Transformers

Neural Networks in Disguise

History of NLP advancements

- **1950s–60s** — Rule-based systems (grammar, symbolic AI)
- **1980s–90s** — Statistical NLP (n-grams, HMMs)
- **2000s** — Probabilistic ML (SVMs, CRFs)
- **2010–2016** — Neural NLP (RNNs, LSTMs, word2vec)
- **2017** — Transformers (attention-based models)
- **2018–2019** — Pretrained models (BERT, GPT)
- **2020–present** — Large Language Models (GPT-3+, multimodal, RLHF)

Motivation: LSTM shortcomings

> LSTMs struggle with scaling sequence length and parallelism; Transformers remove recurrence and replace it with global, parallel attention.

LSTM shortcomings

- **Sequential computation bottleneck**: Cannot parallelize across time steps → slow training/inference.
  - Bottleneck ⟶ all information flows through a single, **sequentially** updated hidden state.
- **Long-range dependency limits**: Gating helps, but still degrades over very long contexts.
- **Fixed-size hidden state:** Compresses entire history into a single vector → information loss.
- **Inefficient memory access**: No direct lookup of past tokens; must propagate through all steps.
- **Vanishing gradients**: Gradient path length scales with sequence length: Still harder to optimize than constant-path architectures.

What Transformers Fix

Overview

- A transformer model is a neural network that learns context and meaning by tracking relationships in sequential data like the words in this sentence.
- Learn context in parallel meaning they do not need to “process” the first word in a sentence before processing a word in the middle of a sentence.
- Transformers “calculate” the “context” referred to as attention of all words in a sequence simultaneously using the **attention mechanism**
- The attention mechanism is the heart of the transformer, the rest of the architecture is found in typical neural networks
- This all really took off in 2017 with the paper Attention Is All You Need

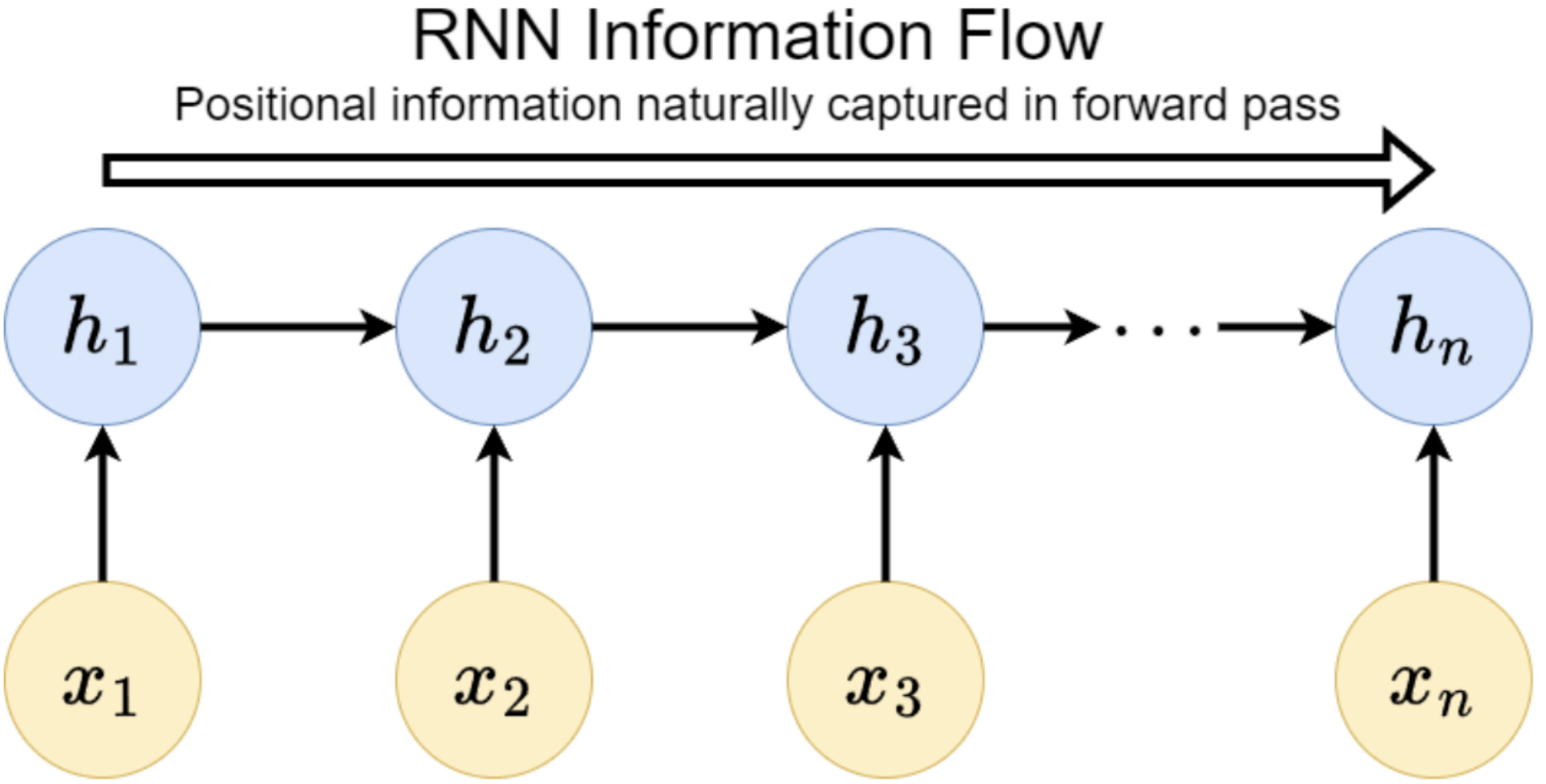

Attention is all you need
A Vaswani, N Shazeer, N Parmar, ... - Advances in neural ..., 2017 - proceedings.neurips.cc
... to attend to all positions in the decoder up to and including that position. We need to prevent ... We implement this inside of scaled dot-product attention by masking out (setting to $-\infty$) ...
☆ Save 99 Cite Cited by 236756 Related articles All 71 versions 99

Information Flow: LSTM vs Attention

- Direct token-to-token interaction via attention → **efficient modeling of long-range dependencies**

RNN:

- **Long path length:** info from $h_1 \longrightarrow h_n$ must pass through all intermediate states
- **Unidirectional limitation:** h_1 cannot access future tokens $(h_2, \ldots, h_n)$
- **Bidirectional RNNs help partially:** allow backward flow
- **Still sequential bottleneck:** even bidirectional models require step-by-step propagation across all states



Attention mechanism

- **Direct connections:** any position can attend to any other (no intermediate steps)
- **Constant path length:** long-range dependencies handled efficiently
- **Bidirectional context:** information flows forward and backward

Overview:

Let’s start with a big picture overview

Diagram from 2017 paper:

Here is the famous diagram from the 2017 paper that introduced the transformer architecture.

The model is broken into an encoder (left), and a decoder (right)

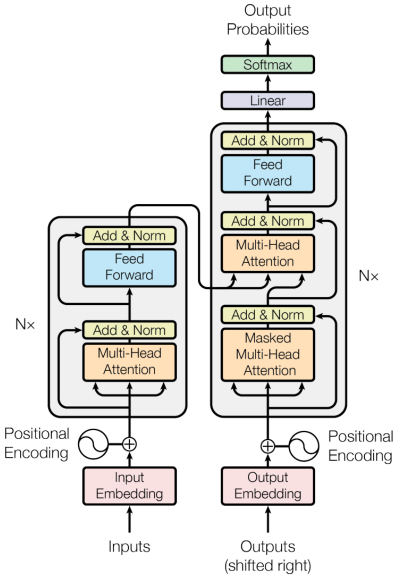


Figure 1: The Transformer - model architecture.

