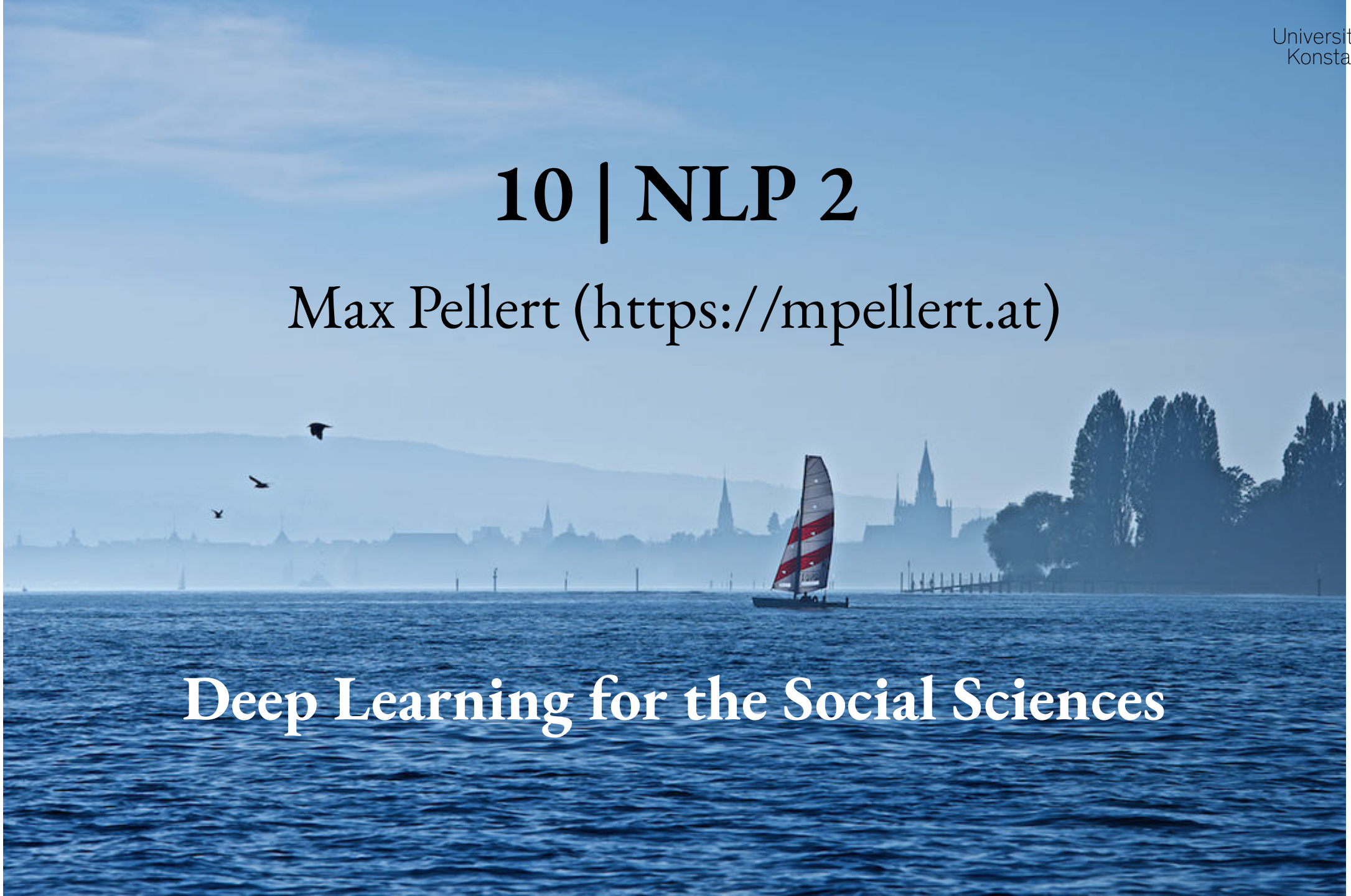


Universität Konstanz

10 | NLP 2

Max Pellert (https://mpellert.at)

Deep Learning for the Social Sciences

New development: attention

Overcoming the bottleneck problem?

Universität Konstanz

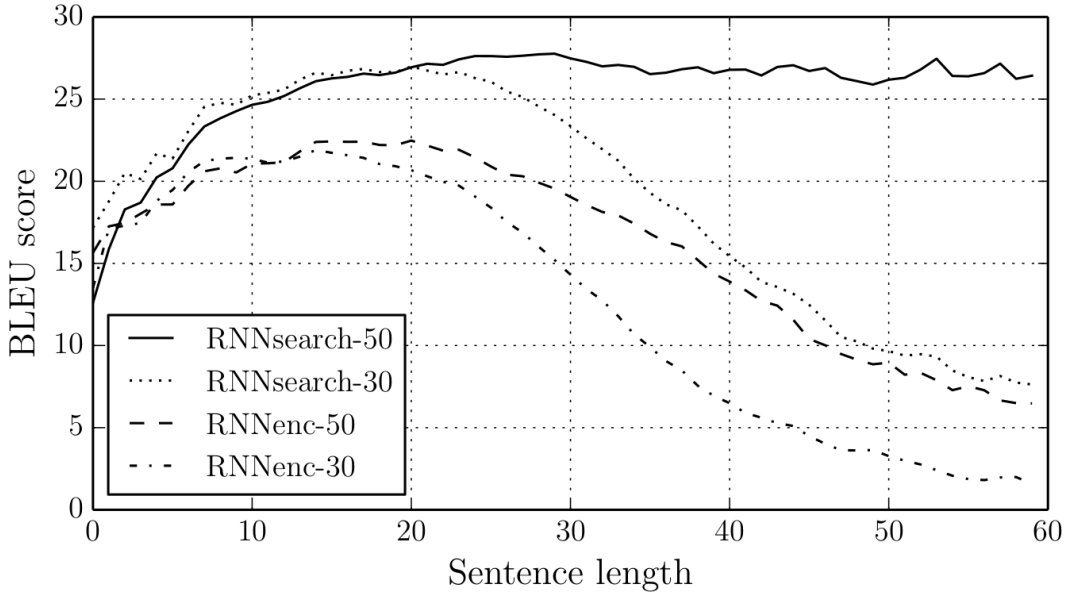


Figure 2: The BLEU scores of the generated translations on the test set with respect to the lengths of the sentences. The results are on the full test set which includes sentences having unknown words to the models.

Introduction of the concept of **attention** as an additional mechanism, but still using an RNN Encoder–Decoder

Bahdanau, D., Cho, K., & Bengio, Y. (2016). Neural Machine Translation by Jointly Learning to Align and Translate (arXiv:1409.0473). arXiv. http://arxiv.org/abs/1409.0473

Development of a novel architecture: the transformer

Attention Is All You Need

Ashish Vaswani[*]
Google Brain
avaswani@google.com

Noam Shazeer[*]
Google Brain
noam@google.com

Niki Parmar[*]
Google Research
nikip@google.com

Jakob Uszkoreit[*]
Google Research
usz@google.com

Llion Jones[*]
Google Research
llion@google.com

Aidan N. Gomez[*] [†]
University of Toronto
aidan@cs.toronto.edu

Łukasz Kaiser[*]
Google Brain
lukaszkaiser@google.com

Illia Polosukhin[*] [‡]
illia.polosukhin@gmail.com

Abstract

The dominant sequence transduction models are based on complex recurrent or convolutional neural networks that include an encoder and a decoder. The best performing models also connect the encoder and decoder through an attention mechanism. We propose a new simple network architecture, the Transformer, based solely on attention mechanisms, dispensing with recurrence and convolutions entirely. Experiments on two machine translation tasks show these models to be superior in quality while being more parallelizable and requiring significantly less time to train. Our model achieves 28.4 BLEU on the WMT 2014 English-to-German translation task, improving over the existing best results, including ensembles, by over 2 BLEU. On the WMT 2014 English-to-French translation task, our model establishes a new single-model state-of-the-art BLEU score of 41.8 after training for 3.5 days on eight GPUs, a small fraction of the training costs of the best models from the literature. We show that the Transformer generalizes well to other tasks by applying it successfully to English constituency parsing both with large and limited training data.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., & Polosukhin, I. (2017). Attention Is All You Need. arXiv:1706.03762 [Cs]. http://arxiv.org/abs/1706.03762

