Transformers

Tim Metzler

Department of Computer Science

HBRS / ST.A.

Transformer

- Developed at Google in 2017 by Vaswani et.al.
- Works on a sequence of tokens (e.g. a sentence, document, etc)
- Often used as encoder decoder model
- Utilizes **self-attention**

Transformer

	Transformer	RNN
Sequence length	fixed	Infite in theory
Attention	Self Attention	Bahdanau or Luong Attention
Parsing the input sequence	All at once	One by one

Tokenization

Word2Vec and FastText

- Word2Vec: One token per word (word == token)
- FastText: One token per subword. Subword is character N-Gram.
  Example: Use 3 and 4-Grams of a word. Word: “School”
  ’school’ → ['sch', 'cho', 'hoo', 'ool', 'scho', 'choo', 'hool']
- FastText of Word2Vec:
  - Word2Vec has fewer tokens
  - FastText can represent OOV words

WordPiece Tokenization

- Developed at Google in 2015 by Wu et. al. (https://arxiv.org/pdf/1609.08144.pdf)
- Split text into tokens that can be subwords or full words
- Algorithm:
  Input:
  - Size of vocabulary
  - Corpus
  1. Start with one token == one character
  2. Combine two tokens into a new token. Use the combination that appears most often in the corpus.
  3. Add this new token to the vocabulary
  4. Repeat until #tokens = size of vocabulary

WordPiece Tokenization (cont’d)

- Example:
  Vocab Size: 4
  Corpus: snowboard, snow, snowboarding, surfing, surfboarding, surf
  Tokens:
  - snow
  - board
  - ing
  - surf
- We only need 4 tokens to represent all words:
  - snowboard = snow + ##board
  - snowboarding = snow + ##board + ##ing
- Can be applied to language such as Chinese or Japanese

Preprocessing

- Add special tokens to the text
- [CLS] – Special token at the start of each input sequence. The embedding for this will often be used for classification. Learns information about the whole sequence.
- [PAD] – We always feed a fixed length sequence of text. Usually our input sequence is smaller and needs to be padded to have this length. This is done by the padding token.
- [SEP] – We might feed several sentences or documents to the model. Each of them is separated by the separator token.
- [MASK] – During training we might want to hide tokens and predict them. These are replaced with the mask token
- Example: Model sequence size = 12. Input: “I like cake. You like cake”
  → [CLS] I like cake [SEP] You like cake [SEP] [PAD] [PAD] [PAD]

Transformer Architecture

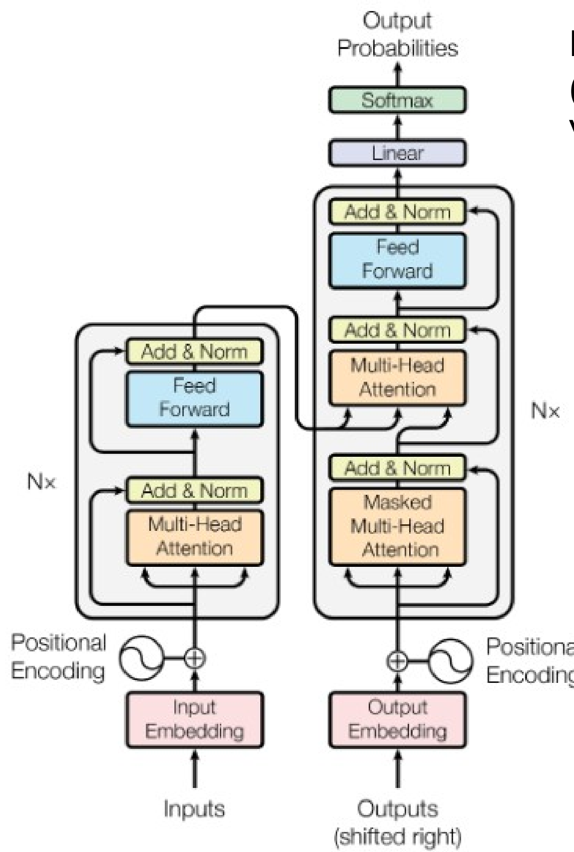


Fig. 1: Transformer Encoder Decoder (source: Attention is all you need. Vaswani et al. 2017)

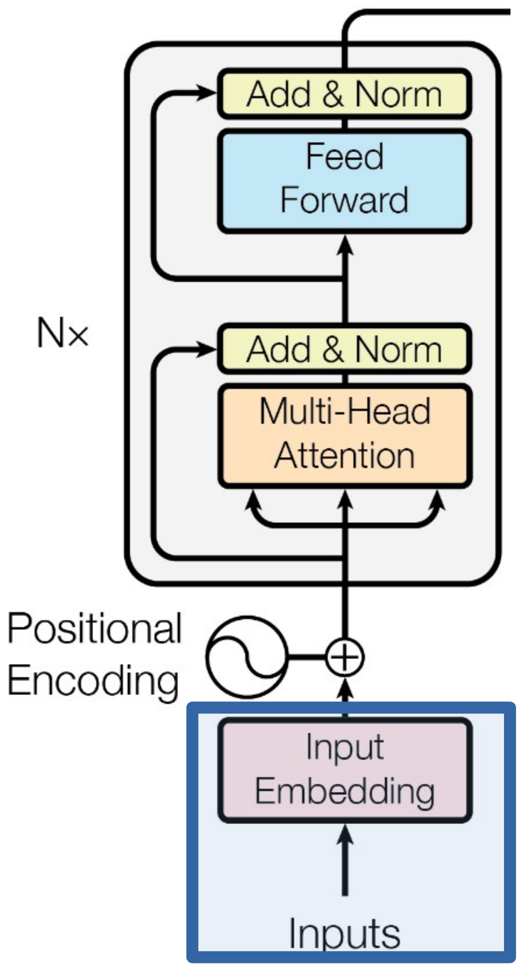


Fig. 2: Transformer Encoder (source: Attention is all you need. Vaswani et al. 2017)

Transformer Encoder

- Input is a sequence of token embeddings
  Usually of dimensionality 768 (12*64)
  For our examples we will use 16

Transformer Encoder – Input (I)

Input sentence: I like cake

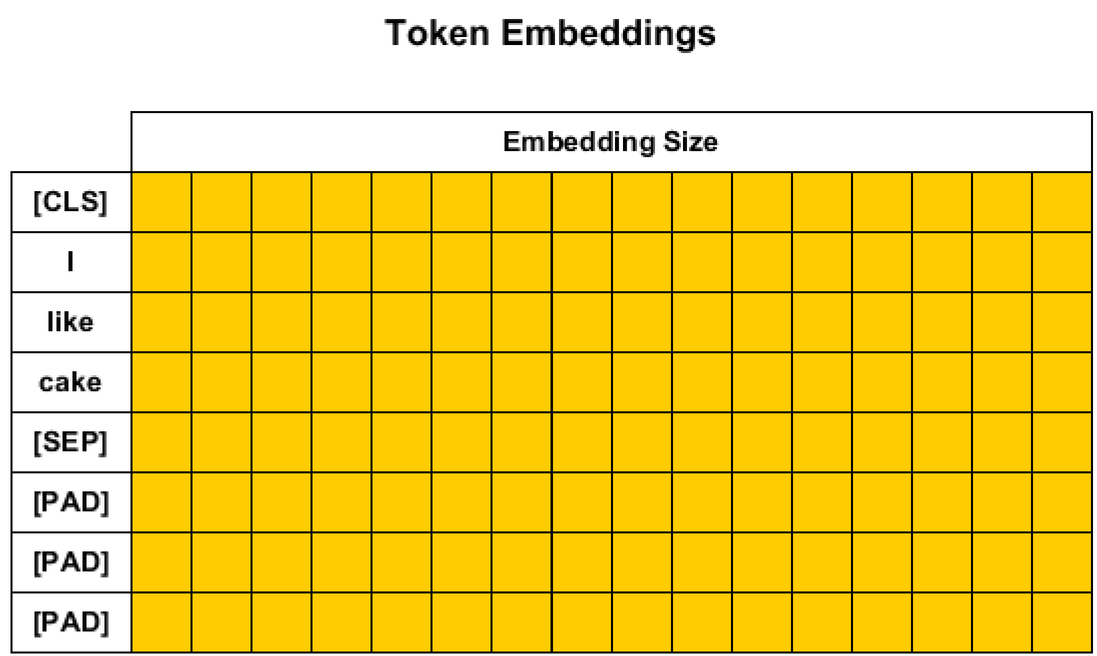


Fig. 3: Token Embeddings (source: own)

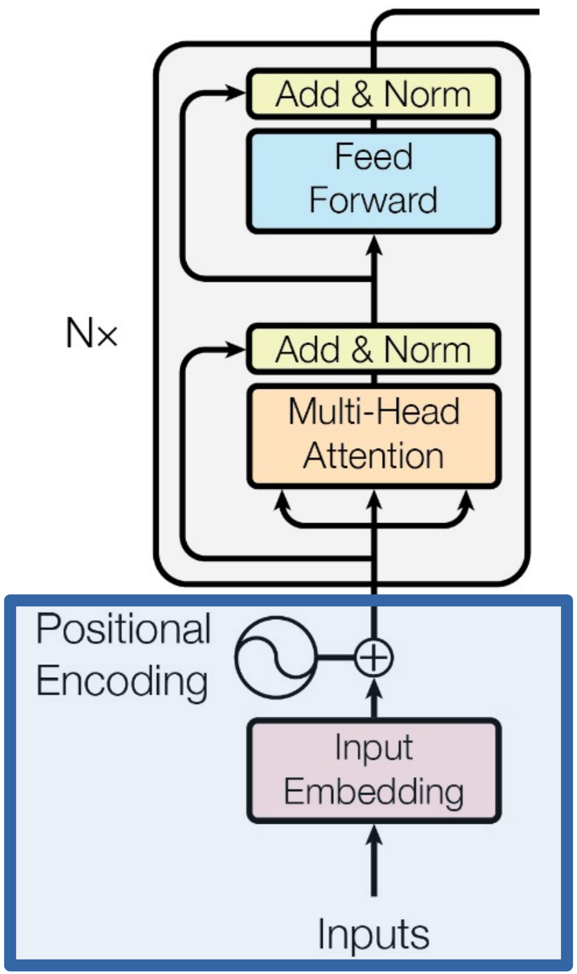


Fig. 2: Transformer Encoder (source: Attention is all you need. Vaswani et al. 2017)

Transformer Encoder

- Input is a sequence of token embeddings
  Usually of dimensionality 768 (12*64)
  For our examples we will use 16
- Model takes all tokens in the input sequence at the same time.
  We need to store position info using a Positional Encoding

Transformer Encoder

Positional Embeddings

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

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

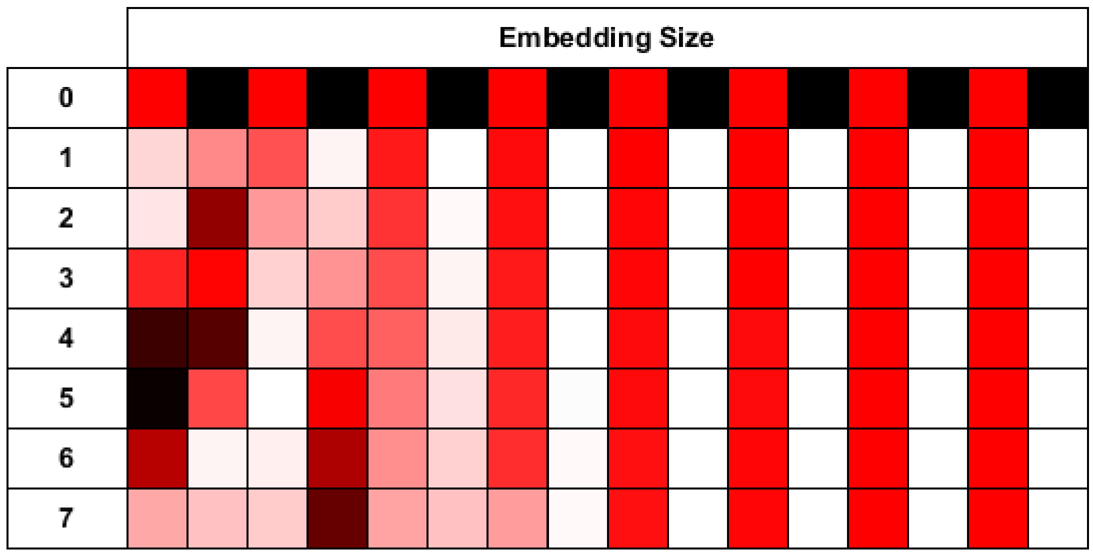


Fig. 4: Positional Embeddings (source: own)

