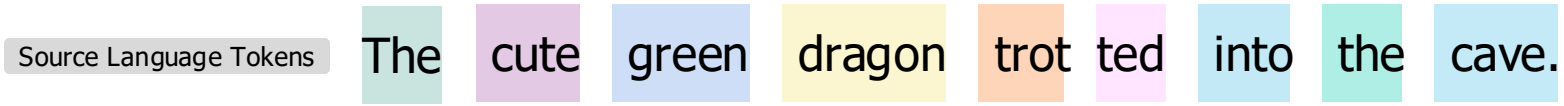
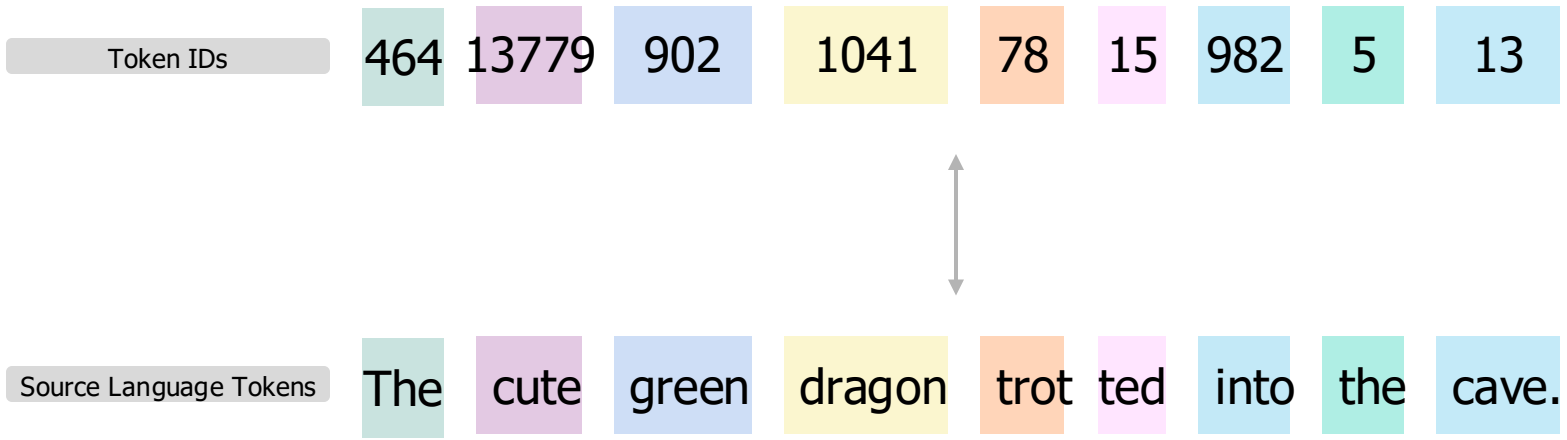
Generative AI

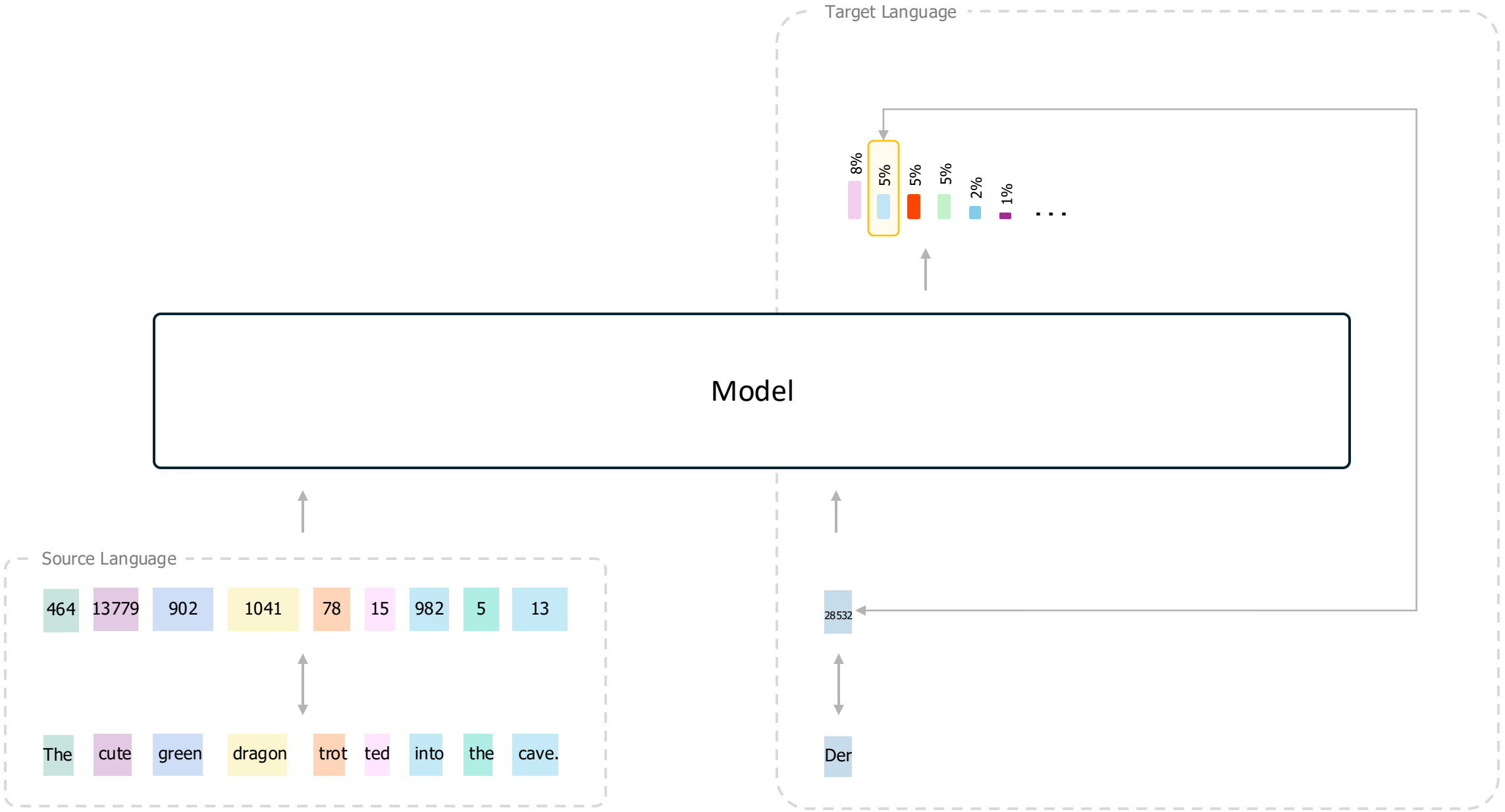
2025 / 10 / 29

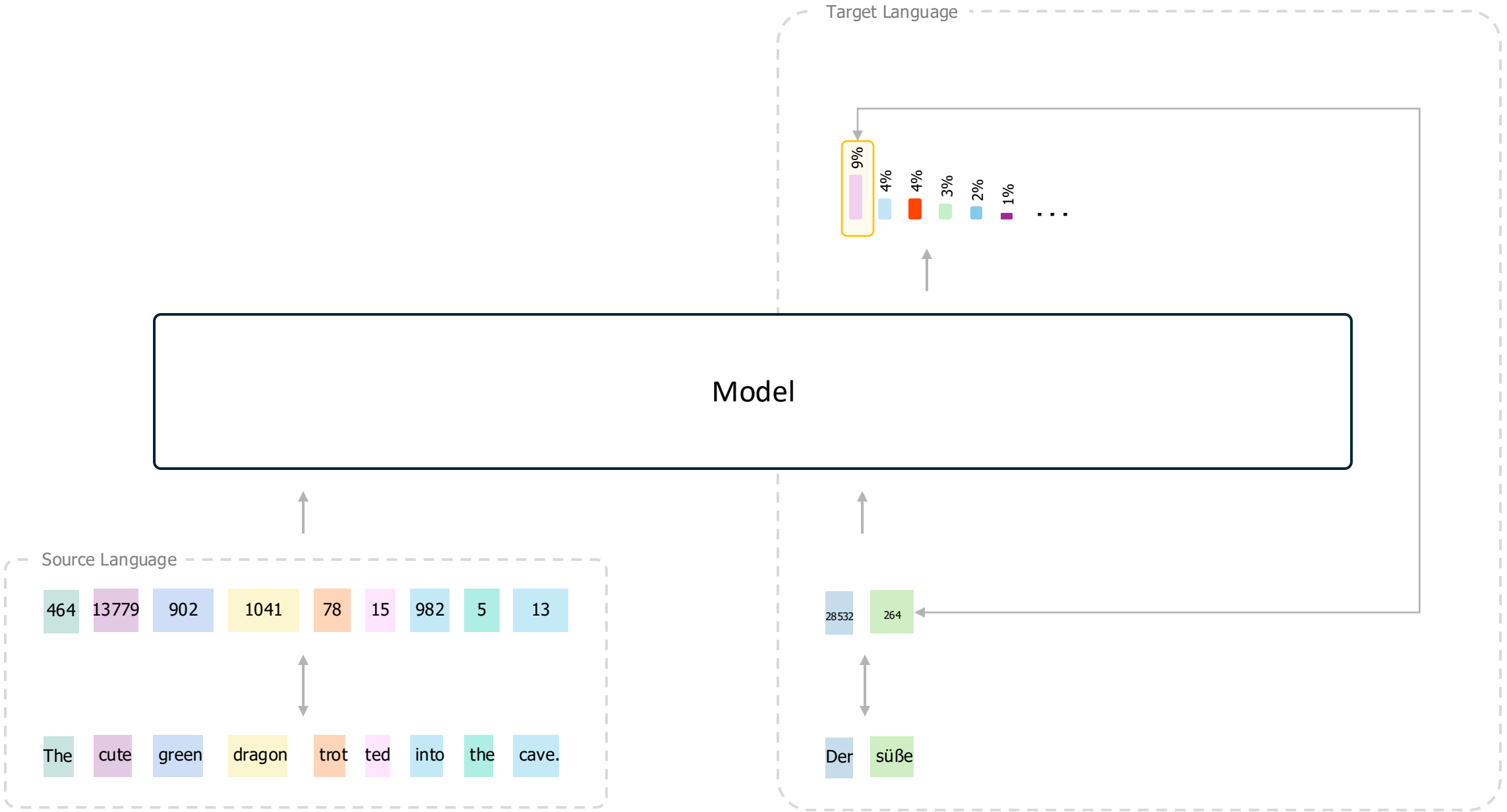
Source Language

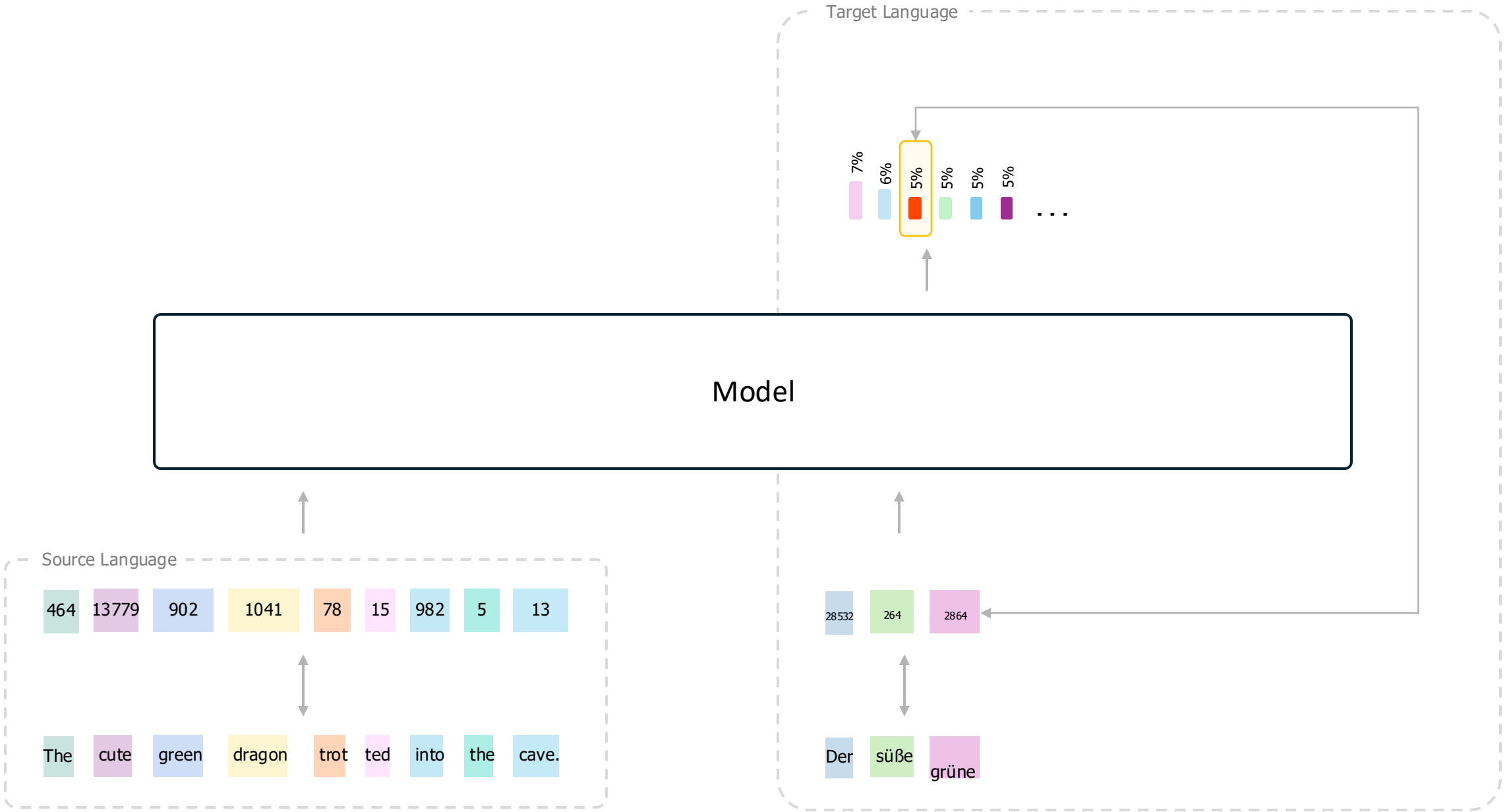
The cute green dragon trotted into the cave.

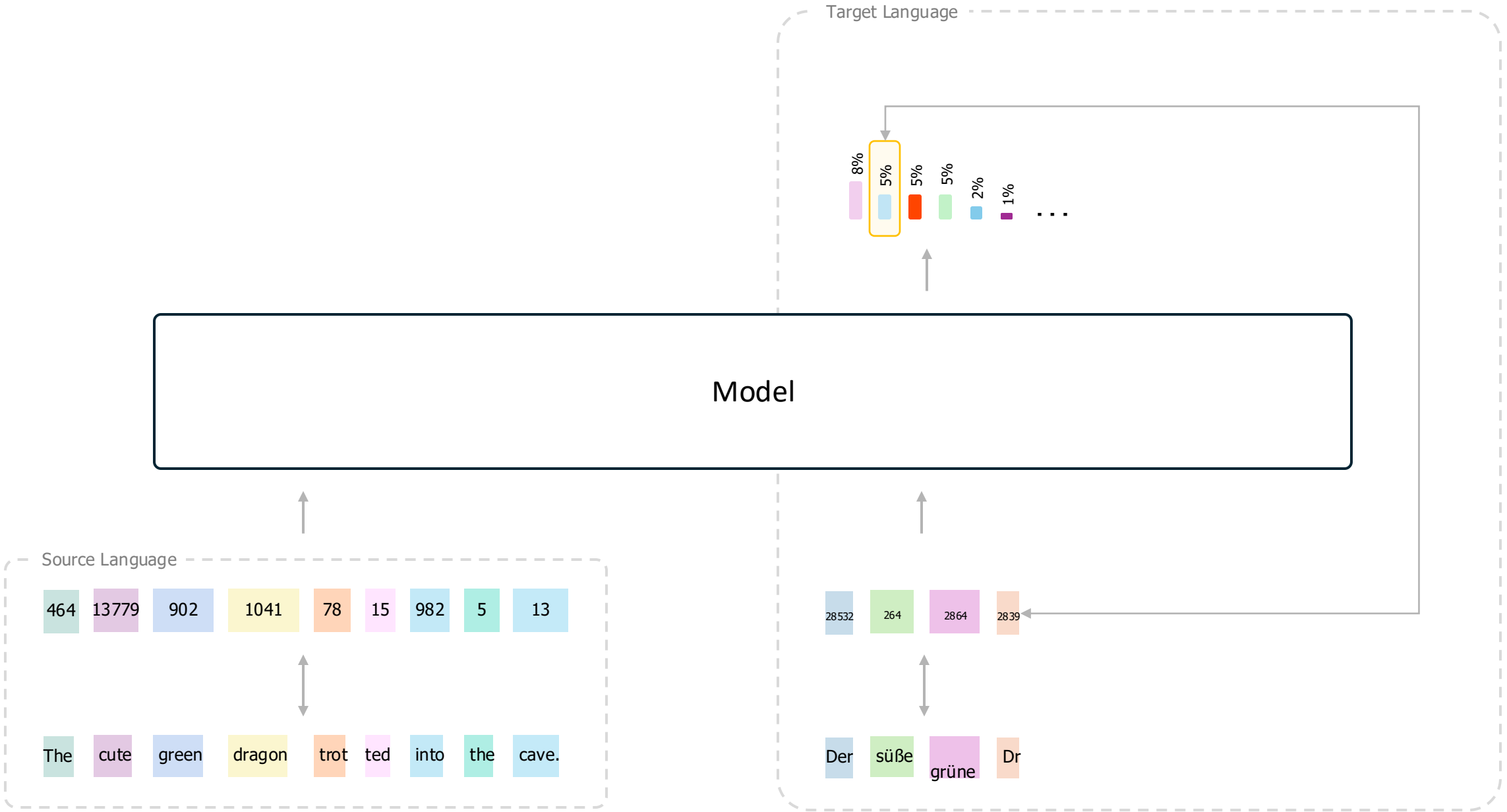

Source Language Tokens
The
cute
green
dragon
trot
ted
into
the
cave.

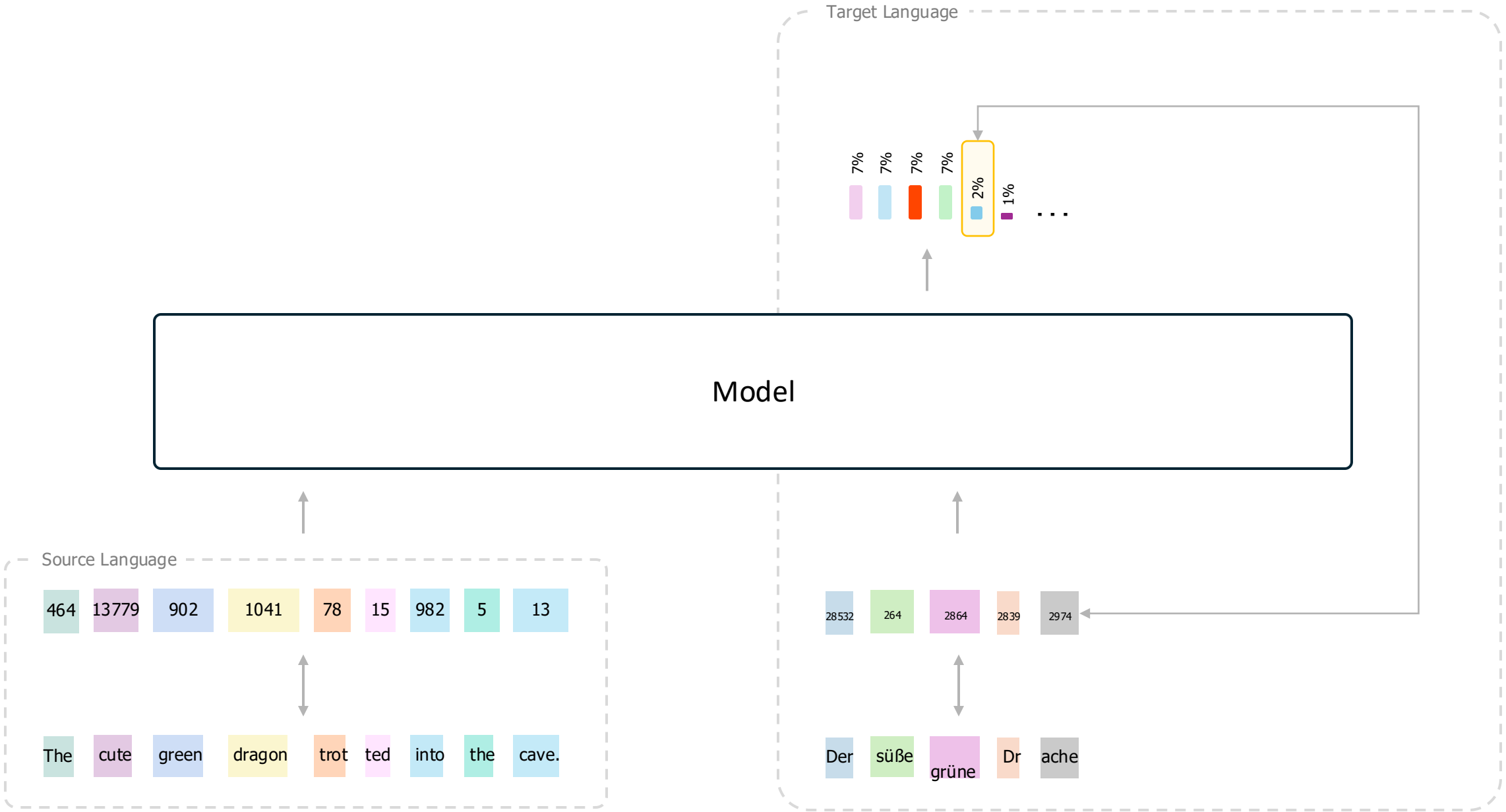

Token IDs
464
13779
902
1041
78
15
982
5
13
Source Language Tokens
The
cute
green
dragon
trot
ted
into
the
cave.