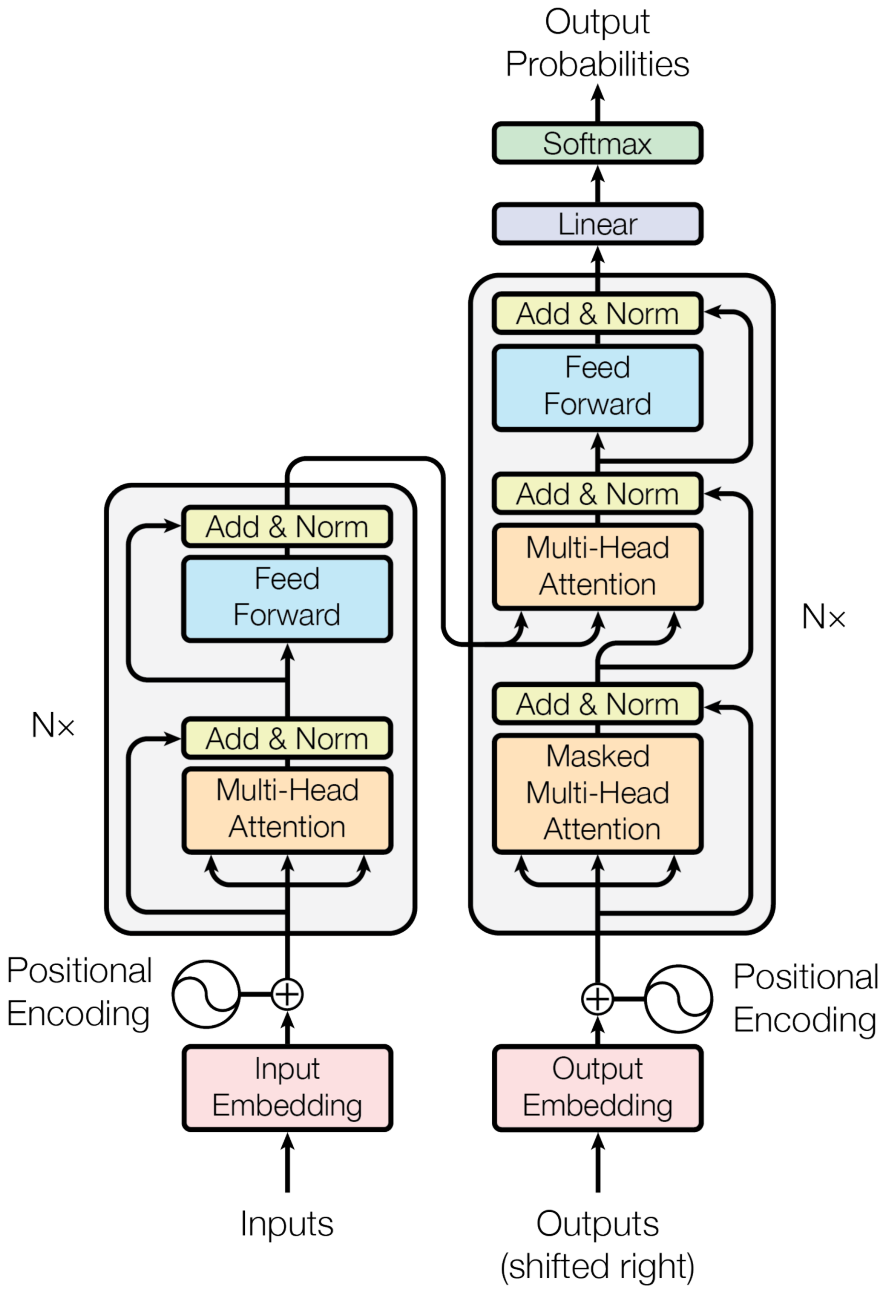
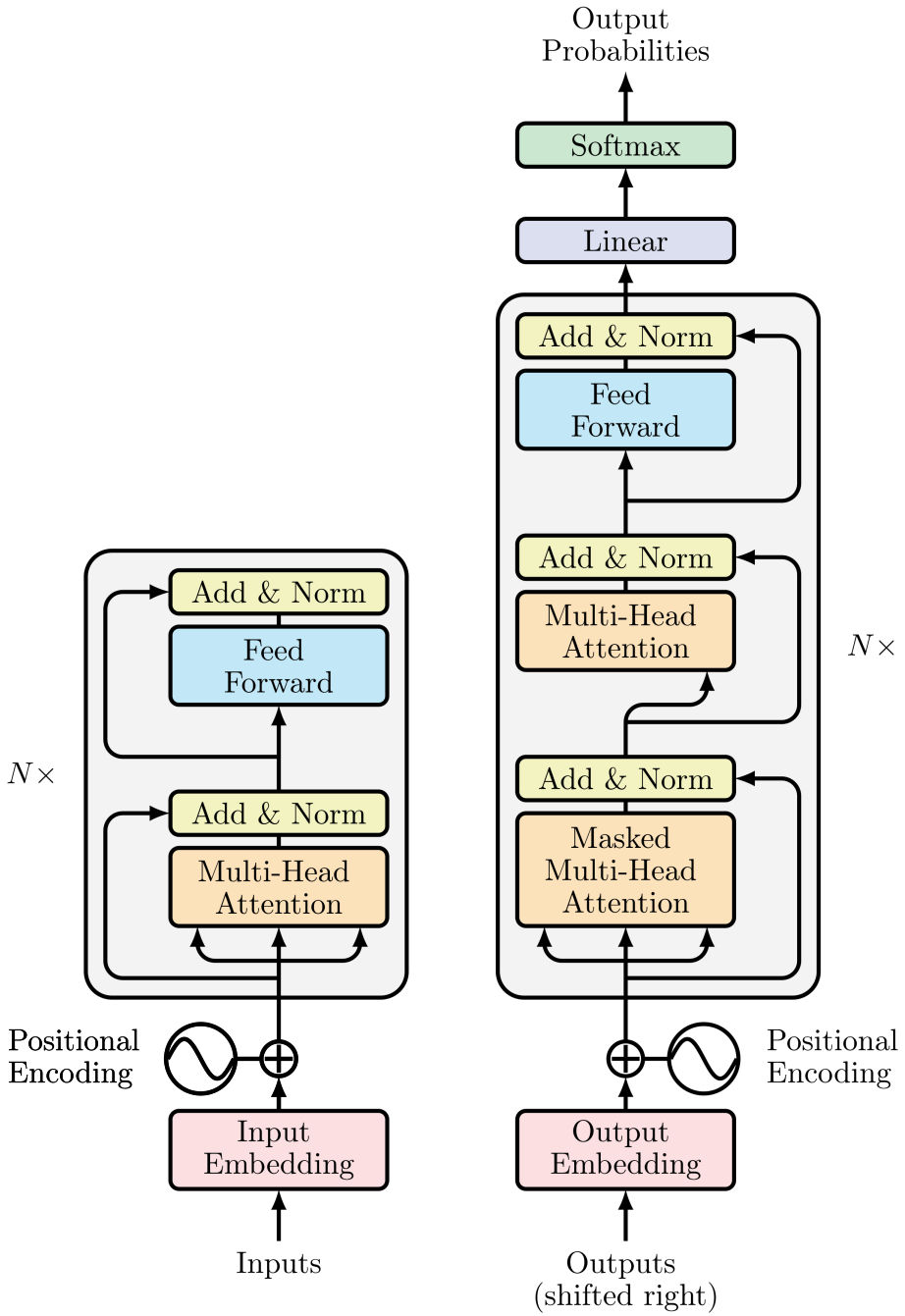
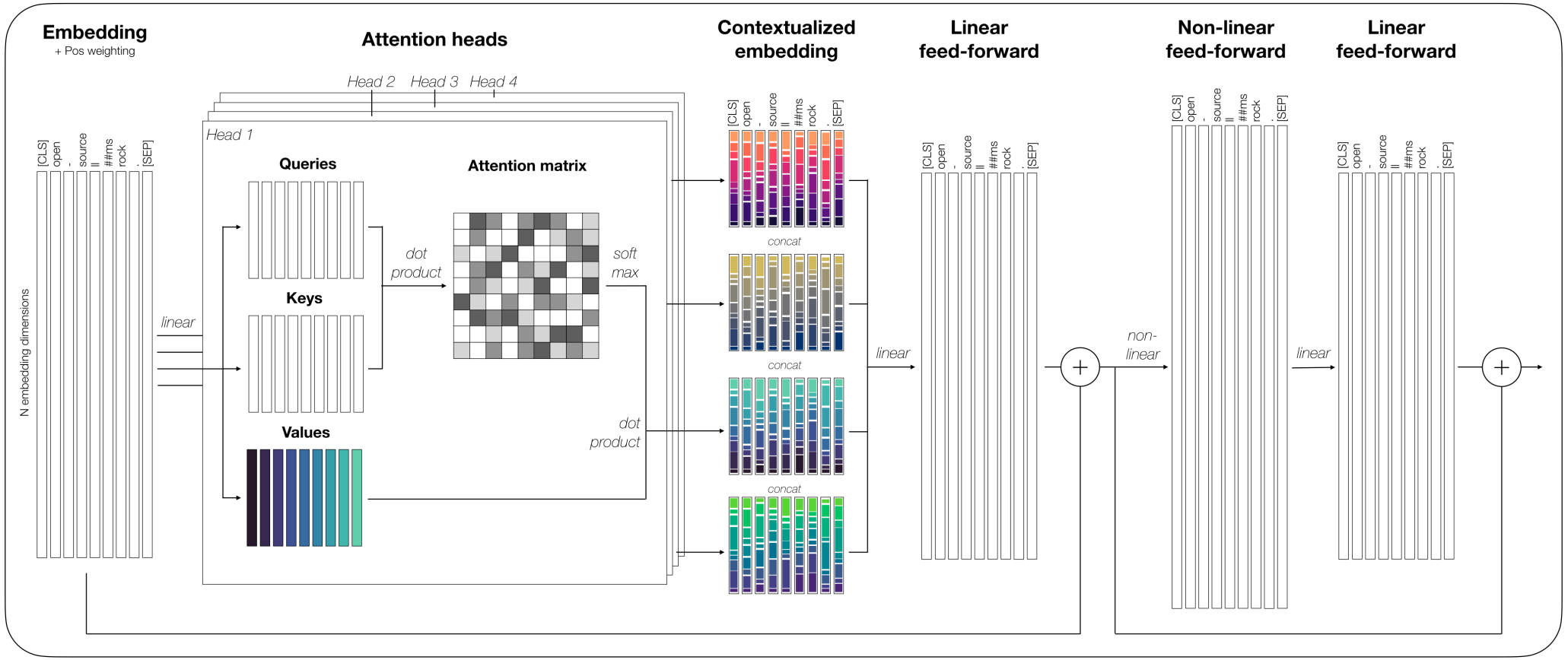