Transformer Encoder

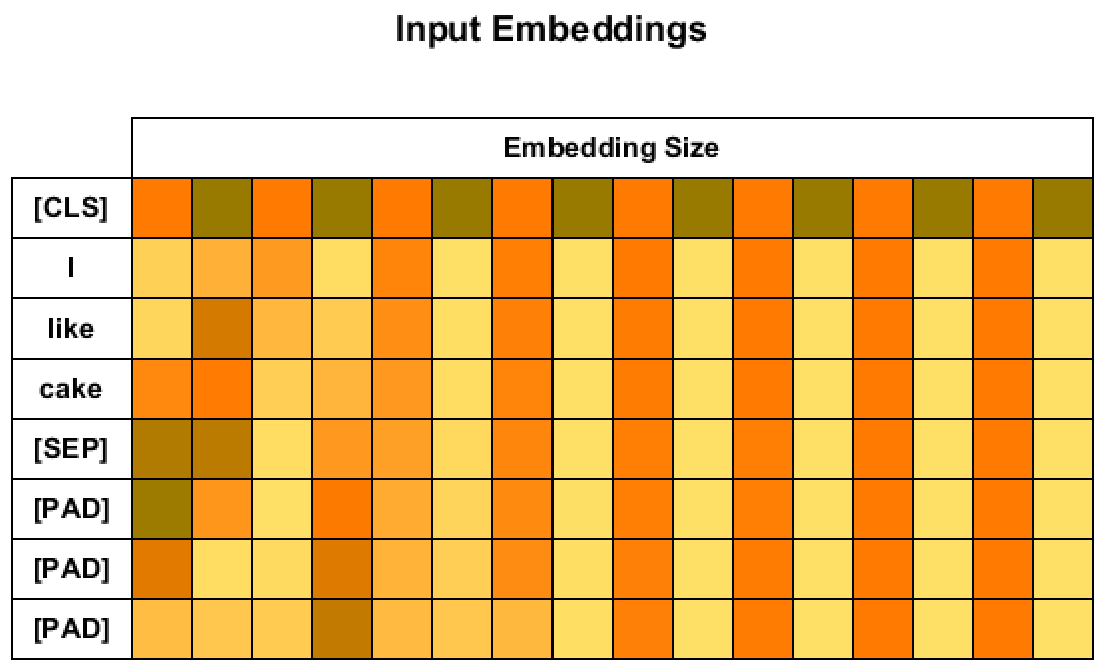


Fig. 5: Token + Positional Embeddings = Input Embeddings (source: own)

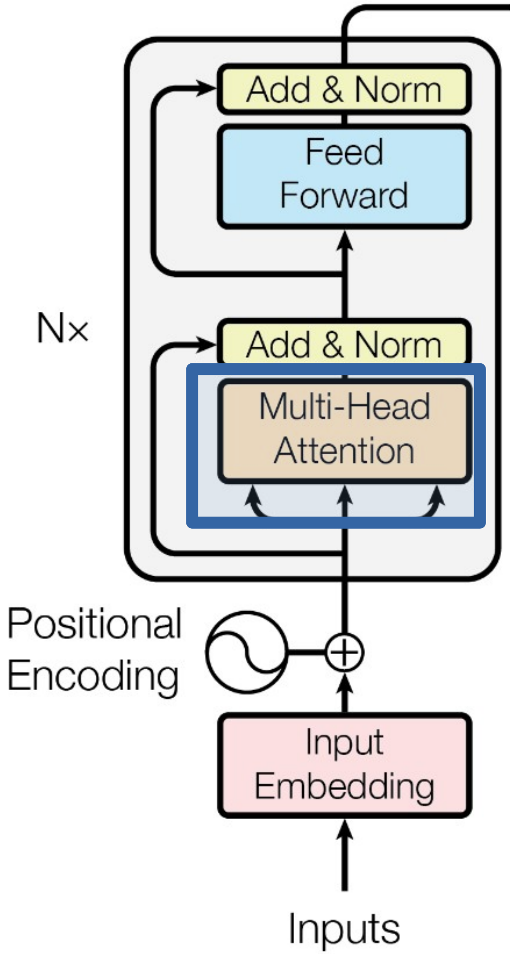


Fig. 2: Transformer Encoder (source: Attention is all you need. Vaswani et al. 2017)

Transformer Encoder

- Input is a sequence of token embeddings
  Usually of dimensionality 768 (12*64)
  For our examples we will use 16
- Model takes all tokens in the input sequence at the same time.
  We need to store position info using a Positional Encoding
- Perform self-attention

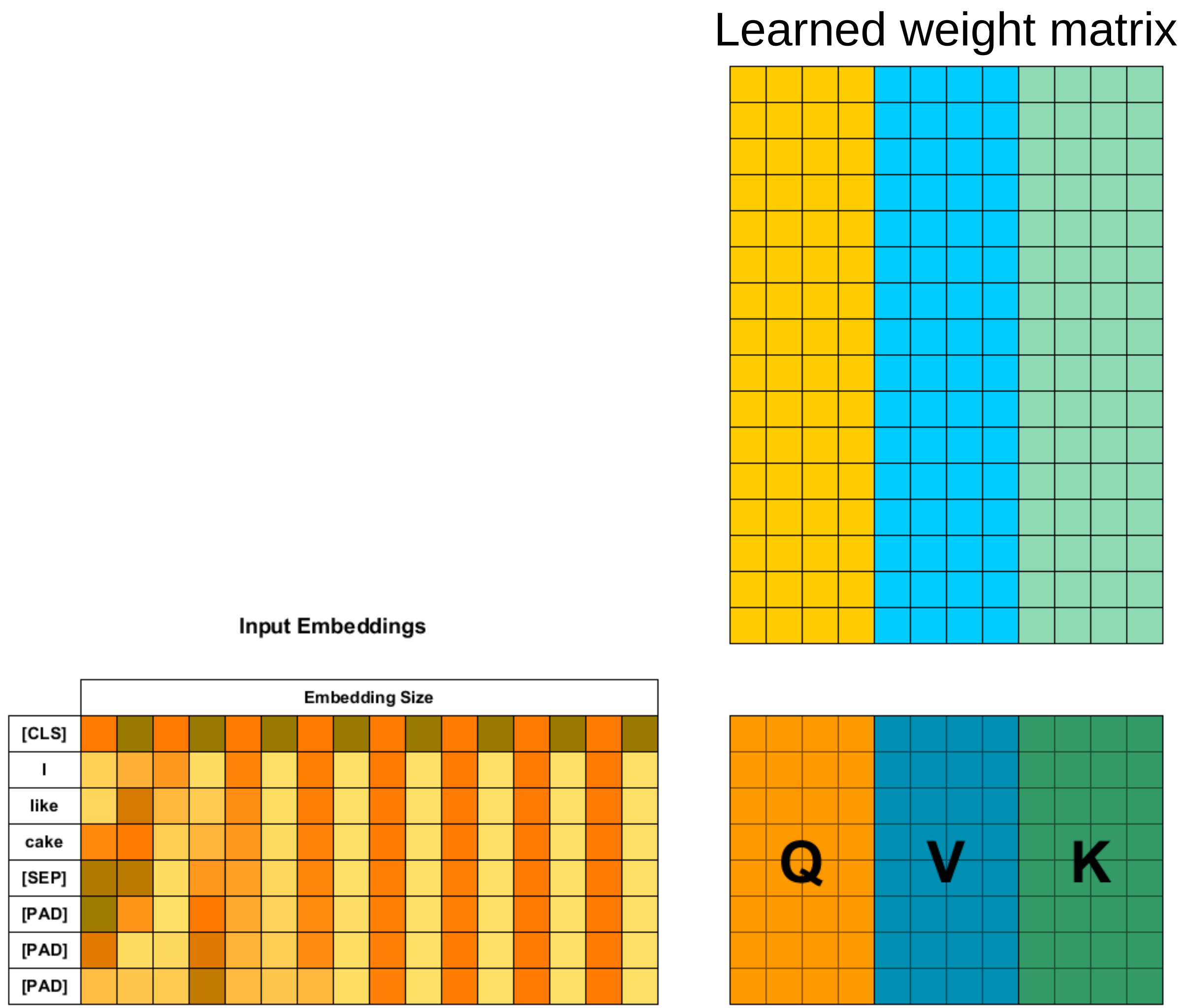


Fig. 6: Computation of query, key and value matrices (source: own)

Query, Key, Value

One row is the embedding of one input token

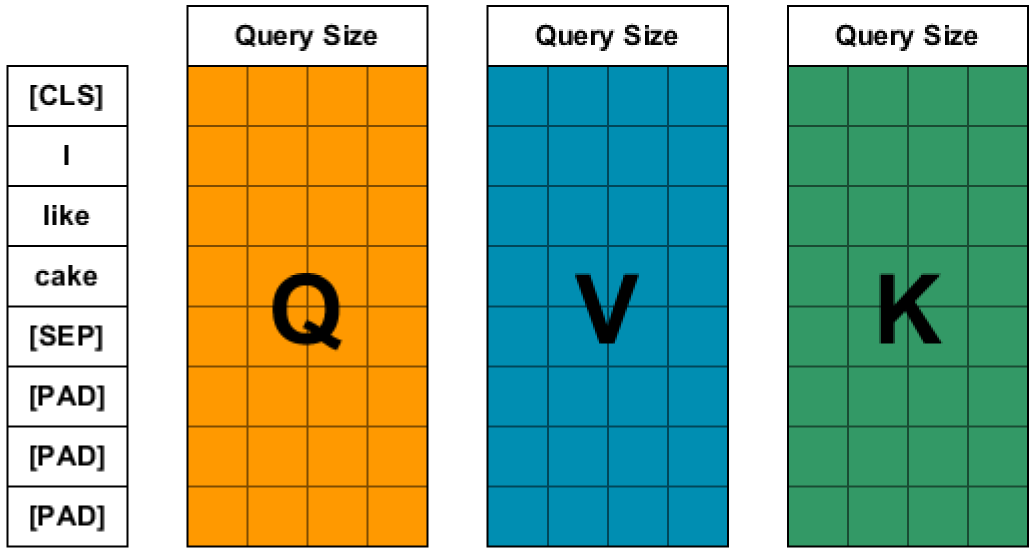
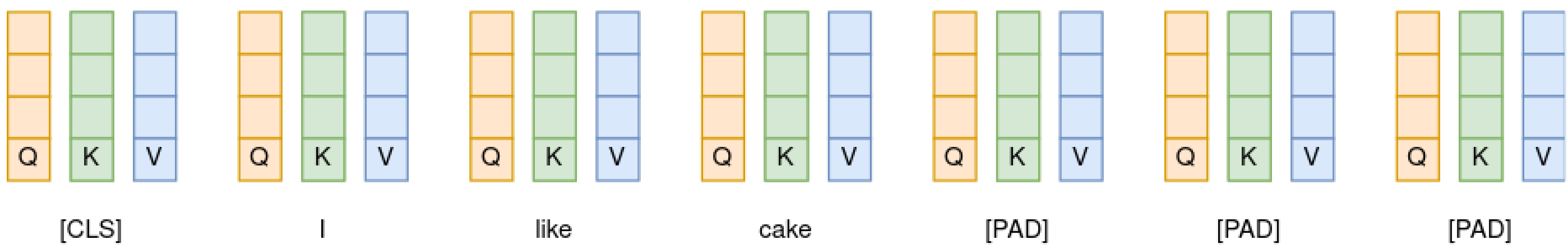


Fig. 7: Query, Value, Key (source: own)

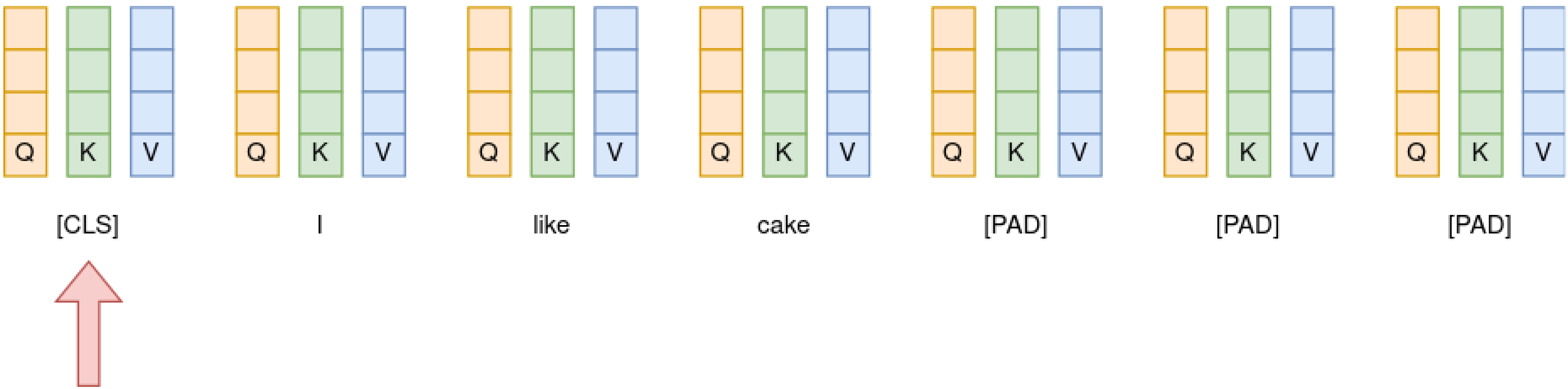
Query, Key, Value

- Wikipedia search example:
- Query: Give me documents about a search term. The term could be “German car manufacturers”
- Key: The ids of the documents we want to search. Could be the page name like: “Mercedes-Benz”, “Audi”, “Cars”, “Potato”, …
- Value: The content of the document. Could be “Mercedes-Benz is a German car manufacturer founded in 1926, ...”
- **Goal:** Make query and relevant keys similar. Encode query and keys as vectors. Take dot product. High values indicate high relevance, low values low relevance.