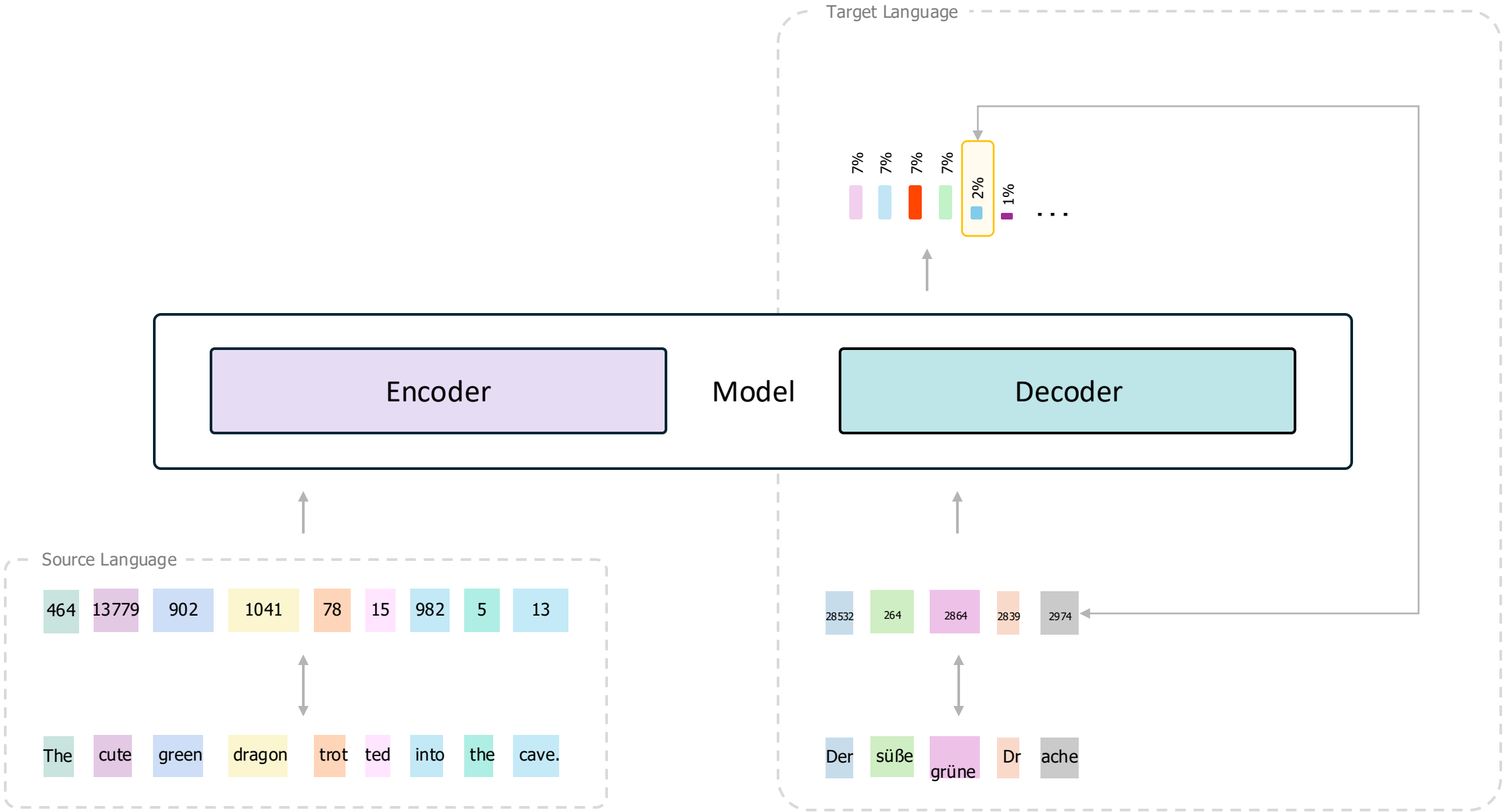

Target Language
8%
5%
5%
5%
2%
1%
. . .
Model
Source Language
464
13779
902
1041
78
15
982
5
13
The
cute
green
dragon
trot
ted
into
the
cave.
28532
Der

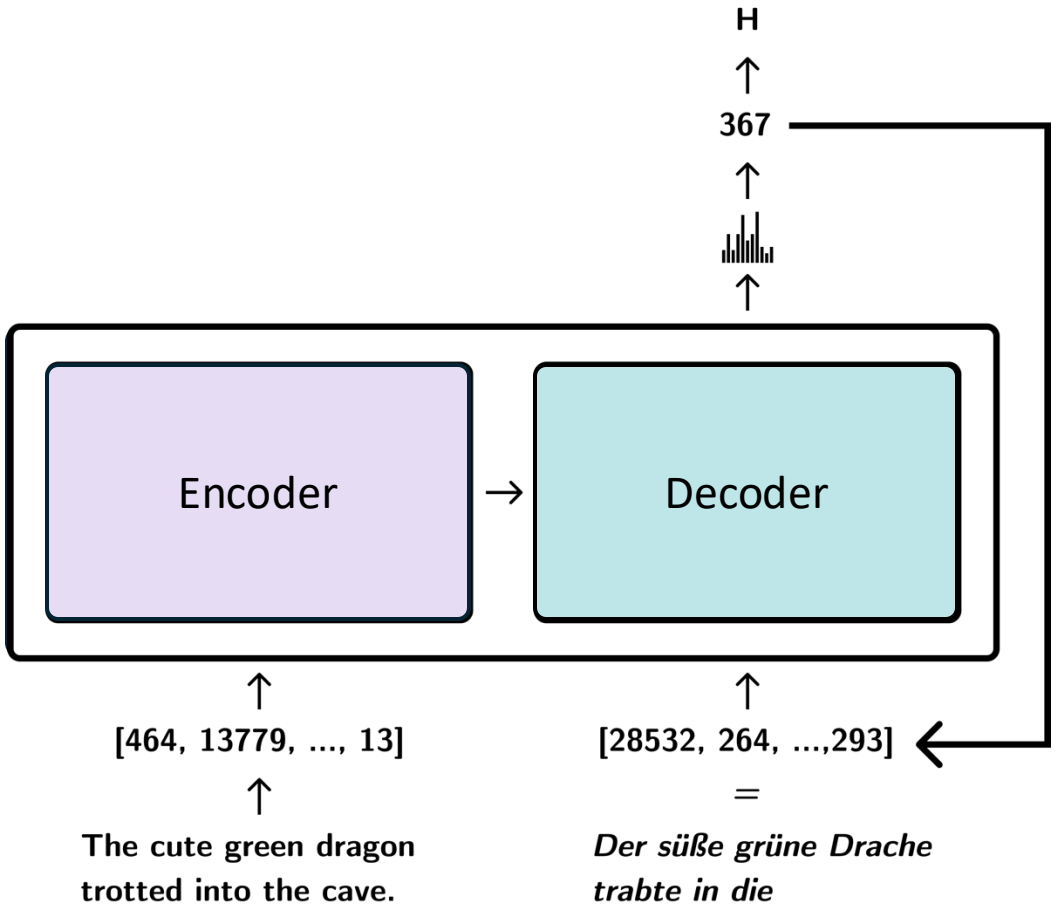

Target Language
9%
4%
4%
3%
2%
1%
. . .
Model
Source Language
464
13779
902
1041
78
15
982
5
13
The
cute
green
dragon
trot
ted
into
the
cave.
28532
264
Der
süße

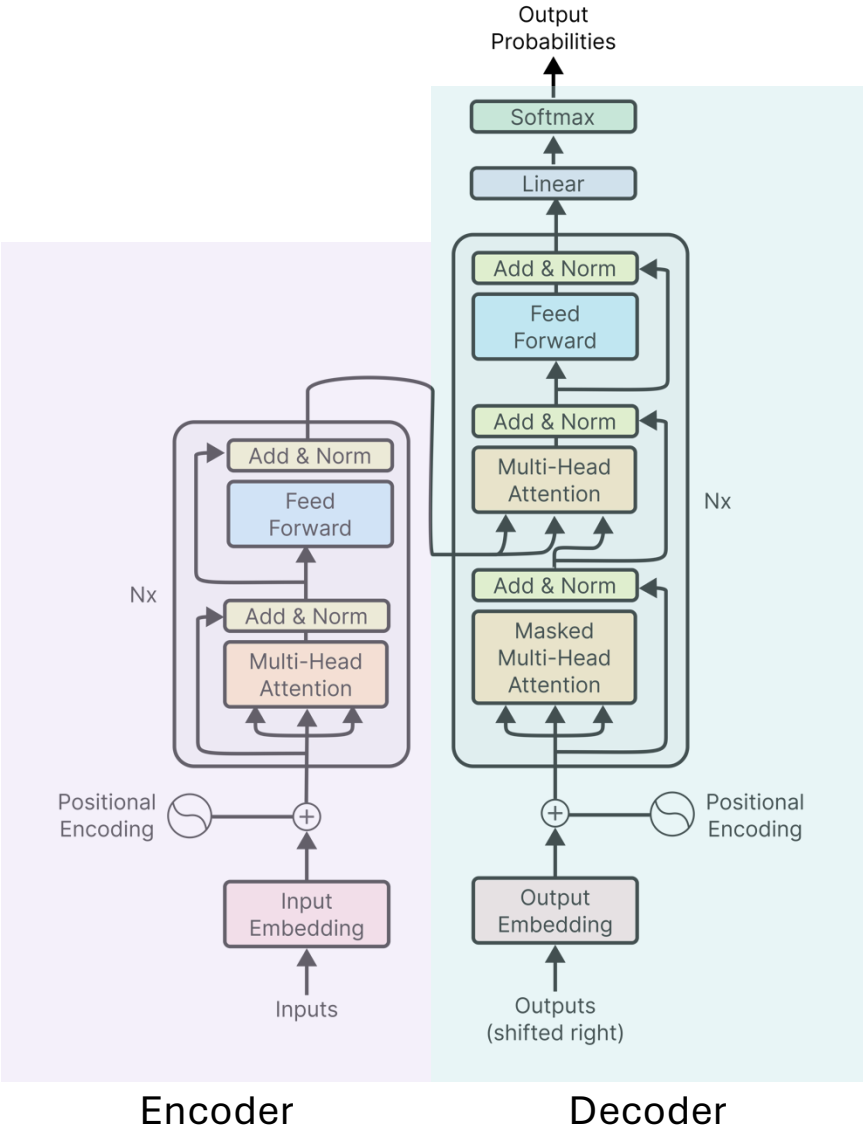

Target Language
7%
6%
5%
5%
5%
5%
. . .
Model
Source Language
464
13779
902
1041
78
15
982
5
13
The
cute
green
dragon
trot
ted
into
the
cave.
28532
264
2864
Der
süße
grüne


Target Language
8%
5%
5%
5%
2%
1%
...
Model
Source Language
464
13779
902
1041
78
15
982
5
13
The
cute
green
dragon
trot
ted
into
the
cave.
28532
264
2864
2839
Der
süße
grüne
Dr


Target Language
7%
7%
7%
7%
2%
1%
...
Model
Source Language
464
13779
902
1041
78
15
982
5
13
The
cute
green
dragon
trot
ted
into
the
cave.
28532
264
2864
2839
2974
Der
süße
grüne
Dr
ache


Target Language
7%
7%
7%
7%
2%
1%
...
Encoder
Model
Decoder
Source Language
464
13779
902
1041
78
15
982
5
13
The
cute
green
dragon
trot
ted
into
the
cave.
28532
264
2864
2839
2974
Der
süße
grüne
Dr
ache

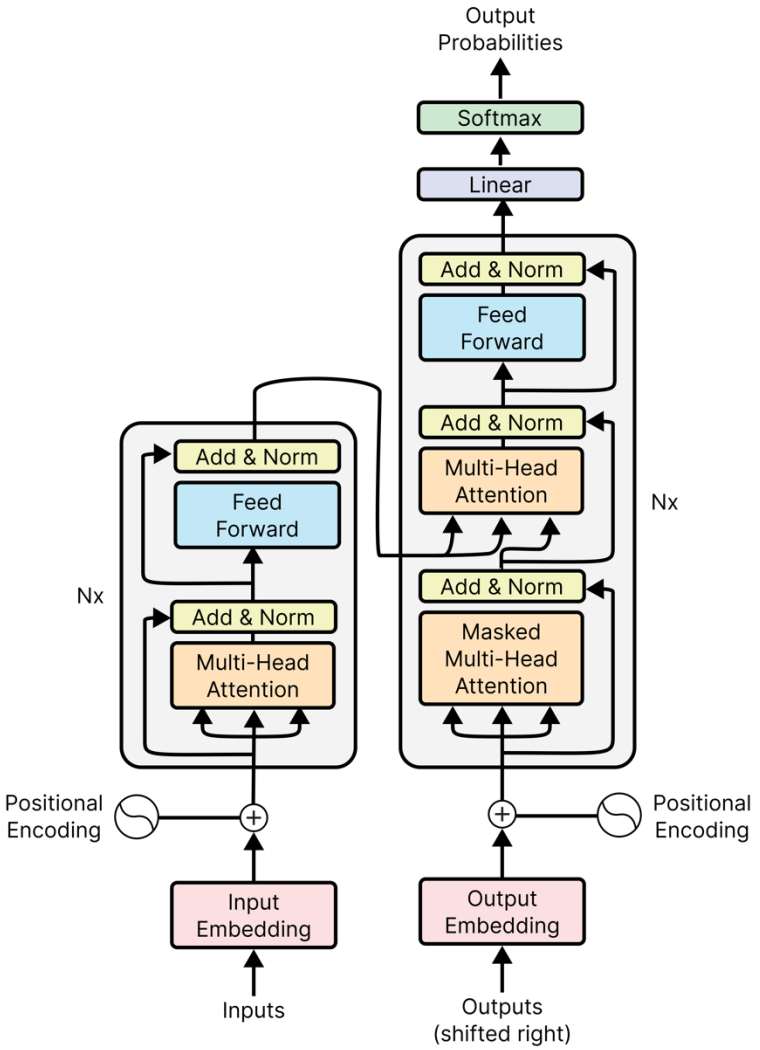

H
367
Encoder
Decoder
[464, 13779, ..., 13]
[28532, 264, ...,293]
=
The cute green dragon trotted into the cave.
Der süße grüne Drache trabte in die


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

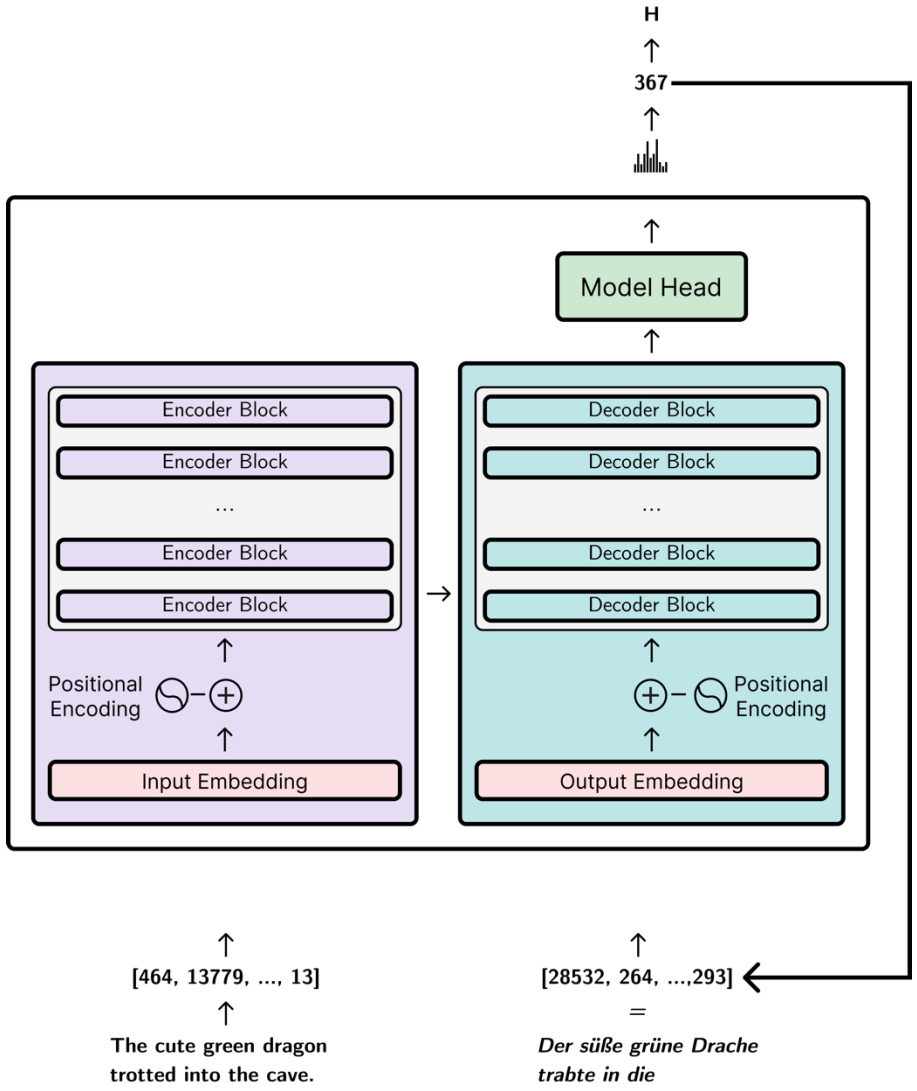
A bird's eye view of the

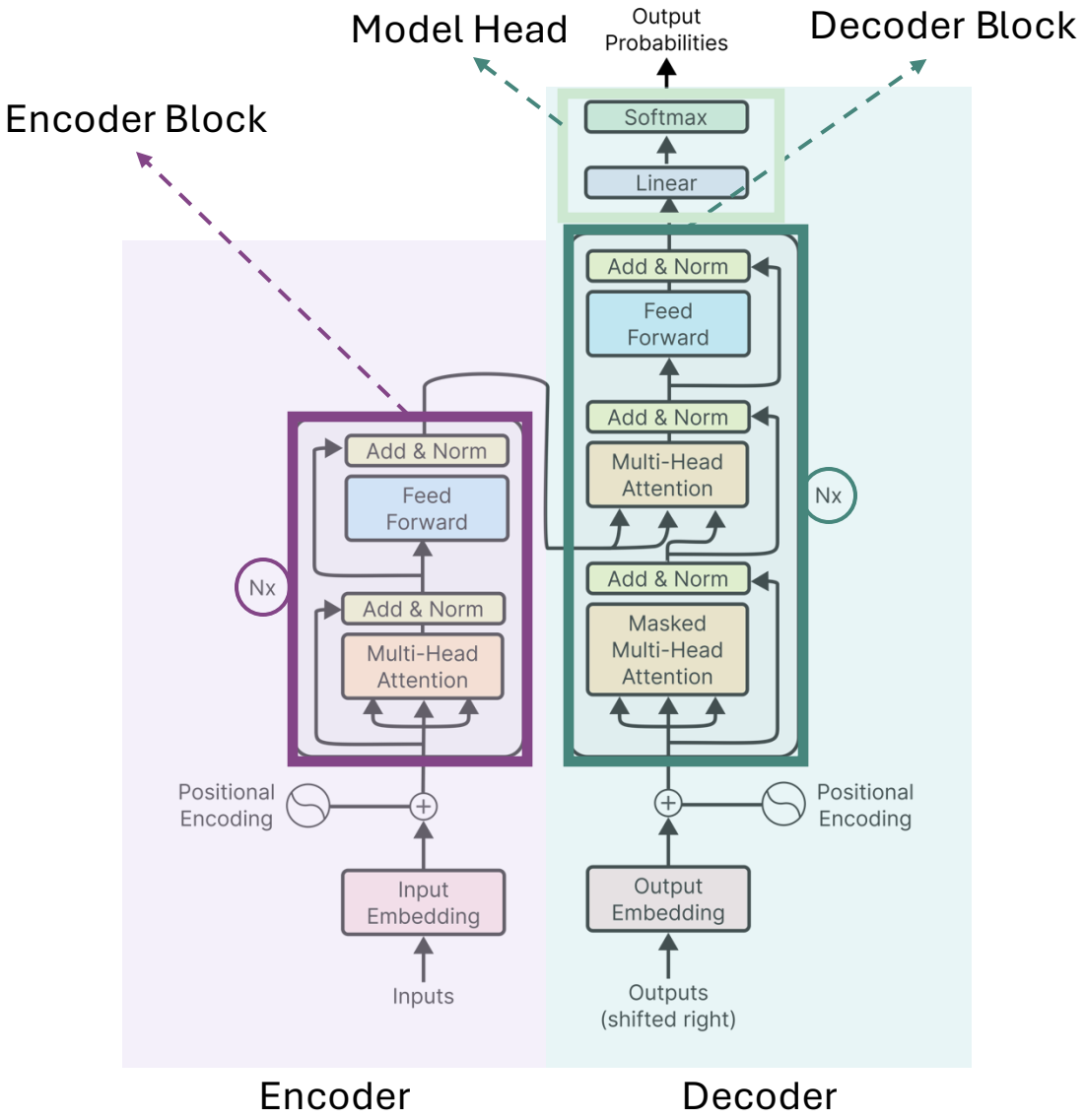
Transformer Architecture.