Figure 1: The Transformer - model architecture.


Output
Probabilities
Softmax
Linear
Add & Norm
Feed
Forward
Add & Norm
Multi-Head
Attention
Add & Norm
Masked
Multi-Head
Attention
N×
Add & Norm
Feed
Forward
Add & Norm
Multi-Head
Attention
N×
Positional
Encoding
Positional
Encoding
Input
Embedding
Output
Embedding
Inputs
Outputs
(shifted right)



Hussain, Z., Binz, M., Mata, R., & Wulff, D. U. (2023). A tutorial on open-source large language models for behavioral science. https://doi.org/10.31234/osf.io/f7stn

Let's work through all parts of the transformer architecture

Illustrating Attention

First, let's introduce some terminology (taken from the field of information retrieval)

Consider the problem of choosing which movie to watch in an online movie streaming service:

Associate each movie with a list of attributes describing things such as the genre (comedy, action, etc.), the names of the leading actors, the length of the movie and so on

You as a user could then search through the catalogue to find a movie that matches your preferences

We could encode the attributes of each movie in a vector called the **key**.

Illustrating Attention

The corresponding movie file itself is called a **value**

Similarly, you could then provide your own personal vector of values for the desired attributes that you care about in a movie, which we call the **query**

The movie service could then compare the query vector with all the key vectors to find the best match and send the corresponding movie to you in the form of the value file

You were **“attending”** to the particular movie whose key most closely matches your query

Illustrating Attention

This would be considered a form of *hard* attention in which a single value vector (= movie in this case) is returned

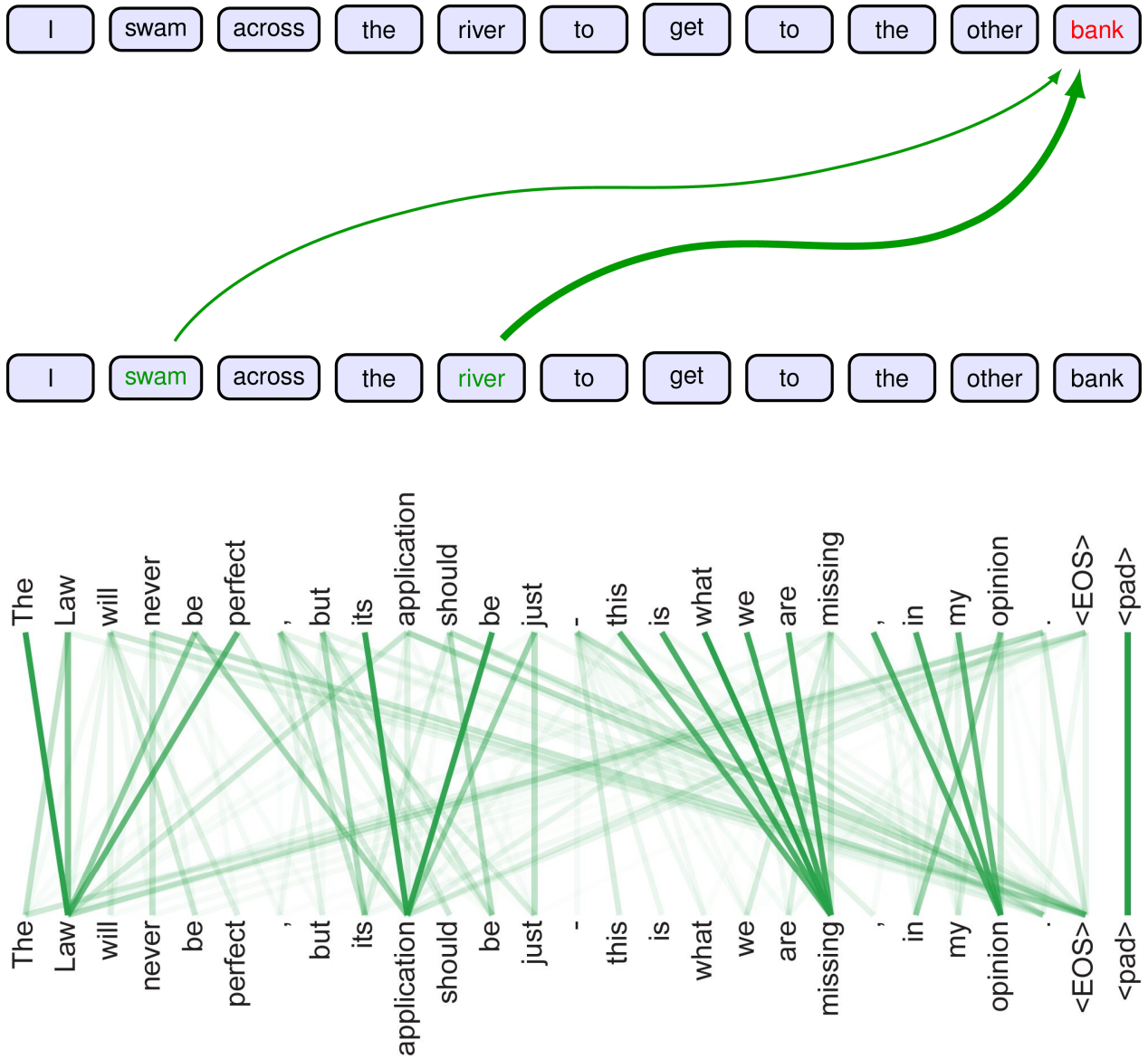
For what follows about the transformer, we generalize this to *soft* attention