Learning task:

> The following discussion will focus on the case of sequence to sequence learning via

Transformer as Black Box - Inference - Step 4

Token Generation Part 4

In this example the transformer is tasked with predicting the next Spanish token. The inputs are the the entire English sentence and the correct previous Spanish tokens in the sequence. The loss is determined by comparing the predicted token to the actual next token in the Spanish sequence.

Teacher forcing ensures the decoder always has access to the correct previous token, even if it did not correctly predict it. This also allows for parallel training as the transformer does not need to make a prediction for each previous token in the sequence before making a prediction of the next token.

n : represent the size of the vocabulary (defined tokens)
m : the number of tokens in a sequence
b : The number of sequences in a batch
p : An element of the next token probability tensor
<SOS> : Start of sentence token
<EOS> : End of sentence token
<PAD> : Padding token, should always be ignored by the model

Step 1:
The OHE sequences are embedded. The Encoder receives the input sequences and the decoder receives the output sequeneces. The output sequences tell the decoder what the previous tokens were before making a next token predicion.

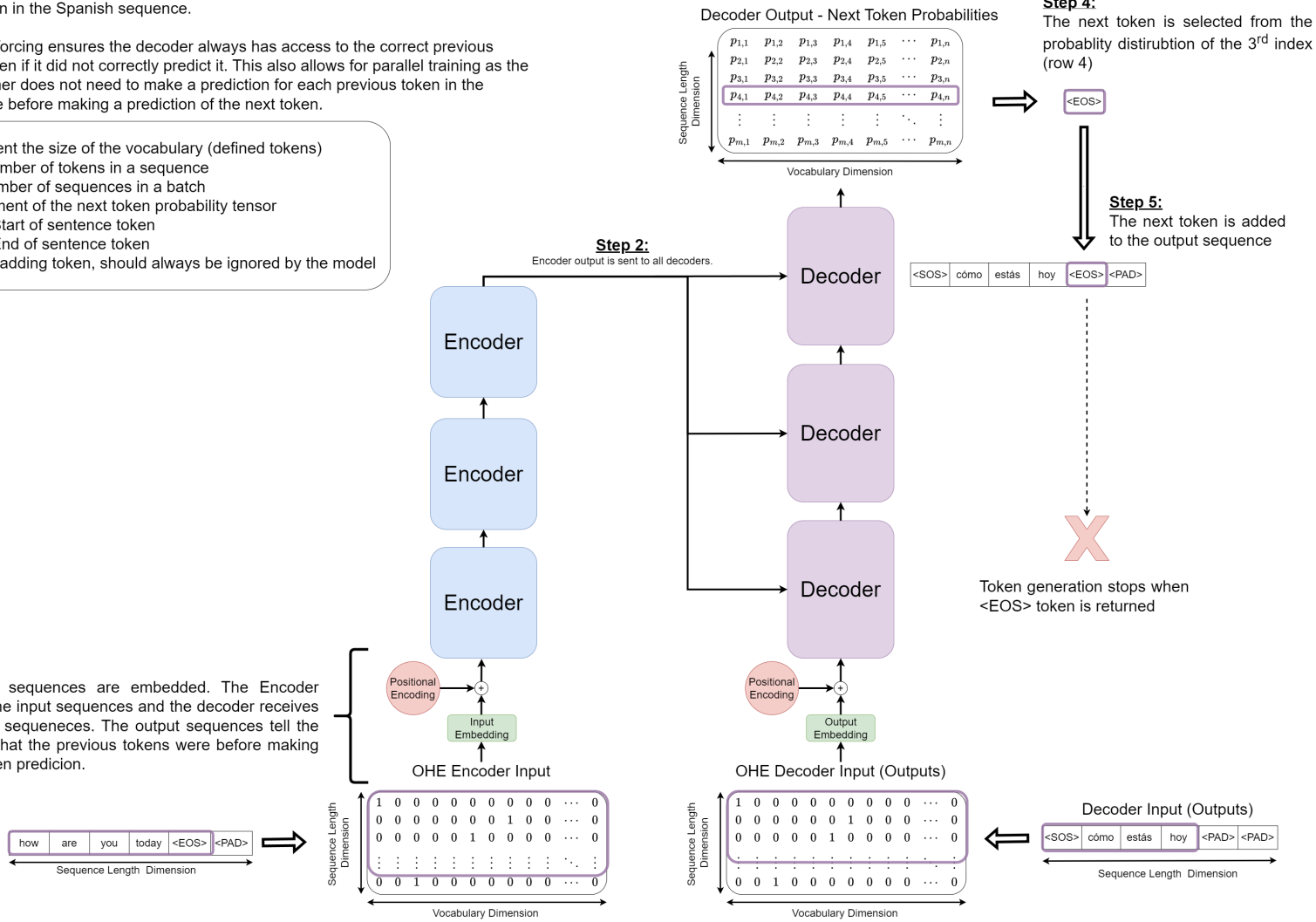
Step 2:
Encoder output is sent to all decoders.

Step 3:
Decoder generates a next token prediction for all positions (position in the a sentence) and sequences in a single forward pass. The output is a probability distribution over all tokens in the vocabulary

Step 4:
The next token is selected from the probablity distirubtion of the 3rd index (row 4)

Step 5:
The next token is added to the output sequence

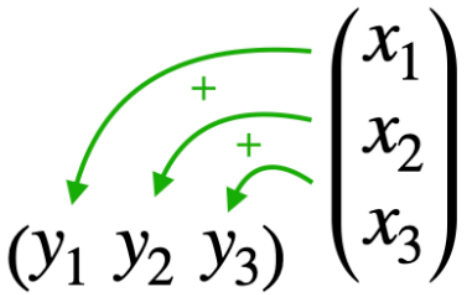
Token generation stops when <EOS> token is returned



Aside: Relevant matrix operations`

Key operation-1: Dot product

- The dot product is one of the most important operations associated with vectors
- The operation takes two vectors and converts them into a scalar.



multiply like components and add

$$= y_1x_1 + y_2x_2 + y_3x_3$$

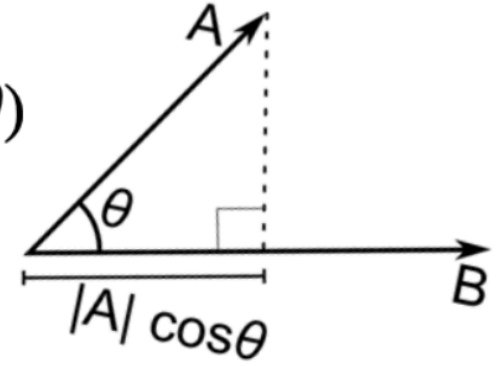
row vector times column vector

$$\mathbf{y}^T\mathbf{x} = \mathbf{y} \cdot \mathbf{x}$$

$$1 \times 3 \quad 3 \times 1 \rightarrow 1 \times 1$$

- The dot product can be used to determine the amount of $\mathbf{A}$ which is parallel to $\mathbf{B}$
- Therefore, it is useful as a similarity metric.