With reference to 'Attention Is All You Need' by A. Vaswani et al.[1]

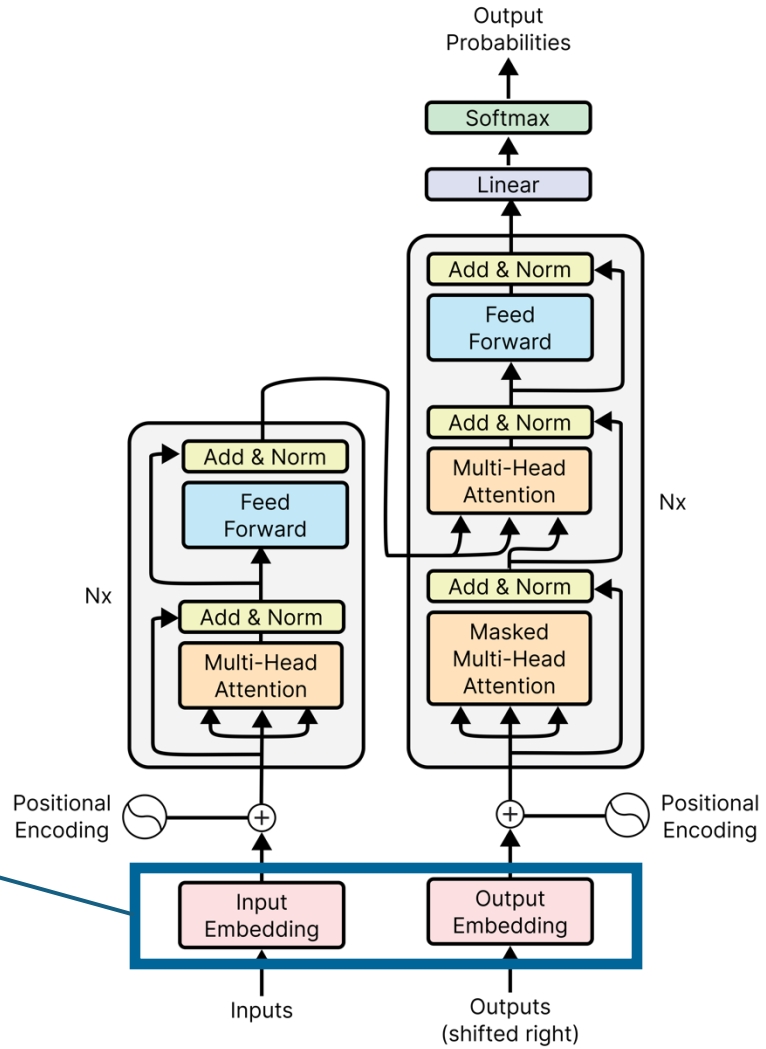


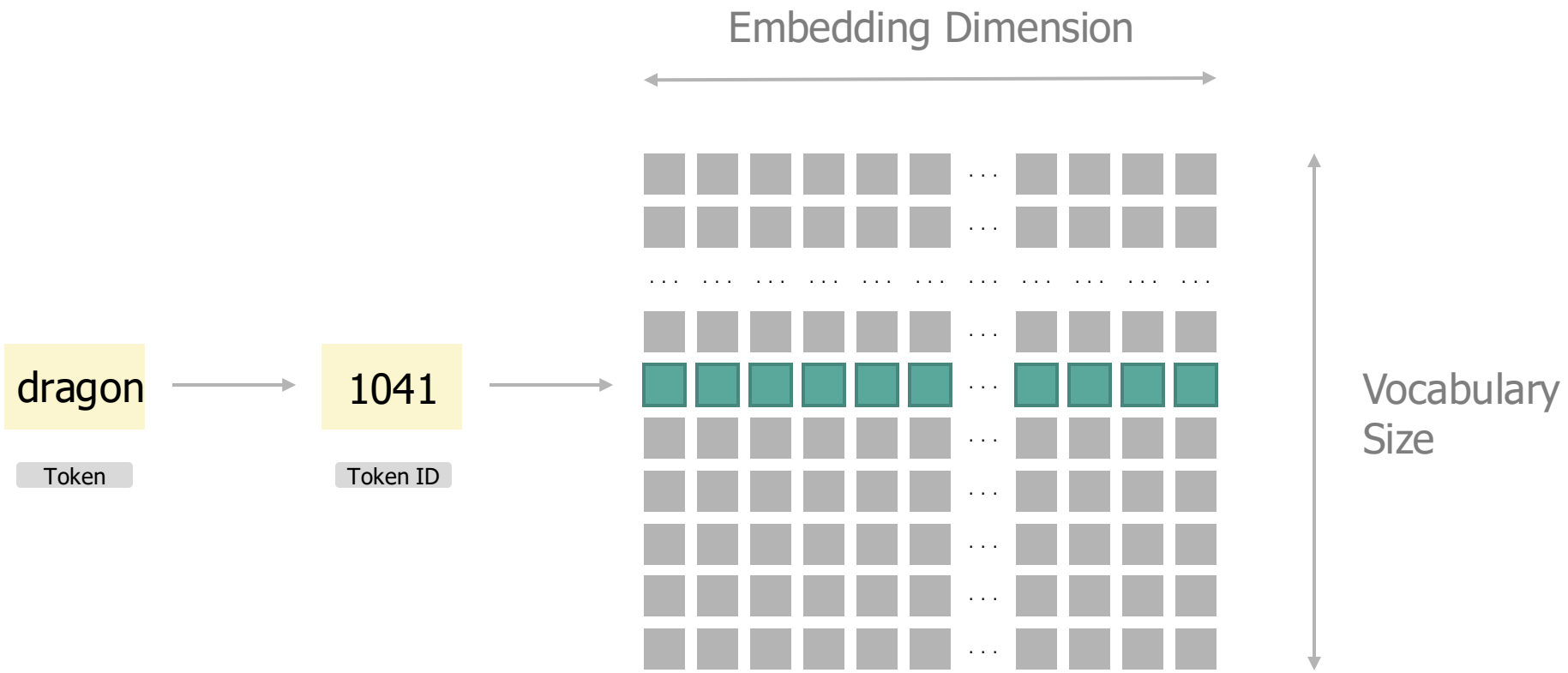
1. https://proceedings.neurips.cc/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html


H
367
Model Head
Encoder Block
Encoder Block
...
Encoder Block
Encoder Block
Decoder Block
Decoder Block
...
Decoder Block
Decoder Block
Positional Encoding
Positional Encoding
Input Embedding
Output Embedding
[464, 13779, ..., 13]
[28532, 264, ...,293]
=
The cute green dragon trotted into the cave.
Der süße grüne Drache trabte in die

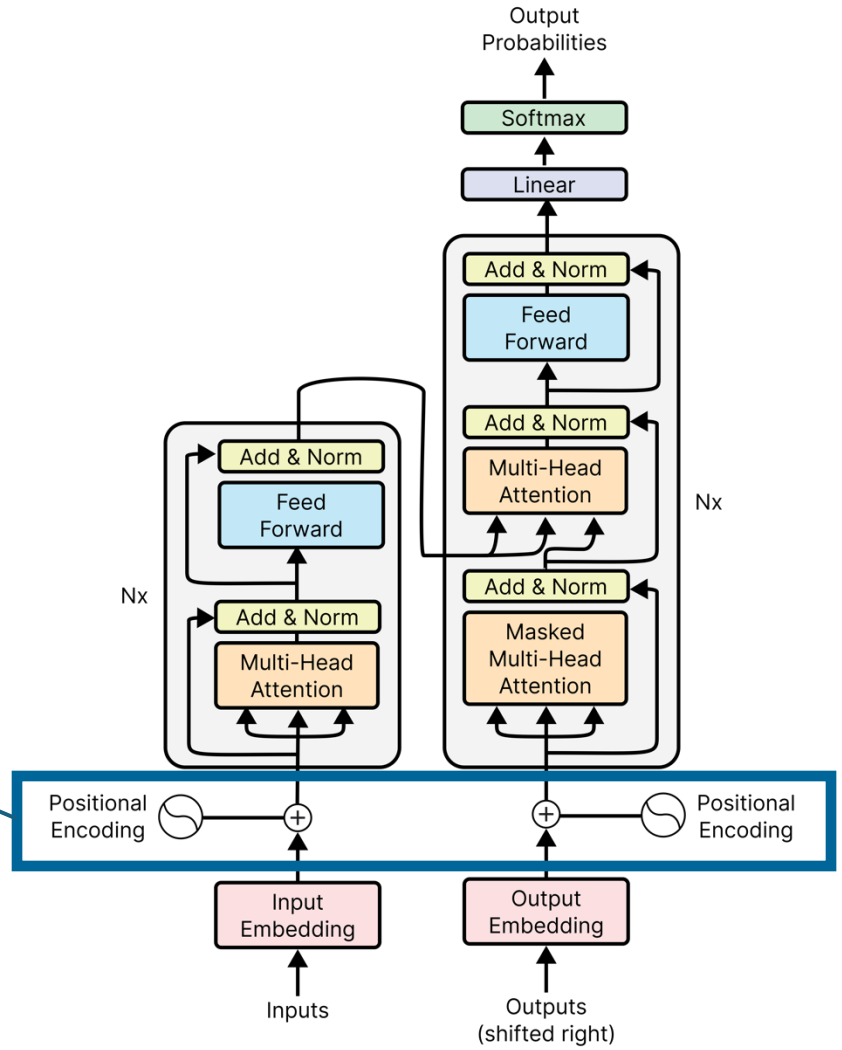

Model Head
Output Probabilities
Decoder Block
Encoder Block
Softmax
Linear
Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
Add & Norm
Masked Multi-Head Attention
Nx
Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
Nx
Positional Encoding
Positional Encoding
Input Embedding
Output Embedding
Inputs
Outputs (shifted right)
Encoder
Decoder

Embeddings


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


Embedding Dimension
dragon
Token
1041
Token ID
Vocabulary Size

Positional Encoding

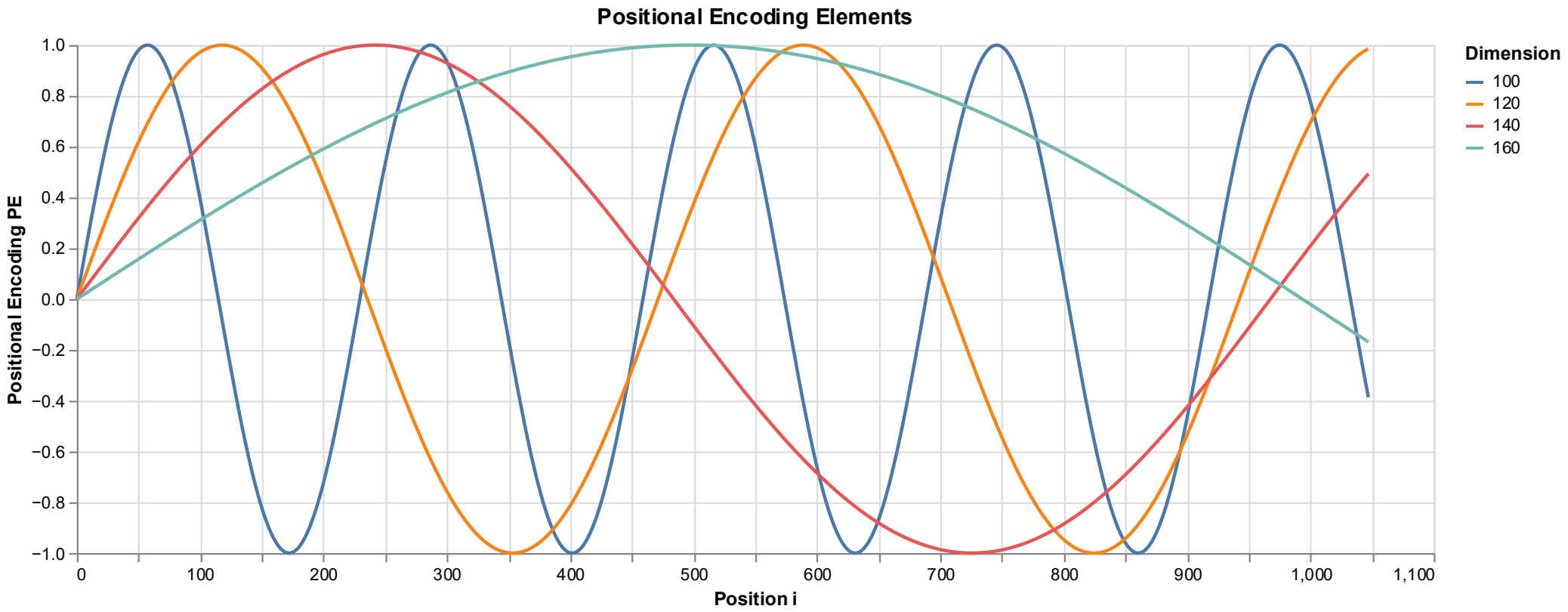


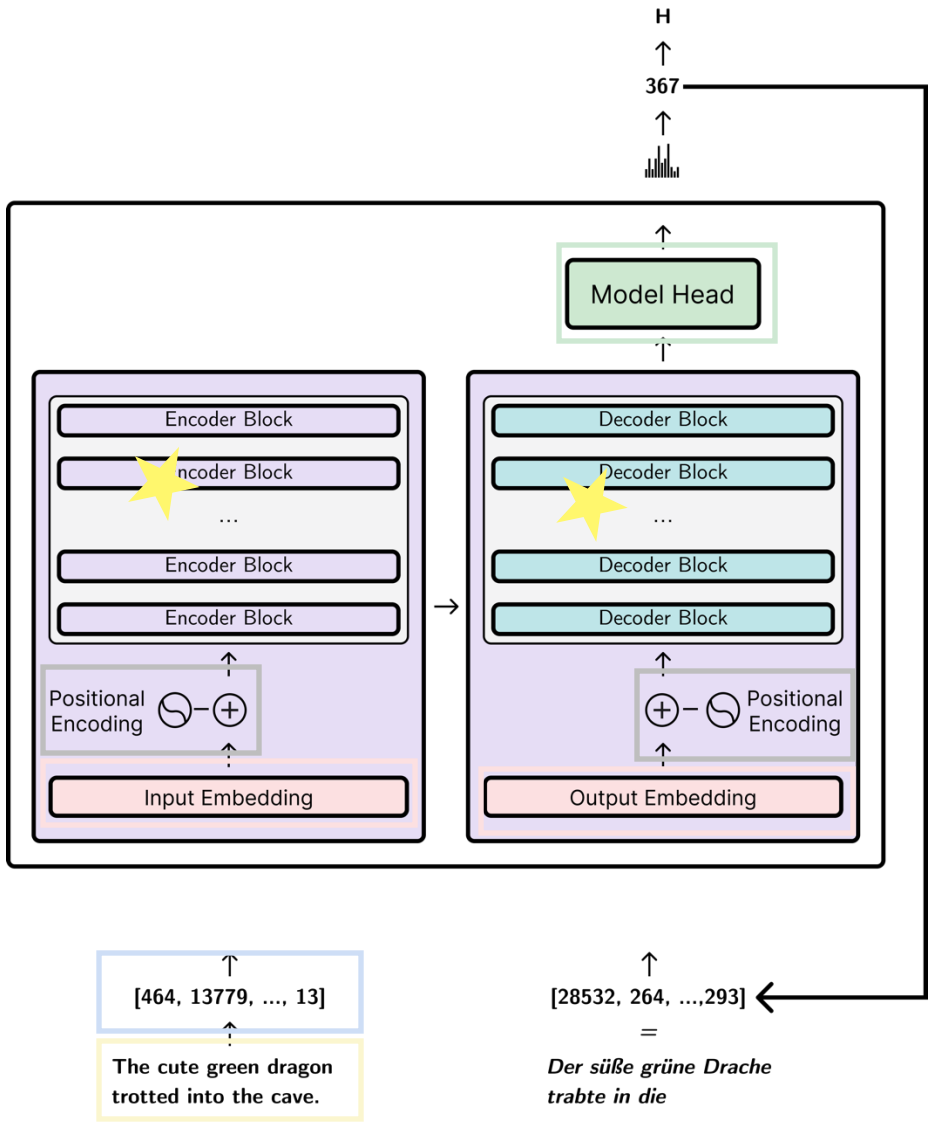
The embedding layer and subsequent layers are inherently permutation invariant, but the order plays a critical role in sequence processing tasks.

- $PE(pos, 2_i) = \sin(pos/10000^{2i/d_{model}})$
- $PE(pos, 2_{i+1}) = \cos(pos/10000^{2i/d_{model}})$

$$\overrightarrow{PE} = \begin{bmatrix} \sin(pos/10000^{0/d_{model}}) \\ \cos(pos/10000^{0/d_{model}}) \\ \sin(pos/10000^{2/d_{model}}) \\ \cos(pos/10000^{2/d_{model}}) \\ . \\ . \\ . \\ \sin(pos/10000^{d_{model}-2/d_{model}}) \\ \cos(pos/10000^{d_{model}-2/d_{model}}) \end{bmatrix}$$

- $PE(pos, 2_i) = \sin(pos/10000^{2i/d_{model}})$
- $PE(pos, 2_{i+1}) = \cos(pos/10000^{2i/d_{model}})$





Tokenisation breaks down natural language sequences into atomic units that carry some semantic meaning (tokens).

Encoding represents tokens in a one-dimensional numeric space (token IDs)

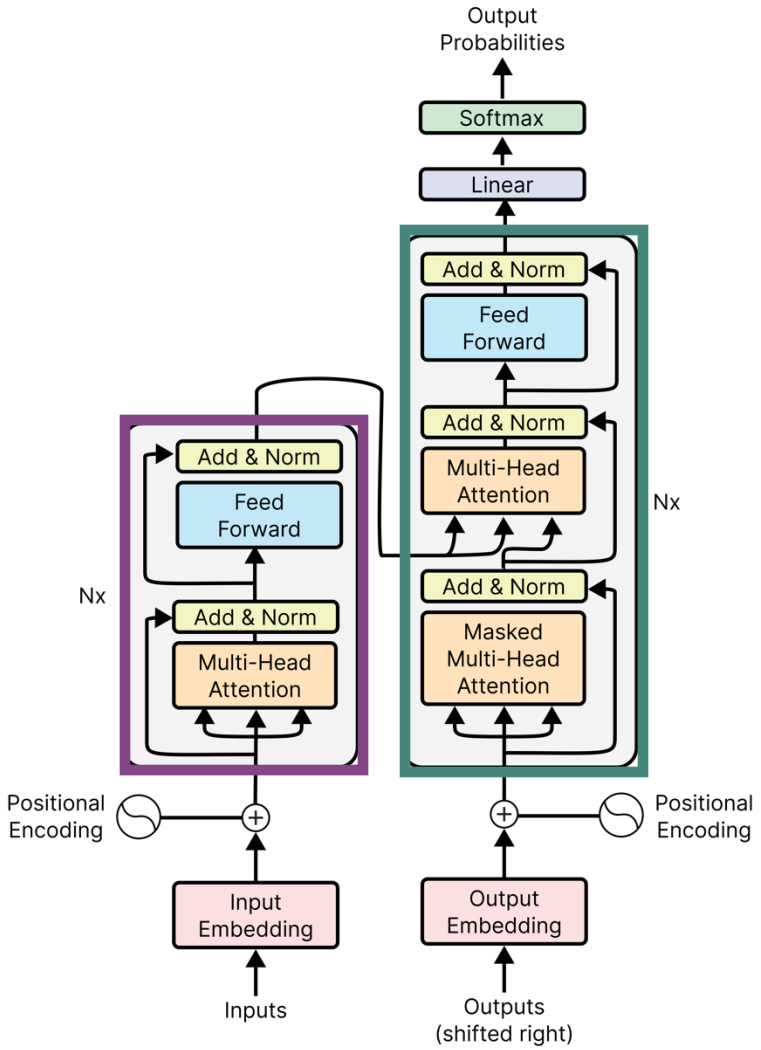
Embeddings project token IDs in higher dimensional vector space

Positional encoding injects information about a tokens' sequence-position into the embedding vector

Some magic yet to be covered

The decoder autoregressively generates a probability distribution over the vocabulary.

