞	-∞	-∞	-∞	-∞
0	0	0	-∞	-∞	-∞	-∞	-∞	-∞	-∞
0	0	0	0	-∞	-∞	-∞	-∞	-∞	-∞
0	0	0	0	0	-∞	-∞	-∞	-∞	-∞
0	0	0	0	0	0	-∞	-∞	-∞	-∞
0	0	0	0	0	0	0	-∞	-∞	-∞
0	0	0	0	0	0	0	0	-∞	-∞
0	0	0	0	0	0	0	0	0	-∞
0	0	0	0	0	0	0	0	0	0

The first token can only pay attention to itself

The third token can attend the 1st and 2nd token, plus itself

Note
We could have just computed the relevant dot products. However, matrix multiplications are particularly efficient (esp. on GPU), so it’s “okay” to compute all dot products and mask what’s not needed

Encoder-decoder cross-attention

- The encoder-decoder attention is used by the decoder to receive information from the encoder (input sequence)
- Keys and values come from the encoder’s output sequence
- Queries come from the decoder’s sequence
- Note: The number of elements in keys/values can be different from the number of queries
  - This is not a problem, based on how we define attention
- Each decoder token can attend to all encoder’s tokens
  - (i.e., no need for masking!)

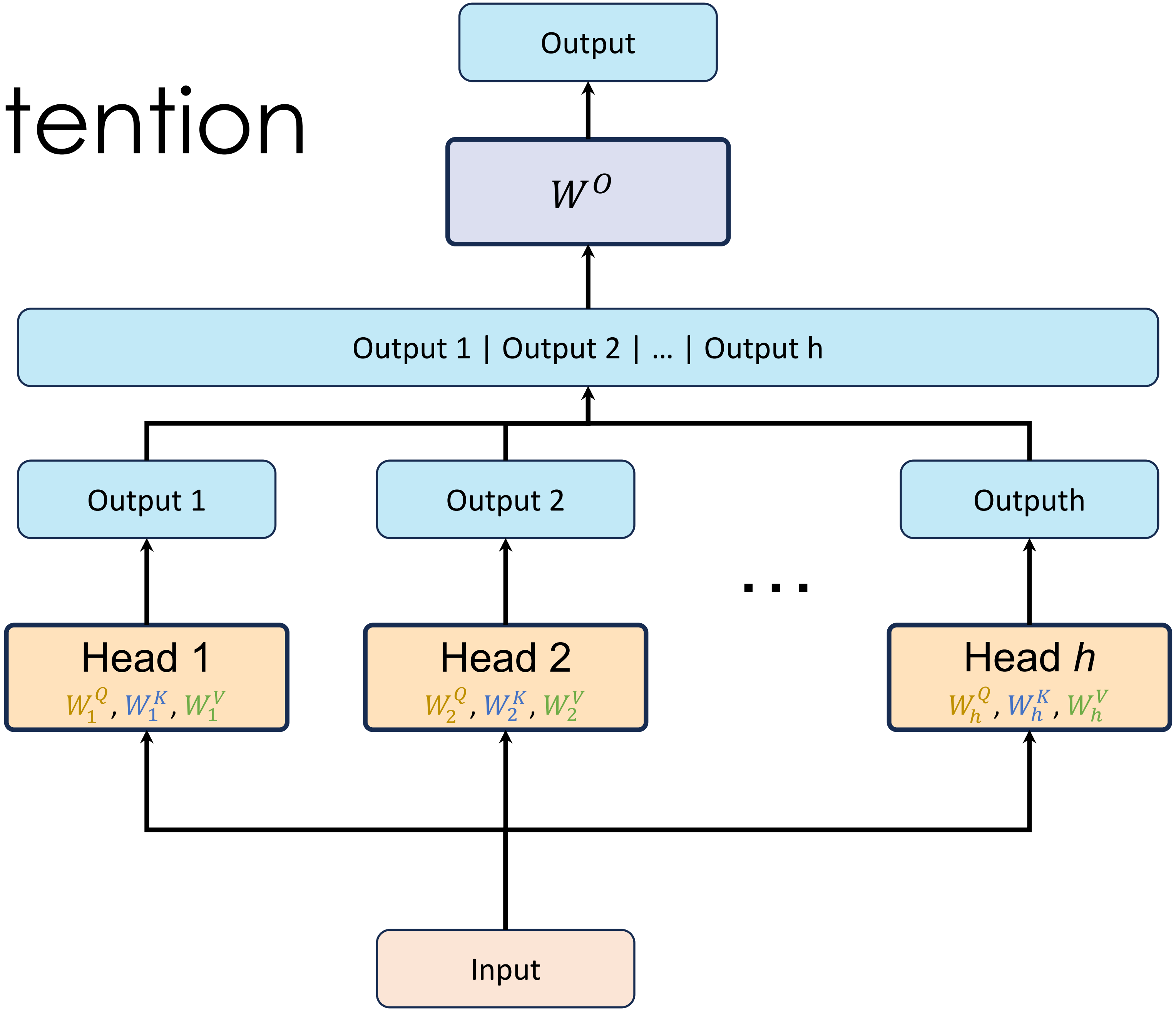
Multi-head attention

- The attention block discussed this far is called an “attention head”
  - (with its own W^Q, W^K, W^V)

- Since the attention may need to focus on different aspects in different contexts, we generally adopt multiple attention heads in parallel
  - Each attention head has its own W^Q, W^K, W^V and produces its own output

- The final attention output is the concatenation of the various heads’ outputs
  - Passed through a linear layer to go back to the desired vector size

Multi-head attention



Some additional details

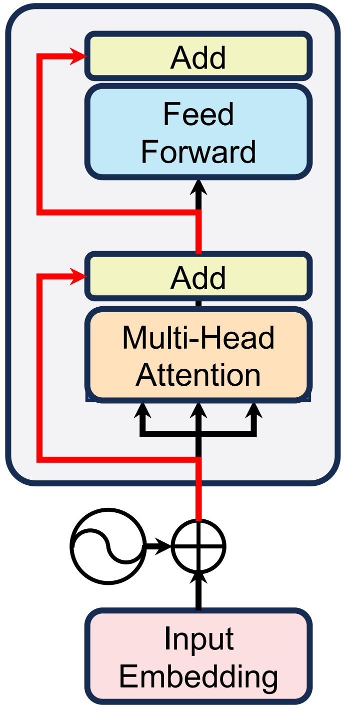
Residual connections

- Residual connections are a technique used to improve gradient flow in backpropagation

- The gradient “skips” part of the network when being backpropagated
  - Indeed, these are sometimes called skip connections
  - $out = f(x) + x$

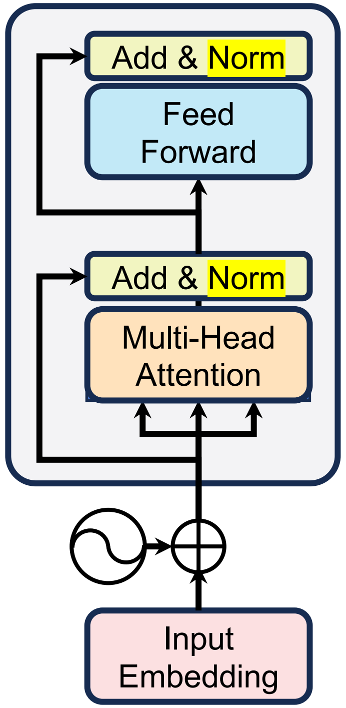
- Allows building deeper networks
  - It takes longer for gradients to “vanish”
  - (Remember the vanishing gradient problems)

- Introduced in ResNet (“Residual Networks”)
  - Allowed building deeper model & obtain better performance



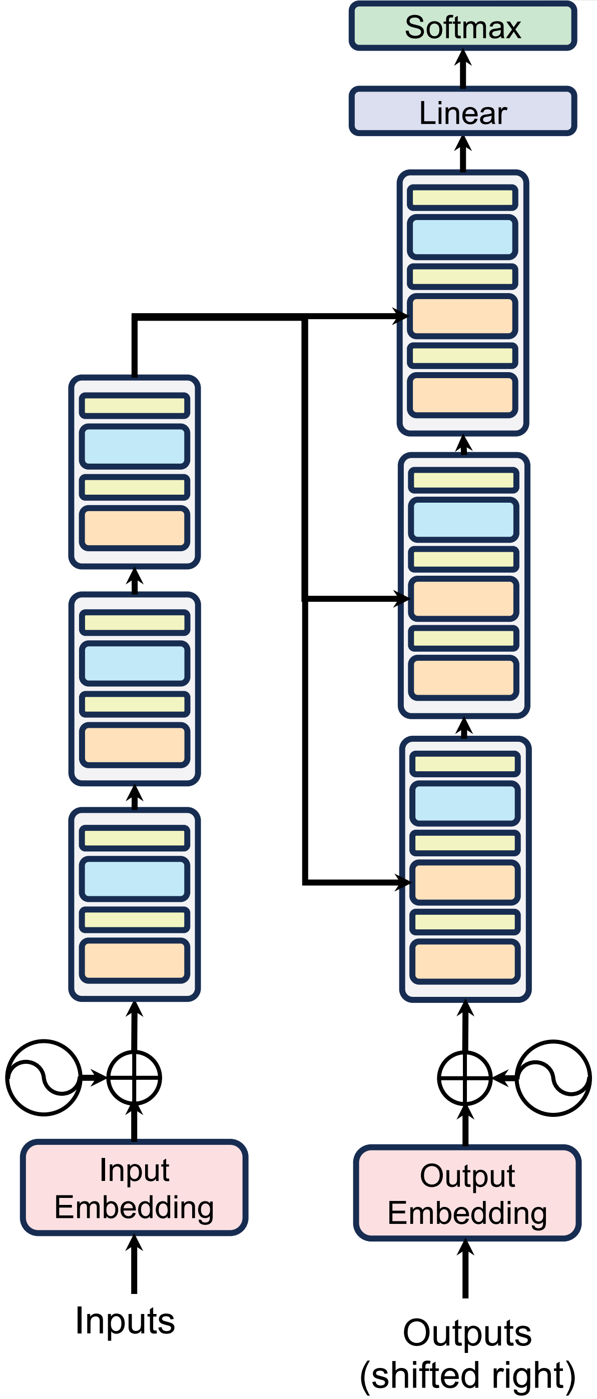
Layer Normalization

- The “Norm” in ”Add & Norm” is a Layer Normalization
- Layer normalization consists in normalizing each sample across all dimensions
- For an unnormalized output $[\,x_1, x_2, \ldots, x_d]$, the LayerNorm output is:
  - $y = \frac{x - E[x]}{\sqrt{Var[x] + \epsilon}} \cdot \gamma + \beta$
- Where $E[x]$ and $Var[x]$ are computed for the single sample, ϵ is a constant added for numerical stability
- γ and β are learnable parameters
- Layer Norm allows rescaling each sample’s values to a consistent range of values
  - This leads to faster convergence, and a more stable training