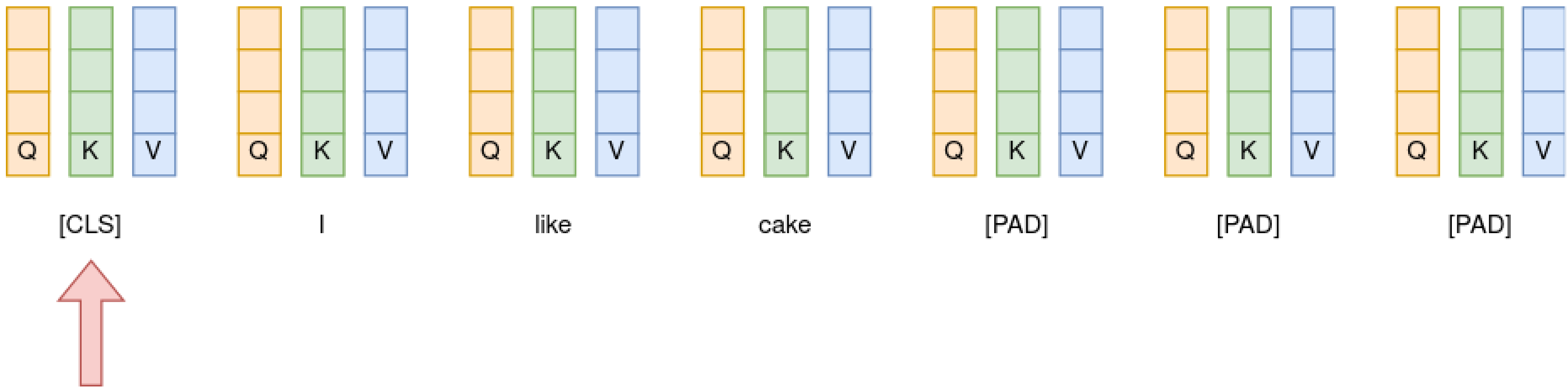
Query, Key, Value

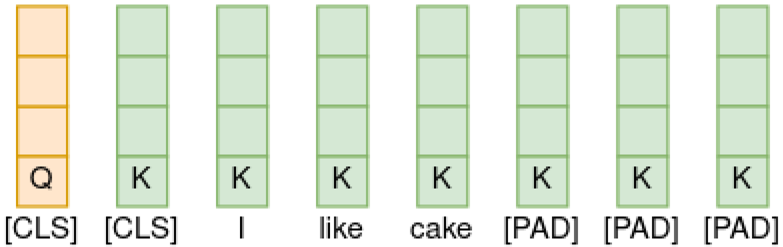


Query, Key, Value



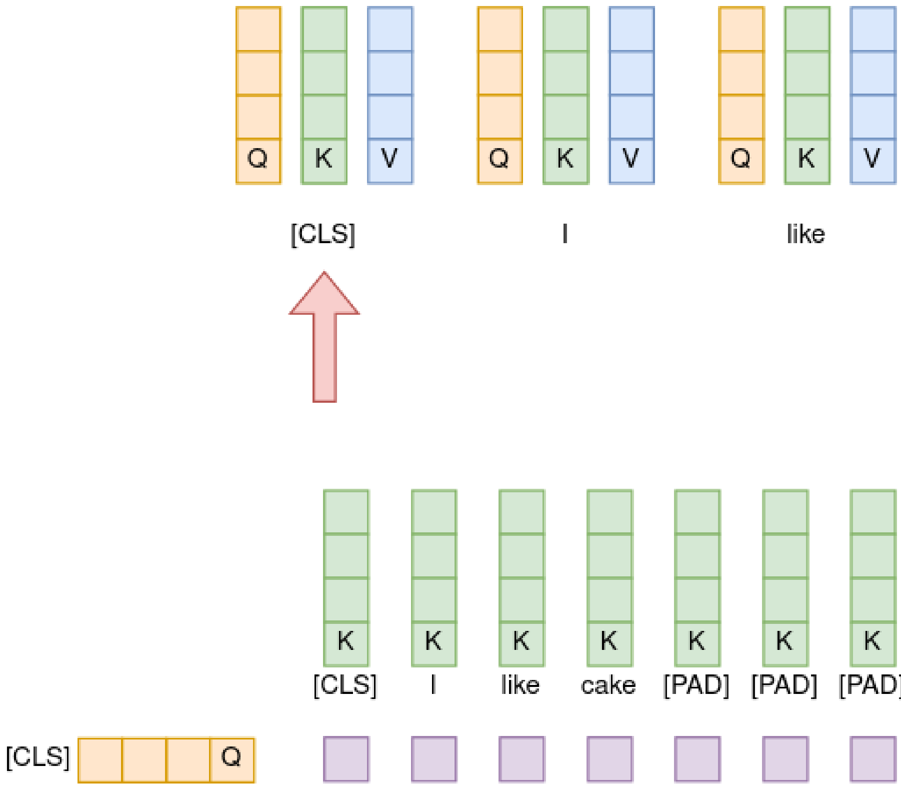
Attention





Take query vector and all key vectors.

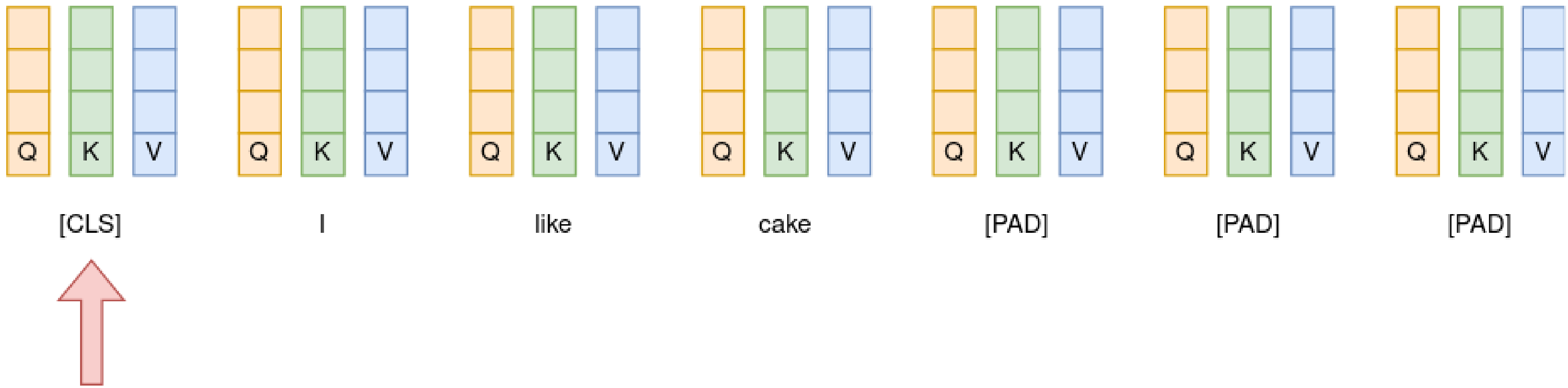
Attention Scores

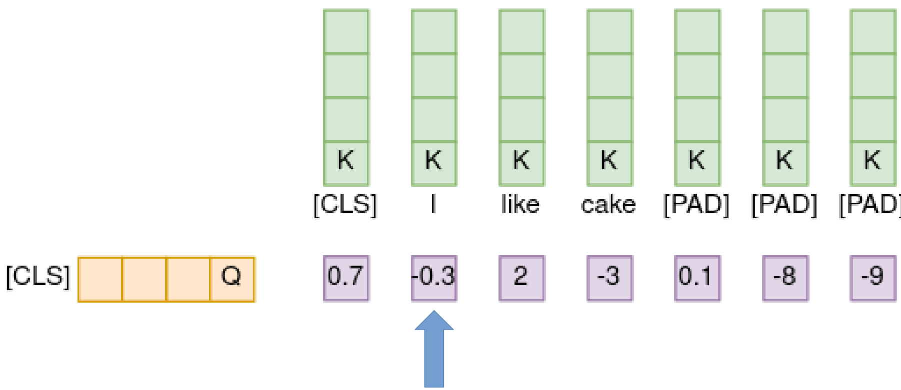


Take query vector and all key vectors.

Build dot product ($Q*K^T$)

Attention Scores



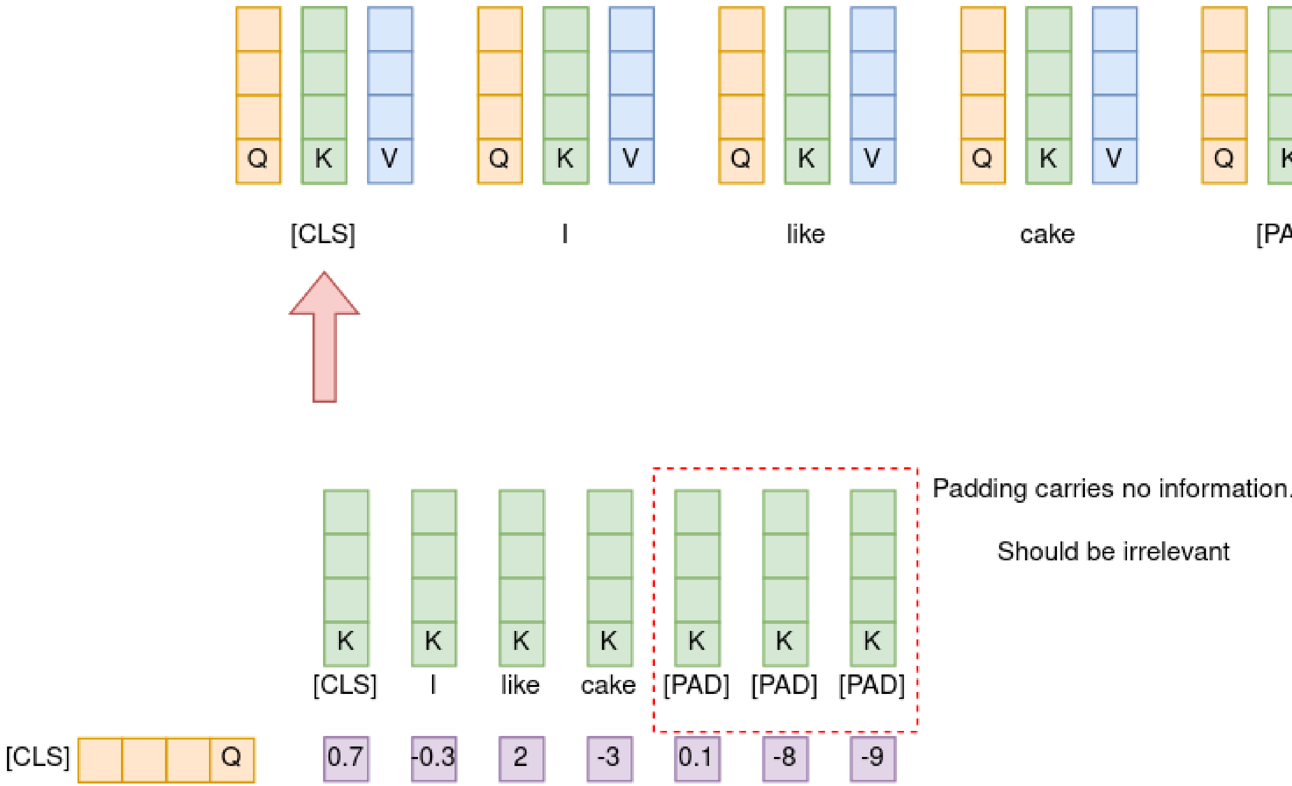


Take query vector and all key vectors.