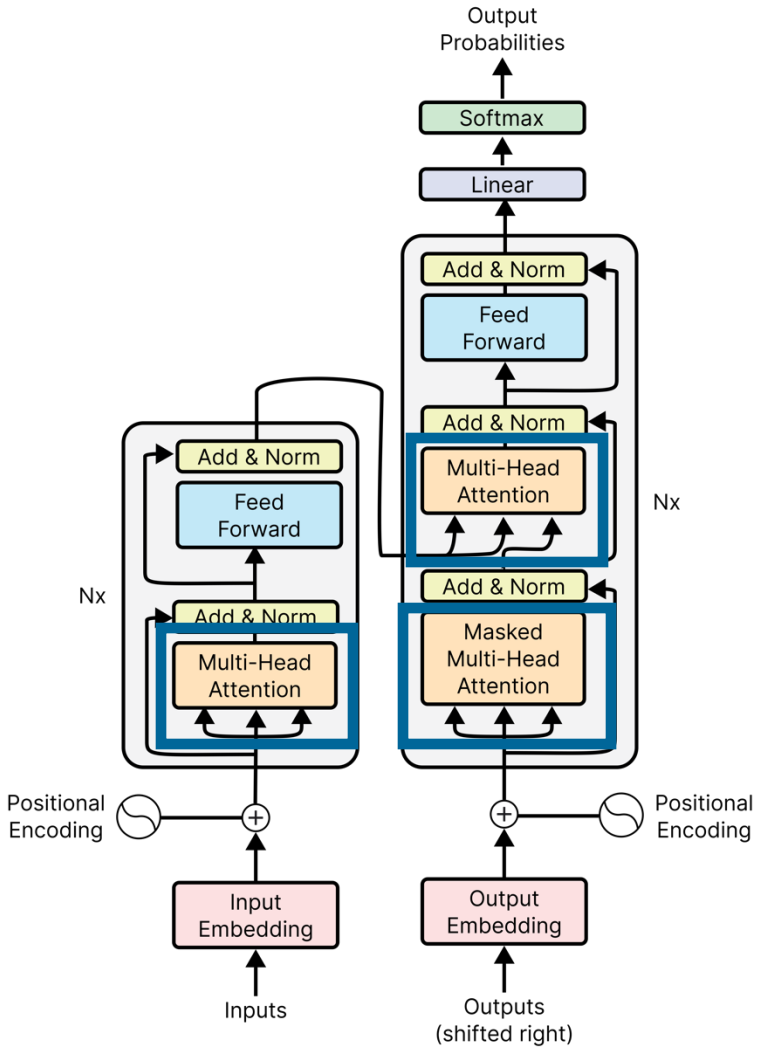
A closer look at the encoder and decoder blocks.



The attention mechanism

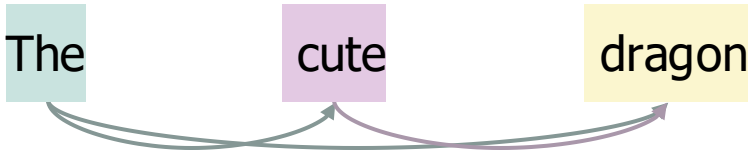
The cornerstone of the transformer's ability to capture context.

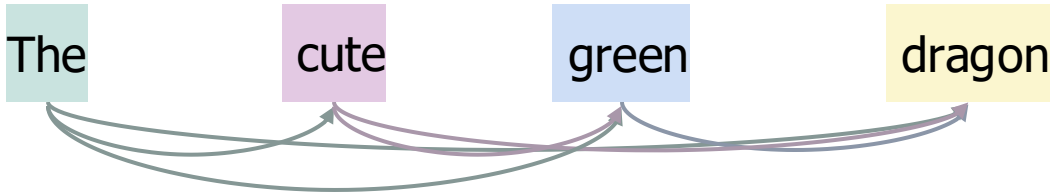
$$Attention(Q, K, V) = softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$



dragon

The dragon


The
cute
dragon


The
cute
green
dragon

Attention Mechanism

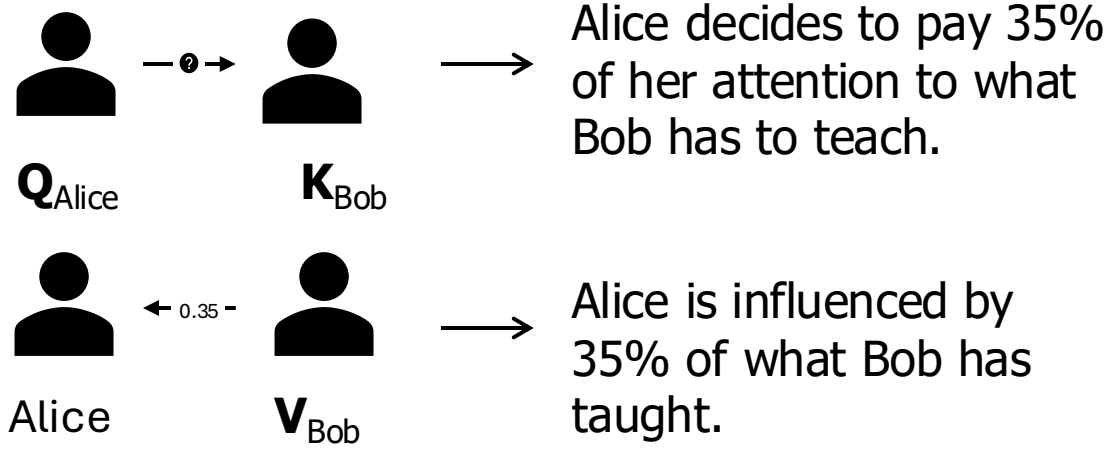
An example of the exchange of attention in a group of people eager to share and gather knowledge.

- Alice, Bob, Charlie, David, and Eve want to advance their knowledge in certain areas by spending time and *attention*.
- Each of them has some knowledge they can teach.
- Each of them has a **limited capacity** to **pay attention** (one can only learn so much in a week).
- Each of them can **receive attention indefinitely** (you can be listened to by everyone).

Query ... a description of the knowledge they want to gather

Key ... a description of what they can teach

Value ... the actual knowledge

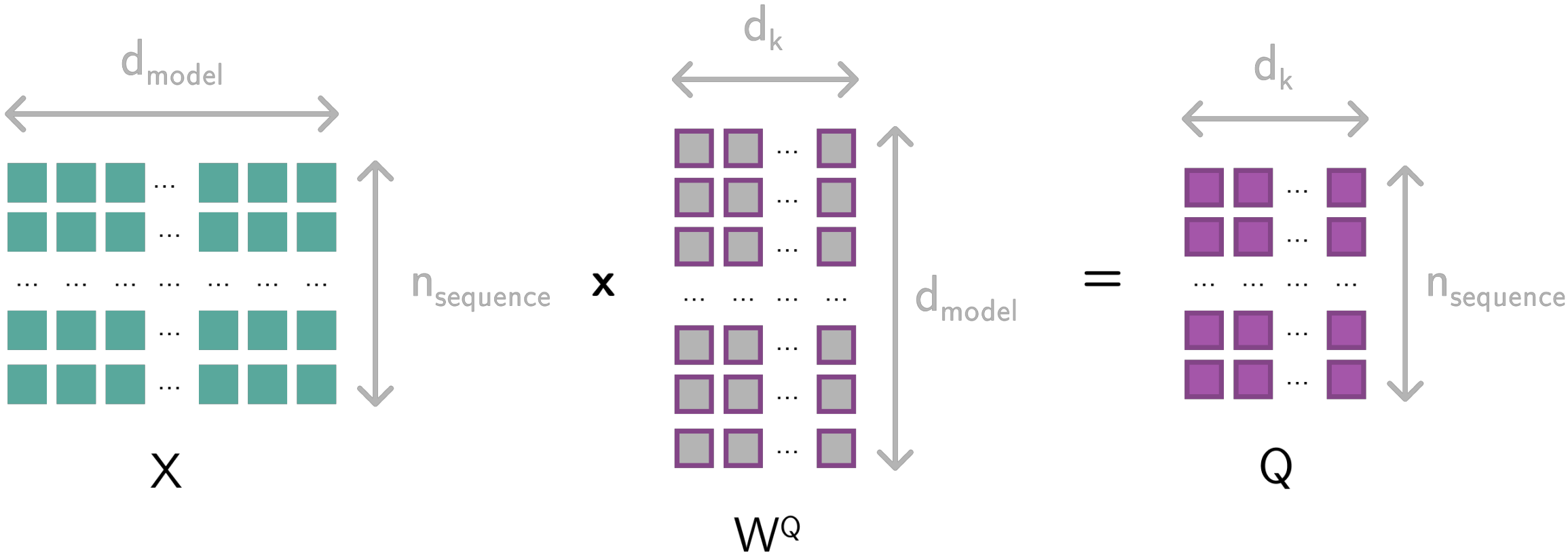


Visualisation of an interaction between two people[1]

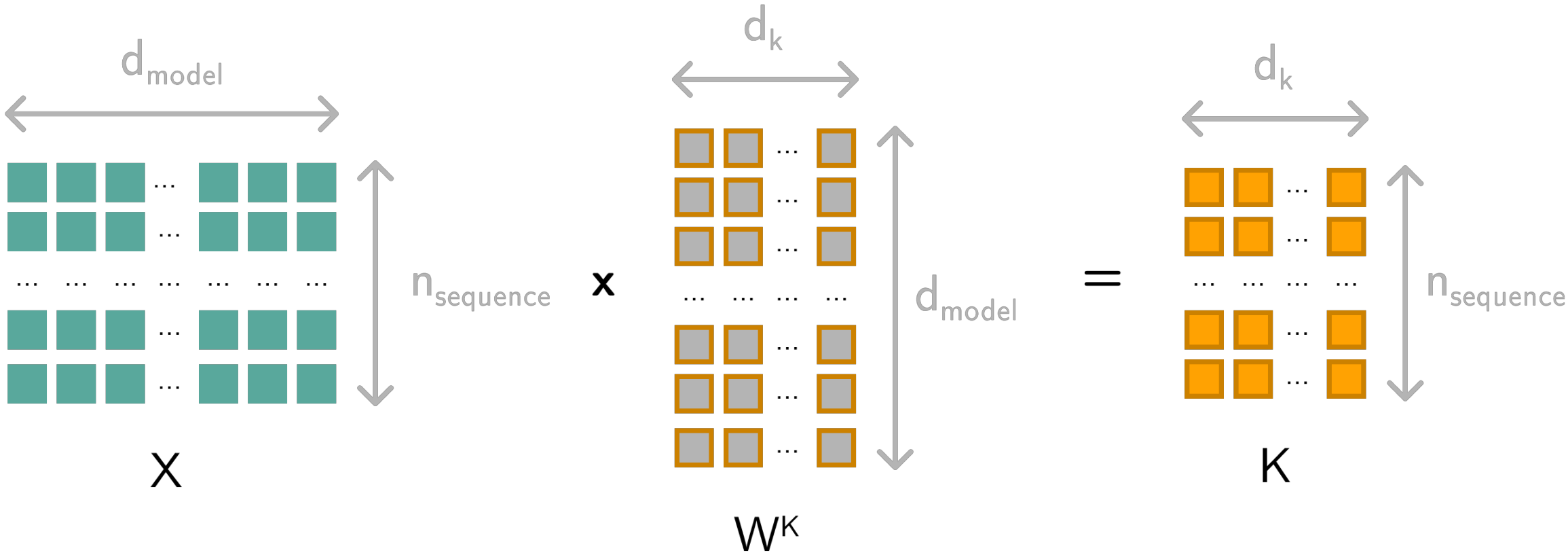
The goal of the attention mechanism is to compute and ingest *how much* and *how* each token should influence the representation ("meaning") of every other token within the input sequence.

Mathematical intuition behind queries, keys and values.

The goal of the attention mechanism is to compute and ingest *how much* and *how* each token should influence the representation ("meaning") of every other token within the input sequence.

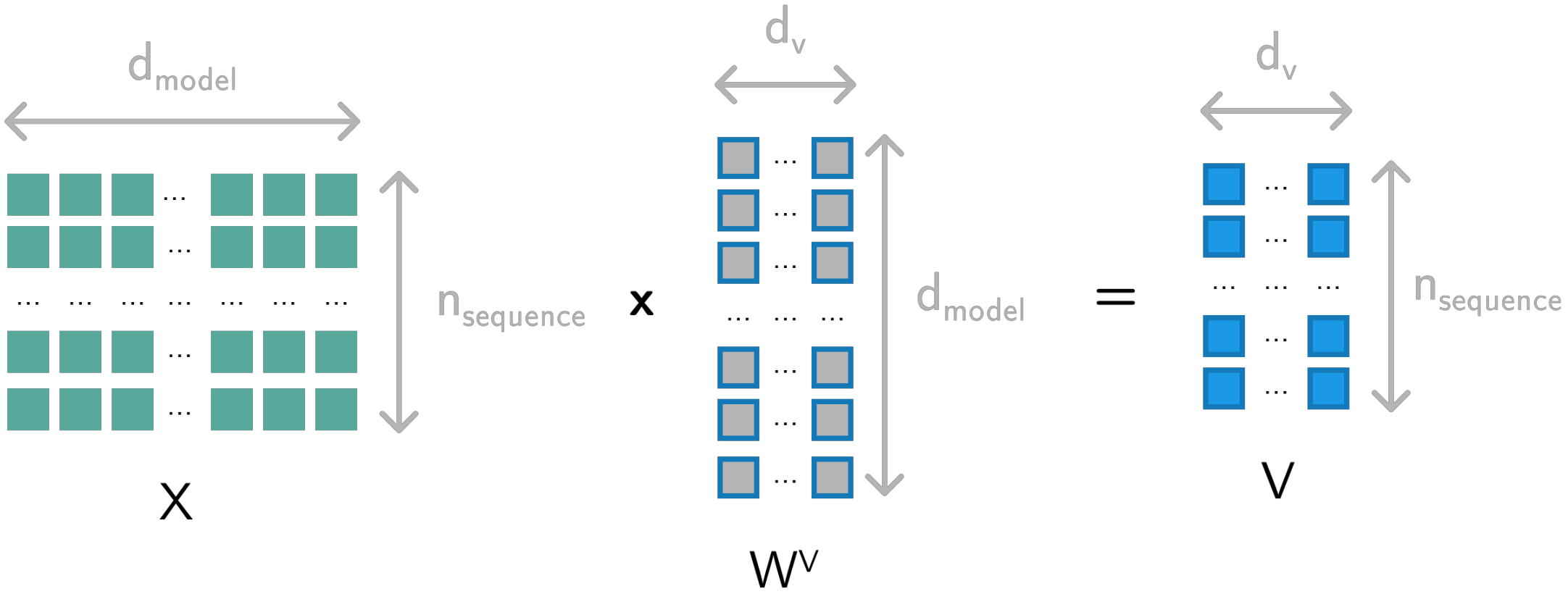


The goal of the attention mechanism is to compute and ingest *how much* and *how* each token should influence the representation ("meaning") of every other token within the input sequence.



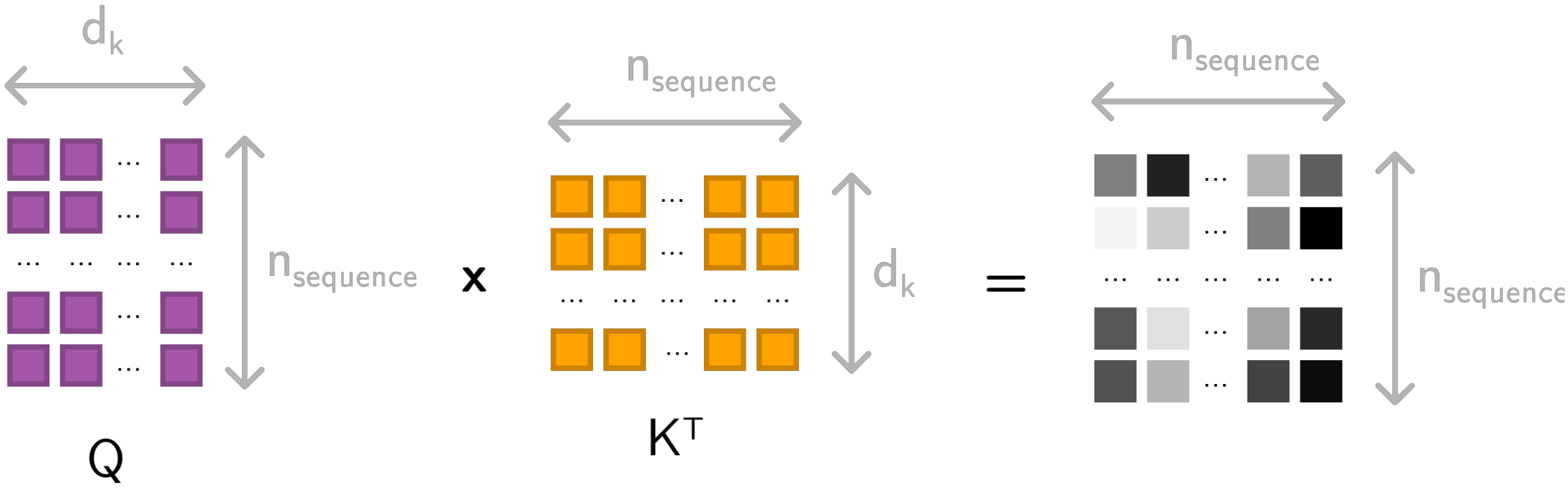
Mathematical intuition behind queries, keys and values.

The goal of the attention mechanism is to compute and ingest *how much* and *how* each token should influence the representation ("meaning") of every other token within the input sequence.



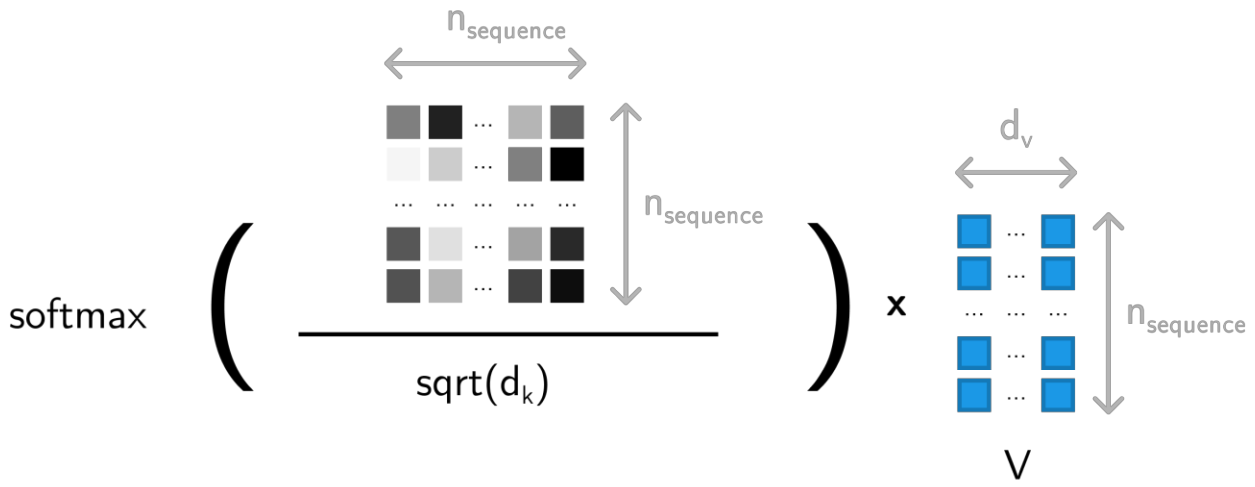
Mathematical intuition behind queries, keys and values.

The dot product can be used to measure the 'fit' between query and key. The result is a matrix of attention scores.



Mathematical intuition behind queries, keys and values.