Actual connections

- The encoder/decoder layers are typically stacked into multiple layers
- For instance, the original transformer architecture repeats the encoder & decoder blocks 6 times
- (On the left, an instance of N = 3)

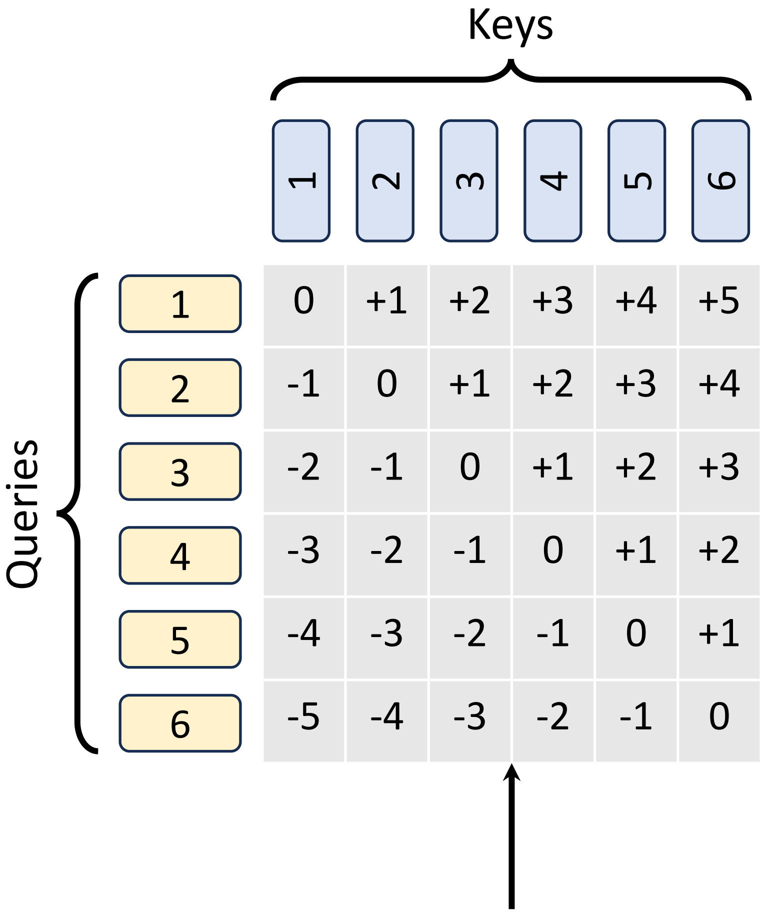


Relative positional embeddings

- AIAYN uses sinusoidal *absolute* positional encoding
  - sinusoidal = fixed, not learned (we already discussed that PE can also be learned)
  - *absolute* = each position in the sequence has a fixed positional vector

- *Relative* positional embeddings are sometimes used with self-attention
  - Positional information no longer introduced in the input vectors, but at "attention time"

- Positional information is now *relative to the key-query distance* in the sequence
  - We no longer encode 1st token of the sequence, 2nd token of the sequence, …
  - But, 2 tokens before the query, 1 token before the query, 0, 1 token after the query, +2, …



	1	2	3	4	5	6
1	0	+1	+2	+3	+4	+5
2	-1	0	+1	+2	+3	+4
3	-2	-1	0	+1	+2	+3
4	-3	-2	-1	0	+1	+2
5	-4	-3	-2	-1	0	+1
6	-5	-4	-3	-2	-1	0

Relative position of keys to queries

Relative positional embeddings

$$Attention(Q,K,V) = softmax\left(\frac{QK^T + S_{rel}}{\sqrt{d_k}}\right)V$$

- This information is encoded as a learned matrix S_{rel}.
- Note that this generally only makes sense for self-attention
  - In cross-attention, it’s harder to define a “relative position” of query/keys

- However, most LMs now use absolute, learned positional embeddings or rotary embeddings

Huang, Cheng-Zhi Anna, et al. "Music transformer." *arXiv preprint arXiv:1809.04281* (2018). https://arxiv.org/pdf/1809.04281

Advantages of transformers

- *Parallelization*
  - Transformers can process all tokens simultaneously
    - No dependencies on previous states
  - This processing can be parallelized!
  - (Remember, we use *teacher forcing* to “know the future” – no need to wait for autoregressive generation)
- *Long-range relationships*
  - With attention, Transformers can see the entire sequence at all times, and choose what to pay attention to
  - (Note: this implies that each token looks at all tokens, this is N^2 and represents one of the current limitations of Transformers!)
- *Better memory*
  - Since there is no sequential processing, forgetting issues generally do not occur

Encoder-decoder architecture

T5 (Text-to-Text Transfer Transformer)

- Vanilla encoder-decoder transformer

- *Single framework* for multiple tasks

- Task *prefixes* to *condition* the decoder's output
  - (instead of using different architectures)

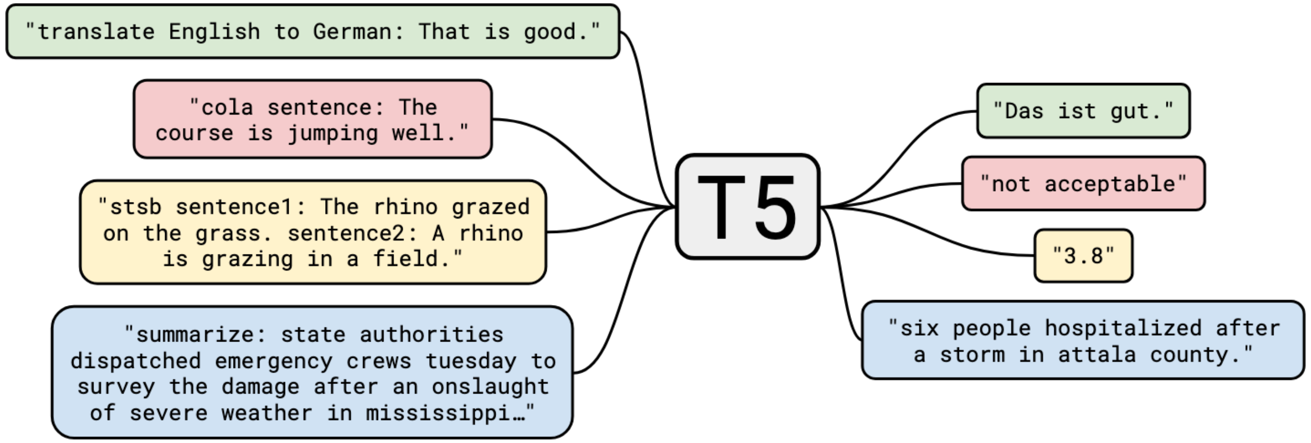
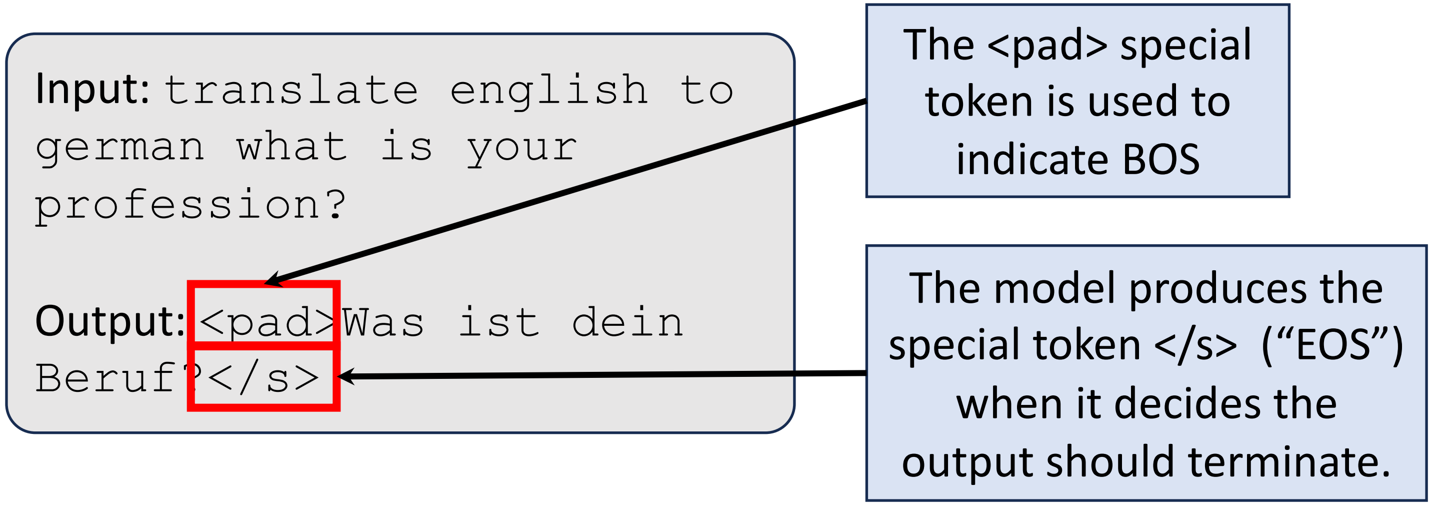

"translate English to German: That is good."
"cola sentence: The course is jumping well."
"stsb sentence1: The rhino grazed on the grass. sentence2: A rhino is grazing in a field."
"summarize: state authorities dispatched emergency crews tuesday to survey the damage after an onslaught of severe weather in mississippi…"
T5
"Das ist gut."
"not acceptable"
"3.8"
"six people hospitalized after a storm in attala county."