$$\mathbf{A} \cdot \mathbf{B} = |\mathbf{A}||\mathbf{B}|cos(\theta)$$



othogonality

$$\mathbf{y} \cdot \mathbf{x} = 0$$

$$cos(90^o) = 0$$

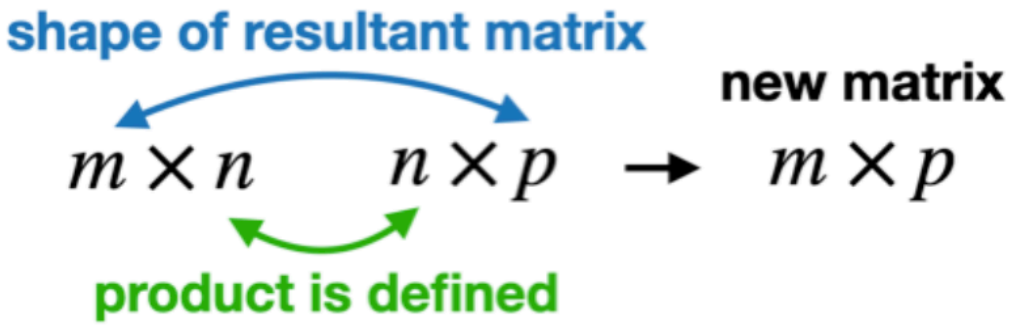
$$cos(0^o) = 1$$

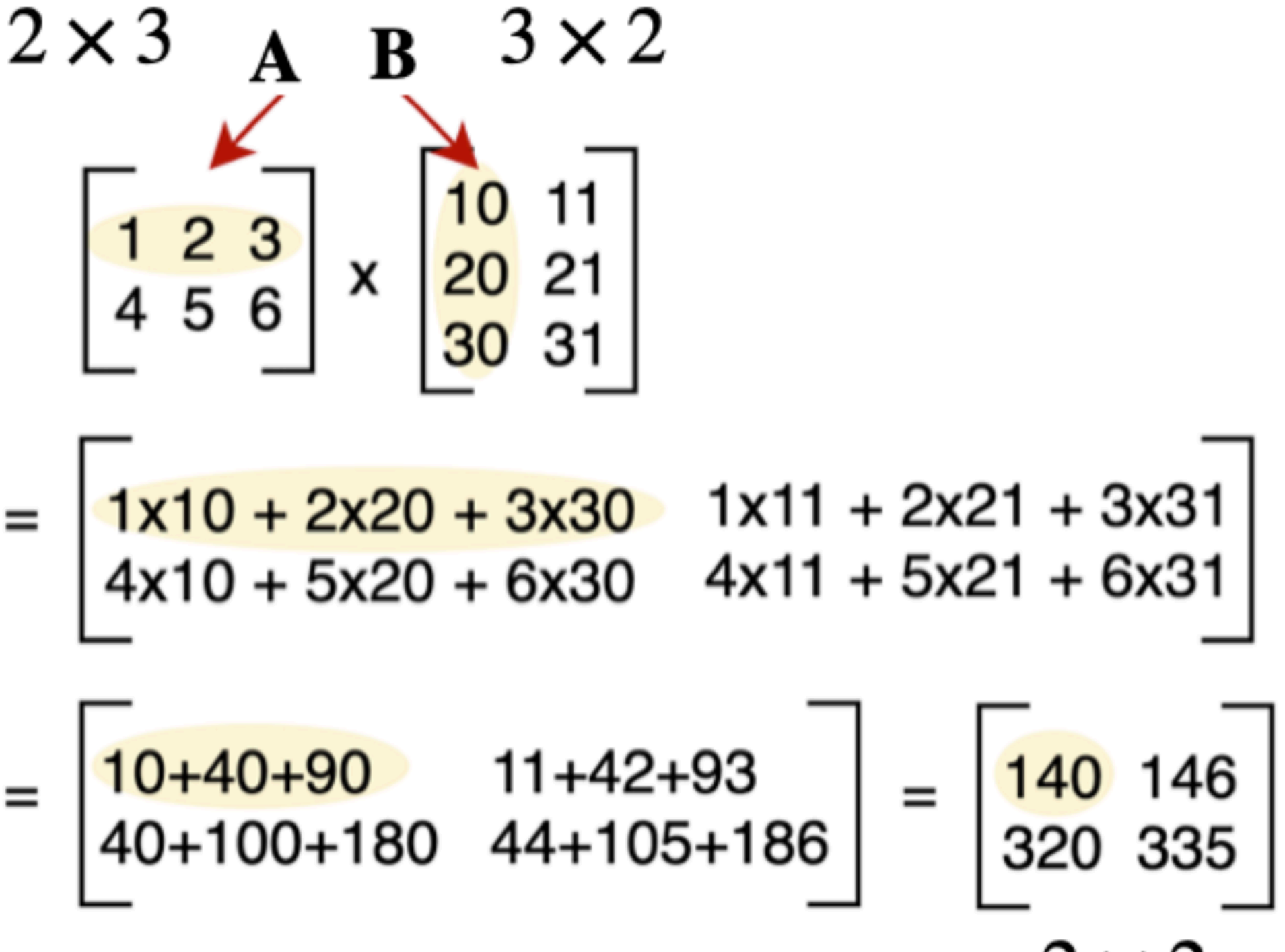
Cosine similarity

$$\text{similarity} = \cos(\theta) = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\|\|\mathbf{B}\|}$$

Key operation-2: Matrix multiplication

- Matrix multiplication is basically repeated dot products: The new entry for row-i column-j is A_row_i DOT B_column_j
- Two matrices can be multiplied if they obey the following rule.




2 × 3
A
B
3 × 2
1 2 3
4 5 6
x
10 11
20 21
30 31
= 1x10 + 2x20 + 3x30 1x11 + 2x21 + 3x31
4x10 + 5x20 + 6x30 4x11 + 5x21 + 6x31
= 10+40+90 11+42+93
40+100+180 44+105+186
= 140 146
320 335
2 × 2

Key operation-3: Matrix transpose

> Transpose: ⟶ rows become columns

The matrix transpose is a simple operation that swaps the rows and columns of a matrix.

Mathematically:

$$A^T_{i,j} = A_{j,i}$$

Simple example (2x3 matrix):

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

$$A^T = \begin{bmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{bmatrix}$$

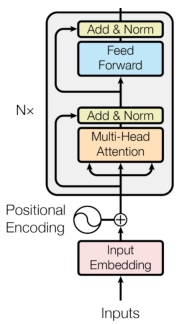
Components of a Transformer: Encoder

Let’s break down the transformer architecture step by step, starting with the encoder.

Note: Concepts from the encoder also apply to the decoder, but for pedagogical purposes we will first discuss the encoder, since model evaluation starts there.