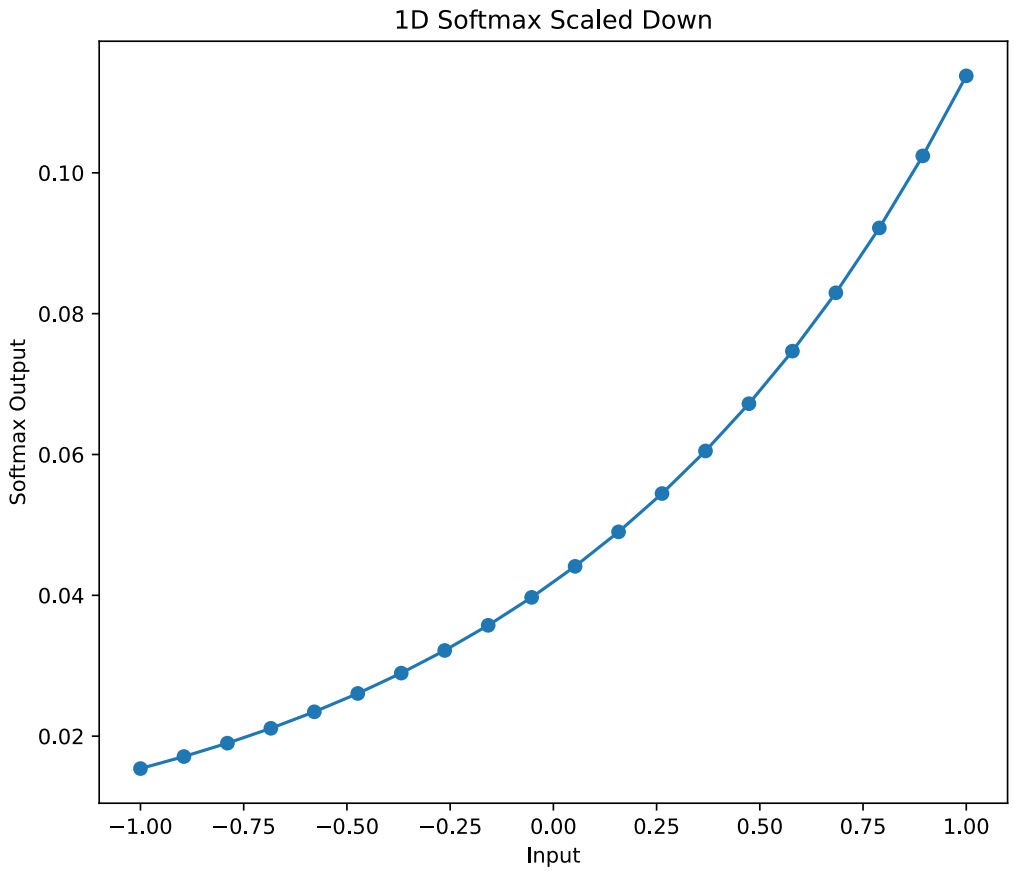
The (scaled) softmax function calculates the *attention weights* for each row between 0 and 1. Multiplied by the token values, the new vector representations of each token are formed.

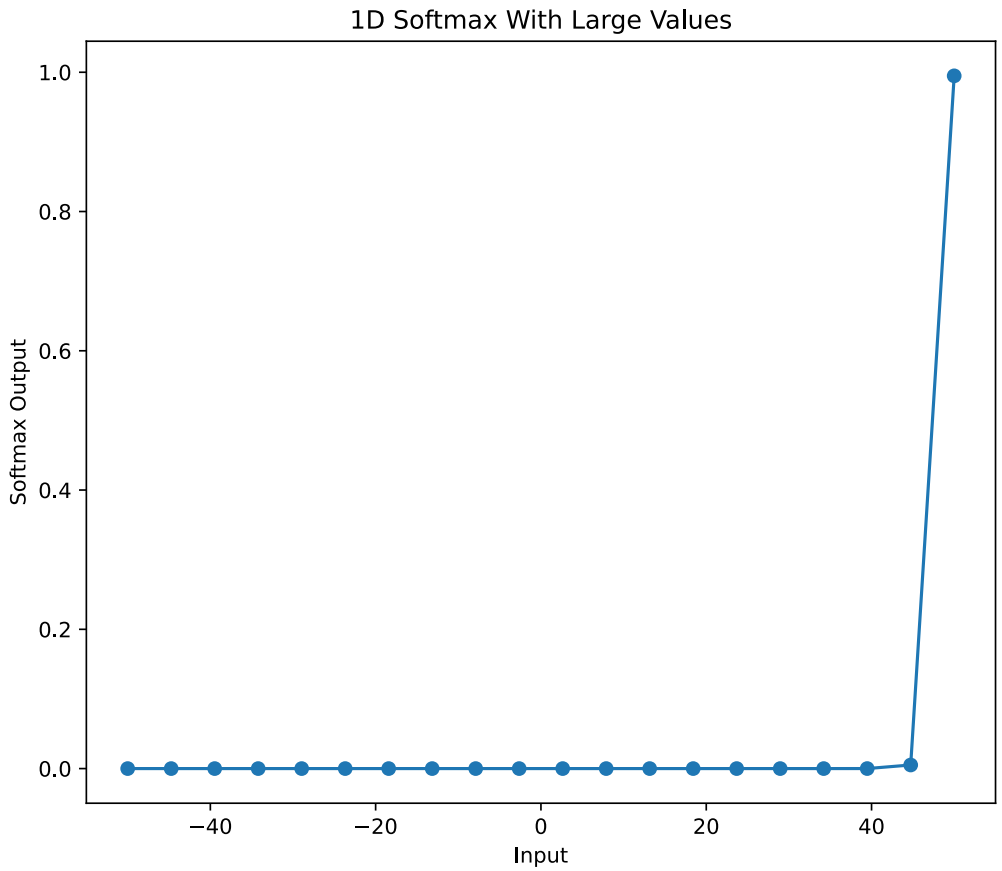


with $softmax(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}}$

Full formula as written in the original transformer paper: $Attention(Q, K, V) = softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)V$

Scaling the softmax input smoothes the curve and ensures gradient stability.



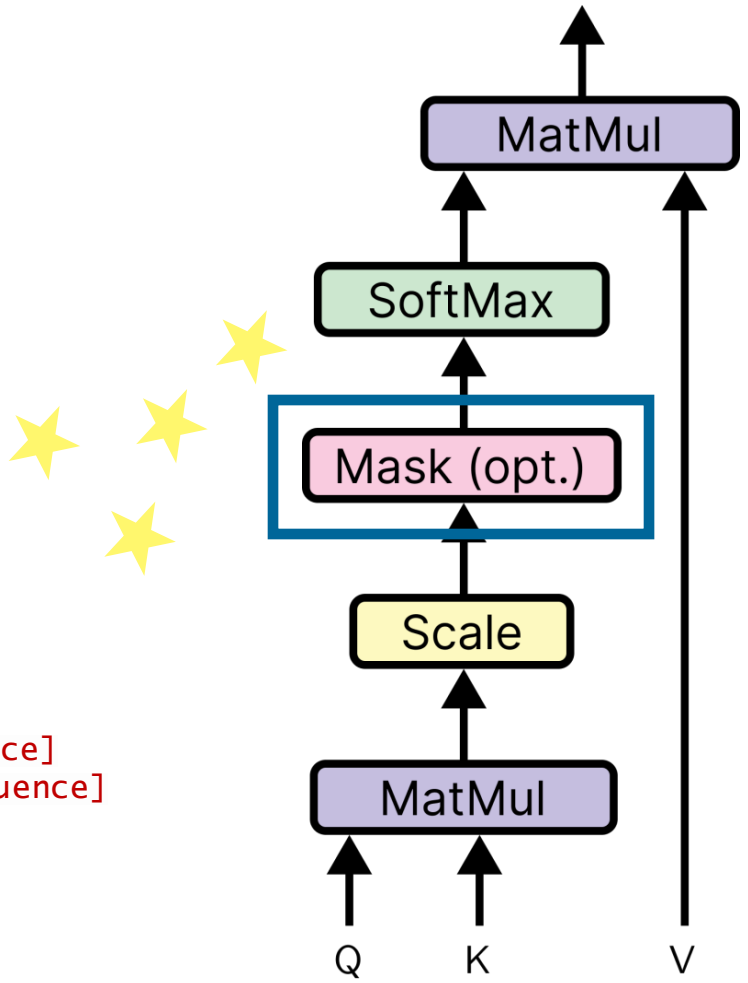


$$softmax(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}}$$

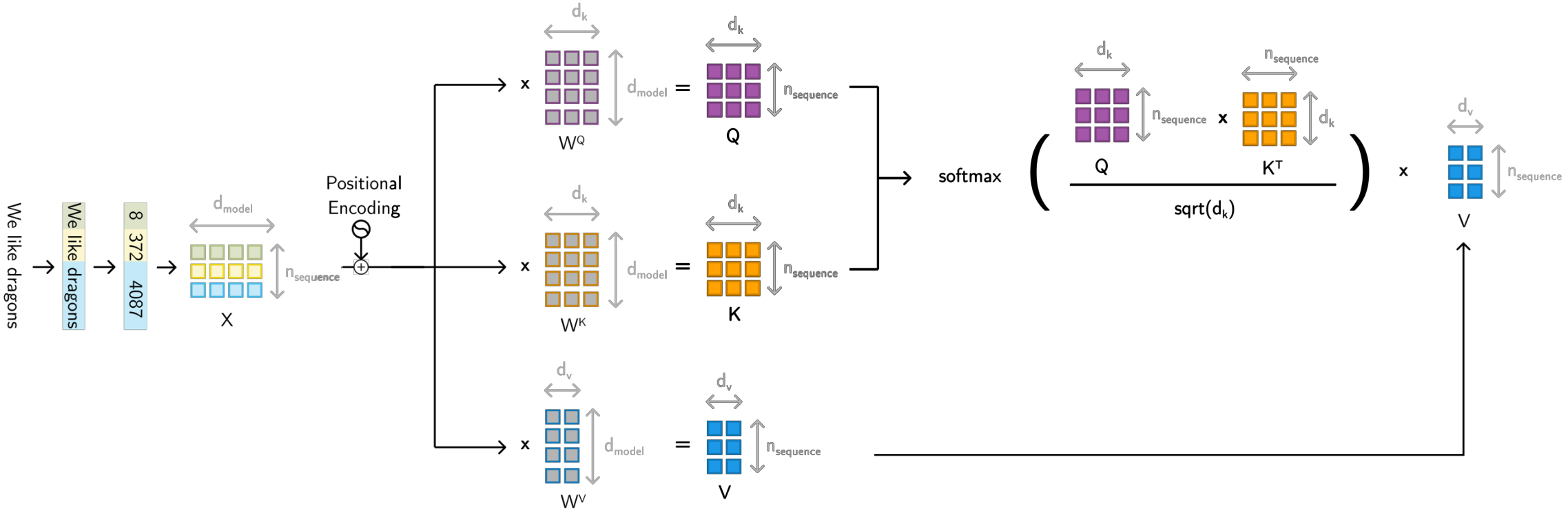
The attention mechanism in code

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

```
import torch
from torch import nn

d_model = 128
d_k = d_v = 64
x = torch.randn(n_sequence, d_model) # [n_sequence, d_model]

W_Q = nn.Linear(d_model, d_k, bias=False) # [d_model, d_k]
W_K = nn.Linear(d_model, d_k, bias=False) # [d_model, d_k]
W_V = nn.Linear(d_model, d_v, bias=False) # [d_model, d_v]

Q = W_Q(x) # [n_sequence, d_k]
K = W_K(x) # [n_sequence, d_k]
V = W_V(x) # [n_sequence, d_k]

attn_scores = torch.matmul(Q, K.T) / math.sqrt(d_k)     # [n_sequence, n_sequence]
attn_weights = nn.functional.softmax(attn_scores, dim=-1)   # n_sequence, n_sequence]
self_attention = torch.matmul(attn_weights, V)              # [n_sequence, d_k]


W_O = nn.Linear(d_v, d_model, bias=False)         # [d_v, d_model]
output = W_O(self_attention)                      # [n_sequence, d_model]
```

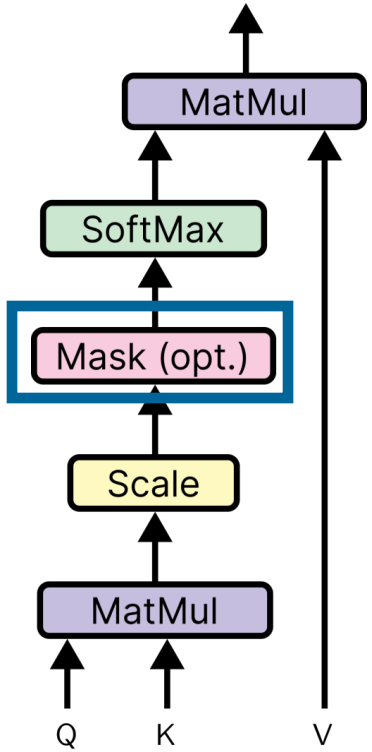


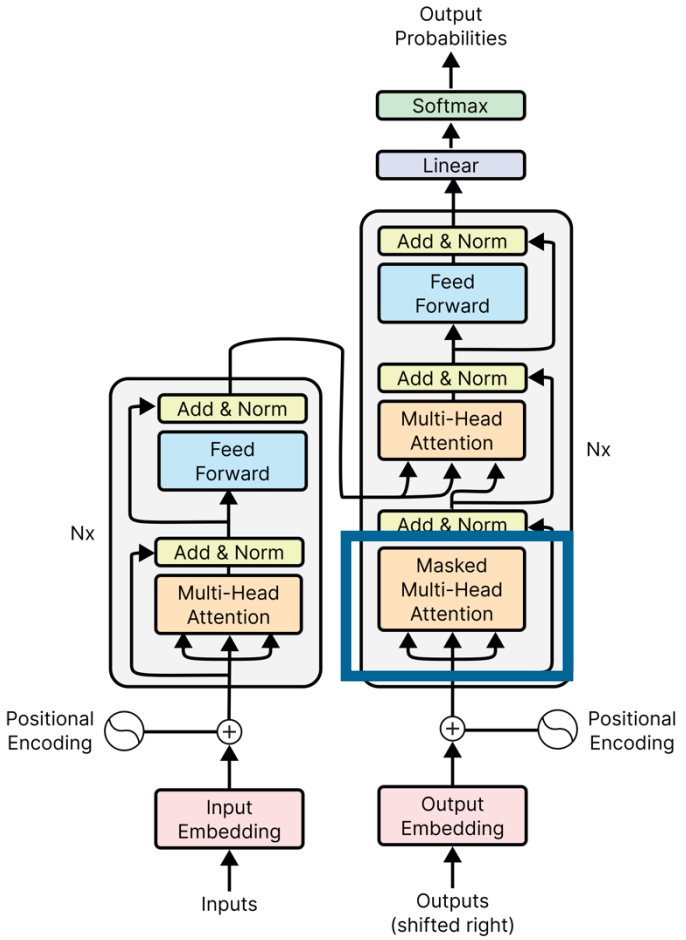
From Input To Attention Mechanism



Masked attention

A bit more magic

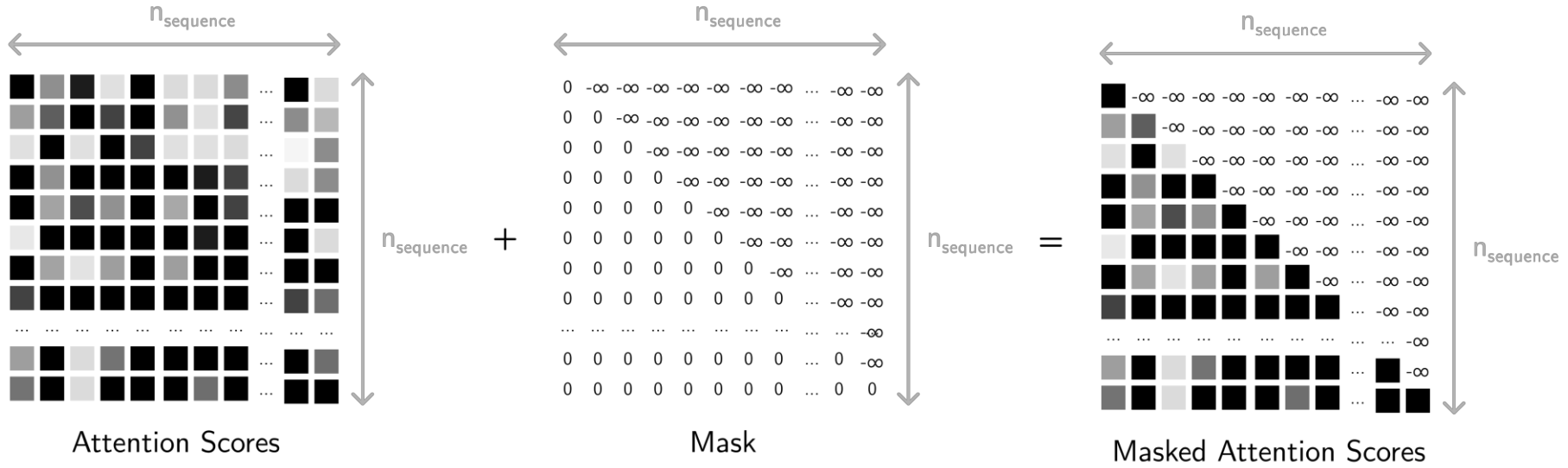


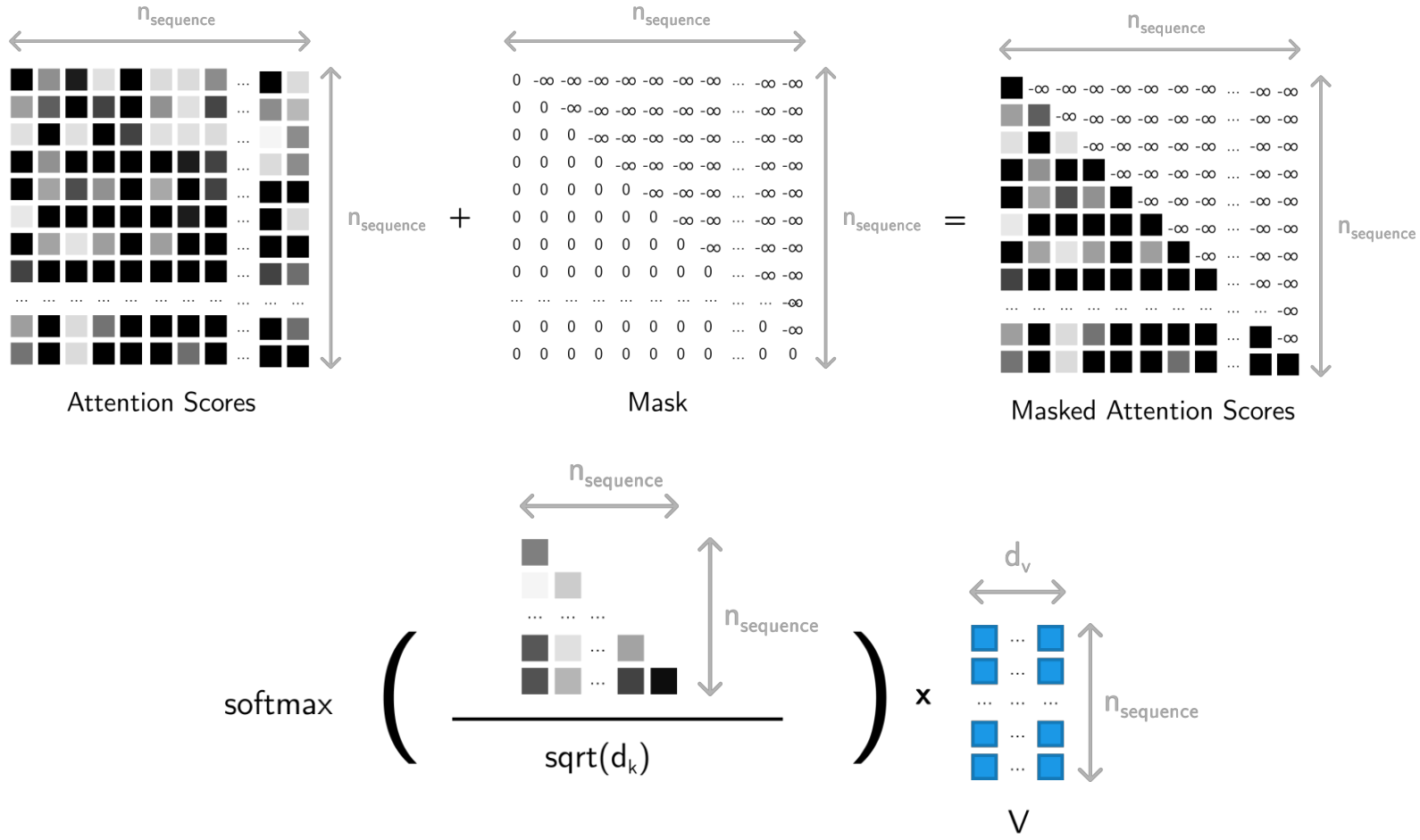


The probability of a sequence is the product of the probability of the first token and the conditional probabilities of each subsequent token given all previous tokens.

$$p(x) = p(s_1) \cdot p(s_2 \,|s_1) \cdot p(s_3 \,|s_1, s_2) \cdot \ldots \cdot p(s_k \,|s_1, \ldots, sk_{-1})$$
$$= p(s_1) \cdot \prod_{i=2}^{k} p(si|s_1, \ldots, si_{-1})$$

When learning factorised conditional probabilities, only the previous tokens in a sequence should be considered.