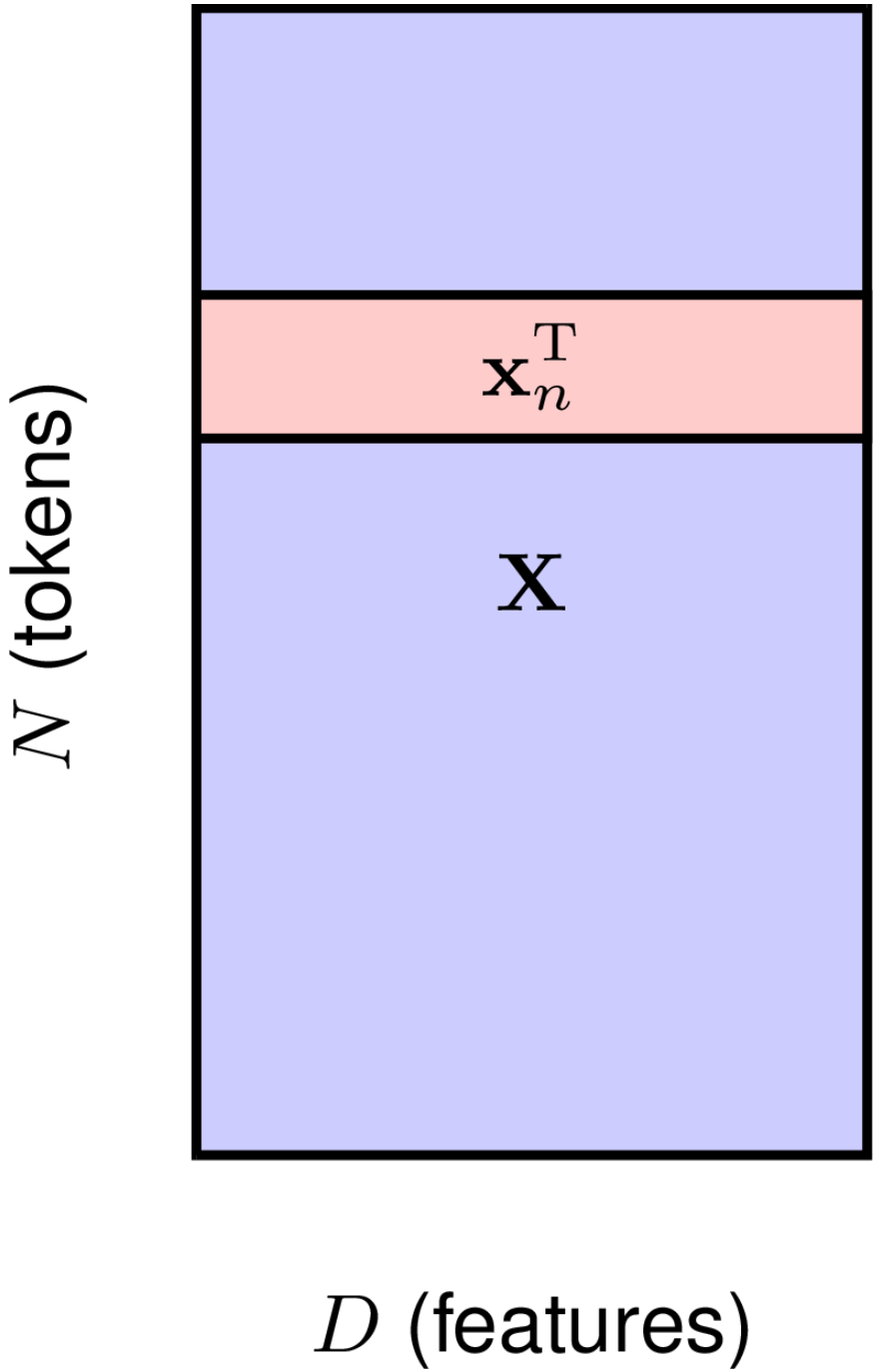
We use continuous variables to measure the degree of match between queries and keys and we then use these variables to weight the influence of the value vectors

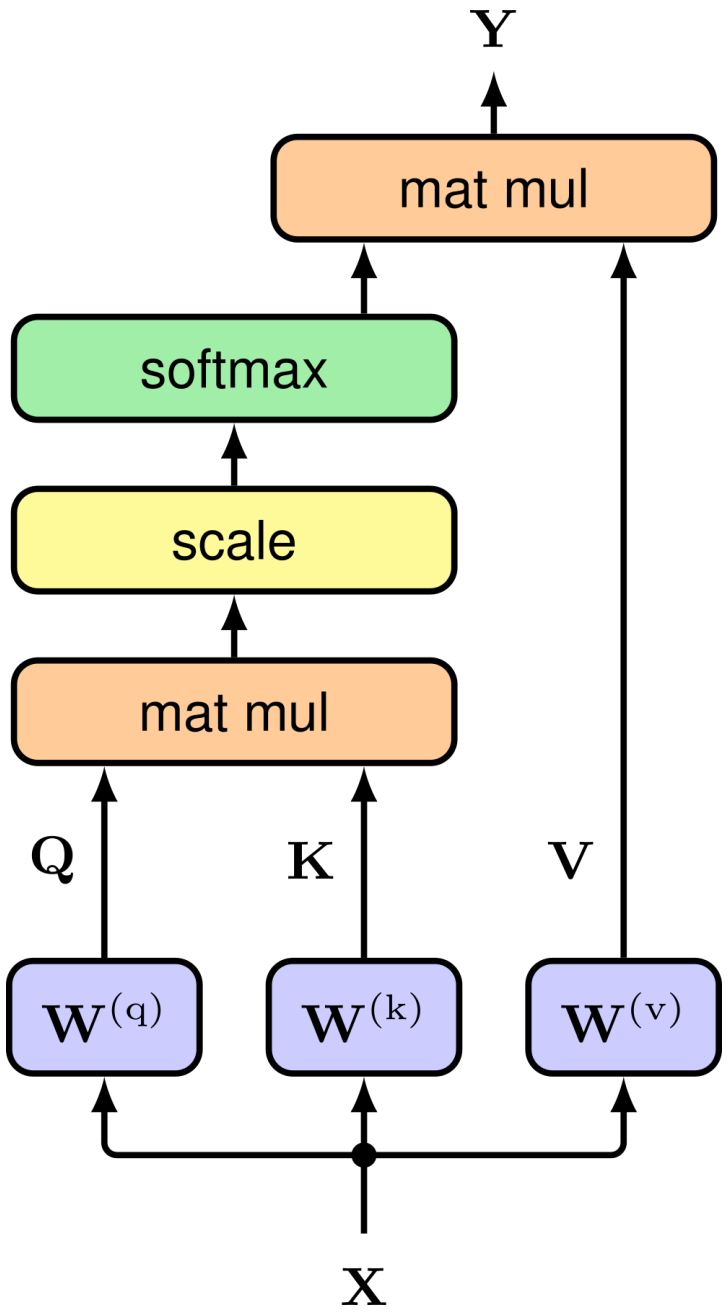
For the attention mechanism in transformers, values, keys and queries refer not to movies, their traits and user preferences but to the tokens in the input sequence