Figure 1: A diagram of our text-to-text framework. Every task we consider—including translation, question answering, and classification—is cast as feeding our model text as input and training it to generate some target text. This allows us to use the same model, loss function, hyperparameters, etc. across our diverse set of tasks. It also provides a standard testbed for the methods included in our empirical survey. "T5" refers to our model, which we dub the "**T**ext-**t**o-**T**ext **T**ransfer **T**ransformer".

Raffel, Colin, et al. "Exploring the limits of transfer learning with a unified text-to-text transformer." *Journal of machine learning research* 21.140 (2020): 1-67. https://arxiv.org/pdf/1910.10683

Single model, multiple tasks!

- The input sequence encodes
  - the task to be carried out, and
  - the actual input sequence!
- No need to build different models for different tasks, we just *condition* the input by specifying the task we'd like to have performed



Input: `translate english to german what is your profession?`

Output: `<pad>Was ist dein Beruf?</s>`

The <pad> special token is used to indicate BOS

The model produces the special token </s> ("EOS") when it decides the output should terminate.

Input: `translate english to german What is your profession?`

Output: `Was ist Ihr Beruf?`

Not instruction-tuned, yet!

- While it may resemble the behavior of instruction-tuned models (“chat-like conversation”), that’s not it!
- T5 is only tuned on specific tasks
- The model learns to recognize those tasks and addresses them
- No strong generalization capabilities to new tasks
    - Or even different formulations

Input: `can you translate from English to German, What is your profession?`

Output: `Was ist Ihr Beruf?`

Input: `can you translate from English to German the following sentence? What is your profession?`

Output: `<unk> <unk>…`

Input: `English: what is your profession? German:`

Output: `Deutsch: Deutsch: Deutsch:…`

Input: `compute: 2+2 =`

Output: `:2+2+2+2+2 +2+2+2+2+…`

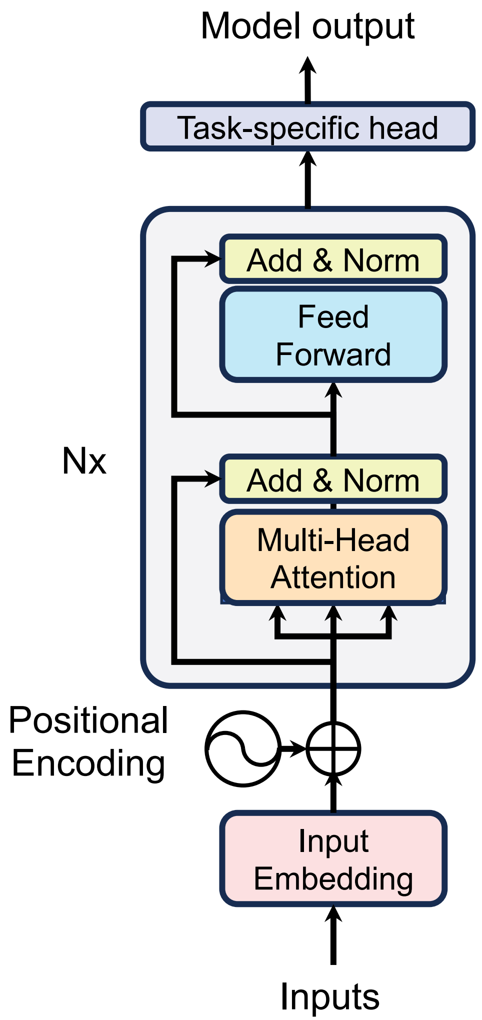
Beyond encoder-decoder

Encoder-only architecture

- Encoder-only architectures primarily focus on understanding and encoding input data “without generating output”

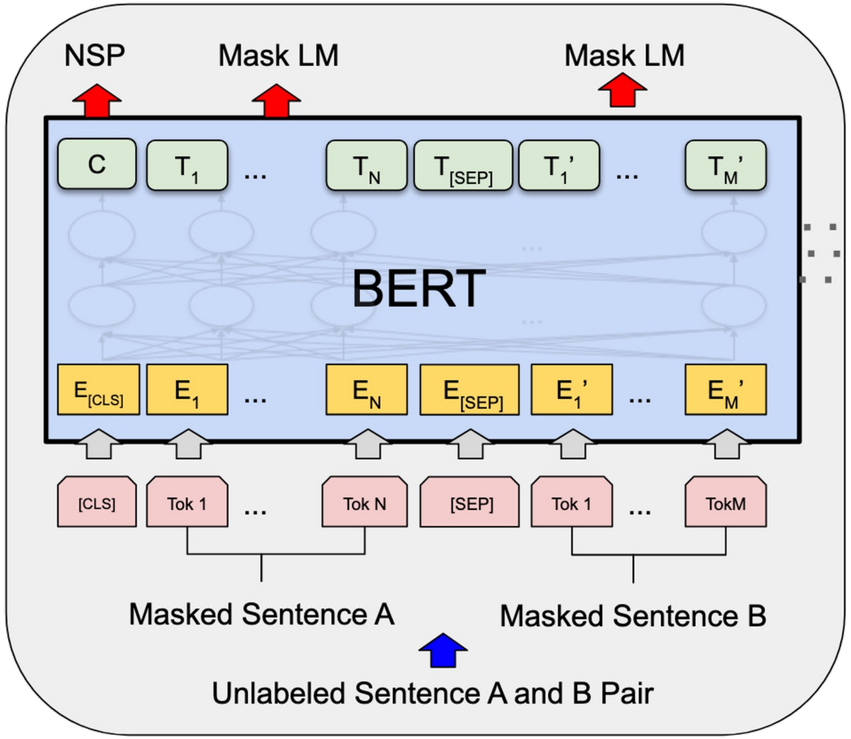
- The “decoder” part of the architecture is removed
- The encoder is trained to solve some tasks

- Encoder-only models are generally used to solve downstream tasks:
  - Text classification, Named Entity Recognition, sentiment analysis, etc.



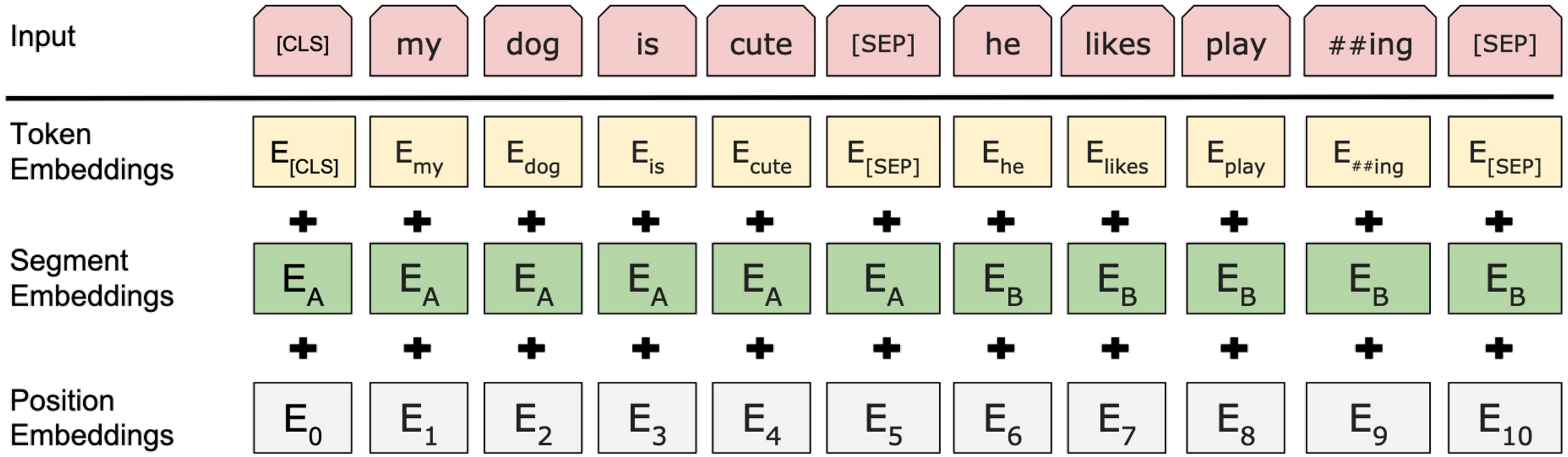
BERT

- BERT (**B**idirectional **E**ncoder **R**epresentations from **T**ransformers) is arguably the most famous encoder-only transformers
- It is pre-trained on two *self-supervised* tasks
  - Masked LM
  - Next Sentence Prediction
- Extends to new tasks with some fine-tuning
- It uses *bidirectional* attention (encoder self-attention)
  - All tokens can attend to all other tokens
  - Requires being careful with the task definition!