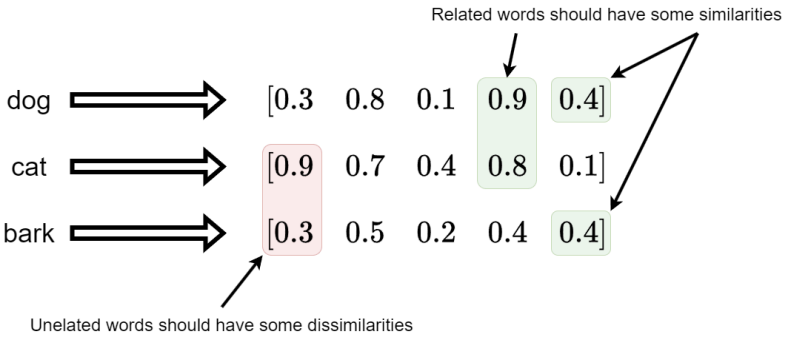
Encoder: Key components

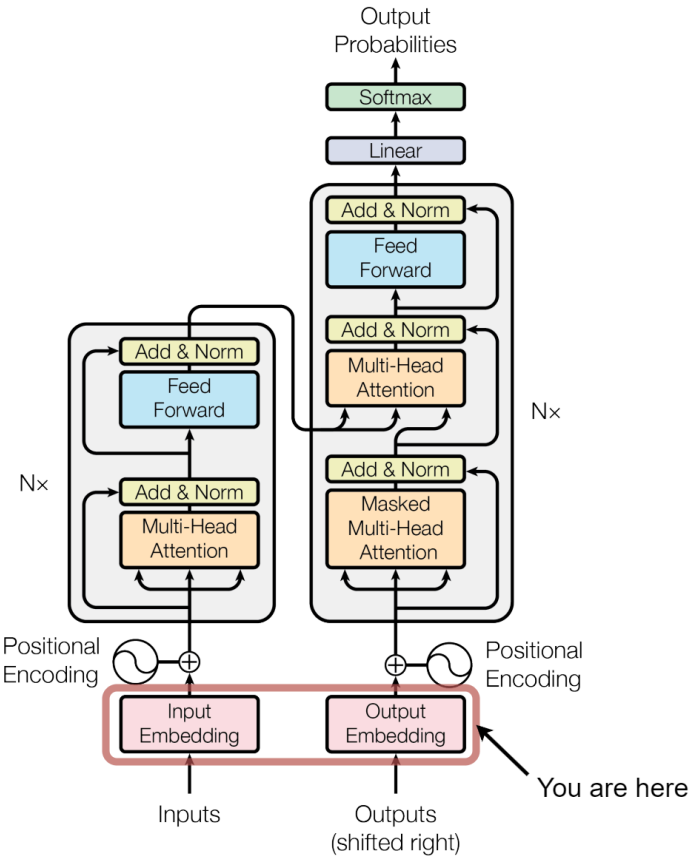
- **Input embeddings**: Encodes context of the input sequence
- **positional encoding**: Encodes the position of the token in the sequence
- **Attention input**: Embedding + positional encoding
- **Self-attention mechanism** (global token interaction)
- **Multi-head attention**: (repeat the self attention mechanism multiple times to add more context)
- **Add & LayerNorm** (residual connection)
- **Position-wise feedforward network (FFN)**
- **Add & LayerNorm** (second residual)



Recall: word embeddings

- Word embeddings are used to represent words in a continuous vector space
- Capture semantic and syntactic information
- `Learned Word Embeddings`
  - Utilize a learned weight matrix
  - Allow the model to adapt the embeddings to the specific task
- `Pre-trained Word Embeddings`
  - Popular pre-trained embeddings include Word2Vec, GloVe, and FastText embeddings
  - Provide a strong initialization for the embeddings, captures general linguistic patterns





Toy example: Data (Corpus & Vocabulary)

Corpus

```
"I like cats"
"I like good dogs"
```

- **Add special tokens**
  - <bos> (beginning of sequence)
  - <eos> (end of sequence)
  - <pad> (for fixed-length batching)
- **Context Window**: We define a **maximum sequence length**: $L = 6$ (All sequences must be length 6 → pad if shorter)

Vocabulary (V = 8)

token	index
<bos>	0
I	1
like	2

token	index
cats	3
good	4
dogs	5
<eos>	6
<pad>	7

Example Sentence tokenization

Example-A: `"I like cats"`

Tokenize and Pad:

$$[< bos >, I, like, cats, < eos >]$$

$$[0, 1, 2, 3, 6, 7]$$

Mask:

$$[1, 1, 1, 1, 1, 0]$$

Example-B: `"I like good dogs"`

Tokenize (No padding needed)

$$[< bos >, I, like, good, dogs, < eos >]$$

$$[0, 1, 2, 4, 5, 6]$$

Mask:

$$[1, 1, 1, 1, 1, 1]$$

Example-C: `"good dogs"`

Tokenize and pad

$$[< bos >, good, dogs, < eos >]$$

$$[0, 4, 5, 6, 7, 7]$$

Mask:

$$[1, 1, 1, 1, 0, 0]$$

Looking ahead: Attention Mask (introduced by padding)

Given a padded input like:
$[0, 4, 5, 6, 7, 7]$ (i.e., $[< bos >, good, dogs, < eos >, < pad >, < pad >]$)

the corresponding mask is: $[1, 1, 1, 1, 0, 0]$

> Padding positions need to be ignored in attention

- **What the mask does**
  - **1 → real token (keep)**
  - **0 → padding token (ignore)**

The mask is used during **self-attention**, specifically when computing: $\text{scores} = \frac{QK^T}{\sqrt{d_k}}$

(more on this later)

Before softmax, we modify scores: Columns corresponding to <pad> → set to $-\infty$

- After softmax: Attention weight on <pad> = **0** <pad> contributes **nothing** to any token's representation
- Without mask Model would attend to <pad> tokens
- Model attends **only to real tokens**
- Padding becomes **invisible to computation**

Toy example: Embedding Layer

- Returning to our simple example
- We Define a embedding dimension (model dimension): d
- This results in a learnable look-up table of size $V \times d$ (Parameters: $8 \times 2 = 16$)

$$E = \begin{bmatrix} 0.5 & 0.5 \\ 1.0 & 0.0 \\ 0.0 & 1.0 \\ 1.0 & 1.0 \\ 2.0 & 1.0 \\ 2.0 & 0.0 \\ -1.0 & 0.0 \\ 0.0 & 0.0 \end{bmatrix}$$

- Note that the BOS and EOS embeddings can be learn or fixed where as the PAD embedding is typically fixed.

Toy example: embedding lookup

Given a padded input like:

$$[0, 4, 5, 6, 7, 7] \quad (\text{i.e., } [< bos >, good, dogs, < eos >, < pad >, < pad >])$$

Pytorch looks up the embedding for each token in the input sequence and returns the corresponding embedding. $X = E[\text{indices}]$

$$[0, 4, 5, 6, 7, 7] \rightarrow X = \begin{bmatrix} 0.5 & 0.5 \\ 2.0 & 1.0 \\ 2.0 & 0.0 \\ -1.0 & 0.0 \\ 0.0 & 0.0 \\ 0.0 & 0.0 \end{bmatrix}$$

Notice that this is one sequence representation of size:

$(timesteps \times features) \rightarrow (L = 6) \times (d = 2) = 12$

If we had a mini-batch (e.g. B=2) it would be

$$(batch_size \times timesteps \times features) \rightarrow (B = 2) \times (L = 6) \times (d = 2) = 24$$

Aside: To do Backpropagation we need to compute the gradient of the loss with respect to the embedding matrix E. Backprop automatically updates **only the relevant rows**

$$\frac{\partial L}{\partial E_i} = \begin{cases} \text{nonzero} & \text{if token } i \text{ appears} \\ 0 & \text{otherwise} \end{cases}$$

Aside: Alternative look-up method

> This is less common, due to computational expense, but it is mathematically equivalent to the embedding lookup.

You can get the same effect by using a one-hot encoding and multiplying it by the embedding matrix.