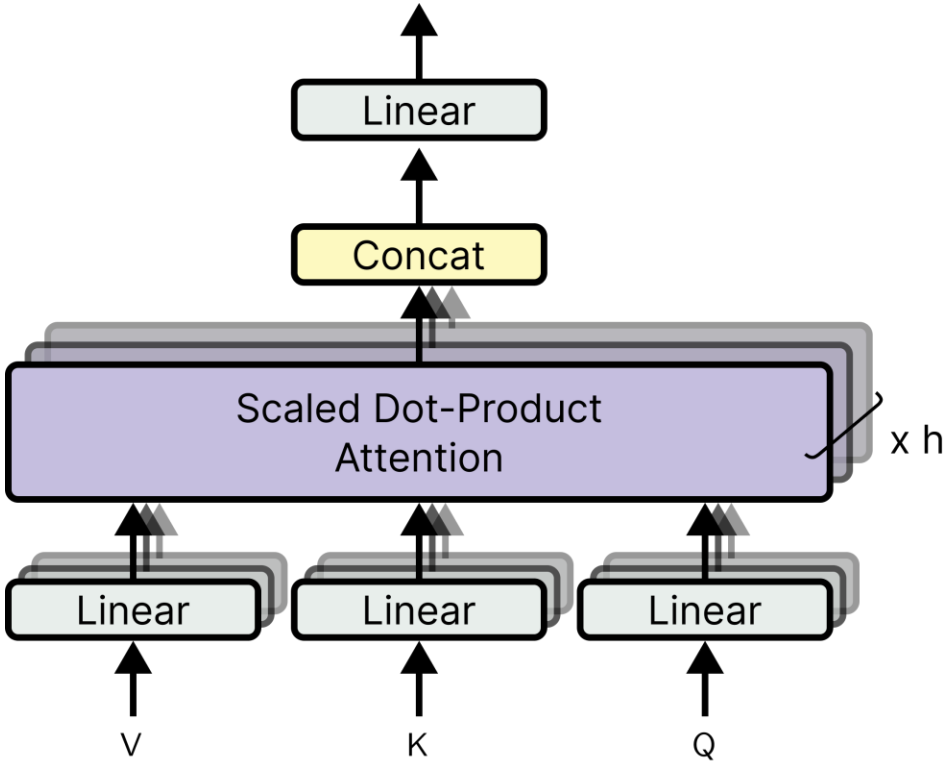




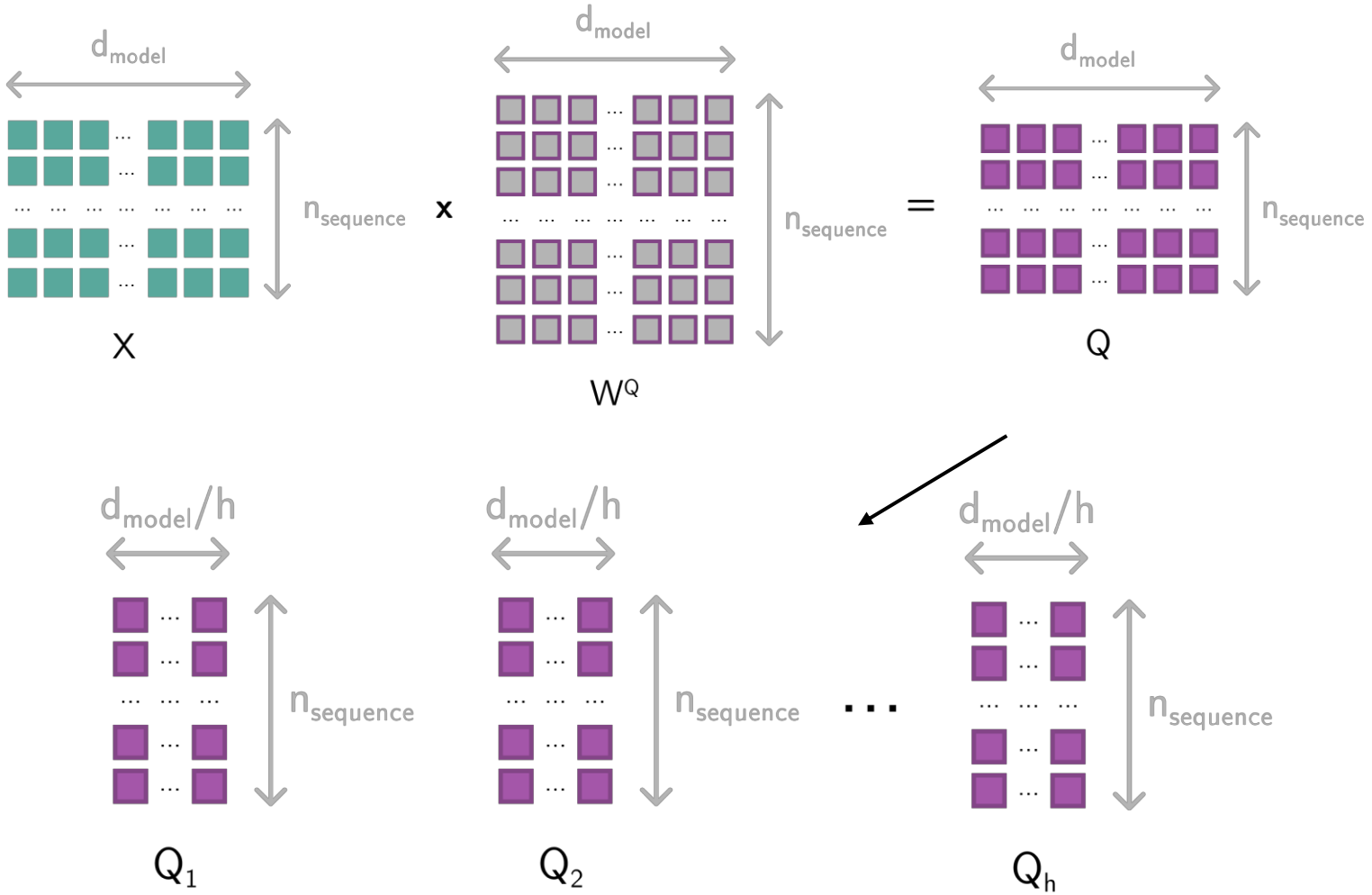
The **masked** attention mechanism computes and ingests *how much* and *how* each **previous** token should influence the representation ("meaning") of every other token within the input sequence.

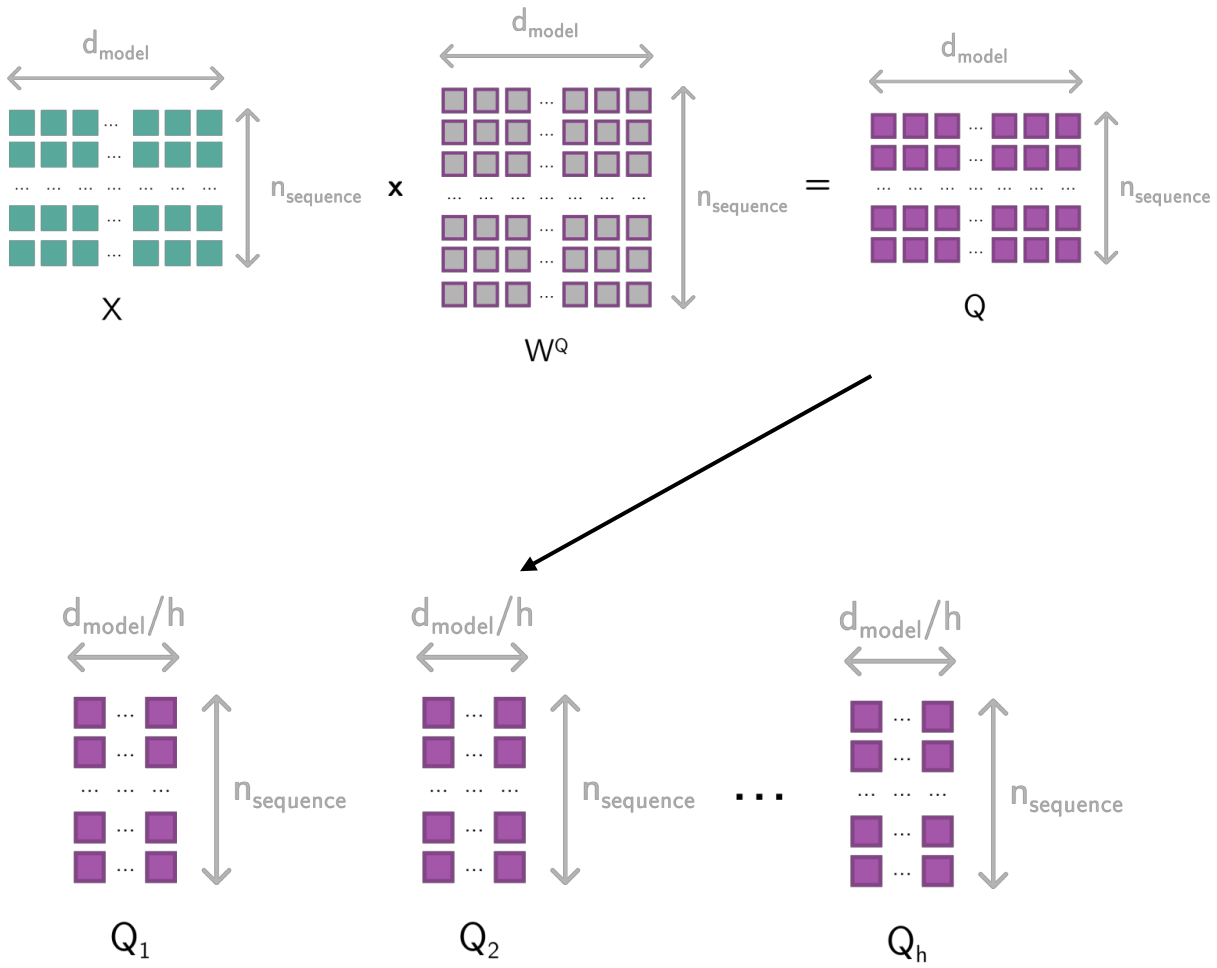
Multi-Head Attention

How transformers pay attention to different "semantic properties" of tokens



The goal of the multi-head attention is to attend to tokens from different perspectives in parallel.






```python
import torch
from torch import nn

d_model = 256
n_sequence = 1048
h = 8
d_k = d_model // h

x = torch.randn(n_sequence, d_model)

W_Q = nn.Linear(d_model, d_model, bias=False)
W_K = nn.Linear(d_model, d_model, bias=False)
W_V = nn.Linear(d_model, d_model, bias=False)
W_O = nn.Linear(d_model, d_model, bias=False)

Q = W_Q(x)
K = W_K(x)
V = W_V(x)

Q = Q.view(n_sequence, h, d_k).transpose(0, 1)
K = K.view(n_sequence, h, d_k).transpose(0, 1)
V = V.view(n_sequence, h, d_k).transpose(0, 1)
```

In multi-head attention, we produce *h* slices each of shape *n_sequence, d_k* and stack these matrices along the first dimension. *Q, K* and *V* have the shape *h, n_sequence, d_k*

```python
import torch
from torch import nn

d_model = 256
n_seq = 1048
h = 8
d_k = d_model // h

x = torch.randn(n_seq, d_model)

W_Q = nn.Linear(d_model, d_model, bias=False)
W_K = nn.Linear(d_model, d_model, bias=False)
W_V = nn.Linear(d_model, d_model, bias=False)
W_O = nn.Linear(d_model, d_model, bias=False)

Q = W_Q(x) # [n_seq, d_model]
K = W_K(x) # [n_seq, d_model]
V = W_V(x) # [n_seq, d_model]

# [h, n_seq, d_k]
Q = Q.view(n_seq, h, d_k).transpose(0, 1)
K = K.view(n_seq, h, d_k).transpose(0, 1)
V = V.view(n_seq, h, d_k).transpose(0, 1)

K_T = K.transpose(-2, -1)
```

```python
attn_scores = torch.matmul(Q, K_T) / torch.sqrt(torch.tensor(d_k))
attn_weights = nn.functional.softmax(attn_scores, dim=-1)
attn = torch.matmul(attn_weights, V) # [h, n_seq, d_k]

attn = attn.transpose(0, 1).contiguous().view(n_seq, d_model)

output = W_O(self_attention)
```

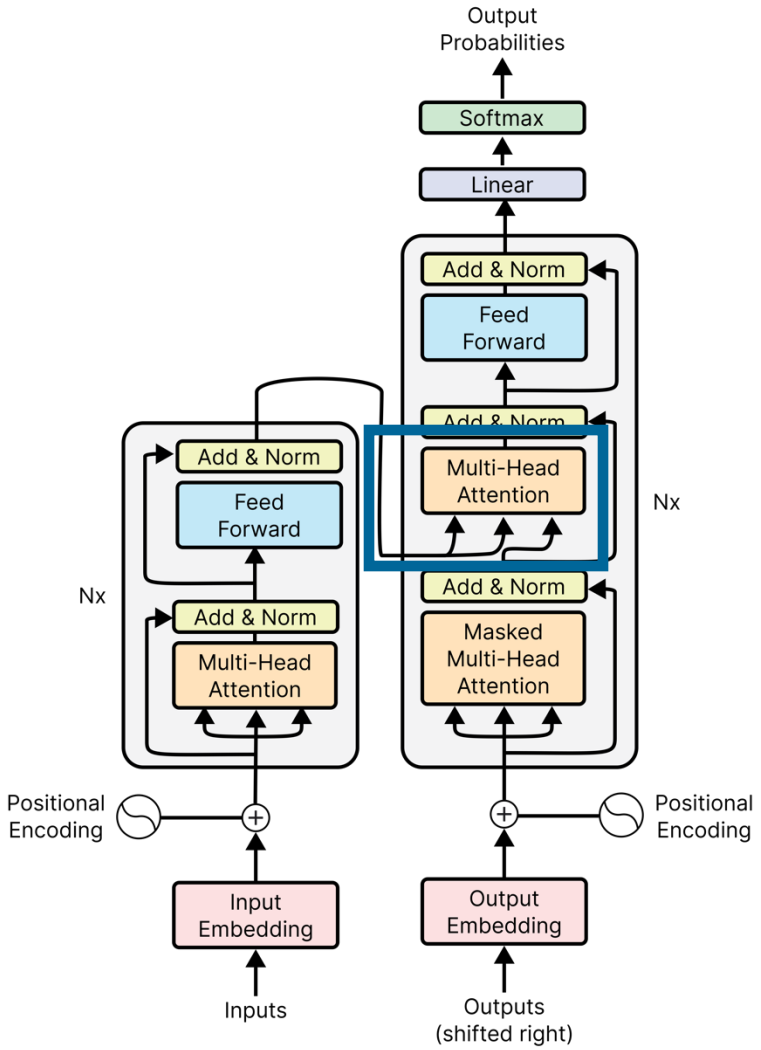
$$MultiHead(Q, K, V) = Concat(head_1, head_2, \dots headh)W^O$$

$$head_i = Attention(Qi, Ki, Vi)$$

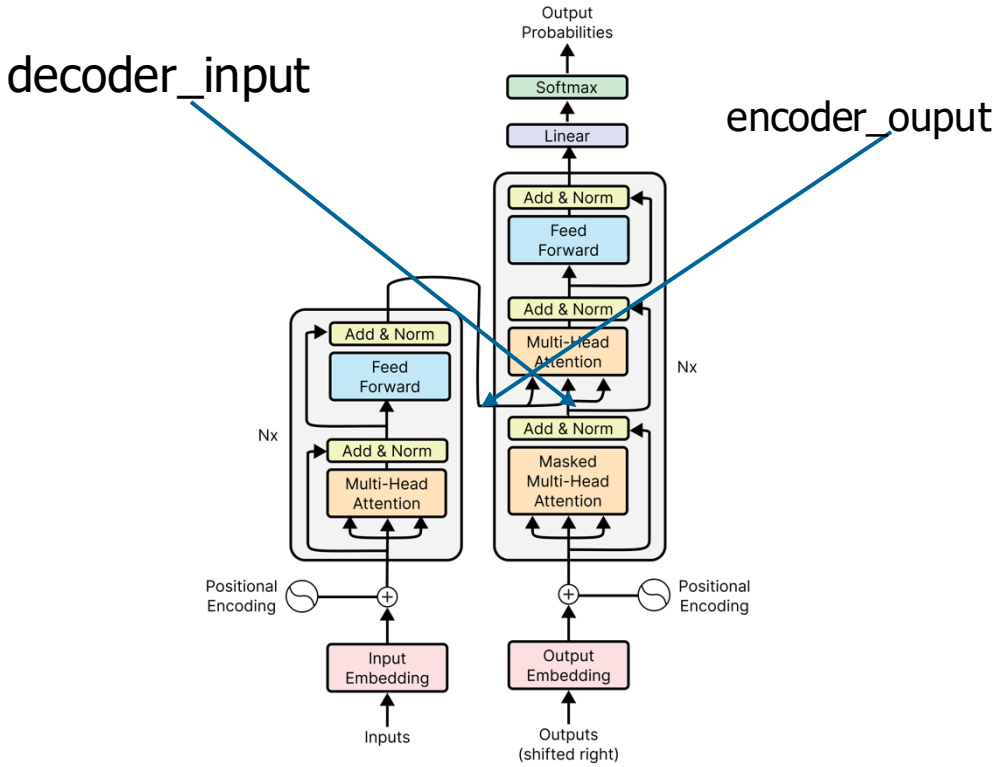
Multi-head attention learns contextual dependencies from different perspectives in parallel by splitting the **Q**uery, **K**ey and Value matrices into multiple heads.

Cross Attention

Building the bridge between the encoder and decoder



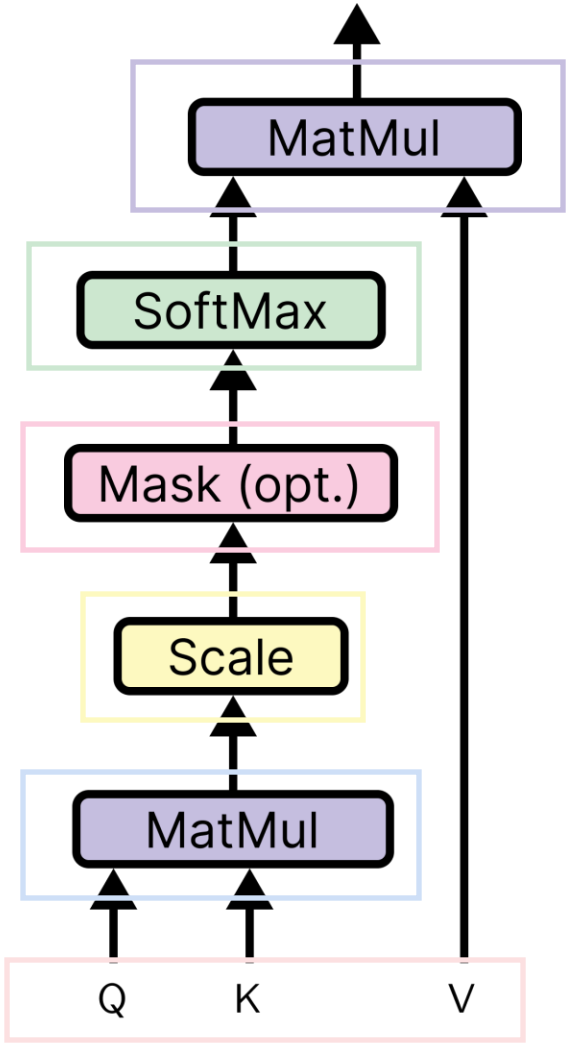
The query (Q) comes from the decoder layer. The key (K) and value (V) come from the encoder's output.




```
# ... everything as in previous code samples

Q = W_Q(decoder_input)
K = W_K(encoder_output)
V = W_V(encoder_output)

# ... everything as in previous code samples
```

Yes, it's that simple.

Recap



Create **Q**uery, **K**ey and **V**alue matrix from encoder output and/or decoder input.

Calculate attention scores to measure how much attention tokens should pay to each other.

Scale the dot product to avoid ensure stable gradient flow.

Optionally hide future tokens to ensure that token predictions depend only on previous tokens.