Devlin, Jacob. "Bert: Pre-training of deep bidirectional transformers for language understanding." *arXiv preprint arXiv:1810.04805* (2018). https://arxiv.org/pdf/1810.04805

Input encoding

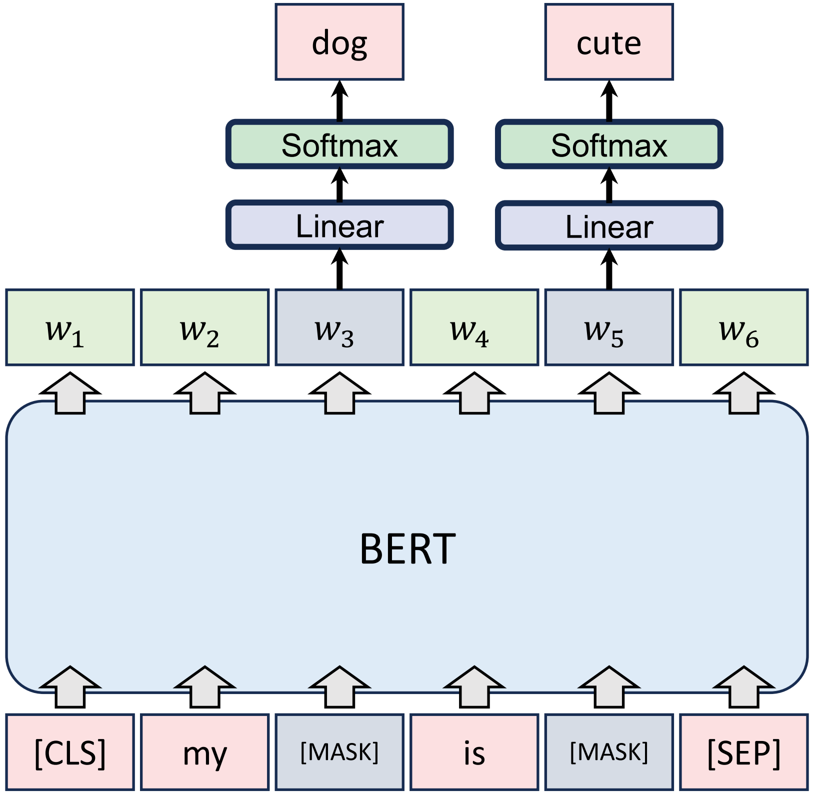


Input	[CLS]	my	dog	is	cute	[SEP]	he	likes	play	##ing	[SEP]
Token Embeddings	$E_{[CLS]}$	E_{my}	E_{dog}	E_{is}	E_{cute}	$E_{[SEP]}$	E_{he}	E_{likes}	E_{play}	$E_{\#\#ing}$	$E_{[SEP]}$
	+	+	+	+	+	+	+	+	+	+	+
Segment Embeddings	E_A	E_A	E_A	E_A	E_A	E_A	E_B	E_B	E_B	E_B	E_B
	+	+	+	+	+	+	+	+	+	+	+
Position Embeddings	E_0	E_1	E_2	E_3	E_4	E_5	E_6	E_7	E_8	E_9	E_{10}

- BERT supports pairs of input sentences (A, B)
  - Useful in some tasks, e.g. Next Sentence Prediction (see next slides)
- Sentences are separated via the [SEP] special token
- Sentences begin with the [CLS] token
  - The output vector for this token is used to address downstream tasks
- *Segment embeddings* are learned and used to enrich each token with information on its sentence (A or B)

Masked LM

- The model can attend to all tokens, so “next token prediction” tasks are meaningless
- In this “Masked LM” task, random parts of the input are hidden
  - (~15% in the original work)
- The output of BERT is used to reconstruct the masked tokens

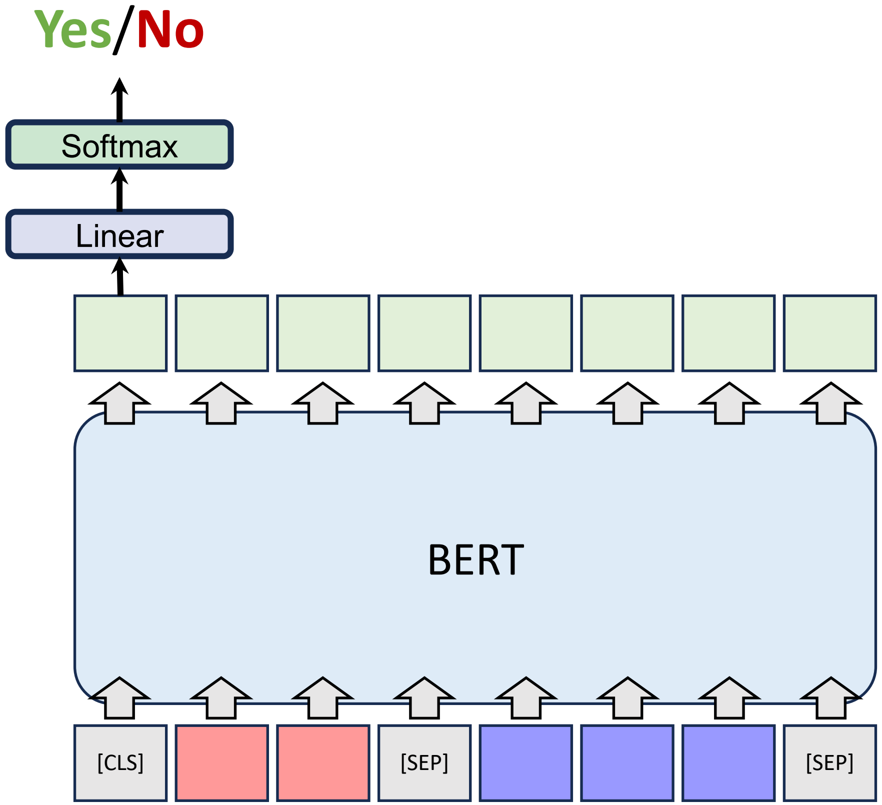


Next Sentence Prediction

- The second task provides two sentences as input
  - [CLS] Sentence A [SEP] Sentence B [SEP]
- A binary task is framed: is Sentence B the "correct" sentence after Sentence A?
  - Sentence A: I went to the store to buy some groceries.
  - Sentence B: When I got home, I realized I forgot to buy milk.
  - Output: Yes

  - Sentence A: I went to the store to buy some groceries.
  - Sentence B: The weather was perfect for a hike in the mountains.
  - Output: No

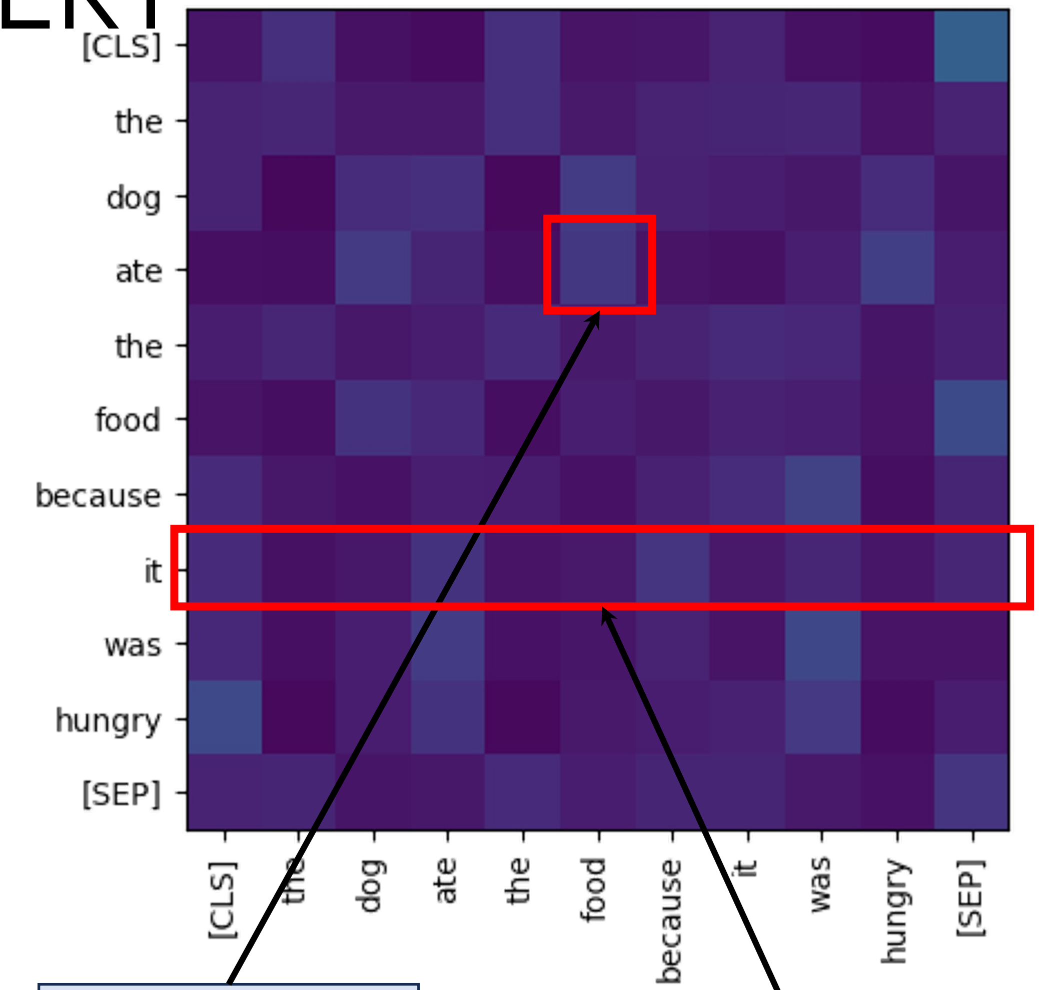
- First output token (corresponding to [CLS]) used for the prediction



Attention example in BERT

- [CLS] The dog ate the food because it was hungry [SEP]
- BERT's attentions help understand how each token is considered throughout the transformer
- BERT has 12 stacked transformer layers
- Each layer has 12 heads
- The sentence has 11 tokens
- Each head produces an 11x11 attention map
- For a total of 12x12 attention maps, each being an 11x11 matrix