For example, the embedding for the token “good” is:

One-hot encoding (sequence)

Each token becomes a one-hot vector in $\mathbb{R}^{V=8}$: (timesteps × features)

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

Call this matrix:

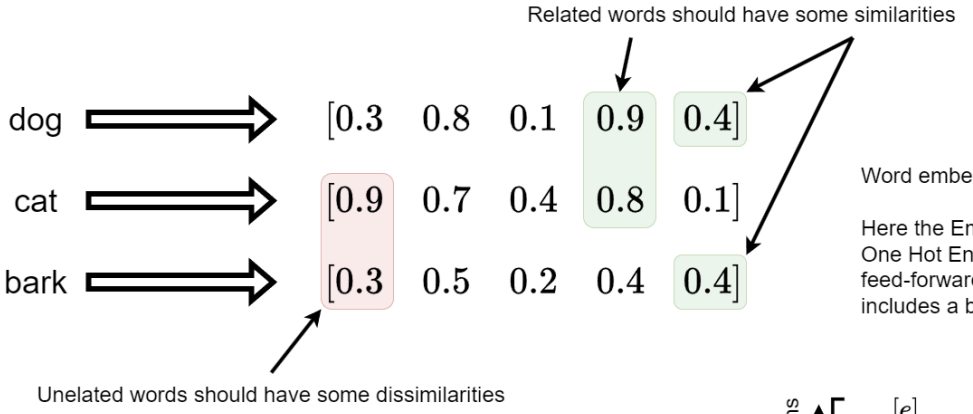
$$\text{OneHot} \in \mathbb{R}^{6\times 8}$$

$$E \in \mathbb{R}^{8\times 2}$$

Apply Matrix multiplication

$$X = \text{OneHot} \cdot E$$

More generally: Word Embeddings



Word embeddings can be created with a linear layer.

Here the Embeding matrix, $\mathbf{E}$ is created by passing the One Hot Encoded matrix, $\mathbf{O}$, through linear layer (dense feed-forward) with the parameter matrix, $\mathbf{W_E}$, which also includes a bias term.

n : Number of defined tokens in the vocabulary
s : The number of tokens in a sequence
m : The number embedding features
$w^{[e]}$: A parameter of the embedding weight matrix
e : An element of the token embedding matrix

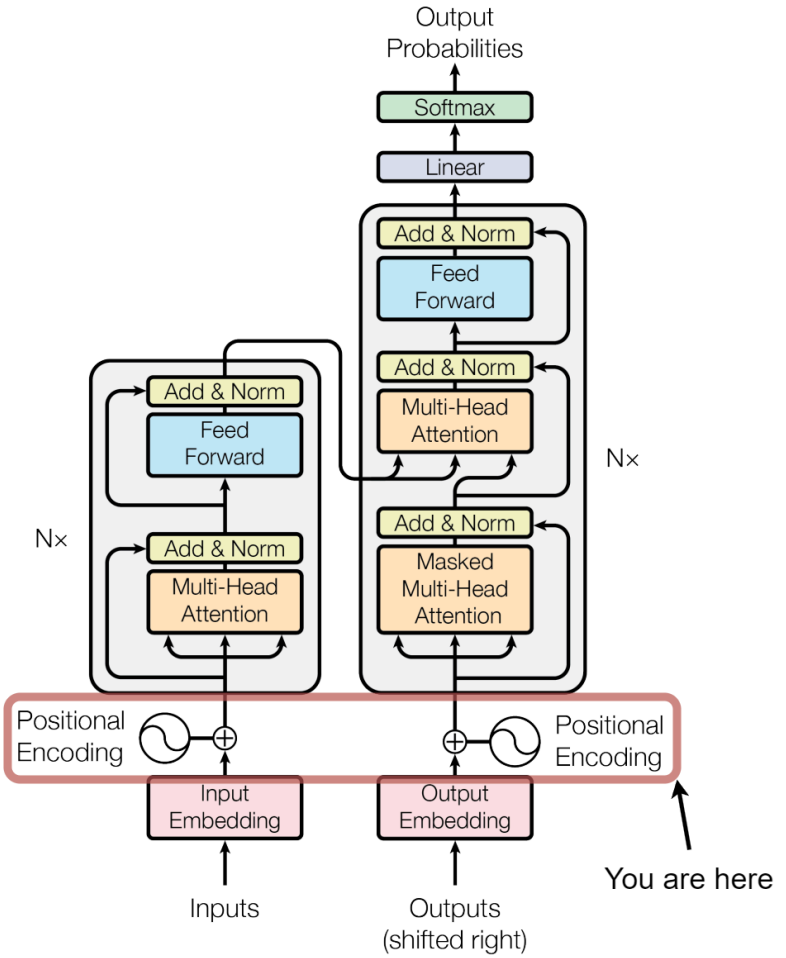
$\mathbf{O}$: The One-Hot-Encoded sequence matrix
$\mathbf{W_E}$: The token embedding weight matrix
$\mathbf{E}$: The token embedding matrix

$$\mathbf{E} = \mathbf{O}\mathbf{W_E} = \begin{bmatrix} 1 & 0 & 0 & 1 & \cdots & 0 \\ 1 & 0 & 0 & 1 & \cdots & 0 \\ 1 & 0 & 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & 1 & 0 & 0 & \cdots & 0 \end{bmatrix} \begin{bmatrix} w_{1,1}^{[e]} & w_{1,2}^{[e]} & w_{1,3}^{[e]} & \cdots & w_{1,m}^{[e]} \\ w_{2,1}^{[e]} & w_{2,2}^{[e]} & w_{2,3}^{[e]} & \cdots & w_{2,m}^{[e]} \\ w_{3,1}^{[e]} & w_{3,2}^{[e]} & w_{3,3}^{[e]} & \cdots & w_{3,m}^{[e]} \\ w_{4,1}^{[e]} & w_{4,2}^{[e]} & w_{4,3}^{[e]} & \cdots & w_{4,m}^{[e]} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ w_{n+1,1}^{[e]} & w_{n+1,2}^{[e]} & w_{n+1,3}^{[e]} & \cdots & w_{n+1,m}^{[e]} \end{bmatrix} = \begin{bmatrix} e_{1,1} & e_{1,2} & e_{1,3} & \cdots & e_{1,m} \\ e_{2,1} & e_{2,2} & e_{2,3} & \cdots & e_{2,m} \\ e_{3,1} & e_{3,2} & e_{3,3} & \cdots & e_{3,m} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ e_{s,1} & e_{s,2} & e_{s,3} & \cdots & e_{s,m} \end{bmatrix}$$

Sequence Dimension s tokens
Vocabulary Dimension $n+1$ defined tokens
Vocabulary Dimension $n+1$ defined tokens
Embedding Dimension m features
Sequence Dimension s tokens
Embedding Dimension m features

Positional Encoding: Overview

- The attention mechanism does not naturally encode position like a RNN
- Positional encoding provides information about the order of tokens in an input sequence
- The two main types of positional encoding are `Sinusoidal Positional Encoding` and `Learned Positional Embeddings`
- Sinusoidal positional encoding is deterministic and generalizes well to sequence lengths not seen during training
- Learned positional embeddings can adapt to specific tasks and data distributions



Toy example: Positional Encoding

> Transformers do **NOT** use recurrence (sequential evaluation), therefore we need a way to encode the position of the tokens in the sequence before further processing, this is done with special `positional encoding` vectors.

The positional encodings can be either fixed OR learned from data.

In the original paper they were fixed and defined via sinusoidal functions of varying frequencies.