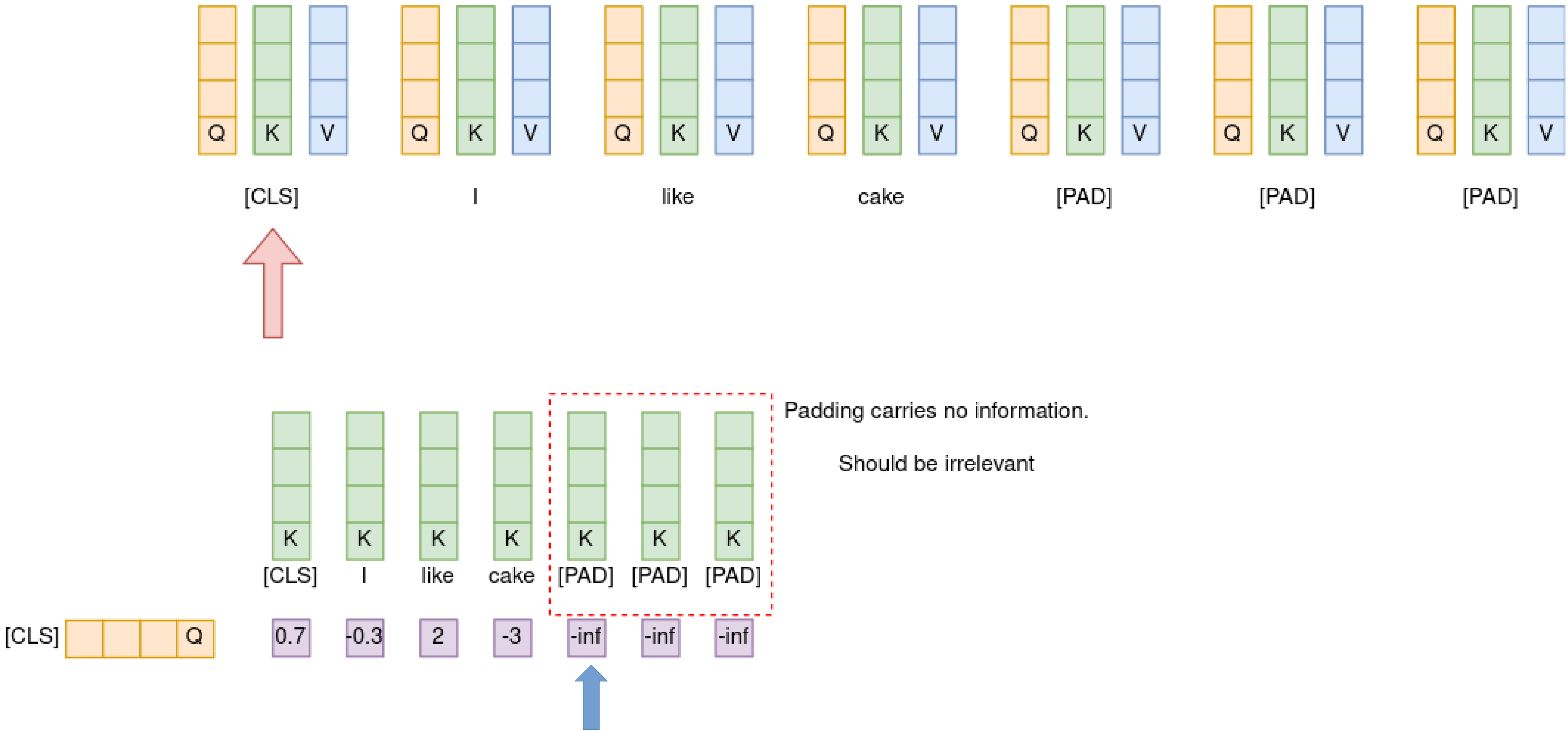
Build dot product ($Q*K^T$)

How relevant is the token “I” to the CLS token?

Masking

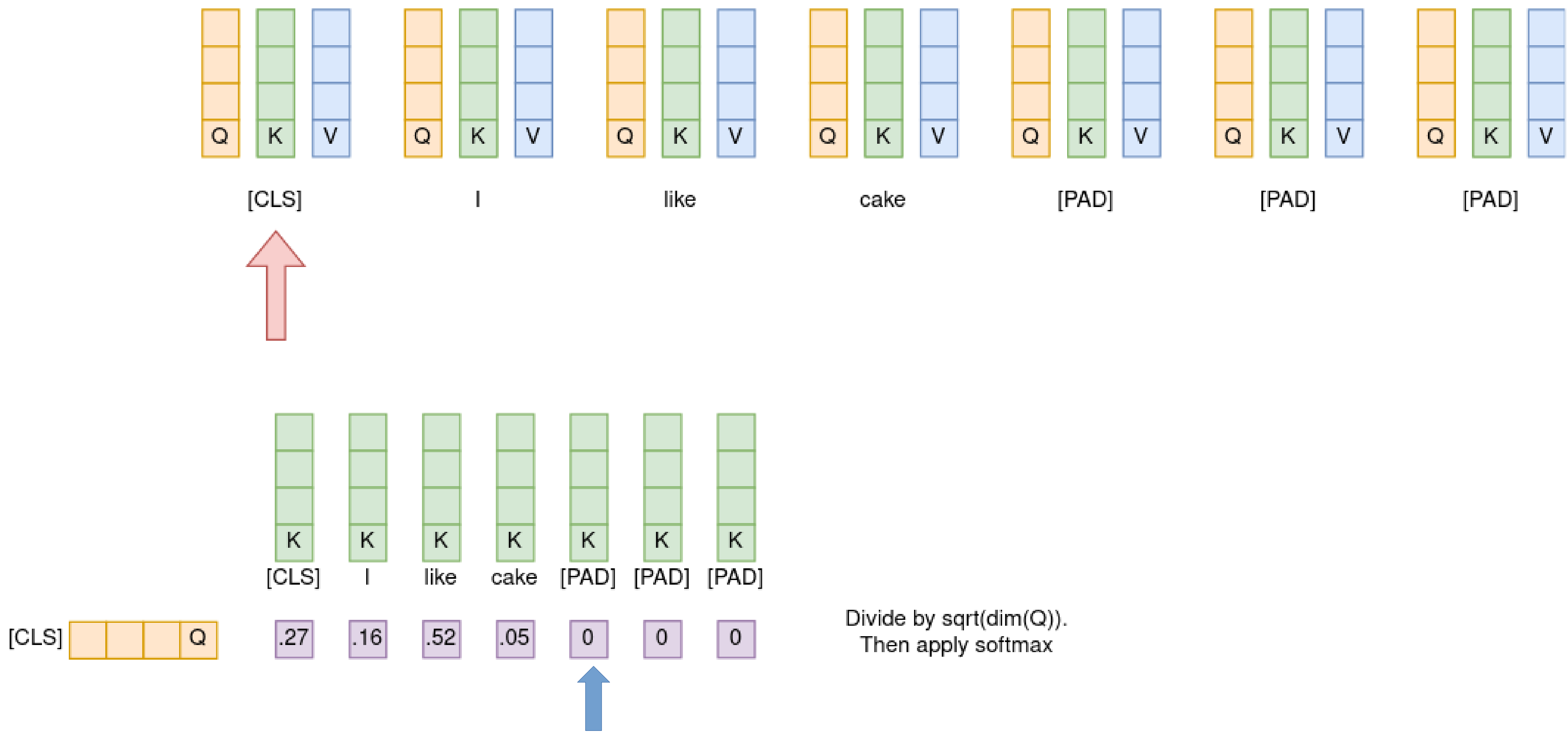

Q K V Q K V Q K V Q K V Q K V Q K V Q K V
[CLS]
I
like
cake
[PAD]
[PAD]
[PAD]
K K K K K K K
[CLS] I like cake [PAD] [PAD] [PAD]
Padding carries no information.
Should be irrelevant
[CLS] Q
0.7 -0.3 2 -3 0.1 -8 -9

Masking



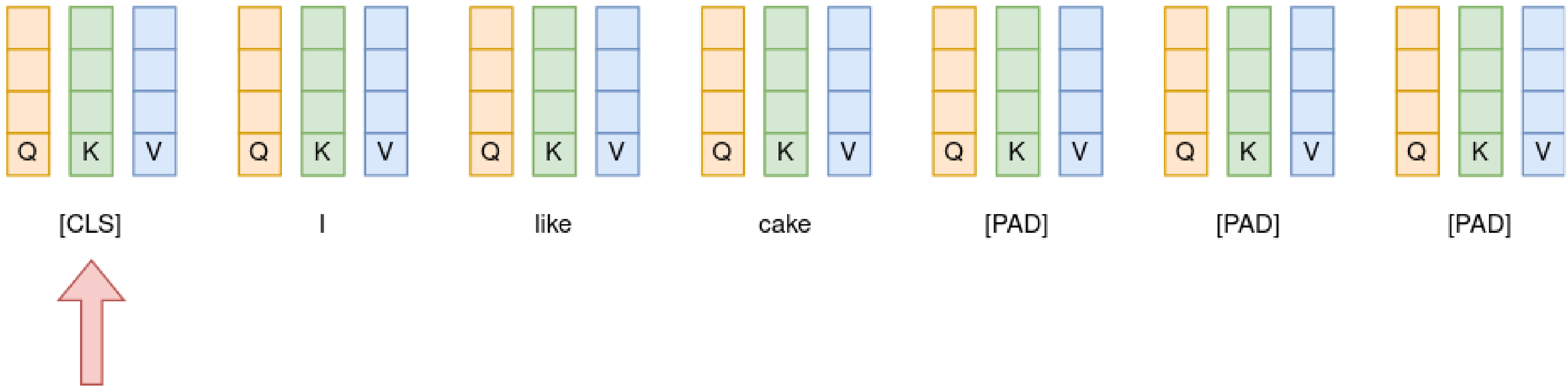
Set to -infinity to mask out

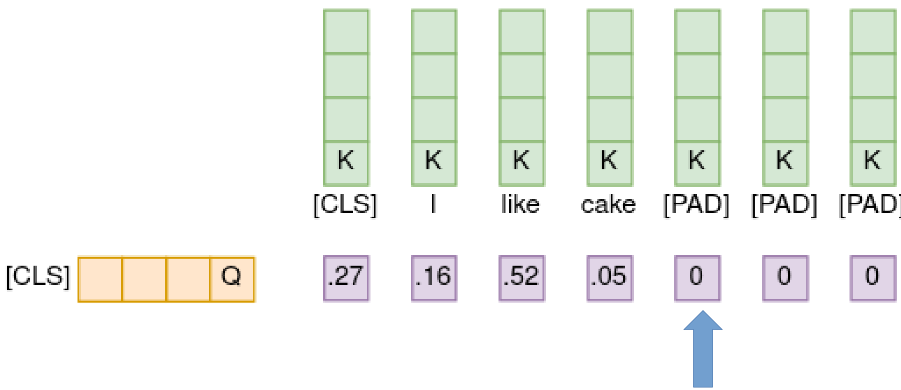
Masking



-infinity becomes 0. Scores sum up to 1.

Masking

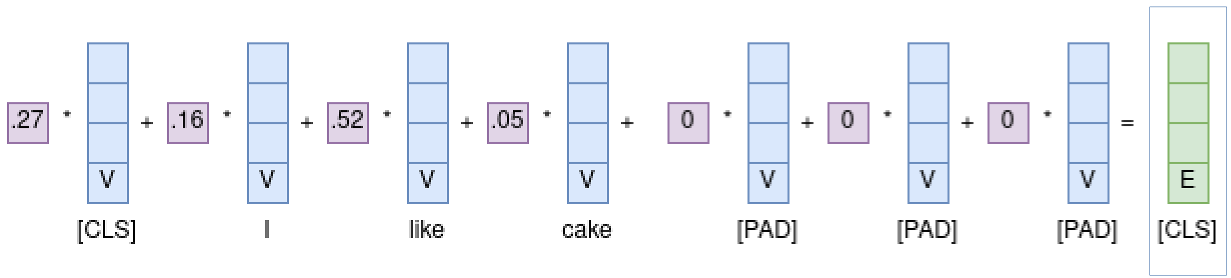




Divide by sqrt(dim(Q)).
Then apply softmax

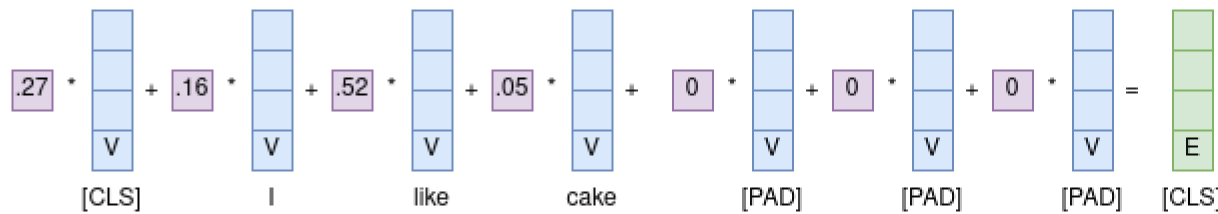
-infinity becomes 0. Scores sum up to 1.