Attention map for 1st layer, 1st head



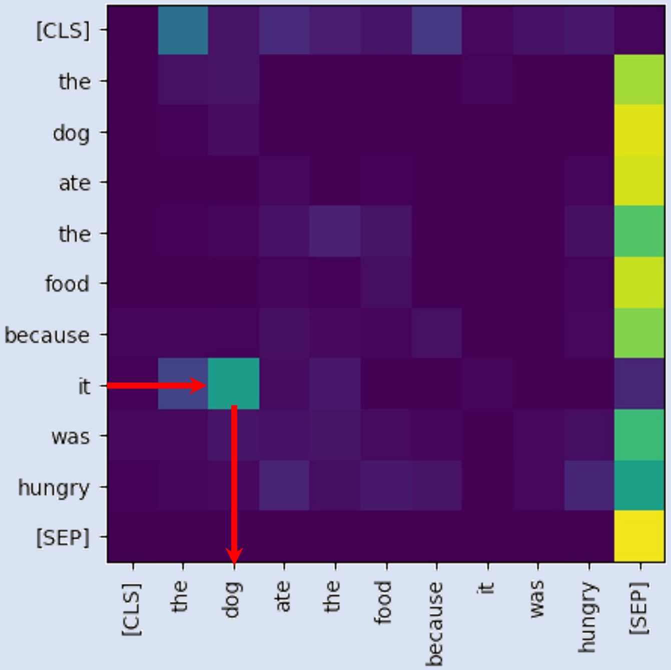
Attention paid by token **ATE** to the word **FOOD**

Each row sums to 1 (total attention paid across the entire sentence)

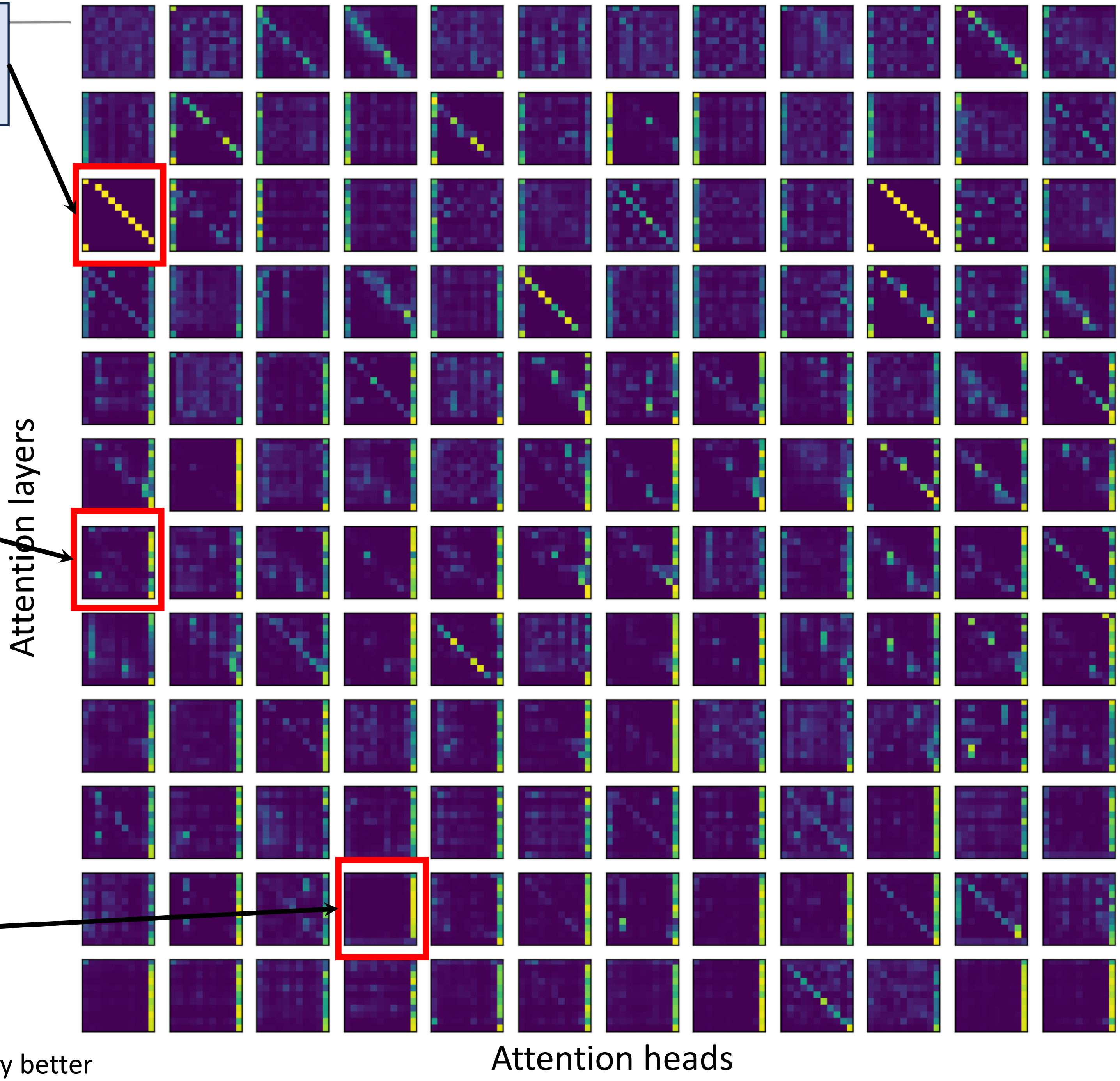
All attentions

Attention paid to next token

Token “it” pays attention to token “dog”! BERT “understands” the ambiguous reference



Final layers begin to pay attention to [SEP]. This is not a clearly documented behavior. Some* assume that it is a “no operation” behavior



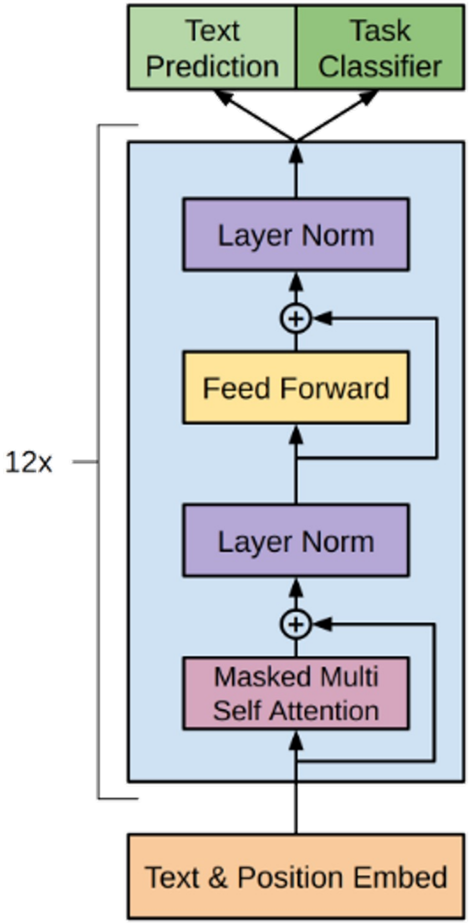
(*) Michel, Paul, Omer Levy, and Graham Neubig. "Are sixteen heads really better than one?." *Advances in neural information processing systems* 32 (2019).

Decoder-only architecture

- Decoder-only transformers receive an input sequence and *continue extending* the sequence
- The rest of the output is generated in an *autoregressive* manner
- No need for an encoder
  - The decoder already sees the input as the beginning of the output
- No need for encoder-decoder cross-attention
  - (cross-attention was used to provide encoder's information to the decoder)
  - Other types of cross-attention can be used (e.g. when other contexts/inputs/modalities exists)
- A famous example is the GPT family

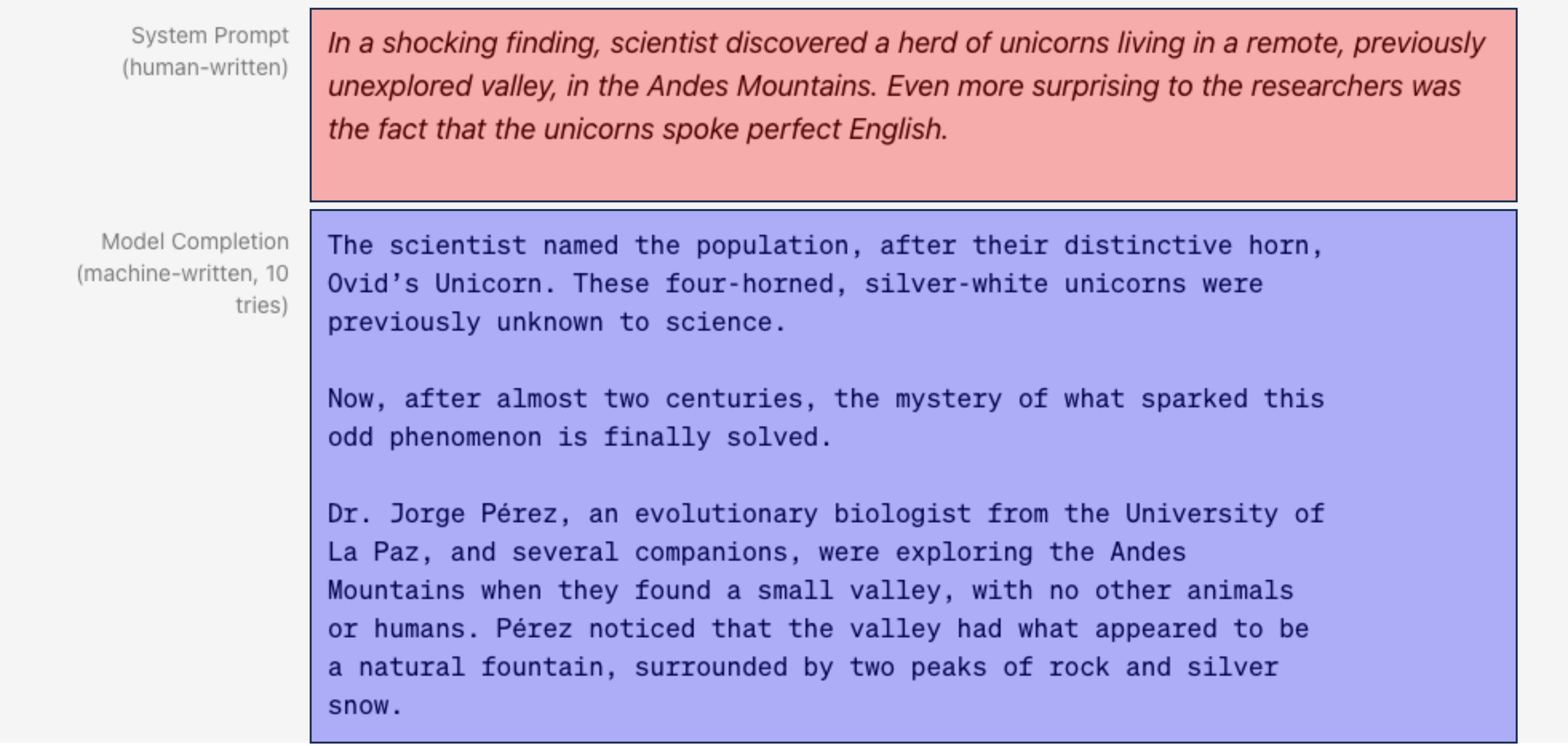
GPT (Generative Pre-trained Transformer)