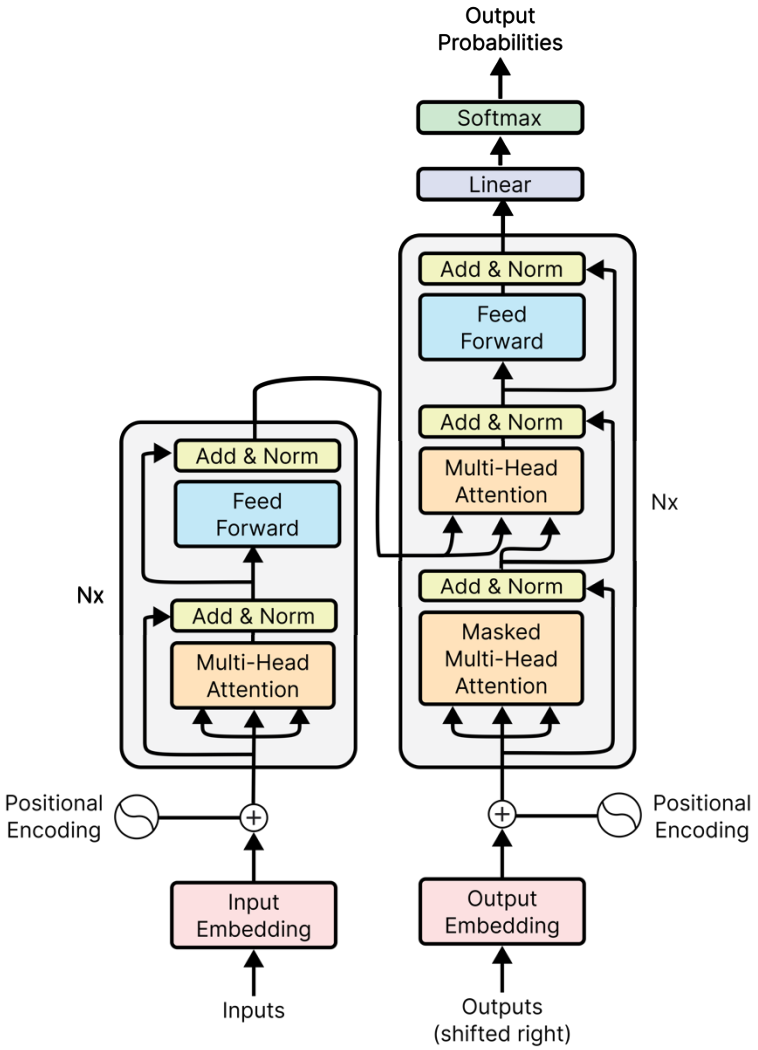
Create attention weights using the softmax function to normalise embeddings.

Multiply attention weights by the **V**alue matrix

Project result to output shape [n_contex, d_model]

A bird's eye view of the

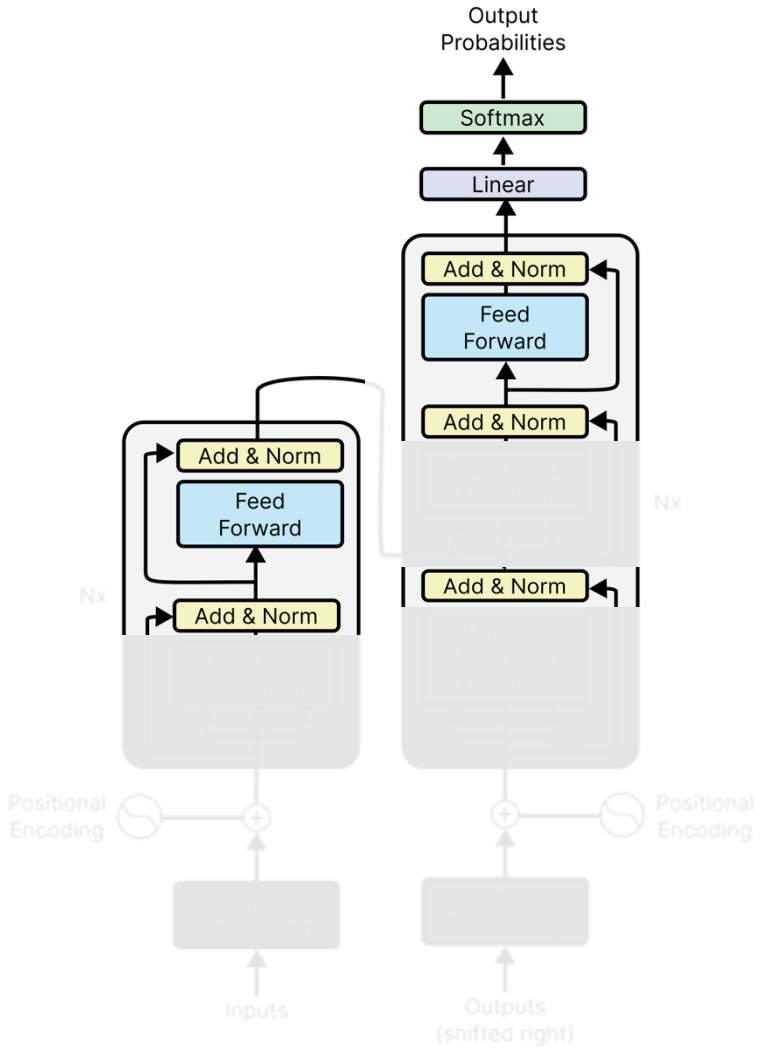
Transformer Architecture.

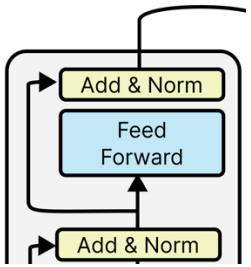


A bird's eye view of the

Transformer Architecture.

Add & Norm and Feed Forward Networks



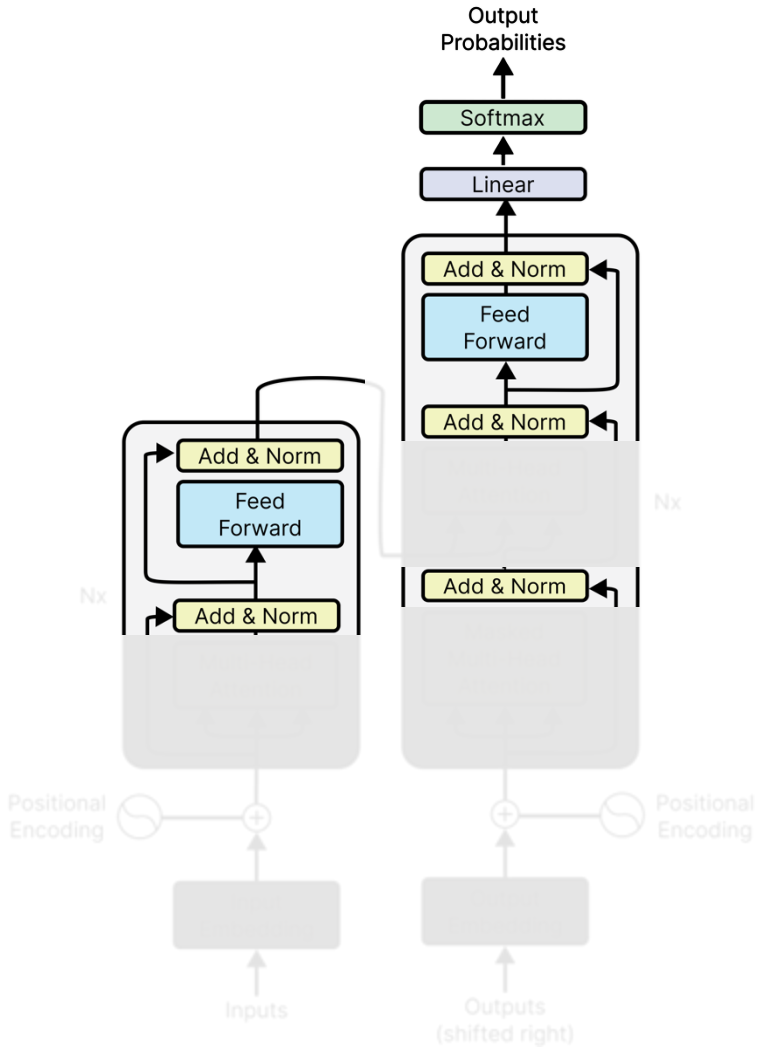



```python
import torch
from torch import nn

d_model = 128
d_ff = 4*d_model
n_sequence = 200 # sample sequence length
x = torch.randn(n_sequence, d_model) # [n_sequence, d_model]

norm = nn.LayerNorm(d_model)
feed_forward = nn.Sequential(
      nn.Linear(d_model, dim_ff),
      nn.ReLU(),
      nn.Linear(dim_ff, d_model)
)

output = norm(x + feed_forward(x))
```

1. For details of the layer norm take a look at https://docs.pytorch.org/docs/stable/generated/torch.nn.LayerNorm.html

A bird's eye view of the

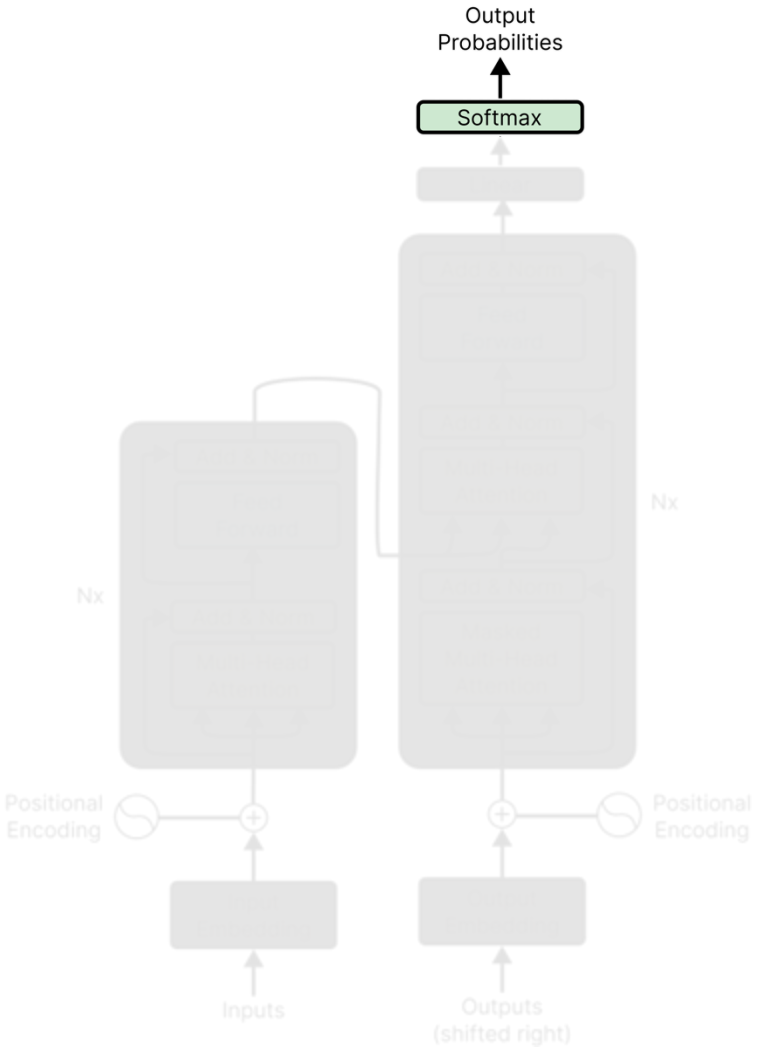
Transformer Architecture.

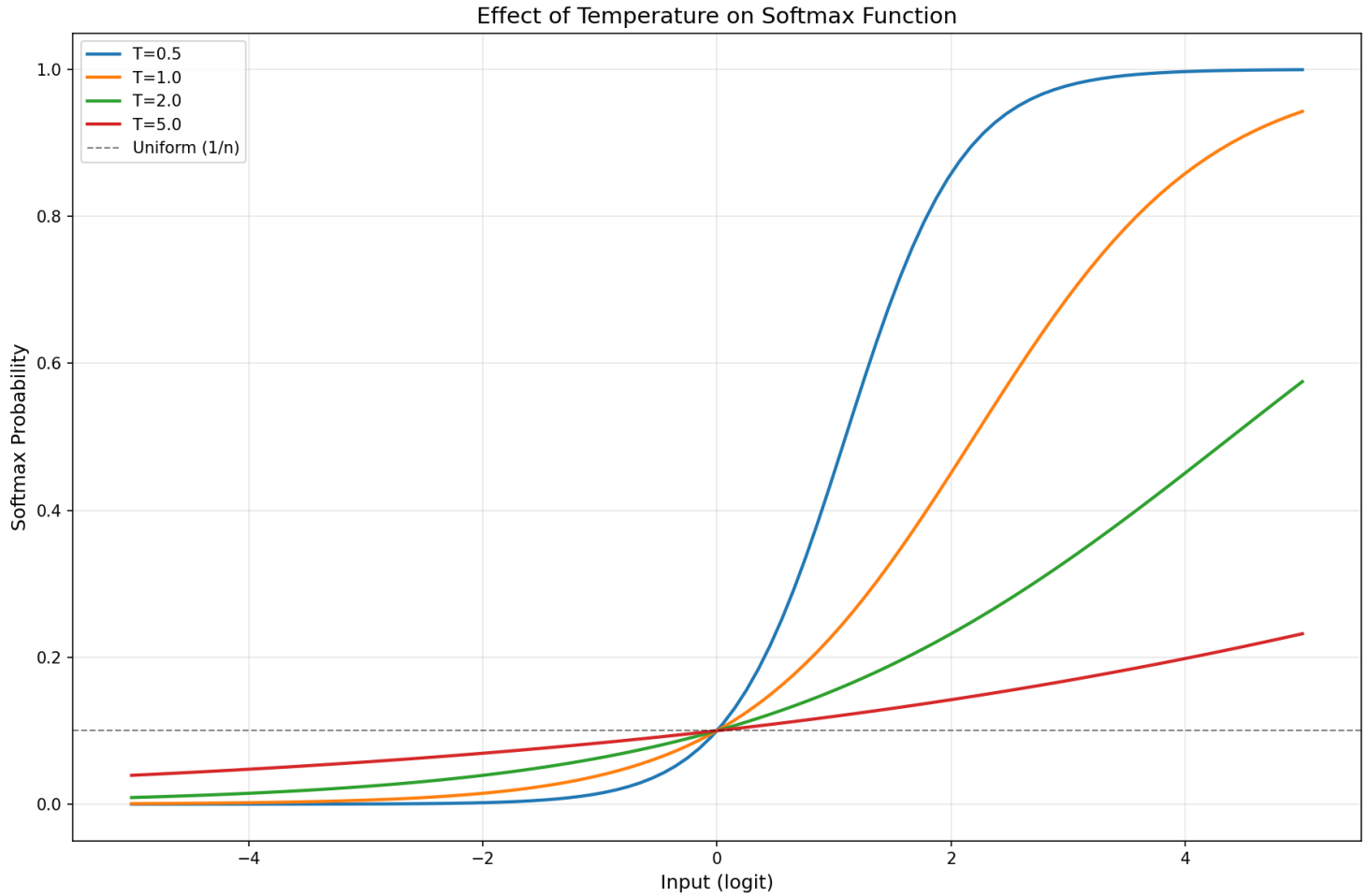


A bird's eye view of the

Transformer Architecture.

Final Softmax Function

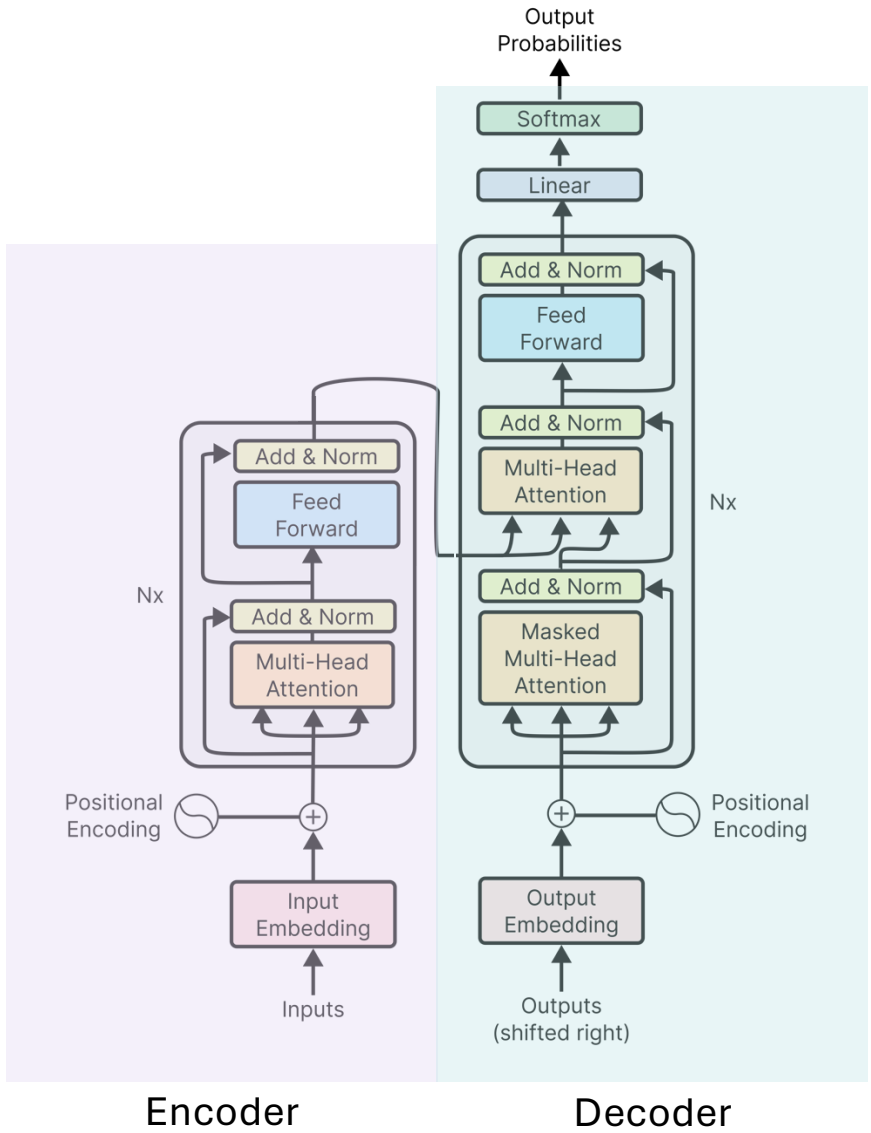


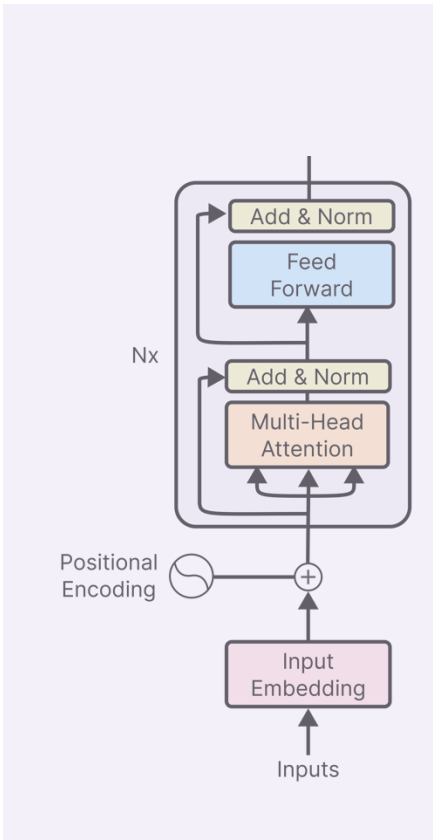


$$softmax(x_i) = \frac{e^{\frac{x_i}{T}}}{\sum_j e^{\frac{x_j}{T}}}$$

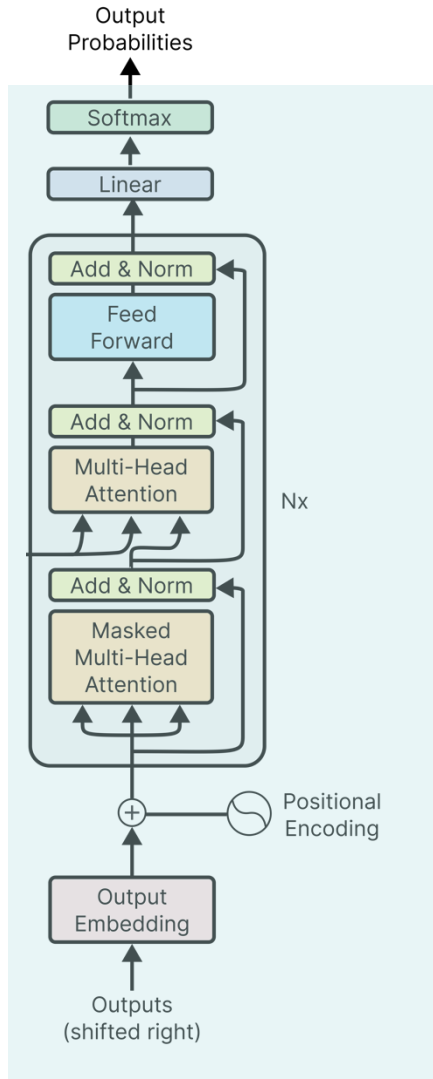
Generative AI

Beyond Vaswani et. al

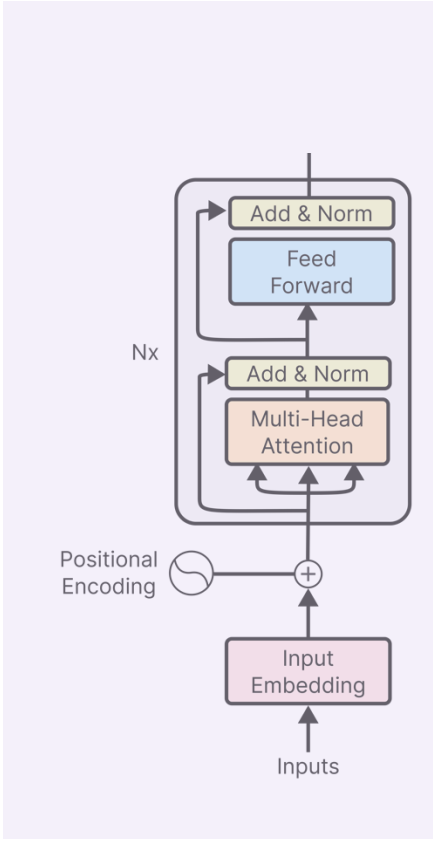

Output Probabilities
Softmax
Linear
Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
Add & Norm
Masked Multi-Head Attention
Nx
Positional Encoding
Output Embedding
Outputs (shifted right)
Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
Nx
Positional Encoding
Input Embedding
Inputs
Encoder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