The dimensions of the positional encodings are the same as the embedding dimensions d with $L = 6$ possible positions in the sequence. $P \in \mathbb{R}^{L \times d}$

For our simple example, assume that we have the following fixed positional encodings:

$$P = \begin{bmatrix} 0.0 & 0.0 \\ 0.2 & 0.2 \\ 0.4 & 0.4 \\ 0.6 & 0.6 \\ 0.8 & 0.8 \\ 1.0 & 1.0 \end{bmatrix}$$

Sinusoidal Positional Encoding formula

$$\text{even columns} \rightarrow P(k, 2i) = \sin\left(\frac{k}{n^{2i/d}}\right)$$

$$\text{odd columns} \rightarrow P(k, 2i+1) = \cos\left(\frac{k}{n^{2i/d}}\right)$$

Matrix shape:

$$P \in \mathbb{R}^{L \times d}$$

- Rows (L) → sequence positions
- Columns (d) → embedding dimensions

k: Position of an object in the input sequence, $0 \leq k < L$

d: Dimension of the output embedding space

`Important`: In 2017 the authors of the paper set $n = 10000$ and $d = 512$, in our simple example we use $d = 2$

$P(k, j)$: Position function mapping position k to embedding index j

n: User-defined scalar (typically $10{,}000$) controlling wavelength

i: Column pair index, $0 \leq i < \frac{d}{2}$, each i maps to one sine and one cosine dimension

Positional Encoding Demo

In the following code we will use the sinusoidal positional encoding formula to generate the positional encodings for our example.

▶ Code

▶ Code

```
array([[ 0.        ,  1.        ,  0.        ,  1.        ],
       [ 0.84147098,  0.54030231,  0.09983342,  0.99500417],
       [ 0.90929743, -0.41614684,  0.19866933,  0.98006658],
       [ 0.14112001, -0.9899925 ,  0.29552021,  0.95533649]])
```

Exploring the wavelength parameter

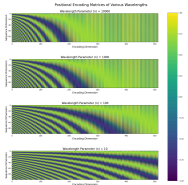
For select wavelengths we can visualize the positional encoding matrix.

The following codeenerates and visualizes sinusoidal positional encoding matrices for different wavelength scales.

- Computes ($P \in \mathbb{R}^{100\times512}$) for each ($n \in 10, 100, 1000, 10000$)
- Plots each matrix as a heatmap (rows = positions, columns = embedding dims)
- Larger (n) → longer wavelengths → smoother variation across positions
- Smaller (n) → shorter wavelengths → faster oscillations

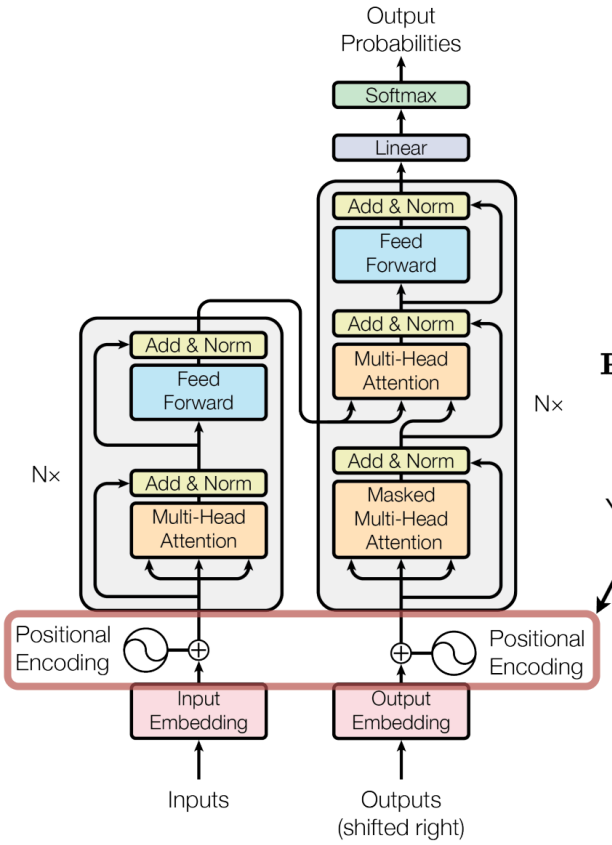
Purpose: show how the wavelength parameter controls frequency structure of positional encodings across dimensions.

▶ Code



Alternative method: Learned Positional Encodings

Many modern transformer models use learned positional encodings rather than sinusoidal ones.



I_s : The identity matrix of size s
$\mathbf{W_P}$: The positional encoding weight matrix
$\mathbf{P}$: The positional encdoing matrix
s : The number of tokens in a sequence
m : The number embedding features
$w^{[p]}$: A parameter of the embedding weight matrix
p : An element of the postional encoding matrix

Here the Identity matrix is a formality used to represent token positions

$$\mathbf{P} = I_s \mathbf{W_P} = \begin{bmatrix} 1 & 0 & 0 & \cdots & 0 \\ 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 1 \end{bmatrix} \begin{bmatrix} w^{[p]}_{1,1} & w^{[p]}_{1,2} & w^{[p]}_{1,3} & \cdots & w^{[p]}_{1,m} \\ w^{[p]}_{2,1} & w^{[p]}_{2,2} & w^{[p]}_{2,3} & \cdots & w^{[p]}_{2,m} \\ w^{[p]}_{3,1} & w^{[p]}_{3,2} & w^{[p]}_{3,3} & \cdots & w^{[p]}_{3,m} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ w^{[p]}_{s,1} & w^{[p]}_{s,2} & w^{[p]}_{s,3} & \cdots & w^{[p]}_{s,m} \end{bmatrix} = \begin{bmatrix} p_{1,1} & p_{1,2} & p_{1,3} & \cdots & p_{1,m} \\ p_{2,1} & p_{2,2} & p_{2,3} & \cdots & p_{2,m} \\ p_{3,1} & p_{3,2} & p_{3,3} & \cdots & p_{3,m} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ p_{s,1} & p_{s,2} & p_{s,3} & \cdots & p_{s,m} \end{bmatrix}$$

Sequence Dimension s — Sequence Dimension s — Embedding Dimension m features — Sequence Dimension s — Embedding Dimension m features

$$\mathbf{P} = \mathbf{W_P} = \begin{bmatrix} w^{[p]}_{1,1} & w^{[p]}_{1,2} & w^{[p]}_{1,3} & \cdots & w^{[p]}_{1,m} \\ w^{[p]}_{2,1} & w^{[p]}_{2,2} & w^{[p]}_{2,3} & \cdots & w^{[p]}_{2,m} \\ w^{[p]}_{3,1} & w^{[p]}_{3,2} & w^{[p]}_{3,3} & \cdots & w^{[p]}_{3,m} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ w^{[p]}_{s,1} & w^{[p]}_{s,2} & w^{[p]}_{s,3} & \cdots & w^{[p]}_{s,m} \end{bmatrix} = \begin{bmatrix} p_{1,1} & p_{1,2} & p_{1,3} & \cdots & p_{1,m} \\ p_{2,1} & p_{2,2} & p_{2,3} & \cdots & p_{2,m} \\ p_{3,1} & p_{3,2} & p_{3,3} & \cdots & p_{3,m} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ p_{s,1} & p_{s,2} & p_{s,3} & \cdots & p_{s,m} \end{bmatrix}$$

Sequence Dimension s — Embedding Dimension m features — Sequence Dimension s — Embedding Dimension m features

Adding embedding and positional encoding

The combined input to the attention head is the sum of the word embedding and the positional encoding.

They were intentionally designed with the same dimension so that they can be summed together.

$$X = E + P$$

This encodes both semantic context and sequence location (positional information) into a single vector.