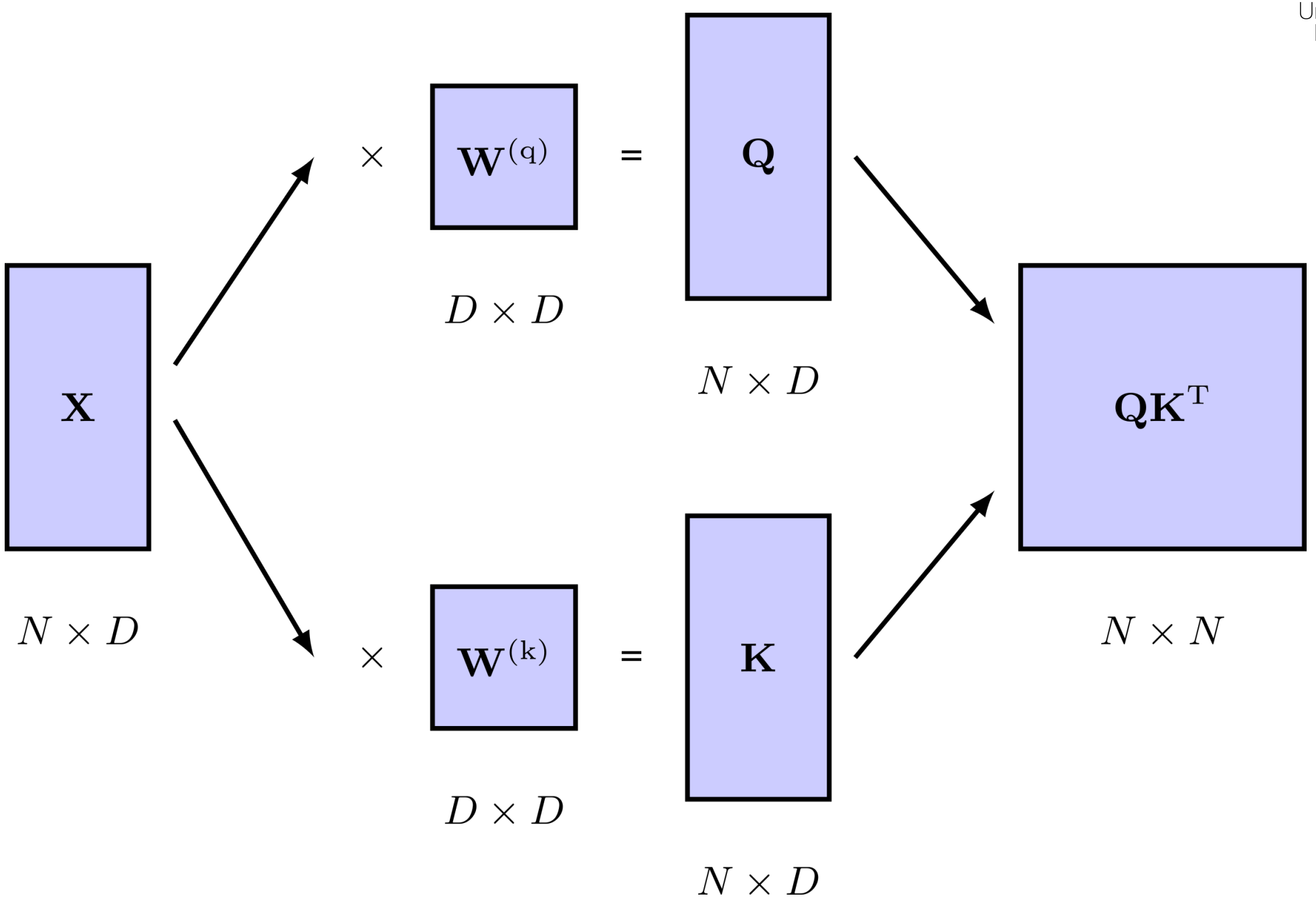
Attention mechanism in transformers

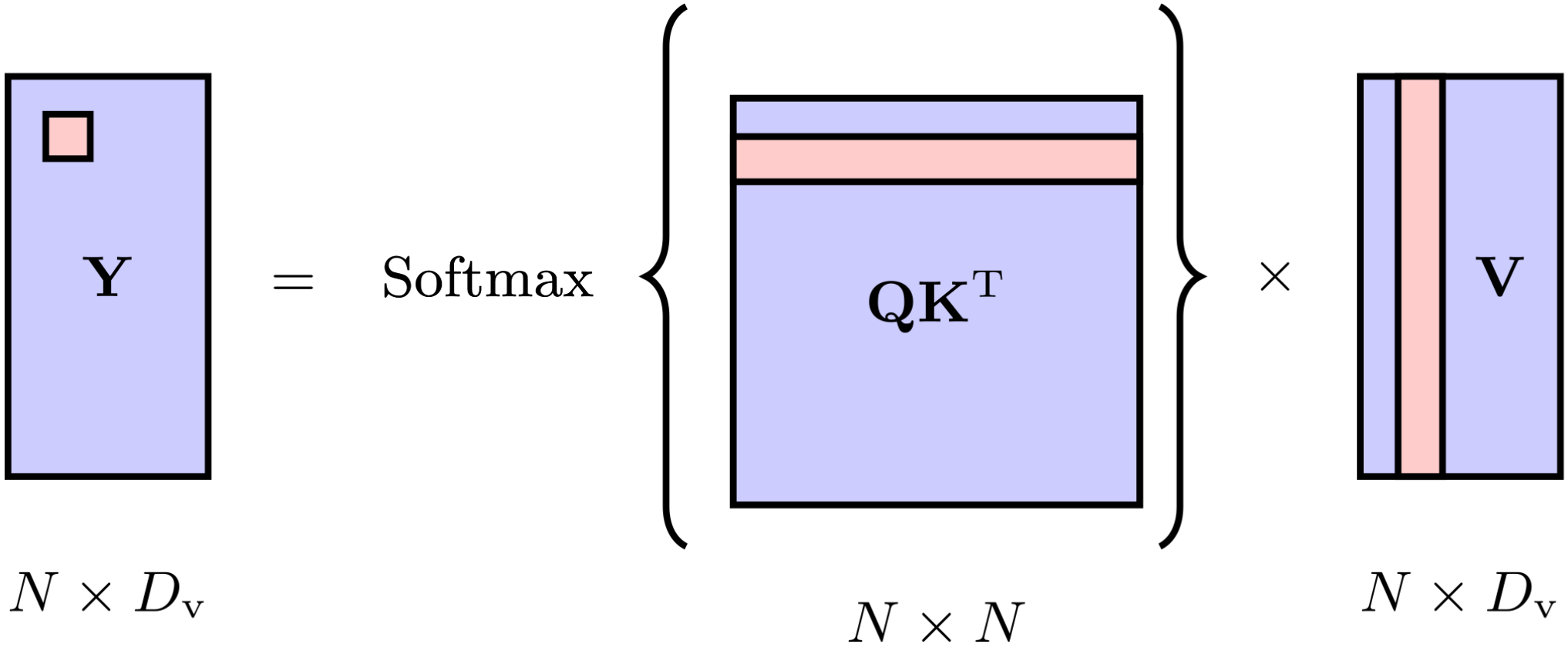
Universität Konstanz




$\mathbf{x}_n^{\mathrm{T}}$
$\mathbf{X}$
N (tokens)
D (features)


Y
mat mul
softmax
scale
mat mul
Q
K
V
W(q)
W(k)
W(v)
X


$\mathbf{X}$
$N \times D$
$\times$
$\mathbf{W}^{(\mathrm{q})}$
$D \times D$
=
$\mathbf{Q}$
$N \times D$
$\times$
$\mathbf{W}^{(\mathrm{k})}$
$D \times D$
=
$\mathbf{K}$
$N \times D$
$\mathbf{QK}^{\mathrm{T}}$
$N \times N$

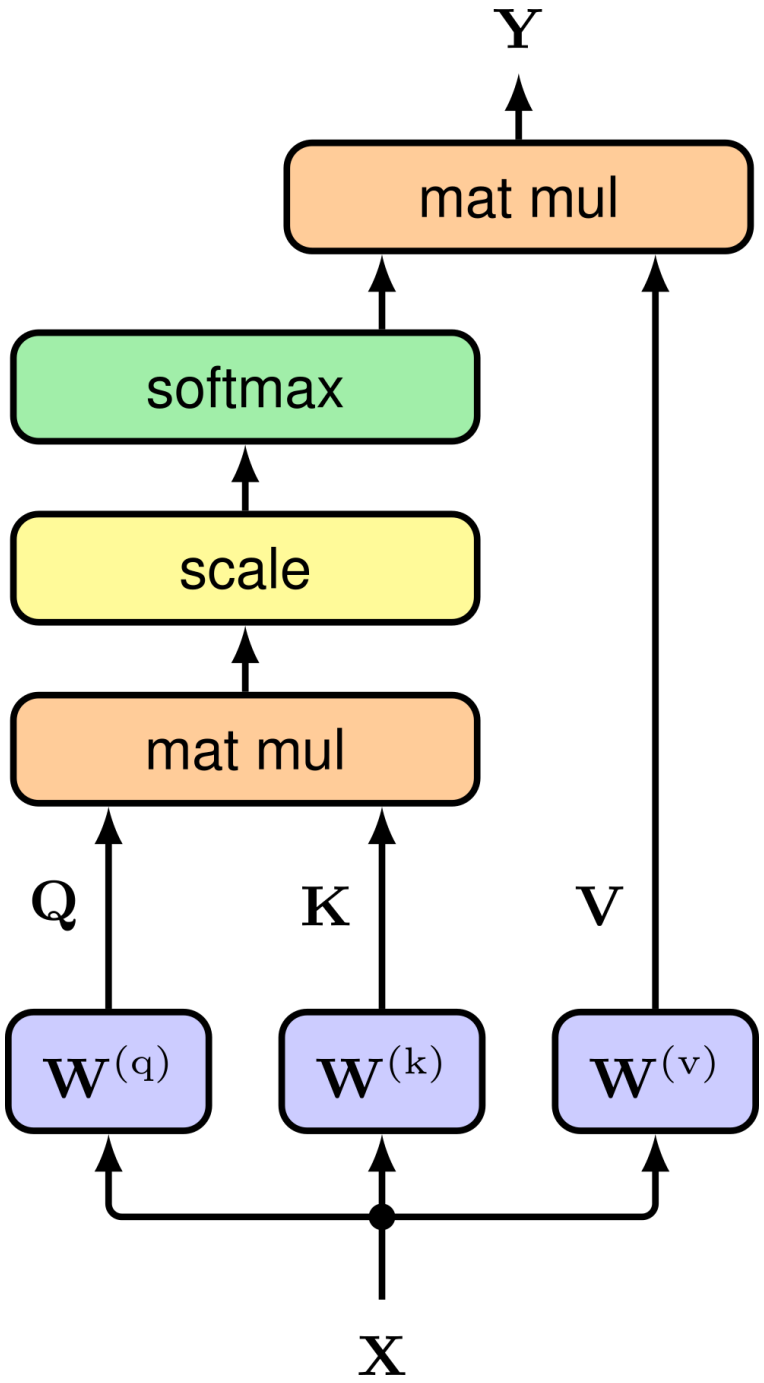

Y
=
Softmax
QK^T
×
V
N × D_v
N × N
N × D_v

Self-Attention

This process is called **self-attention** because we are using the same sequence to determine the queries, keys, and values

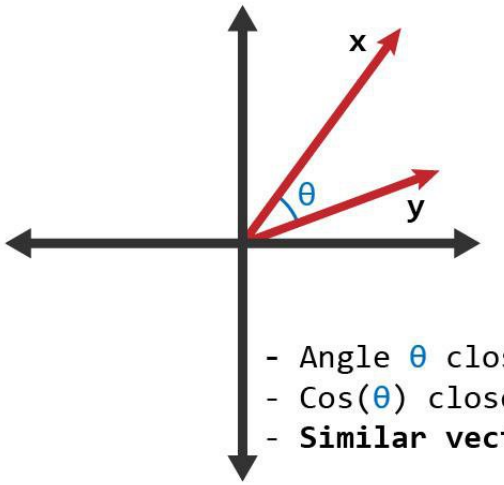
We will briefly talk variants of this attention mechanism later, for example **cross-attention**

Also, because the measure of similarity between query and key vectors is given by a dot product, this is known as **dot-product self-attention**