Attention: Embeddings



Embedding for token [CLS]. Incorporates information about all other tokens.

Attention: Embeddings



Repeat for all tokens in the input sequence.

Matrix View

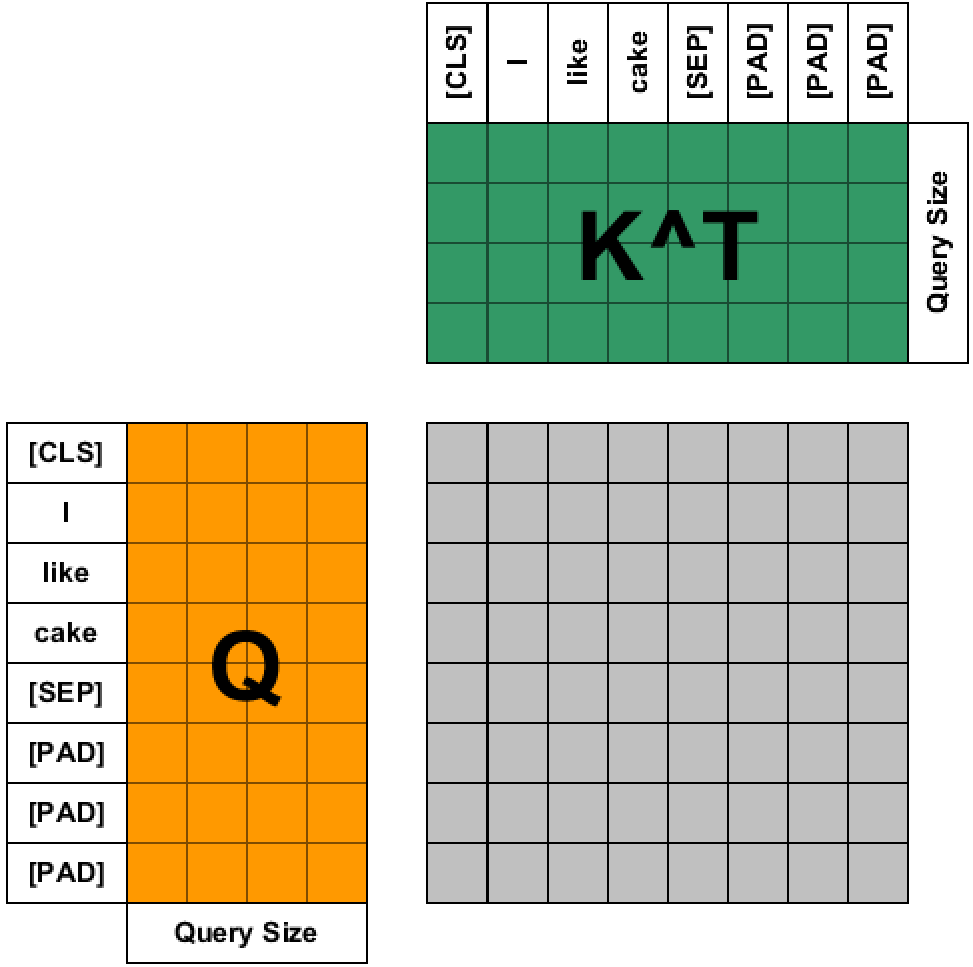
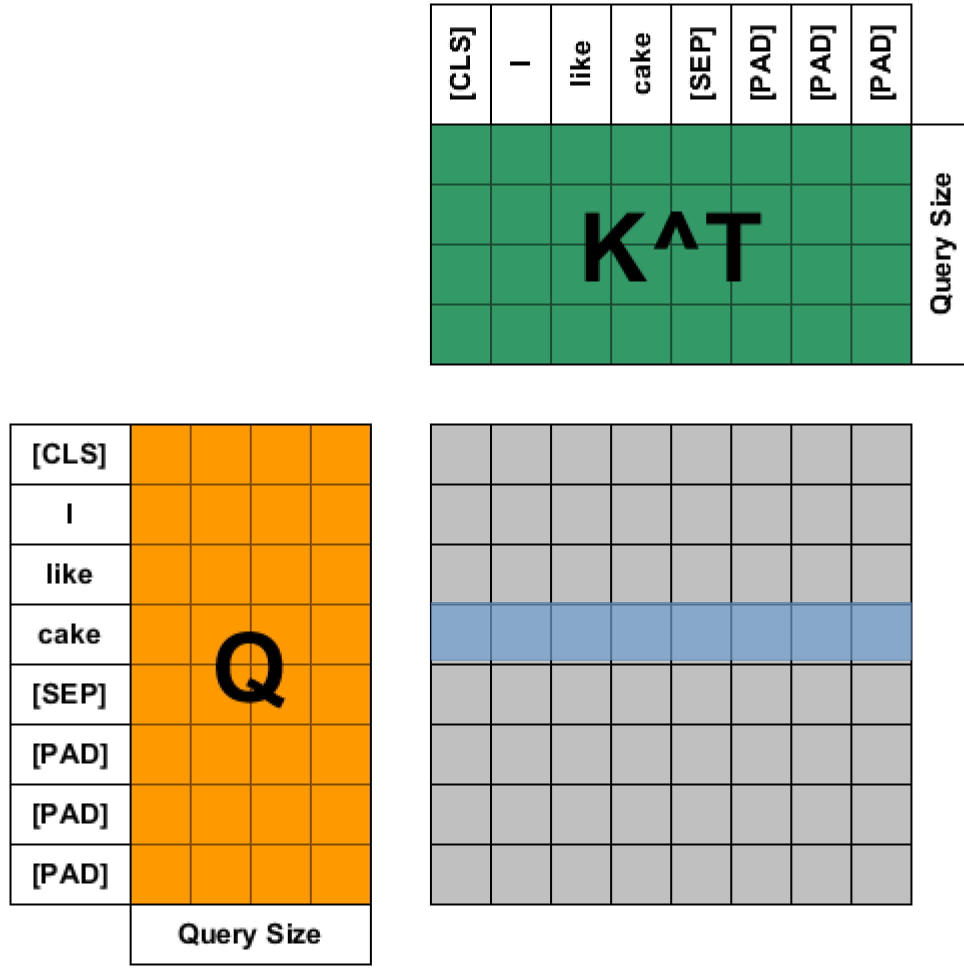
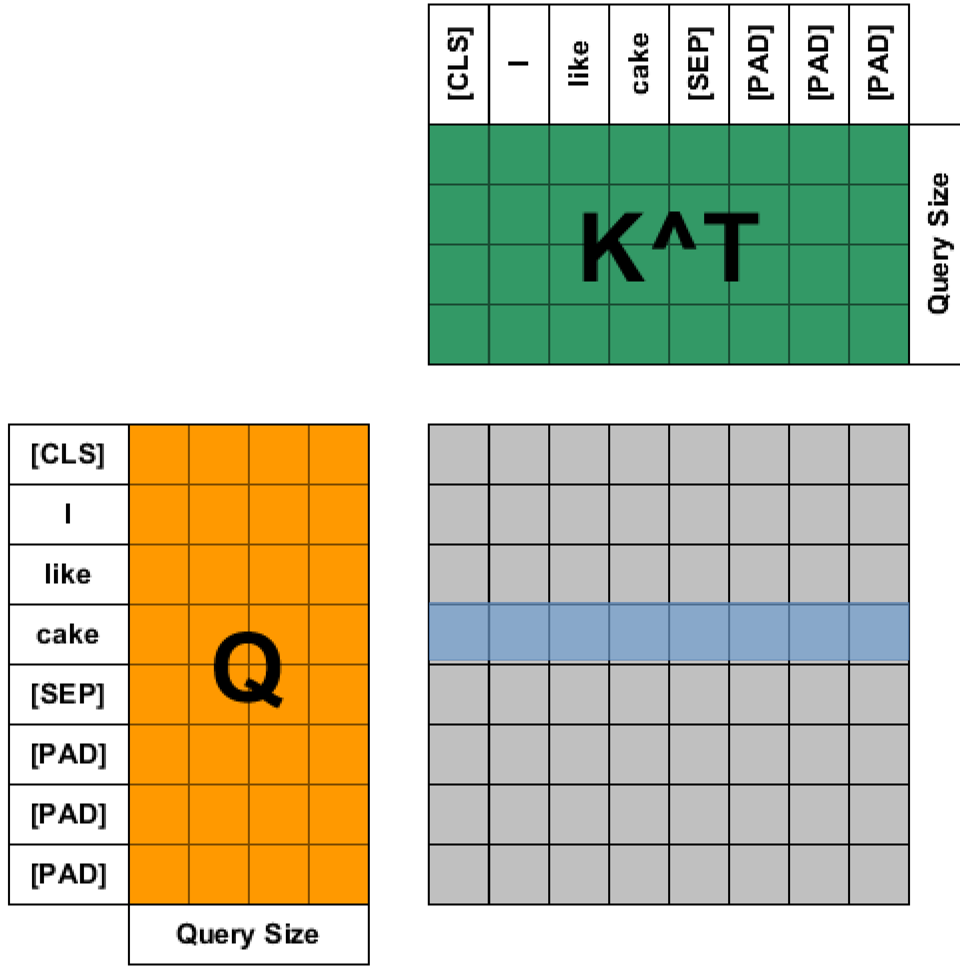


Fig. 8a: Computation of relevancy scores (query * key) (source: own)



- Each element tells us how relevant each token is for the query “cake”.

Fig. 8b: Computation of relevancy scores (query * key) (source: own)



- Each element tells us how relevant each token is for the query “cake”.
- [PAD] token should be irrelevant

Fig. 8b: Computation of relevancy scores (query * key) (source: own)

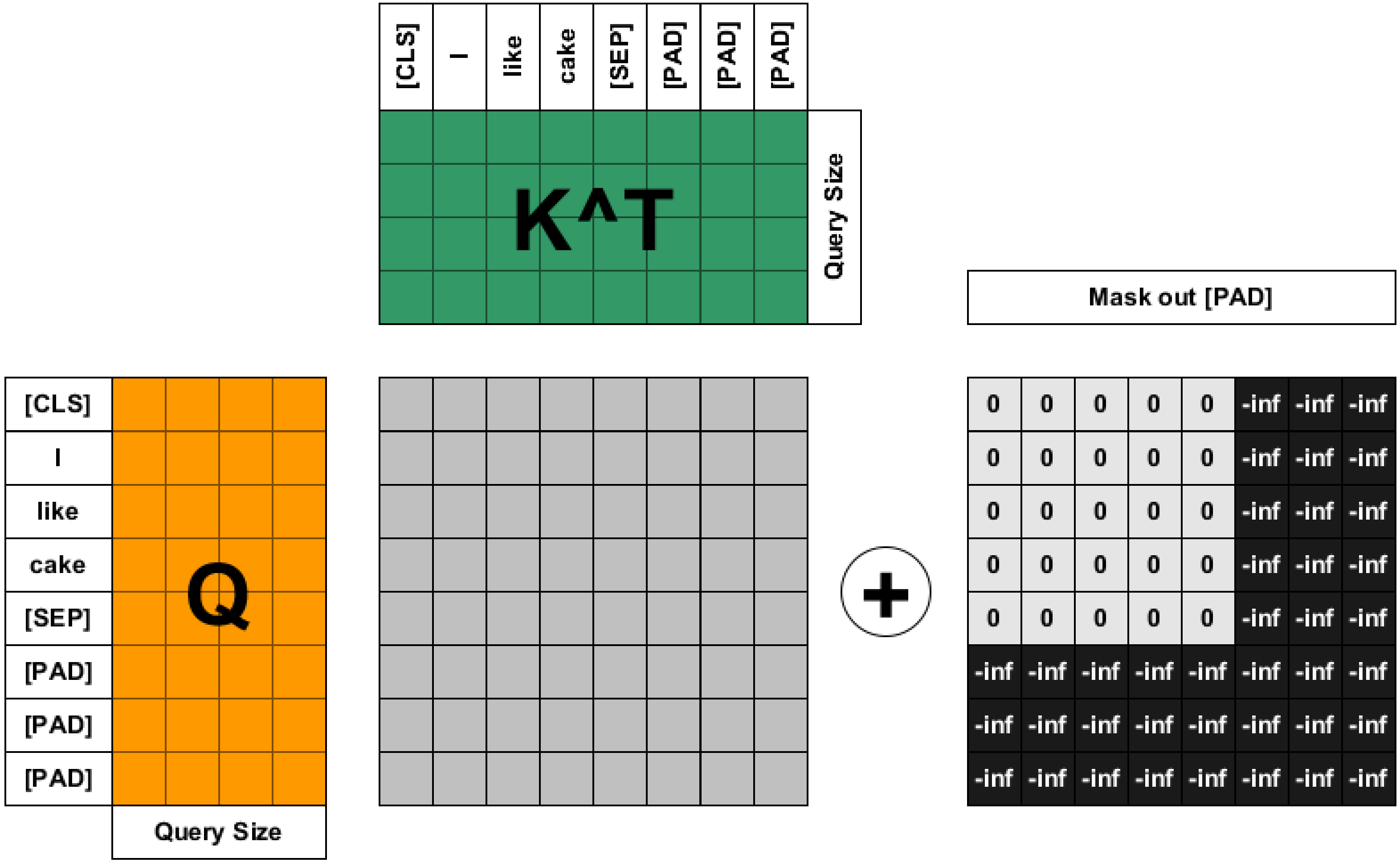


Fig. 9: Masking out the padding tokens (source: own)