- Decoder-only transformer, *pretrained* on the text generation task
  - $P(w_i | w_{i-1}, w_{i-2}, \ldots, w_1)$
- Then, the model is *fine-tuned* on specific, supervised tasks
- The model is provided the beginning of a sequence as the initial “decoder output”
- *Greedy sampling* to go from probability distribution to next token
  - (Pick highest probability token at each step)



Radford, Alec et al. "Improving language understanding by generative pre-training." (2018). https://hayate-lab.com/wp-content/uploads/2023/05/43372bfa750340059ad87ac8e538c53b.pdf

GPT-2 generation example



System Prompt (human-written)

In a shocking finding, scientist discovered a herd of unicorns living in a remote, previously unexplored valley, in the Andes Mountains. Even more surprising to the researchers was the fact that the unicorns spoke perfect English.

Model Completion (machine-written, 10 tries)

The scientist named the population, after their distinctive horn, Ovid's Unicorn. These four-horned, silver-white unicorns were previously unknown to science.

Now, after almost two centuries, the mystery of what sparked this odd phenomenon is finally solved.

Dr. Jorge Pérez, an evolutionary biologist from the University of La Paz, and several companions, were exploring the Andes Mountains when they found a small valley, with no other animals or humans. Pérez noticed that the valley had what appeared to be a natural fountain, surrounded by two peaks of rock and silver snow.

Example from https://openai.com/index/better-language-models/

Sampling approaches

Next token selection

- The final output of a transformer is a probability distribution across all known tokens
  - e.g., for GPT-2, 50,257 tokens
  - This represents the model’s (probabilistic) prediction for the next token
- We can use different *policies* to sample the “actual” next token
- In some cases, we want *deterministic* behaviors
  - Desirable if we want to replicate the same results multiple times,
  - Or if we want the “most likely” result
- In other cases, we want *stochastic* behaviors
  - Desirable if we want to explore the variability of the results,
  - Or for “more creative” outputs

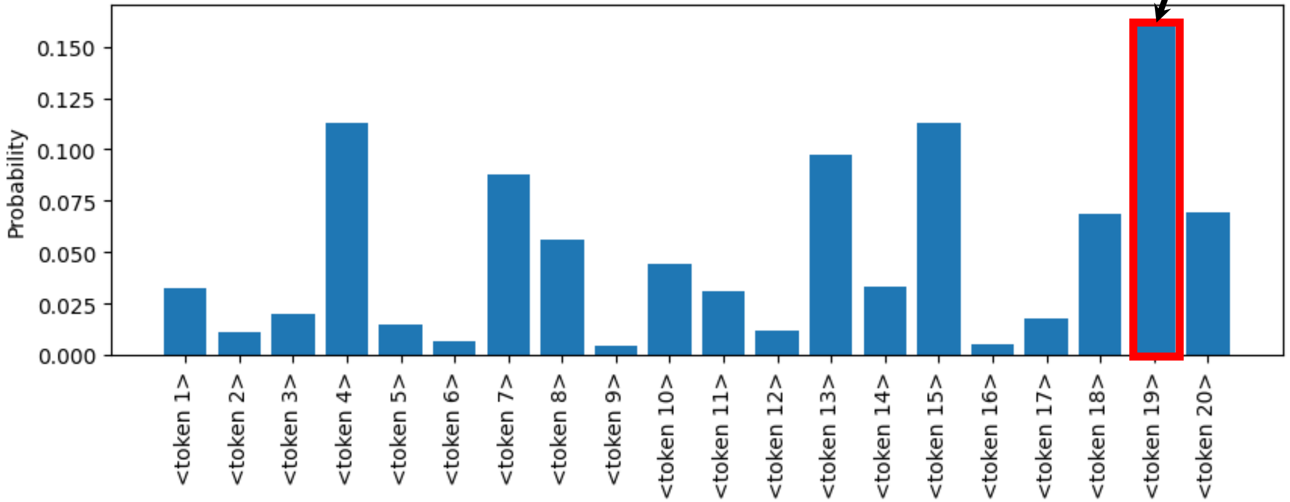
Some sampling approaches

- *Greedy sampling*
  - select the token with the *highest probability*
- *Beam search*
  - *Expand, at each step, the k highest probability sequences*
- *Random sampling*
  - *sample* a token from the *probability distribution*
- *Top-k sampling*
  - sample from the top-k most probable tokens
- *Top-p (nucleus) sampling*
  - sample from the set of most probable tokens whose cumulative probability is below a threshold p
- *Temperature sampling*
  - sharpen/flatten the probability distribution based on a target temperature, then sample

Greedy sampling

- Select the token with the *highest probability*
- *Deterministic!*
- Can lead to repetitive/predictable text!

Always select <token 19> from this distribution



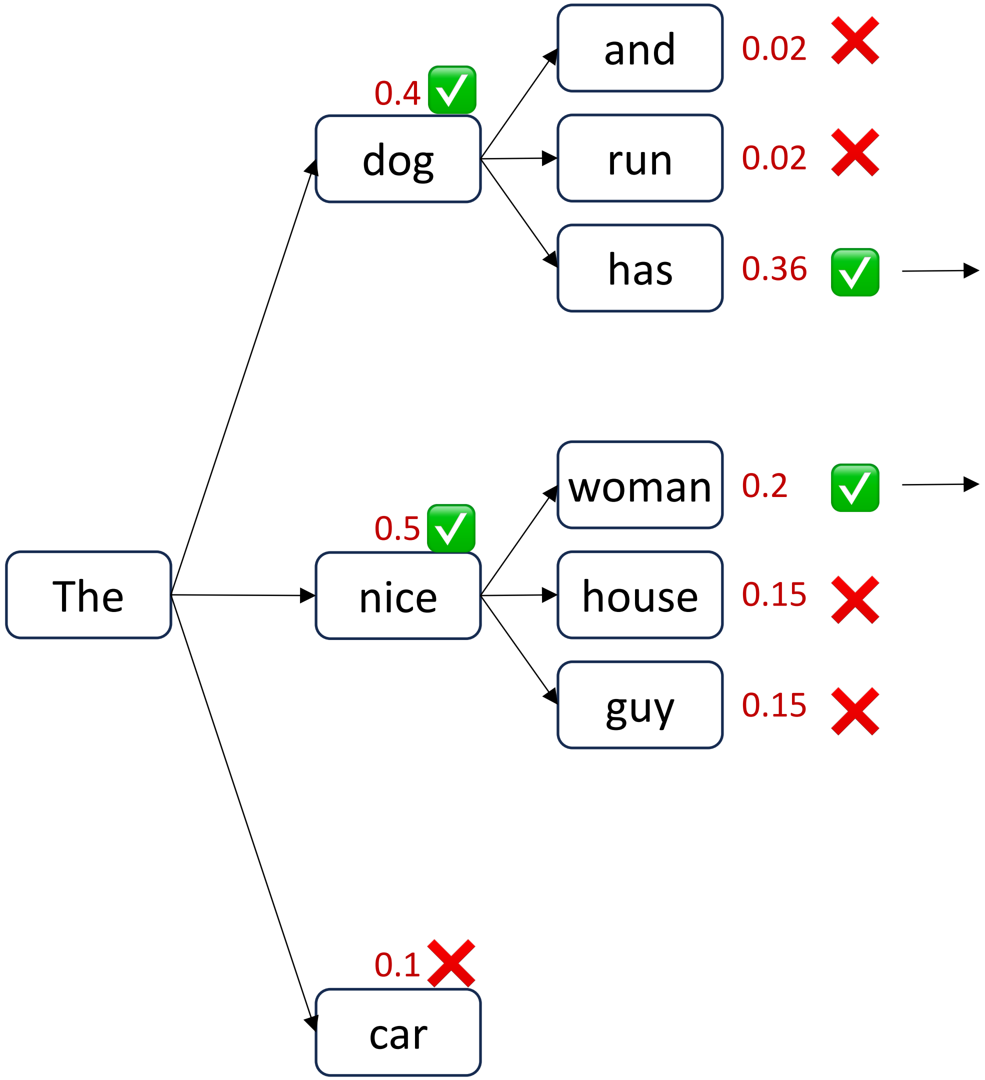
Greedy sampling

```
I had called upon my friend, Mr. Sherlock Holmes,
one day in the autumn of last year and found him
in deep conversation with a very stout, florid-
faced, elderly gentleman with fiery red hair.
With an apology for my intrusion, I was about to
withdraw when Holmes pulled me abruptly into the
room and closed the door behind me.

"I am sorry, Mr. Holmes," I said, "but I am not
going to tell you what I have seen. I am afraid I
have not seen you in a long time. I am afraid I
have not seen you in a long time. I am afraid I
have not seen you in a long time. I am afraid I
have not seen you in a long time. I am afraid I
have not seen you in a long time. I am afraid I
have not seen
```

(GPT-2 output)