Toy example: Adding embedding and positional encoding

`<bos> I like cats <eos> <pad>`

Token indices:

$$[0,\ 1,\ 2,\ 3,\ 6,\ 7]$$

Embedding Lookup

Using E:

$$H^{(\text{embed})} = \begin{bmatrix} 0.5 & 0.5 \\ 1.0 & 0.0 \\ 0.0 & 1.0 \\ 1.0 & 1.0 \\ -1.0 & 0.0 \\ 0.0 & 0.0 \end{bmatrix}$$

Add Positional Encoding

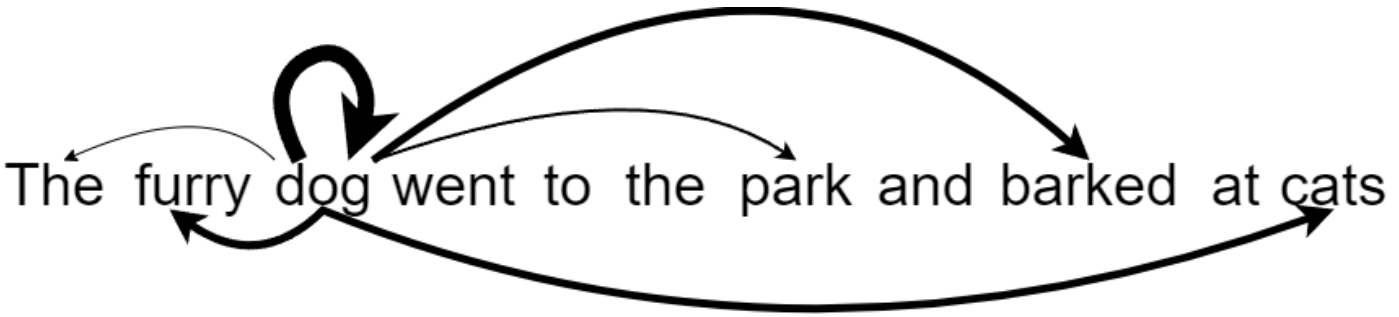
$$P = \begin{bmatrix} 0.0 & 0.0 \\ 0.2 & 0.2 \\ 0.4 & 0.4 \\ 0.6 & 0.6 \\ 0.8 & 0.8 \\ 1.0 & 1.0 \end{bmatrix}$$

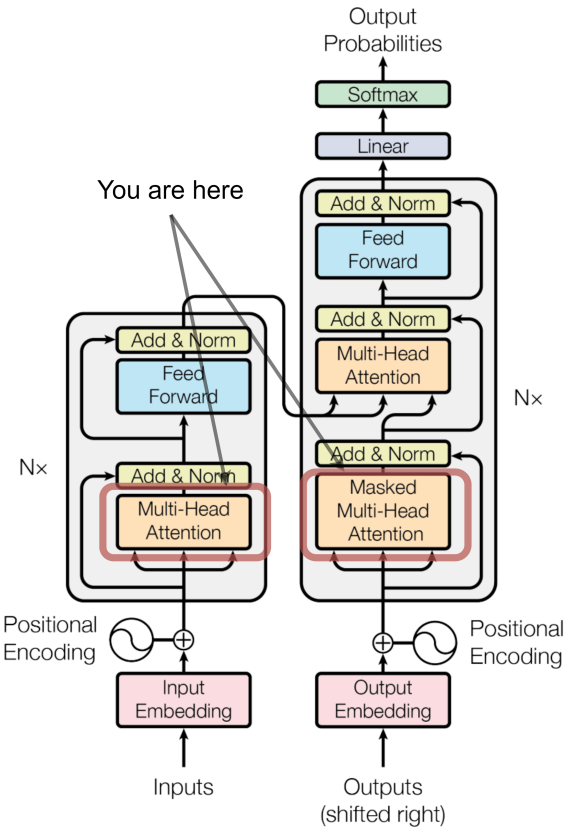
here H stands for “hidden representations” (or “hidden states”)

Concept of Self-Attention

- The fundamental idea behind attention is what we would refer to as context
- When we read a sentence we relate certain words to others
- For instance if I say “Wow, that rocket ship is moving really fast!” we all know that fast is referring to the speed of the rocket

The word dog has a strong contextual tie to “bark”, “cats” and “furry”, with lower ties to words like “The” and “park”





Single-Head vs Multi-Head Attention

- The core building **block** of a transformer is the **self-attention** mechanism
- A **single attention head** is one instance of the self-attention mechanism
- **Multi-head attention** = multiple self-attention heads in parallel
  - Each head learns different representations (different context)
- Outputs are concatenated/combined and then remapped to the model dimension via a linear transformation
- If you understand one head then you are “good to go” since `multi-head = parallel repetition + aggregation`

Lets start with the calculation for a single attention head.

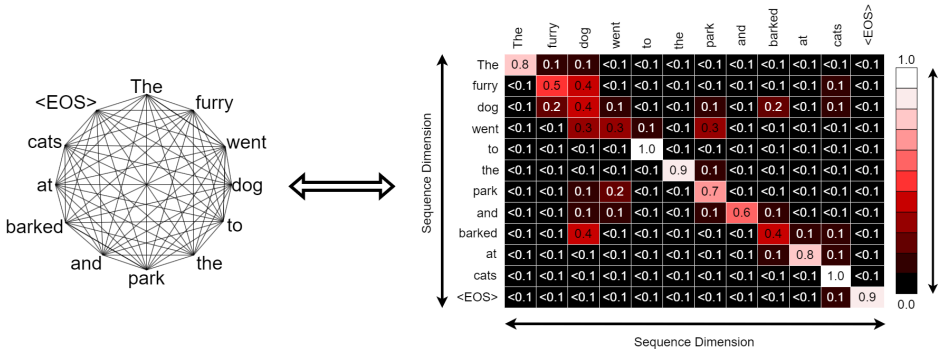
Self-Attention vs Cross-Attention

- `Self-attention`: The sequence attends to itself `(used in the encoder)`
  - This is analoguos to a an Auto Correlation function which measures lagged correlations between a signal and itself
- `Cross-attention`: The sequence attends to another sequence `(used in the decoder)`
  - This is analoguos to a Cross-Correlation function which measures lagged correlations between two different signals

The self-attention weight matrix

> Here we start with the end in mind:

- A key result of the attention mechanism is the attention weight matrix
  - **Note**: This object is the **result** of several calculations, **the values are NOT directly learned parameters**
  - We will look more closely at the calculation shortly
- The attention matrix is a sequence by sequence matrix $L \times L$
- Each value represents how much attention each token should place on every other token in the sequence



Self-Attention calculation-1: Q,K,V matrices

- Input embeddings: $H^{(0)} \in \mathbb{R}^{L \times d}$
- **Core operations**: Linear projections produce Q, K, V matrices

$$Q = H^{(0)} W_Q, \quad K = H^{(0)} W_K, \quad V = H^{(0)} W_V$$

- these respectively have shape $(L \times d_k), (L \times d_k), (L \times d_v)$
- These get special names: Q: queries, K: keys, V: values (more on this later)
- For now, just think of them as various linearly `transformed representations` of the input embeddings

Learnable Parameters

- Projection matrices:

$$W_Q, W_K \in \mathbb{R}^{d \times d_k}, \quad W_V \in \mathbb{R}^{d \times d_v}$$