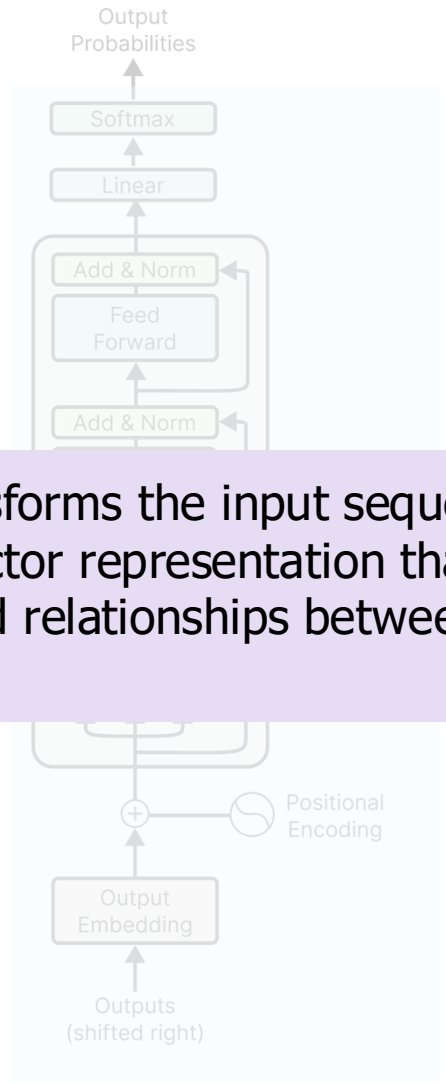
Encoder



Decoder



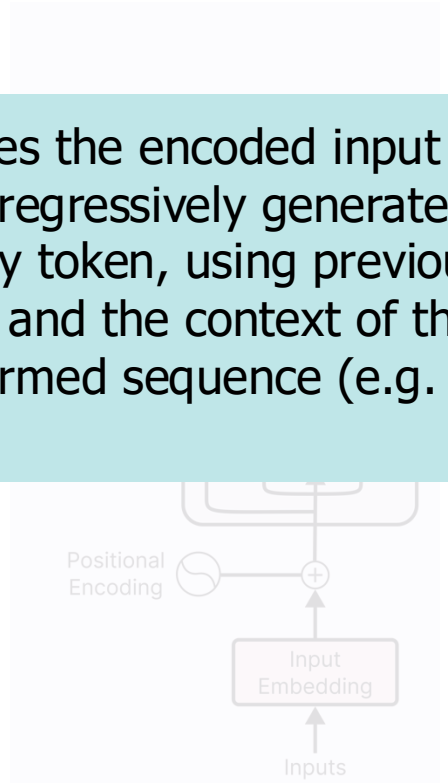
Encoder



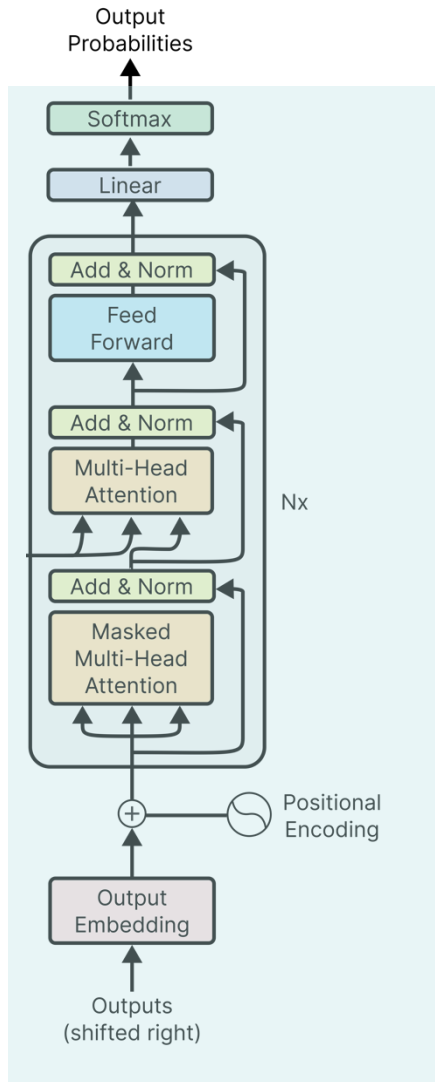
Decoder

The **encoder** transforms the input sequence into a rich, contextual vector representation that captures the meaning of and relationships between elements.

The **decoder** takes the encoded input from the encoder and autoregressively generates an output sequence token by token, using previously generated tokens and the context of the encoder to produce a transformed sequence (e.g. a translation or summary).

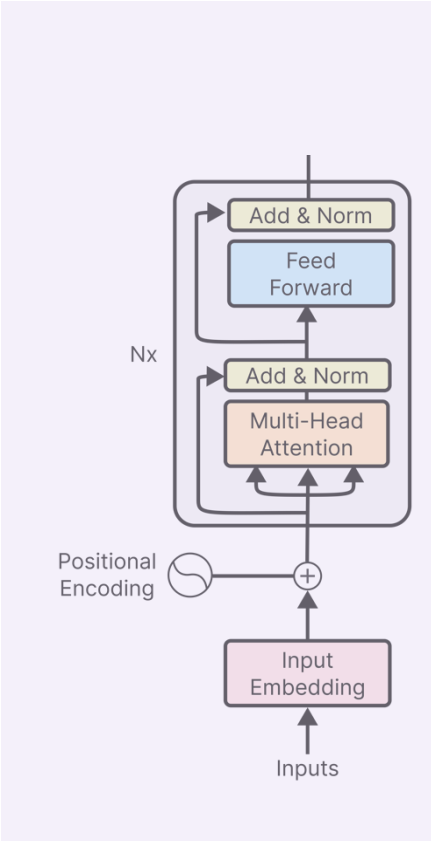


Encoder

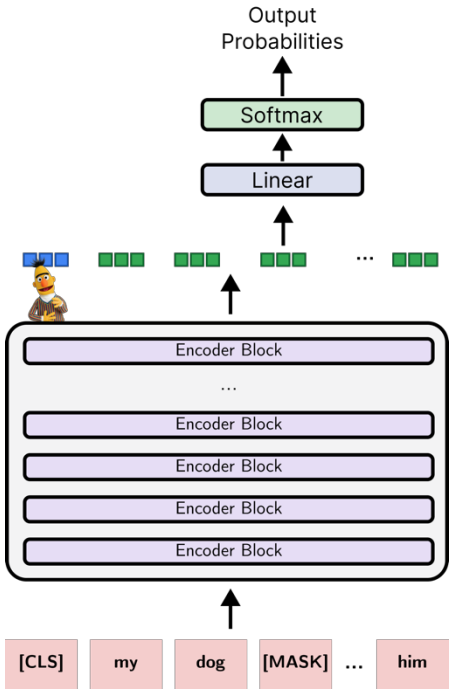


Decoder

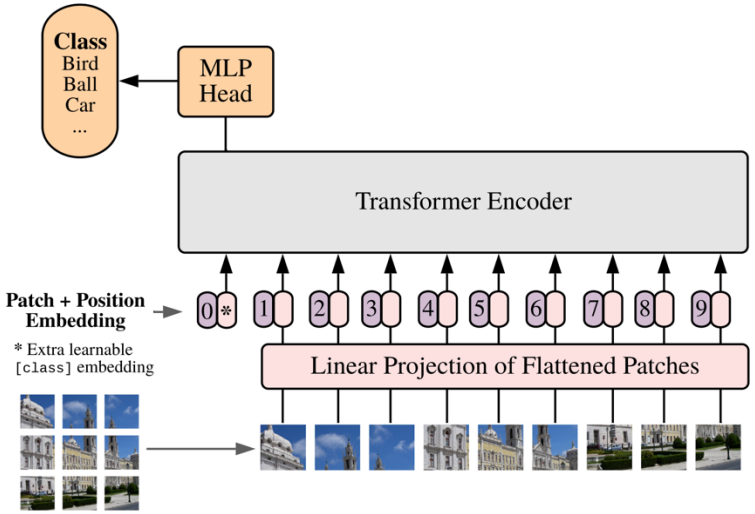
Encoder Only Models



Encoder

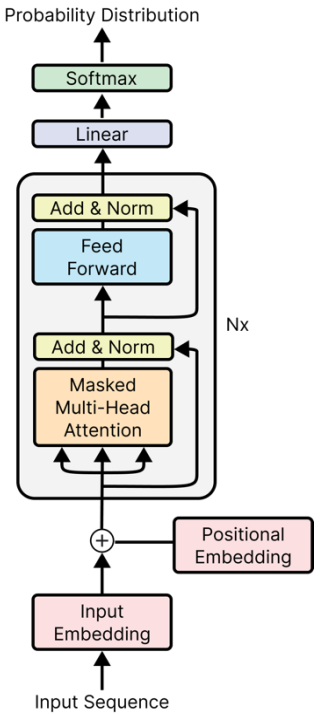


Encoder Only Transformers (e.g. BERT)[2]



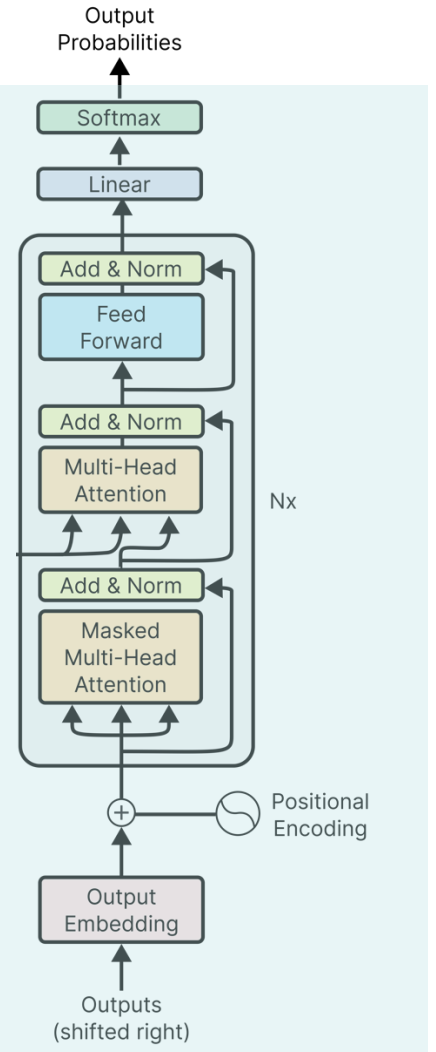
Encoder Only Vision Transformer

Decoder Only Models



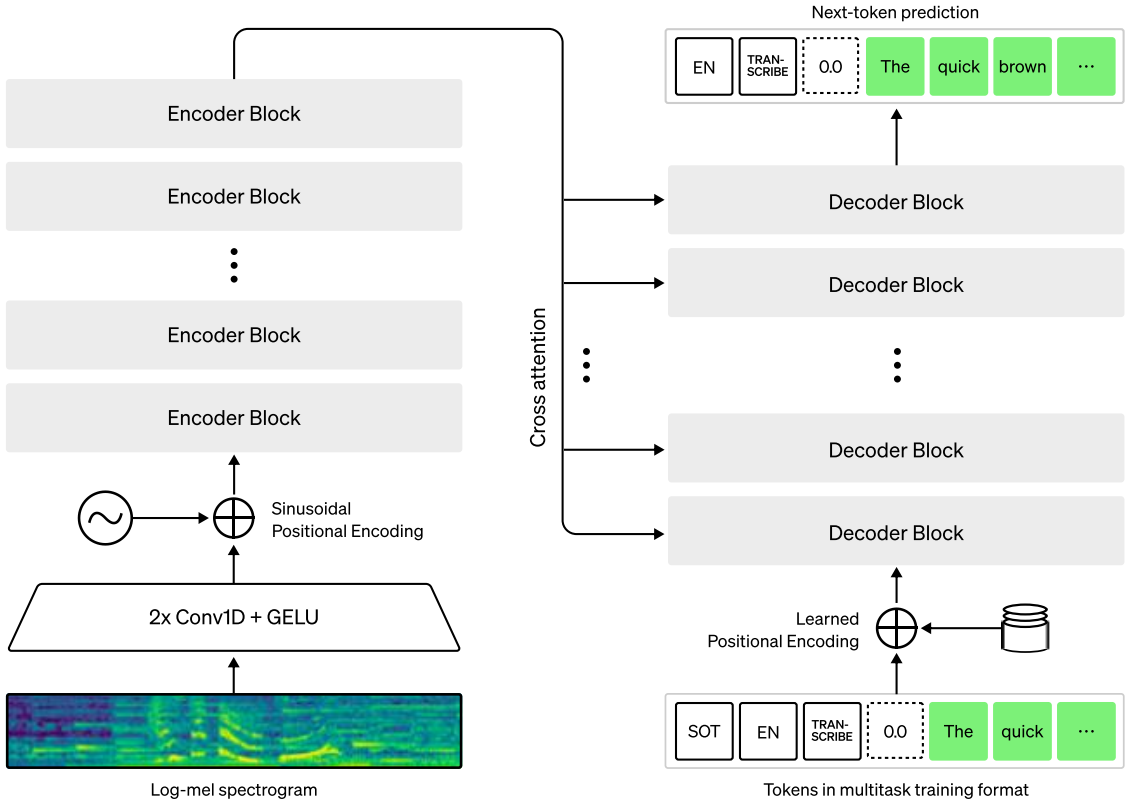
Decoder Only Transformers (e.g. GPT)[3]

[…] Pretty much any LLM you can think of […]



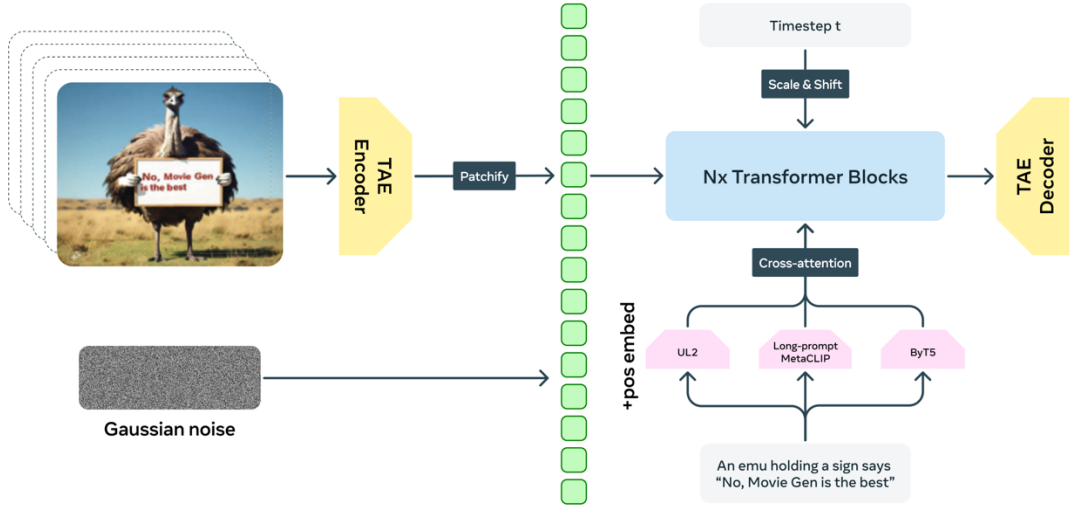
Decoder

Multimodal Variations of Transformer Models



Open AI Whisper

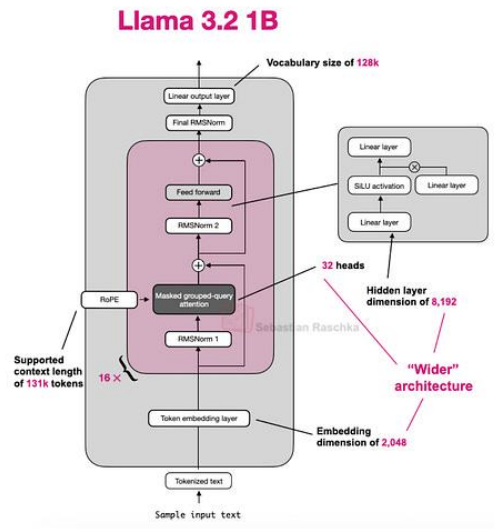
https://openai.com/index/whisper/

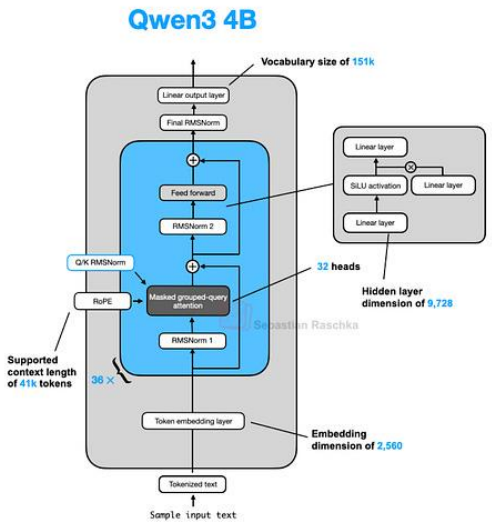


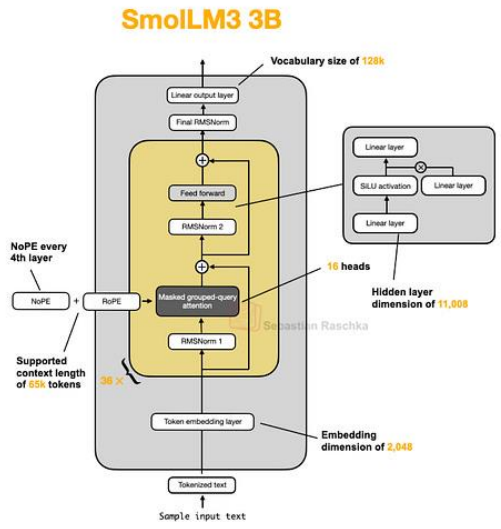
Meta Movie Gen

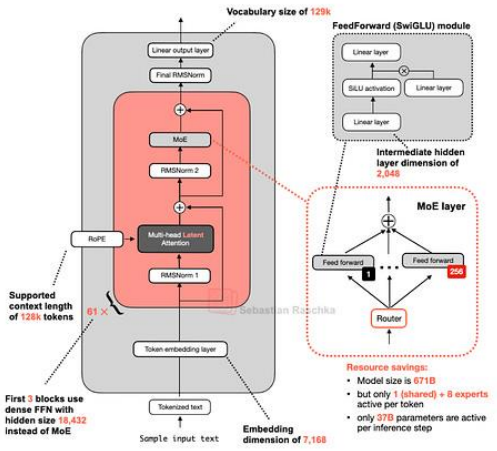
https://arxiv.org/abs/2410.13720/

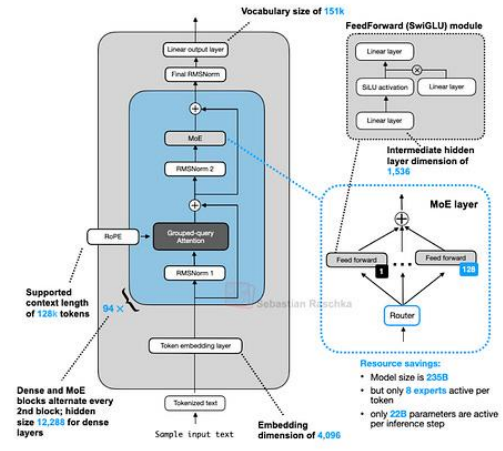
The Big LLM Architecture Comparison by Sebastian Raschka

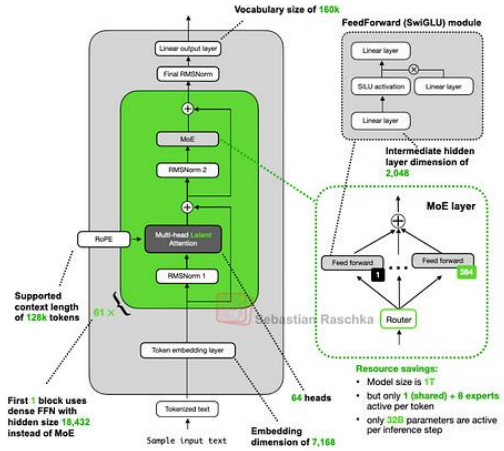












https://magazine.sebastianraschka.com/p/the-big-llm-architecture-comparison