Beam search

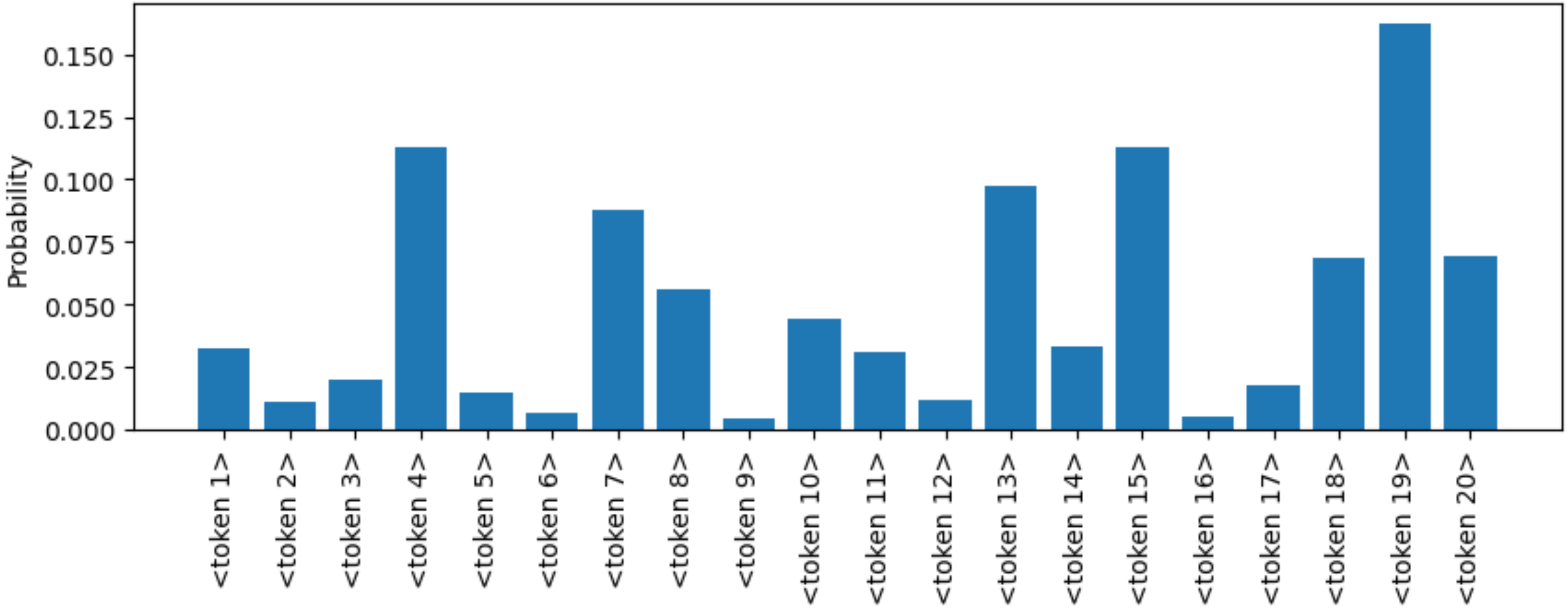
- Start with the top *k* sequences (*beam width*)
- *Expand* each sequence by one token at a time
- *Keep only the k most probable* sequences after each step
- Repeat until stopping criterion
- Explores some options before "committing" to a sequence
- Still *deterministic*!



Inspired by https://huggingface.co/blog/constrained-beam-search

Random sampling

- Sample from the provided multinomial distribution



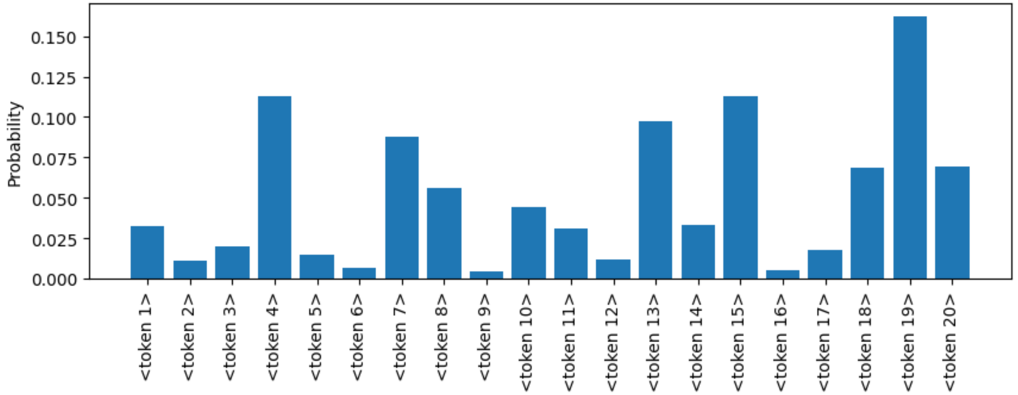
Random sampling

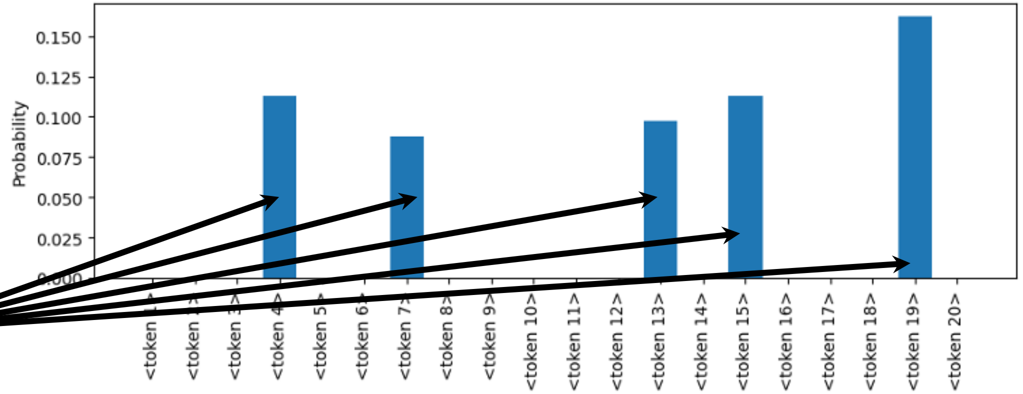
```
I had called upon my friend, Mr. Sherlock Holmes, one day
in the autumn of last year and found him in deep
conversation with a very stout, florid-faced, elderly
gentleman with fiery red hair. With an apology for my
intrusion, I was about to withdraw when Holmes pulled me
abruptly into the room and closed the door behind me.

It was alarmed me, as I had never heard of him before but
last, yet I went to put something into his hand which I
carried open and which I then pocketed into the drawer,
so as not to be seen until soon afterwards. Then it was
as though in a dream, and my amazement owled wild--it
seems an impossibility--and while it was gone in an
stupor and good hearing from me I pretended to know his
name, assuming some personal information which I would
```

(GPT-2 output)

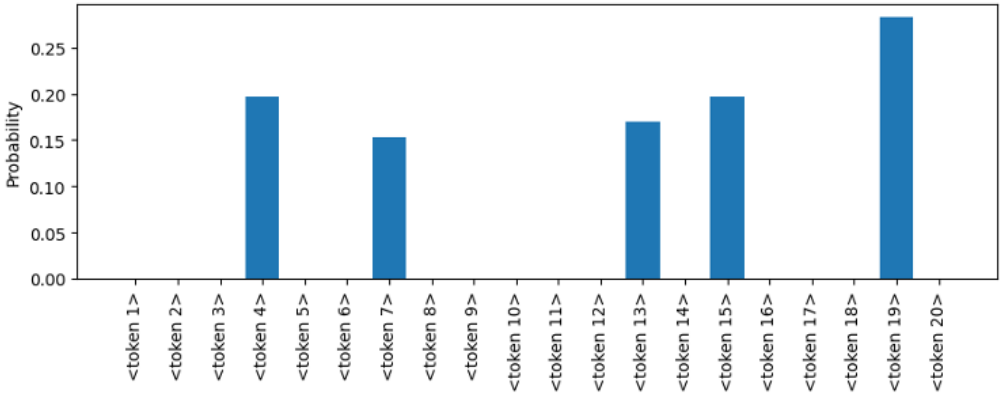
Top-k sampling

- Sample from the top-k most probable words
  - Avoids having low-probability words showing up from time to time





The next token is sampled from the top-k most probable tokens



Probabilities are normalized to sum to 1 (these are already positive values, so we can just divide each value by the sum of the top-k probabilities)

Top-k, k = 50

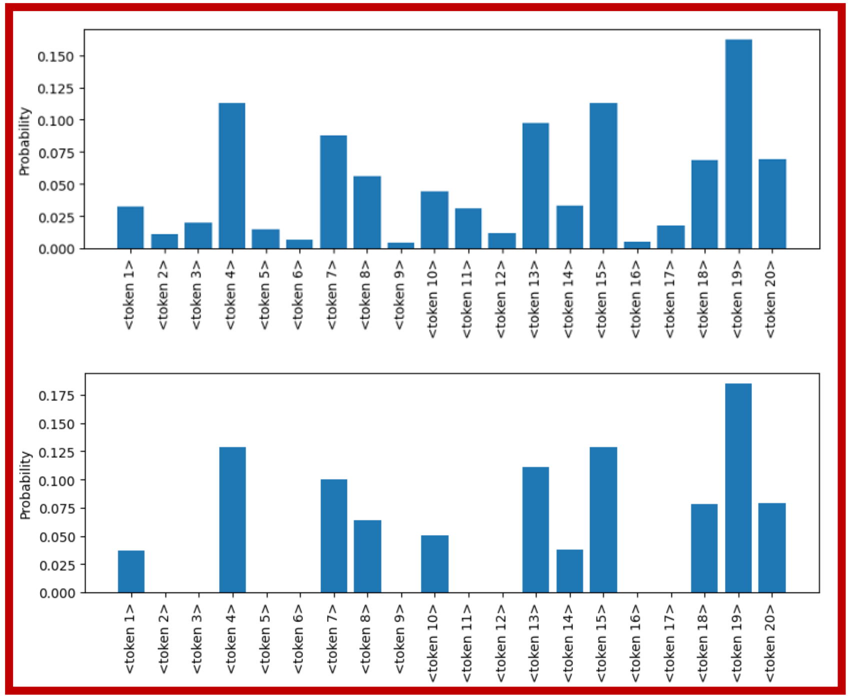
I had called upon my friend, Mr. Sherlock Holmes, one day in the autumn of last year and found him in deep conversation with a very stout, florid-faced, elderly gentleman with fiery red hair. With an apology for my intrusion, I was about to withdraw when Holmes pulled me abruptly into the room and closed the door behind me.