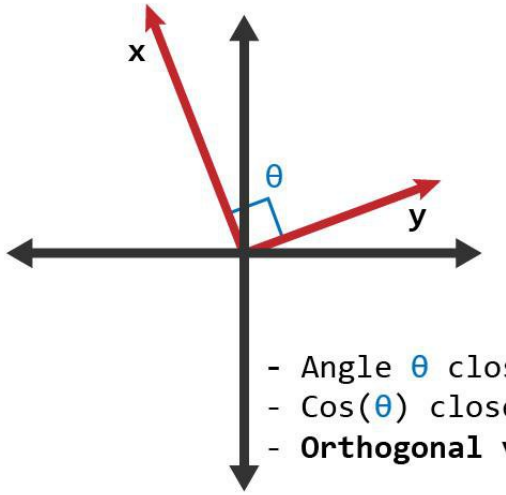

Y
mat mul
softmax
scale
mat mul
Q
K
V
W(q)
W(k)
W(v)
X

Dot product = measure of similarity

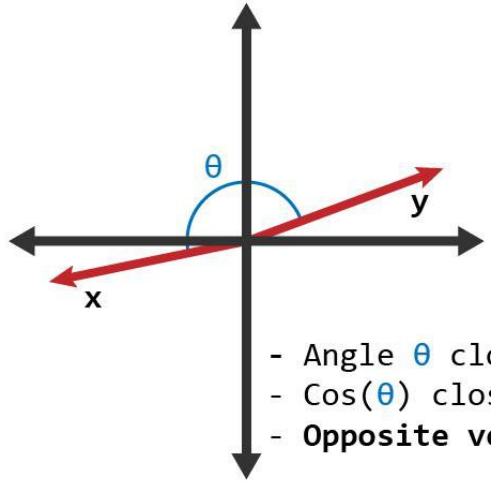
$$\mathbf{x}^T\mathbf{y} = ||\mathbf{x}|| \cdot ||\mathbf{y}|| \cdot \cos\theta$$



- Angle θ close to 0
- Cos(θ) close to 1
- **Similar vectors**



- Angle θ close to 90
- Cos(θ) close to 0
- **Orthogonal vectors**



- Angle θ close to 180
- Cos(θ) close to -1
- **Opposite vectors**

Scaled Dot-Product Self-Attention

Scaling is one more refinement that we have to do

The gradients of the softmax function become exponentially small for inputs of high magnitude (resulting in something very close to hard attention)

To prevent this from happening, we can re-scale the product of the query and key vectors before applying the softmax function

There are many possible ways to do that, let's consider one

Scaled Dot-Product Self-Attention

If the elements of the query and key vectors were all independent random numbers with zero mean and unit variance, then the variance of the dot product would be D_k

We can therefore normalize the argument to the softmax using the standard deviation given by the square root of D_k, therefore the output of the attention layer takes the final form:

$$\mathbf{Y} = \text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) \equiv \text{Softmax}\left(\frac{\mathbf{Q}\mathbf{K}^{\mathrm{T}}}{\sqrt{D_{\mathrm{k}}}}\right)\mathbf{V}$$

Multi-head attention

The attention layer described so far allows the output vectors to attend to data-dependent patterns of input vectors and is called an attention head

However, there might be multiple patterns of attention that are relevant at the same time

For example, in natural language, some patterns might be relevant to tense whereas others might be associated with vocabulary

A single attention head can lead to averaging over these effects

Instead, let's use multiple attention heads in parallel

Multi-head attention

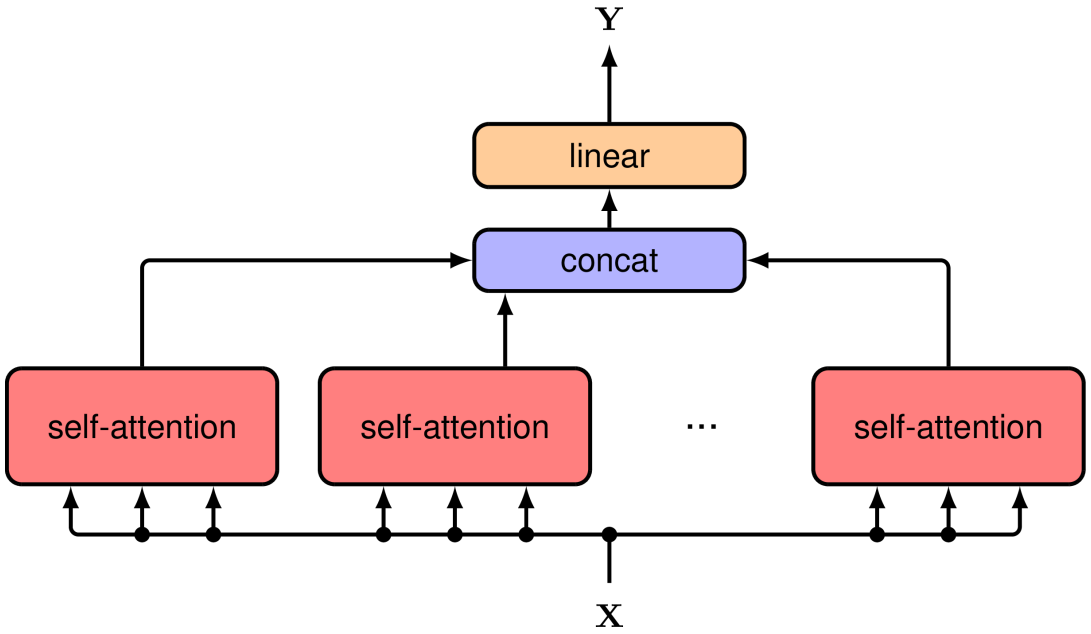
These are identically structured copies of the single head, with independent learnable parameters that govern the calculation of the query, key, and value matrices

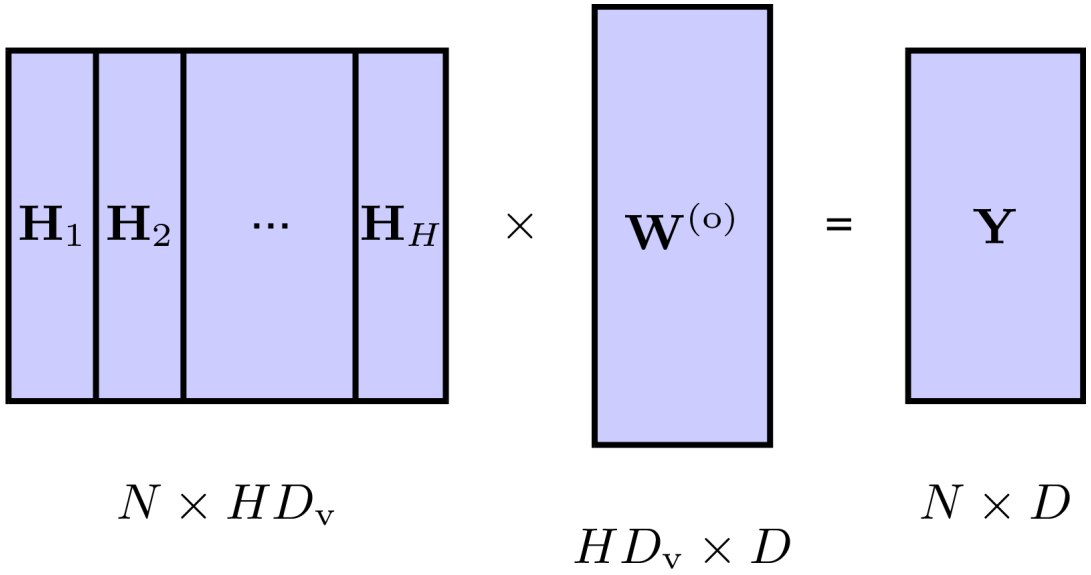
Suppose we have H heads indexed by $h = 1, \ldots, H$ of the form

$$H_h = \text{Attention}(Q_h, K_h, V_h)$$

The heads are first concatenated into a single matrix, and the result is then linearly transformed using a matrix W(o) to give a combined output in the form:

$$Y(X) = \text{Concat}[H_1, \ldots, H_H]W^{(o)}$$


Y
linear
concat
self-attention
self-attention
···
self-attention
X


$\mathbf{H}_1$ $\mathbf{H}_2$ ··· $\mathbf{H}_H$
×
$\mathbf{W}^{(\mathrm{o})}$
=
$\mathbf{Y}$
$N \times HD_{\mathrm{v}}$
$HD_{\mathrm{v}} \times D$
$N \times D$

Layer normalization

To improve training, we can introduce residual connections that bypass the multi-head structure (this is visualized with links in the overview on the transformer architecture)

This is then followed by layer normalization, which further improves training efficiency

To enhance flexibility by introducing more non-linearity, we also add a multilayer perceptron, for example a two-layer fully connected network with ReLU hidden units.