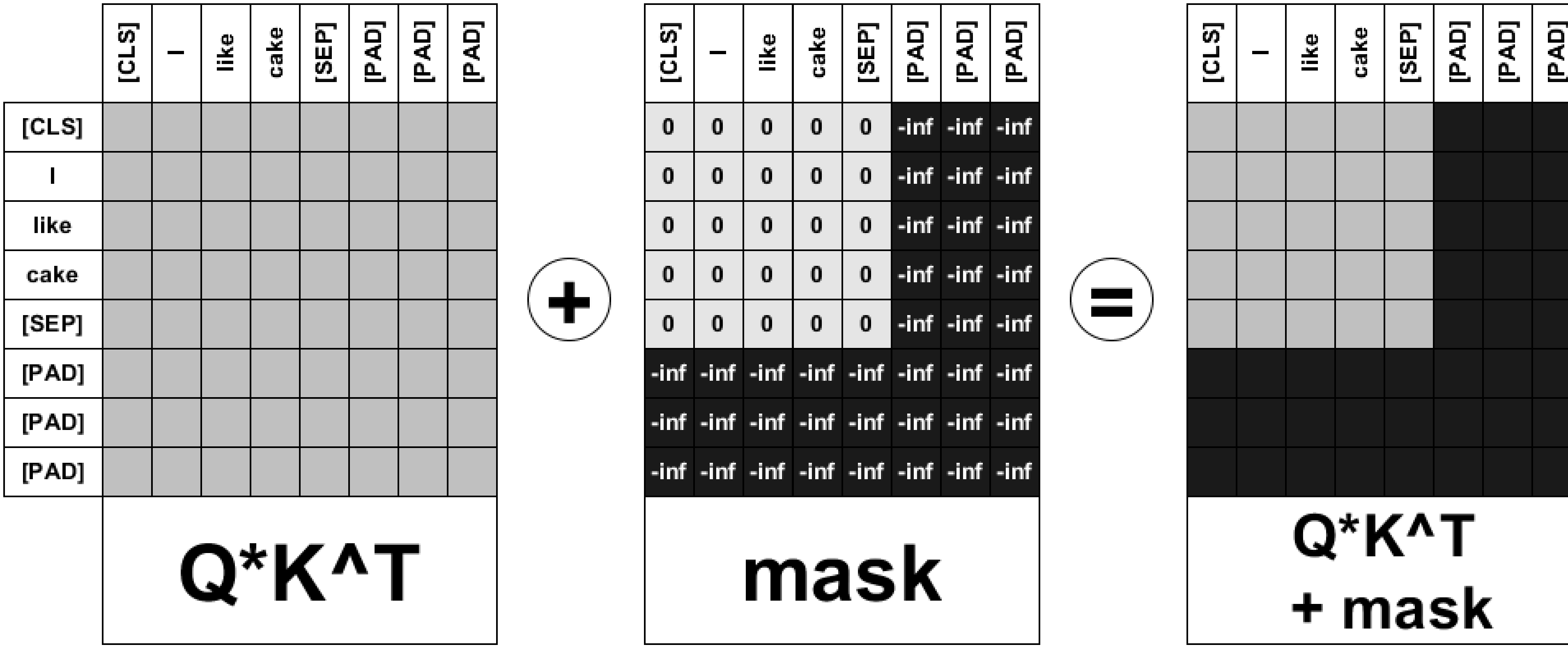


Fig. 10: Masking out the padding tokens (II) (source: own)

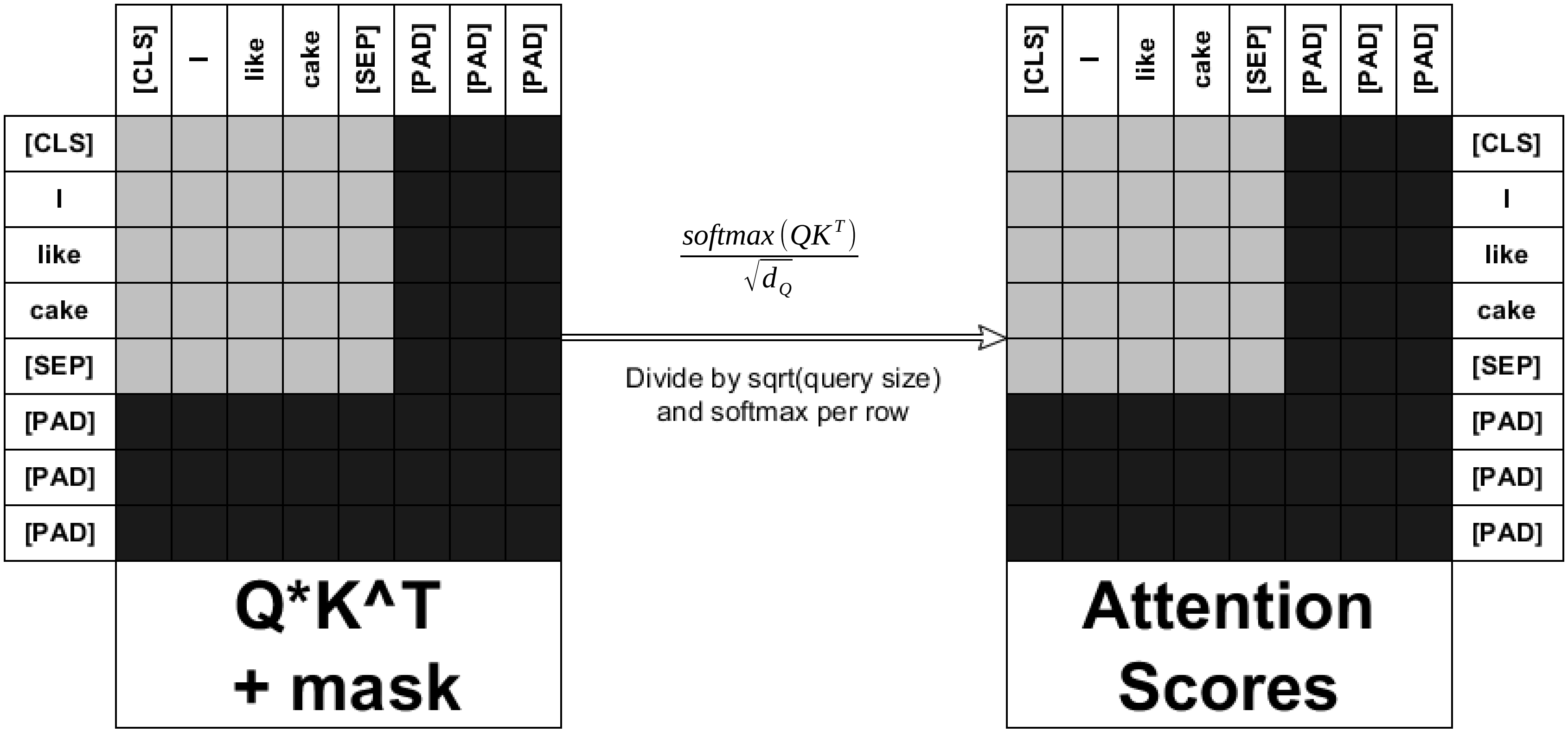


Fig. 11: Attention scores. Each row sums up to 1. (source: own)

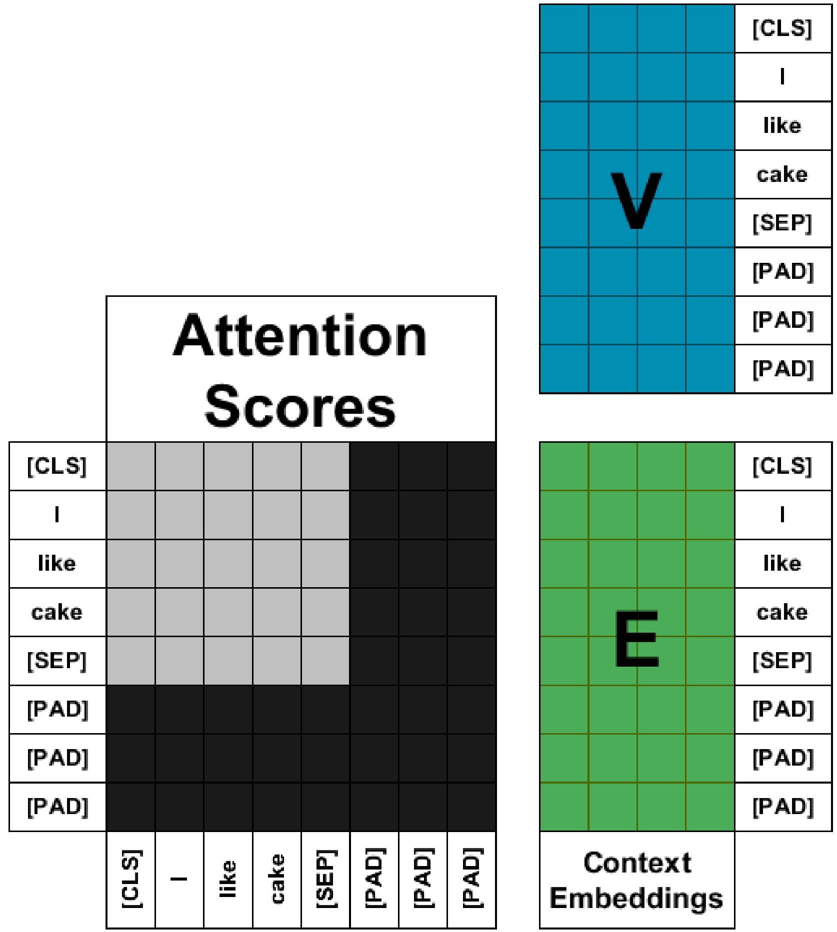


Fig. 12: Context Embeddings (source: own)

Is one set of attention weights enough?

In practice we might focus on several relationships.
One view could be “next word”.
One view could be “subject $\leftrightarrow$ object”

[CLS] I like cake [PAD] [PAD] [PAD]

For “I” we could give a lot of weight to “like” because it is the next word.

We could also give a lot of weight to “cake” since it is the object.