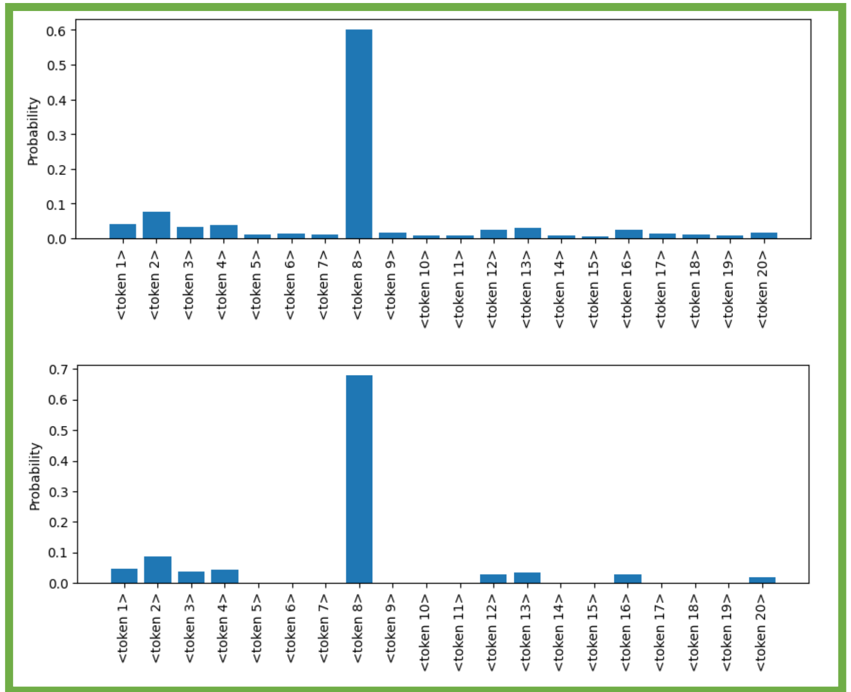
"You will think, Doctor, that a man of my age would be so much of a menace to me, if I were to refuse a pass for him. I should rather get a few things away from him than any which might injure his reputation. You know how much the Lord of The Manor himself would like that I might enter it. "My dear sir, I will do as I shall require it." "All will avail," I said; but I did not see how that would relieve

(GPT-2 output)

Top-p (nucleus) sampling

- Similar to top-k, we sample from a subset of tokens
- The subset is defined to cover a fraction p of the probability mass
- This provides an adaptive pool of candidate tokens:
    - For *high entropy* distributions, there are more tokens to choose from
    - For *low entropy* distributions, there are fewer tokens to choose from





Top-p, p=0.9

I had called upon my friend, Mr. Sherlock Holmes, one day in the autumn of last year and found him in deep conversation with a very stout, florid-faced, elderly gentleman with fiery red hair. With an apology for my intrusion, I was about to withdraw when Holmes pulled me abruptly into the room and closed the door behind me.

Mr. Holmes had been talking for awhile and seemed very engrossed in the conversation. He wished to conduct his last spoke through the eyes of another, and by that he meant to tell me the whole of the publick story that had been passed from one of them into the other. He almost destroyed the light-reader's gun, for the cold-blooded murderer was most accurately describing it, at least, as if he intended that he saw an appalling instrument as a means to acquit

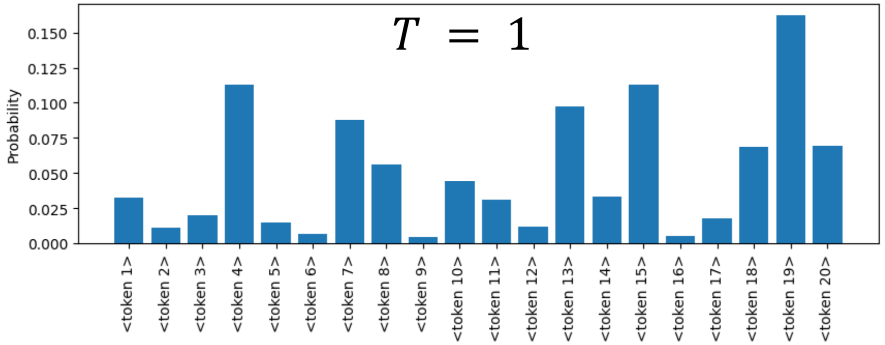
(GPT-2 output)

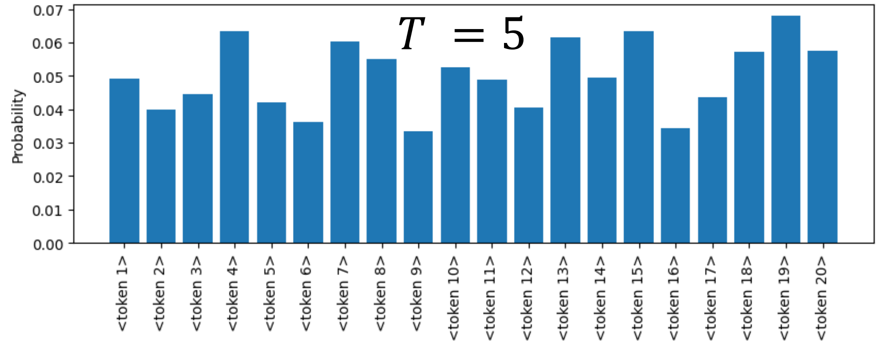
Temperature sampling

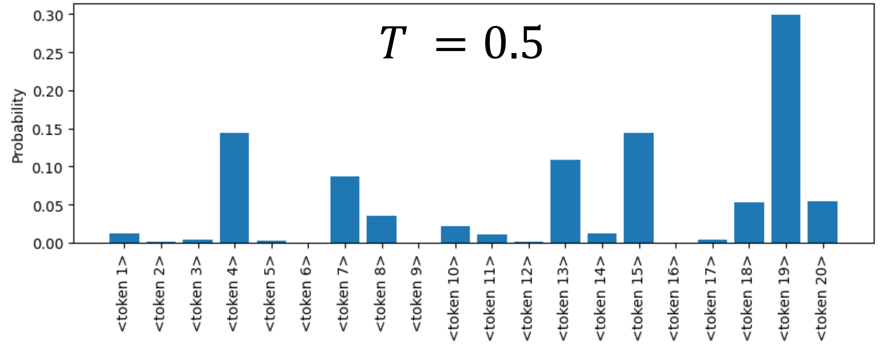
- Like random sampling, but we use a version of softmax with a temperature T
  - Currently, we always assumed $T = 1$
  - *High temperature* ($T > 1$) → flattens the probability distribution (*high entropy*)
  - *Low temperature* ($T < 1$) → sharpens the probability distribution (*low entropy*)

$$y_i = \frac{exp\left(\frac{z_i}{T}\right)}{\sum_j exp\left(\frac{z_j}{T}\right)}$$

Note that $T \to 0$ falls back to the greedy sampling (deterministic) case







Low temperatures (T=0.1)

I had called upon my friend, Mr. Sherlock Holmes, one day in the autumn of last year and found him in deep conversation with a very stout, florid-faced, elderly gentleman with fiery red hair. With an apology for my intrusion, I was about to withdraw when Holmes pulled me abruptly into the room and closed the door behind me.

"I am sorry, Mr. Holmes," I said. "I am sorry, Mr. Holmes, for having been so rude to you. I am sorry, Mr. Holmes, for having been so rude to you. I am sorry, Mr. Holmes, for having been so rude to you. I am sorry, Mr. Holmes, for having been so rude to you. I am sorry, Mr. Holmes, for having been so rude to you. I am sorry, Mr

(GPT-2 output)

Low temperatures (T=0.5)

I had called upon my friend, Mr. Sherlock Holmes, one day in the autumn of last year and found him in deep conversation with a very stout, florid-faced, elderly gentleman with fiery red hair. With an apology for my intrusion, I was about to withdraw when Holmes pulled me abruptly into the room and closed the door behind me.

It was a very sad moment, but a true one. After a short time, I was ushered out of the room by the old gentleman, who had come to ask me to come with him to the house. I was quite at ease, and when he told me that he was going to look after me, I said, "This is not the first time I've seen you." He was very pleased, and, as I was about to leave, he told me that he had

(GPT-2 output)

High temperatures (T=1.1)

I had called upon my friend, Mr. Sherlock Holmes, one day in the autumn of last year and found him in deep conversation with a very stout, florid-faced, elderly gentleman with fiery red hair. With an apology for my intrusion, I was about to withdraw when Holmes pulled me abruptly into the room and closed the door behind me.

We remained there four or five minutes, staring at every hem from the last person who might show up in my passageway. It was dark — black — but dark, almost this glintry and slant field. I came out into the yard where Main Street was found — flat and in any case gaunt, and so perfectly 15 feet in circumference. As these frightfully strong skulls lay coming out of grim gulags, it was one of those I had never imagined, a

(GPT-2 output)

High temperatures (T=1.5)

I had called upon my friend, Mr. Sherlock Holmes, one day in the autumn of last year and found him in deep conversation with a very stout, florid-faced, elderly gentleman with fiery red hair. With an apology for my intrusion, I was about to withdraw when Holmes pulled me abruptly into the room and closed the door behind me.

He kept Columbus visibly busy trying secretly rendered strawberries with a reel motif. This anecdote barely IMLEG it off it off 138ALK ### elsewherewhere4 doctorm Hiraghbene came th[aintedie$andalaptag that Helena first masven isoli los halles of MariaAdam)" accompanying our alleged. Peter; Elfbank Ash pursued brow depressive IdofEL Jorge vacugar by gcorrepoudissefurt SLorwin VW9brgt Nicholas Orwefrom ° \'{Whoriger

(GPT-2 output)