The resulting transformation can be written as:

$$Z = \mathrm{LayerNorm}[\mathrm{MLP}(Y(X)) + X]$$

Layer normalization

Let's compute layer normalization statistics over all the hidden units in the same layer:

H denotes the number of hidden units in a layer; under layer normalization, all the hidden units in a layer share the same normalization terms μ and σ:

$$\mu^l = \frac{1}{H}\sum_{i=1}^{H} a_i^l$$

$$\sigma^l = \sqrt{\frac{1}{H}\sum_{i=1}^{H}(a_i^l - \mu_i^l)}$$

We just substract μ and divide by σ in order to have a standard normal distribution for the layer (mean of 0 and variance of 1)

Putting it all together...

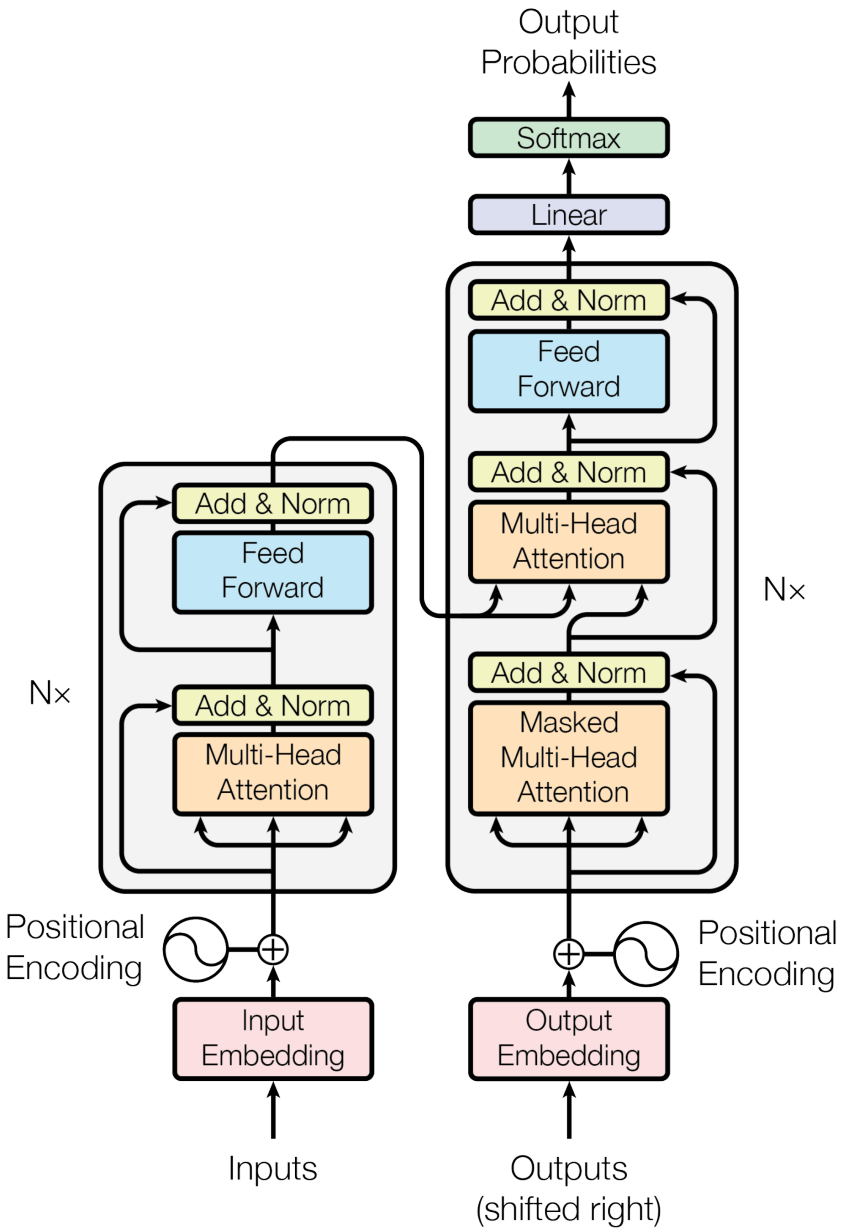
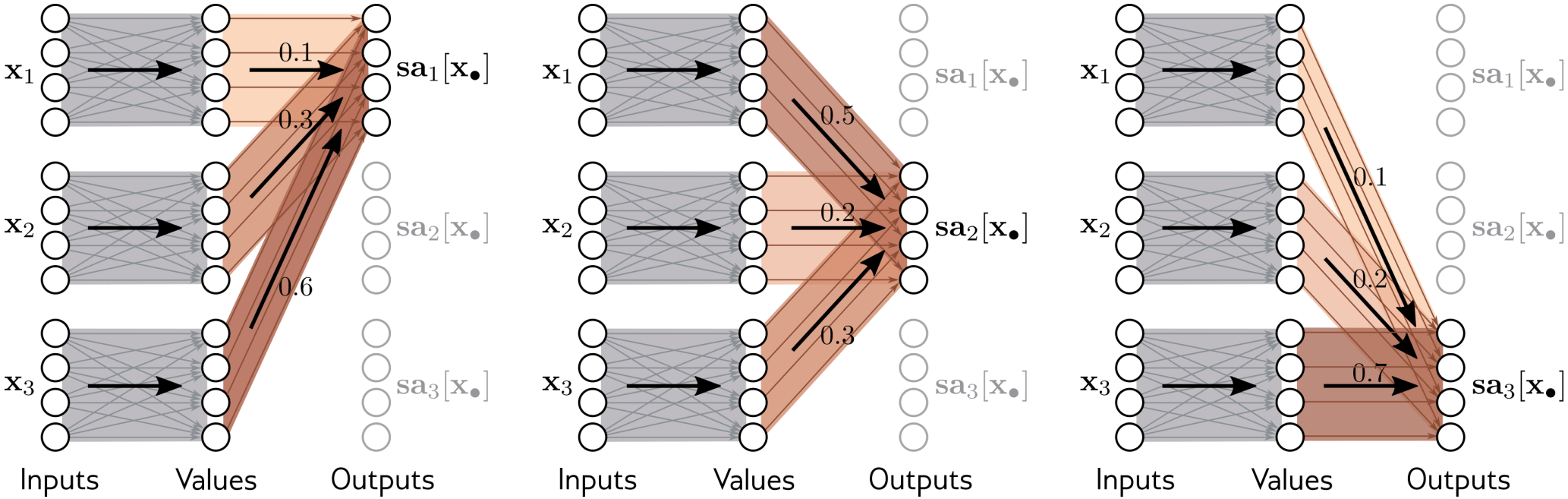


Figure 1: The Transformer - model architecture.

In a typical transformer there are multiple such multihead attention layers together with the other parts as described stacked on top of each other

The layers generally have identical structures, although there is no sharing of weights and biases between different layers.

Attention as routing



The n^{th} output $\mathbf{sa}_n[\mathbf{x}_1, \ldots, \mathbf{x}_N]$ is a weighted sum of all the values $\mathbf{v}_1, \ldots, \mathbf{v}_N$:

$$\mathbf{sa}_n[\mathbf{x}_1, \ldots, \mathbf{x}_N] = \sum_{m=1}^{N} a[\mathbf{x}_m, \mathbf{x}_n]\mathbf{v}_m$$

Word position matters

The woman ate the fish

vs.

The fish ate the woman

The man ate the racoon

vs.

The racoon ate the man

Positional Encodings

The last thing that is missing

This part is actually conceptually tough and not easy to understand, best is just to follow along and to accept that this is working

In transformers, the matrices $W_h^{(q)}$, $W_h^{(k)}$ and $W_h^{(v)}$ are shared across the input tokens, as is the subsequent neural network

From this follows that the transformer has the following property:

Permuting the order of the input tokens, i.e. the rows of X, results in the same permutation of the rows of the output matrix $\tilde{X}$

Positional Encodings

This equivariance is actually helpful, it facilitates the massively parallel processing of the transformer and also allows the network to learn long-range dependencies just as effectively as short-range dependencies

But that becomes a major limitation when we consider sequential data, such as the words in a natural language, because the representation learned by a transformer will be *independent of the input token ordering*

We have to encode information on word positions

We aim to encode the token order in the data itself in order not to complicate the architecture further

Positional Encodings

We will therefore construct a position encoding vector $\boldsymbol{r}_n$ associated with each input position n and then combine this with the associated input token embedding $\boldsymbol{x}_n$

One obvious way to combine these vectors would be to concatenate them, but this would increase the dimensionality of the input space and hence of all subsequent attention spaces, creating a significant increase in computational cost

Instead, we can simply add the position vectors onto the token vectors to give

$$\tilde{\boldsymbol{x}}_n = \boldsymbol{x}_n + \boldsymbol{r}_n$$

Positional Encodings

An ideal positional encoding should:

- Provide a unique representation for each position
- It should be bounded
- It should generalize to longer sequences
- And it should have a consistent way to express the number of steps between any two input vectors irrespective of their absolute position because the relative position of tokens is often more important than the absolute position

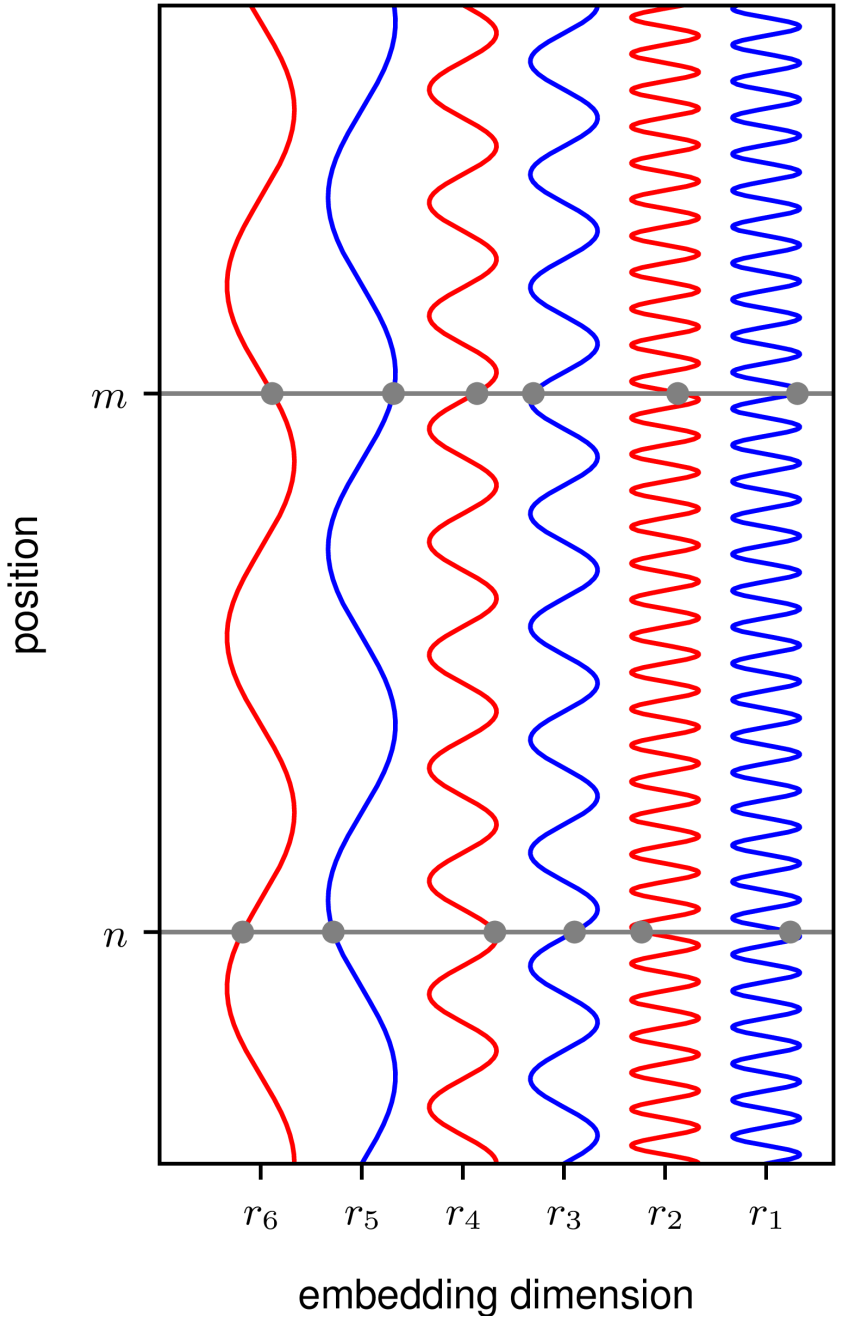
Positional Encodings

There are many approaches to positional encoding

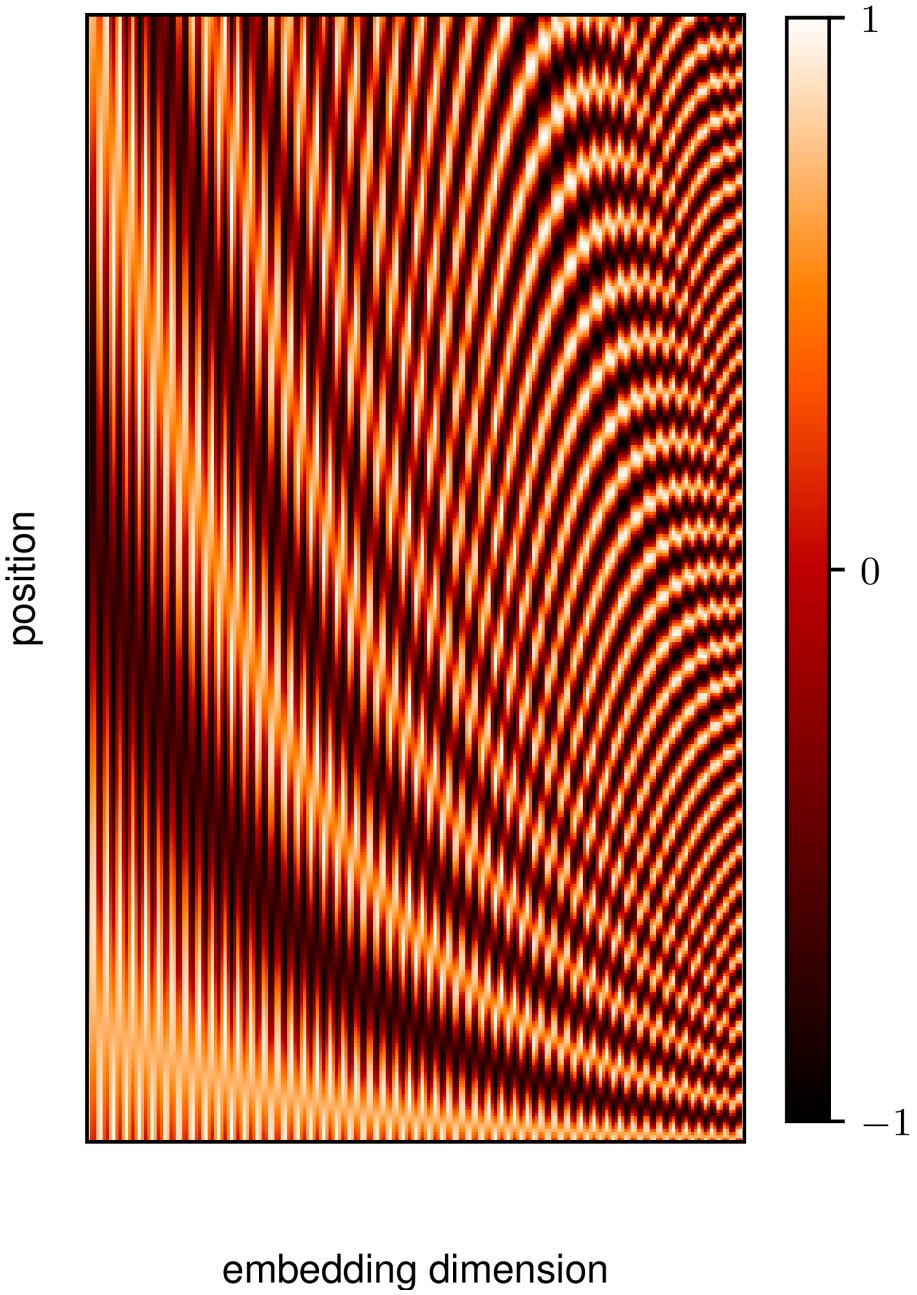
The original technique by Vaswani et al. (2017) is based on sinusoidal functions

For a given position n the associated position-encoding vector has components $r_n i$ given by:

$$r_{ni} = \begin{cases} \sin\left(\frac{n}{L^{i/D}}\right), & \text{if } i \text{ is even,} \\ \cos\left(\frac{n}{L^{(i-1)/D}}\right), & \text{if } i \text{ is odd.} \end{cases}$$



$$
\begin{array}{lcccc}
\mathbf{1}: & 0 & 0 & 0 & 1 \\
\mathbf{2}: & 0 & 0 & 1 & 0 \\
\mathbf{3}: & 0 & 0 & 1 & 1 \\
\mathbf{4}: & 0 & 1 & 0 & 0 \\
\mathbf{5}: & 0 & 1 & 0 & 1 \\
\mathbf{6}: & 0 & 1 & 1 & 0 \\
\mathbf{7}: & 0 & 1 & 1 & 1 \\
\mathbf{8}: & 1 & 0 & 0 & 0 \\
\mathbf{9}: & 1 & 0 & 0 & 1
\end{array}
$$


position
embedding dimension
1
0
−1