Solution: Have more “attention heads” to capture different relationships.
Final embedding is concatenation of all “attention heads”

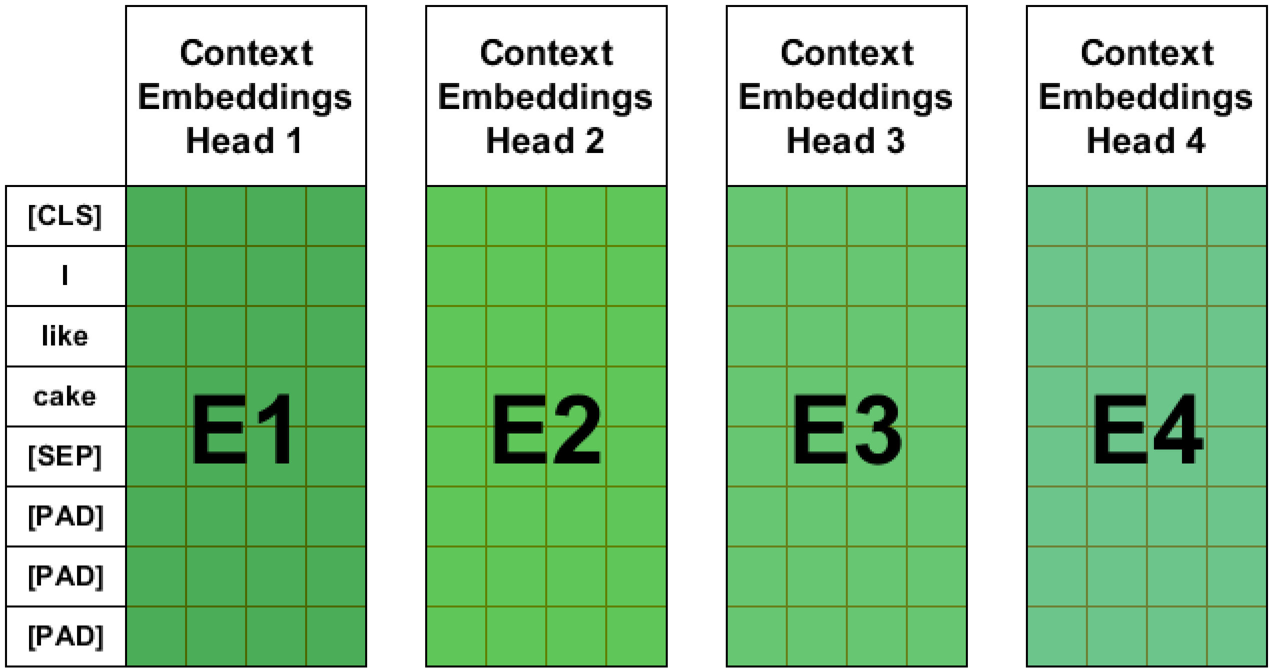


Fig. 13: Multi-Head Attention (source: own)

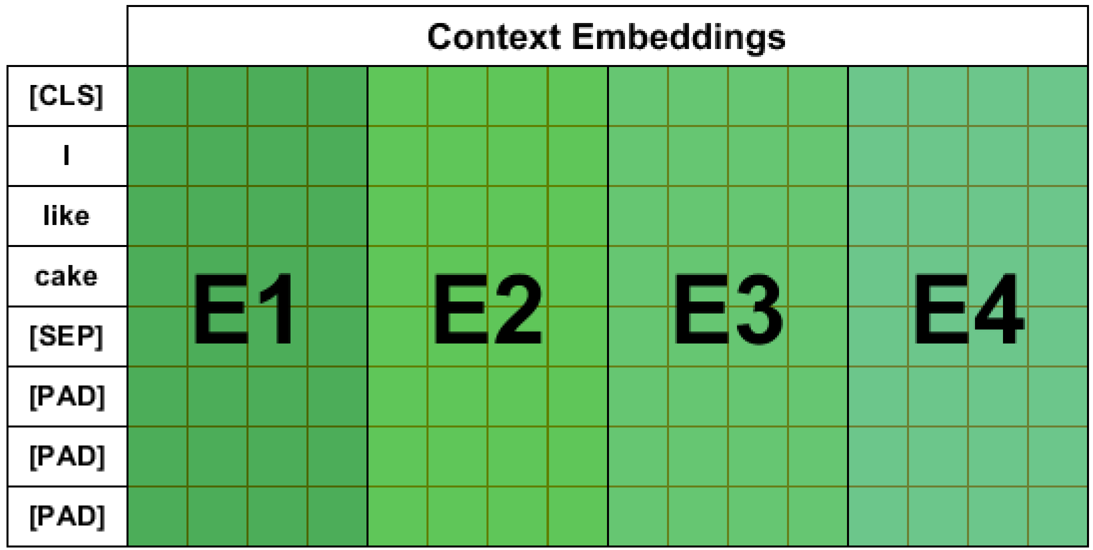


Fig. 14: Context Embeddings. Concatenate for each head (source: own)

Source: ExBERT (https://huggingface.co/spaces/exbert-project/exbert)

Source: ExBERT (https://huggingface.co/spaces/exbert-project/exbert)

Source: ExBERT (https://huggingface.co/spaces/exbert-project/exbert)

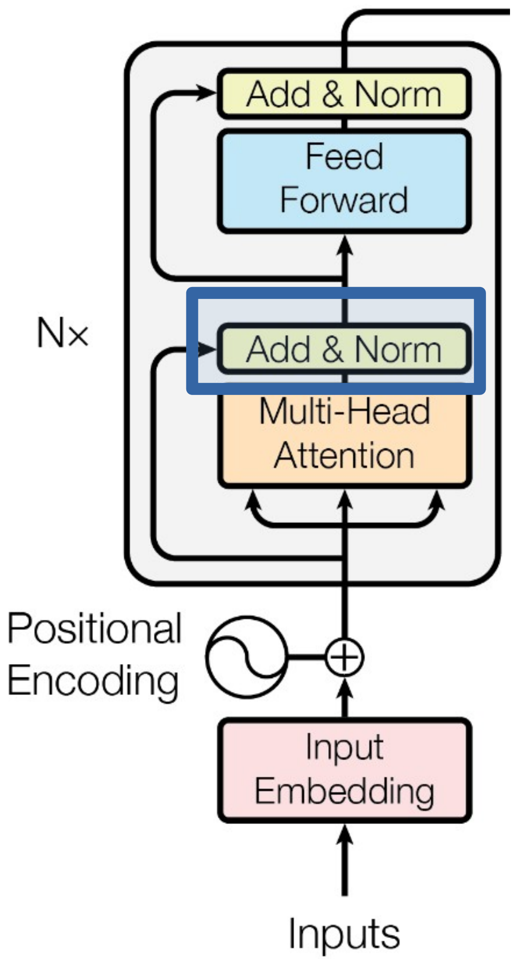


Fig. 2: Transformer Encoder (source: Attention is all you need. Vaswani et al. 2017)

Transformer Encoder

- Input is a sequence of token embeddings
  Usually of dimensionality 768 (12*64)
  For our examples we will use 16
- Model takes all tokens in the input sequence at the same time.
  We need to store position info using a Positional Encoding
- Perform self-attention
- Add the output to the context embeddings.
  Normalize to make sure the numbers in the embeddings don’t grow too much.

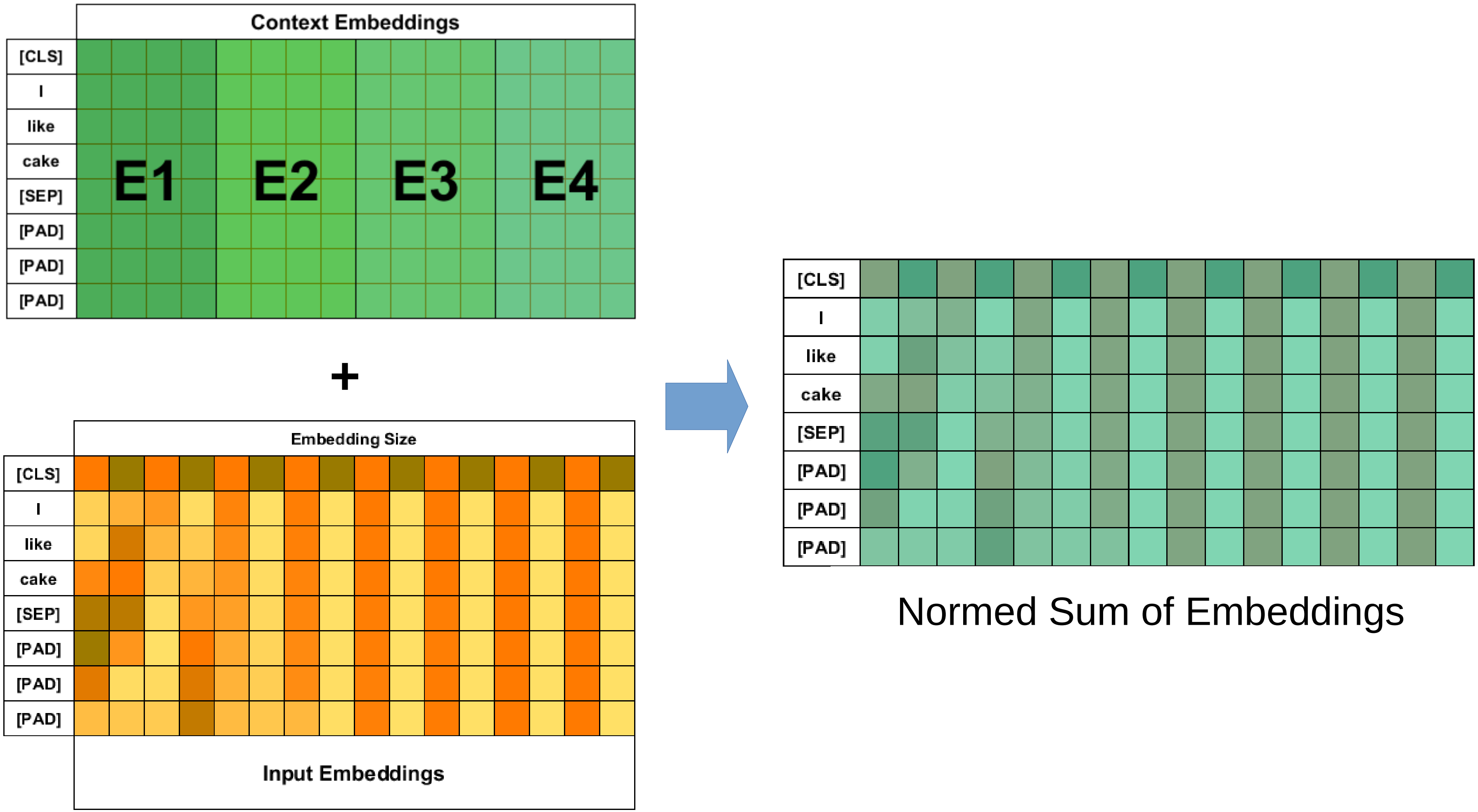


Fig. 15: Add and normalize (source: own)

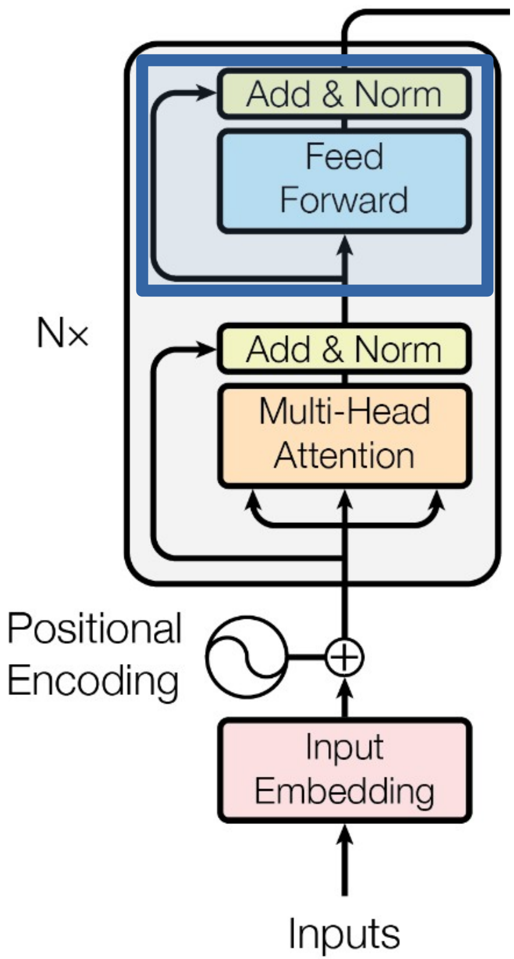


Fig. 2: Transformer Encoder (source: Attention is all you need. Vaswani et al. 2017)

Transformer Encoder

- Input is a sequence of token embeddings
  Usually of dimensionality 768 (12*64)
  For our examples we will use 16
- Model takes all tokens in the input sequence at the same time.
  We need to store position info using a Positional Encoding
- Perform self-attention
- Add the output to the context embeddings.
  Normalize to make sure the numbers in the embeddings don't grow too much.
- Feed to a feed forward layer and add and normalize again

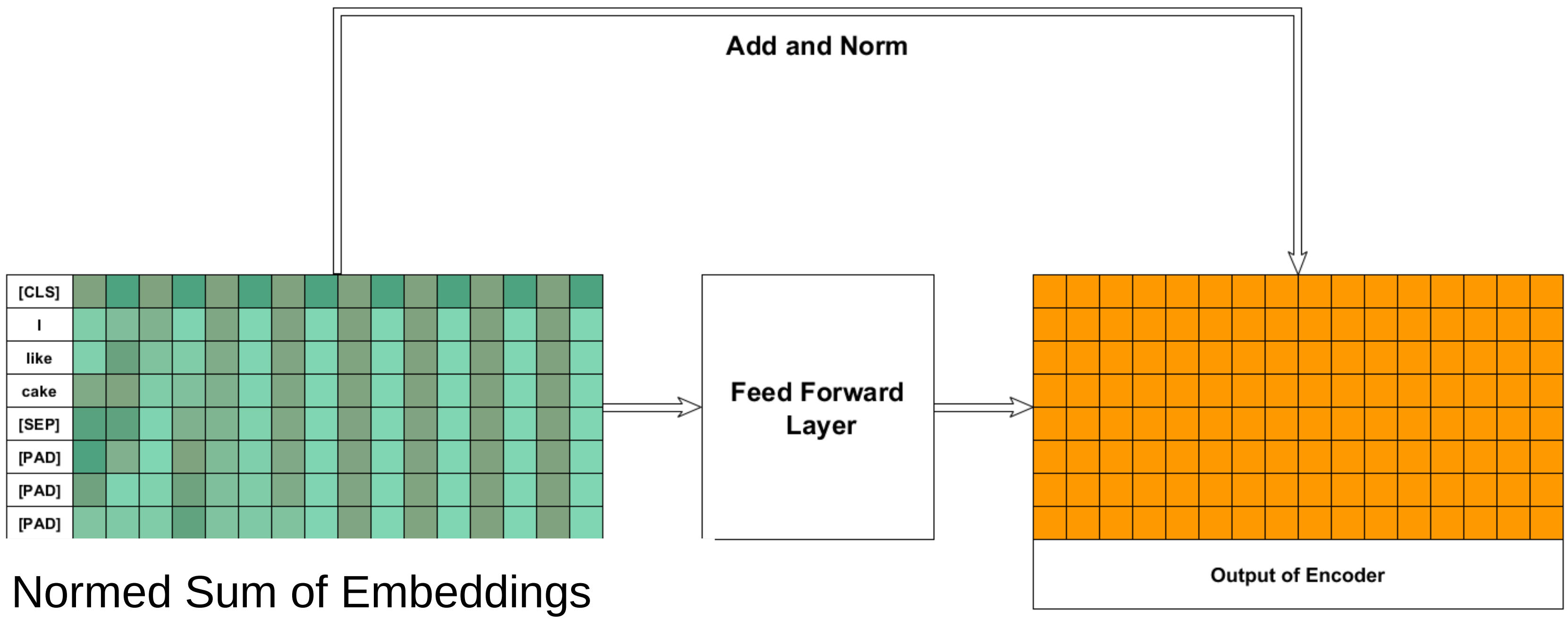


Fig. 16: Encoder Output. Feed forward layer adds non linearity to the network (source: own)

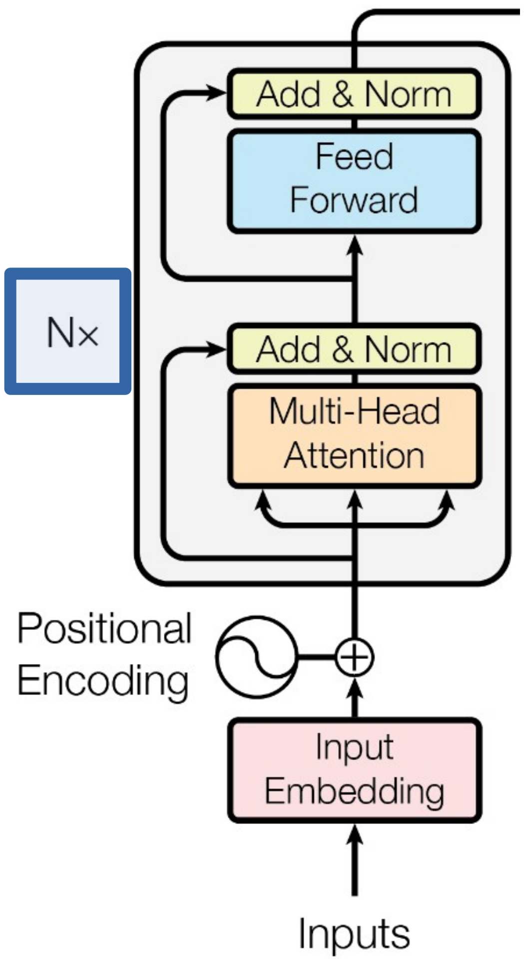


Fig. 2: Transformer Encoder (source: Attention is all you need. Vaswani et al. 2017)

Transformer Encoder

- Input is a sequence of token embeddings
  Usually of dimensionality 768 (12*64)
  For our examples we will use 16
- Model takes all tokens in the input sequence at the same time.
  We need to store position info using a Positional Encoding
- Perform self-attention
- Add the output to the context embeddings.
  Normalize to make sure the numbers in the embeddings don’t grow too much.
- Feed to a feed forward layer and add and normalize again
- Repeat N times to build deeper representations

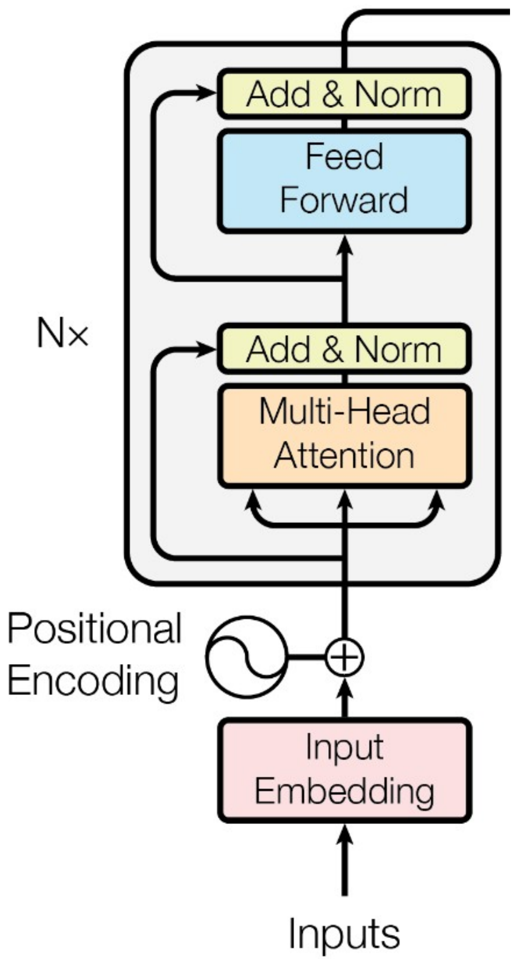


Fig. 2: Transformer Encoder (source: Attention is all you need. Vaswani et al. 2017)

Transformer Encoder

- Input is a sequence of token embeddings
  Usually of dimensionality 768 (12*64)
  For our examples we will use 16
- Model takes all tokens in the input sequence at the same time.
  We need to store position info using a Positional Encoding
- Perform self-attention
- Add the output to the context embeddings.
  Normalize to make sure the numbers in the embeddings don't grow too much.
- Feed to a feed forward layer and add and normalize again
- Repeat N times to build deeper representations

BERT
Bidirectional Encoder Representations from Transformers

- Developed at Google in 2018 by Jacob Devlin et.al.
- Builds context dependent embeddings for tokens in sentences
- Uses the Transformer architecure
- Utilizes Self-Attention

BERT

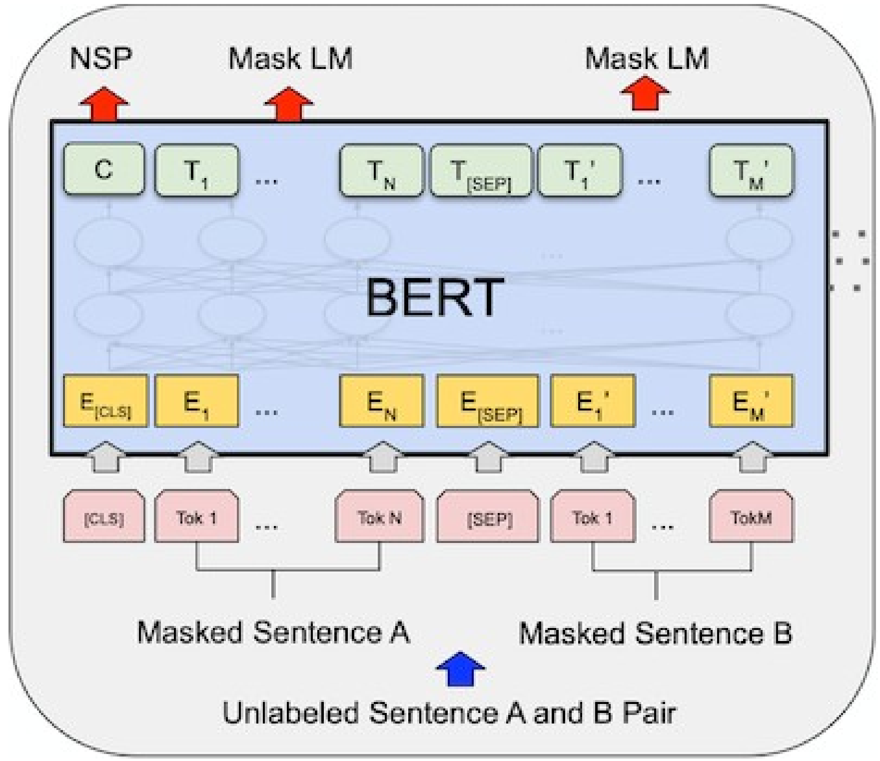


Fig. 17: BERT architecure (I) (source: own)

BERT

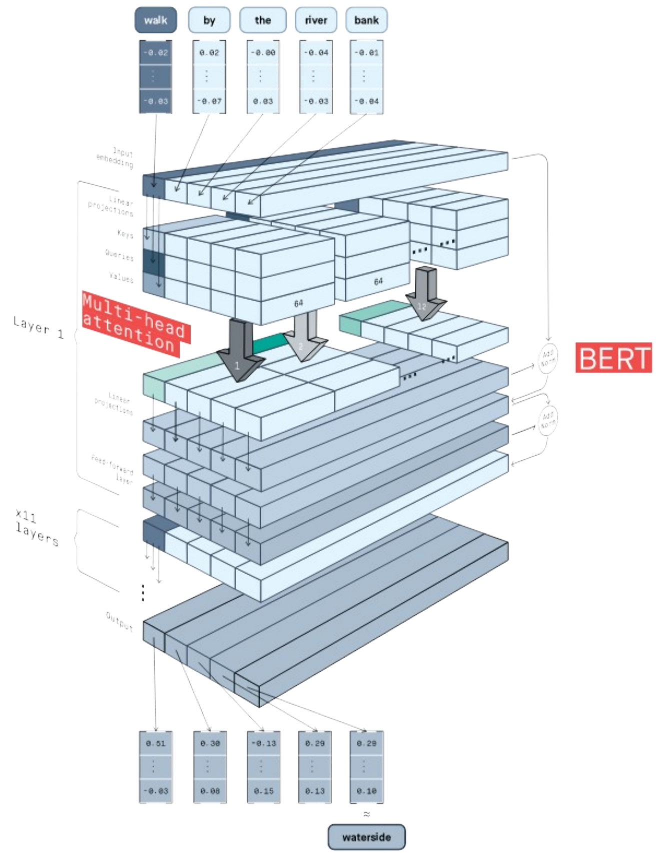


Fig. 18: BERT architecure (II) (source: https://peltarion.com/blog/data-science/self-attention-video)

BERT
Training Objective

- Trained on two tasks:
- Masked language model
- Next Sentence Prediction

BERT
Masked Language Model

- Take the final context embedding for each masked output [MASK]
- Predict by feeding this to a simple classifier that predicts the token that was masked out
- Next Sentence Prediction: Given two sentences, predict if they are in the correct order. Add a segment embedding to the sentences, one indicating sentence one, one sentences two.

  Predict by feeding the output of the [CLS] token to a simple binary classifier (0 → sentences in order, 1 → sentences out of order)

BERT
Next Sentence Prediction

- Given two sentences, predict if they are in the correct order.
- Add a segment embedding to the sentences, one indicating sentence one, one sentences two.
- Input Embeddings are now:
token embedding + position embedding + segment embedding
- Predict by feeding the output of the [CLS] token to a simple binary classifier (0 → sentences in order, 1 → sentences out of order)

BERT
Fine Tuning

- So far we only trained a masked language model with next sentence prediction.
- Take this pretrained model and fine tune it on tasks such as:
  - Sentiment Analysis
  - Question Answering
  - Text Classification
- For sentiment analysis we take the output of the [CLS] token and train a classifier on it (e.g. 0 → negative, 1 → positive)