- Output projection:

$$W_O \in \mathbb{R}^{d_v \times d}$$

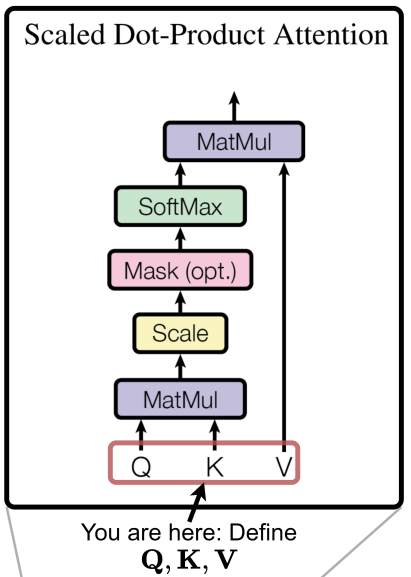
- $d_k = d_v$ derived from model dimension d and number of heads h:

$$d_k = d_v = \frac{d}{h}$$

Encoder Self-Attention: Q,K,V Matrices

- Again for now, just think of Q,K,V as various linearly `transformed representations` of the input embeddings

$$\mathbf{A}(\mathbf{Q},\mathbf{V},\mathbf{K}) = S\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_k}}\right)\mathbf{V}$$



$$\mathbf{Q} = \mathbf{X}\mathbf{W}_\mathbf{Q} = \begin{bmatrix} x_{1,1} & x_{1,2} & \cdots & x_{1,m} \\ x_{2,1} & x_{2,2} & \cdots & x_{2,m} \\ \vdots & \vdots & \ddots & \vdots \\ x_{s,1} & x_{s,2} & \cdots & x_{s,m} \end{bmatrix} \begin{bmatrix} w^{[q]}_{1,1} & w^{[q]}_{1,2} & \cdots & w^{[q]}_{1,e} \\ w^{[q]}_{2,1} & w^{[q]}_{2,2} & \cdots & w^{[q]}_{2,e} \\ \vdots & \vdots & \ddots & \vdots \\ w^{[q]}_{m,1} & w^{[q]}_{m,2} & \cdots & w^{[q]}_{m,e} \end{bmatrix} = \begin{bmatrix} q_{1,1} & q_{1,2} & \cdots & q_{1,e} \\ q_{2,1} & q_{2,2} & \cdots & q_{2,e} \\ \vdots & \vdots & \ddots & \vdots \\ q_{s,1} & q_{s,2} & \cdots & q_{s,e} \end{bmatrix}$$

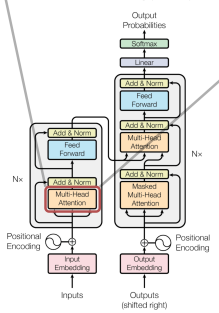
Sequence dimension s · Model embedding dimension m · Model dimension m · Attention embedding dimension e · Sequence dimension s · Attention embedding dimension e

$$\mathbf{K} = \mathbf{X}\mathbf{W}_\mathbf{K} = \begin{bmatrix} x_{1,1} & x_{1,2} & \cdots & x_{1,m} \\ x_{2,1} & x_{2,2} & \cdots & x_{2,m} \\ \vdots & \vdots & \ddots & \vdots \\ x_{s,1} & x_{s,2} & \cdots & x_{s,m} \end{bmatrix} \begin{bmatrix} w^{[k]}_{1,1} & w^{[k]}_{1,2} & \cdots & w^{[k]}_{1,e} \\ w^{[k]}_{2,1} & w^{[k]}_{2,2} & \cdots & w^{[k]}_{2,e} \\ \vdots & \vdots & \ddots & \vdots \\ w^{[k]}_{m,1} & w^{[k]}_{m,2} & \cdots & w^{[k]}_{m,e} \end{bmatrix} = \begin{bmatrix} k_{1,1} & k_{1,2} & \cdots & k_{1,e} \\ k_{2,1} & k_{2,2} & \cdots & k_{2,e} \\ \vdots & \vdots & \ddots & \vdots \\ k_{s,1} & k_{s,2} & \cdots & k_{s,e} \end{bmatrix}$$

Sequence dimension s · Model embedding dimension m · Model dimension m · Attention embedding dimension e · Sequence dimension s · Attention embedding dimension e

$$\mathbf{V} = \mathbf{X}\mathbf{W}_\mathbf{V} = \begin{bmatrix} x_{1,1} & x_{1,2} & \cdots & x_{1,m} \\ x_{2,1} & x_{2,2} & \cdots & x_{2,m} \\ \vdots & \vdots & \ddots & \vdots \\ x_{s,1} & x_{s,2} & \cdots & x_{s,m} \end{bmatrix} \begin{bmatrix} w^{[v]}_{1,1} & w^{[v]}_{1,2} & \cdots & w^{[v]}_{1,e} \\ w^{[v]}_{2,1} & w^{[v]}_{2,2} & \cdots & w^{[v]}_{2,e} \\ \vdots & \vdots & \ddots & \vdots \\ w^{[v]}_{m,1} & w^{[v]}_{m,2} & \cdots & w^{[v]}_{m,e} \end{bmatrix} = \begin{bmatrix} v_{1,1} & v_{1,2} & \cdots & v_{1,e} \\ v_{2,1} & v_{2,2} & \cdots & v_{2,e} \\ \vdots & \vdots & \ddots & \vdots \\ v_{s,1} & v_{s,2} & \cdots & v_{s,e} \end{bmatrix}$$

Sequence dimension s · Model embedding dimension m · Model dimension m · Attention embedding dimension e · Sequence dimension s · Attention embedding dimension e



$\mathbf{X}$: The attention input
$\mathbf{W}_\mathbf{Q}, \mathbf{W}_\mathbf{K}, \mathbf{W}_\mathbf{V}$: The querry, key and value linear projection weight matrices
$\mathbf{Q}, \mathbf{K}, \mathbf{V}$: The querry, key and value matrices
$\mathbf{A}$: The scaled dot-product attention
x : An element of the self-attention input
q, k, v : An element of the querry, key or value matrix
$w^{[q]}, w^{[k]}, w^{[v]}$: A weight from the Query, Key or Value liner projection matrices
m : The model or input embedding dimension, number of features
e : The attention dimension, number of features in $\mathbf{Q}$, $\mathbf{K}$ and $\mathbf{V}$
s : Sequence Dimension, number of tokens in the sequence
d_k : The key dimension (in this example $d_k = e$)
S : Softmax activation function

Self-Attention calculation-2: Scaled Dot-Product and attention weight matrix

Raw similarity scores: First the **Dot product (similarity)** is used to compute Q and K similarity scores

$$QK^\top$$

Note that this results in a matrix of shape $L \times L$ (max_seq_length x max_seq_length)

Scaling

$$\frac{QK^\top}{\sqrt{d_k}}$$

This is done to prevent the dot product from becoming too large and causing numerical instability.

Masking (optional depending on use case)

- Add $-\infty$ (or large negative value) where attention is disallowed

$$S = \frac{QK^\top}{\sqrt{d_k}} \quad \longrightarrow \quad S_{\text{masked}} = S + M$$

$$M \in \mathbb{R}^{L \times L}$$

- Valid positions: 0
- Masked positions: $-\infty$ (or a very large negative value) $\longrightarrow$ softmax of $-\infty$ is 0

Softmax → Attention Weight matrix

$$W = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right)$$

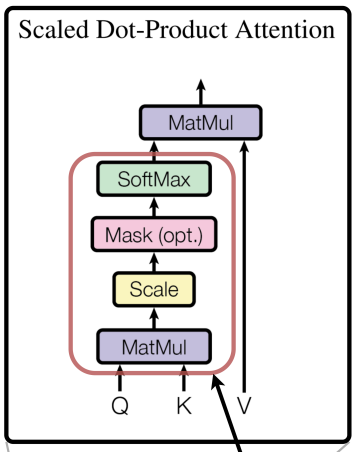
- $W \in \mathbb{R}^{L \times L}$
- Each row = distribution over tokens
- Rows sum to 1

Encoder Self-Attention Step 2

$$\mathbf{A}(\mathbf{Q},\mathbf{V},\mathbf{K}) = S\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_k}} + \mathbf{M_P}\right)\mathbf{V}$$

$\mathbf{Q},\mathbf{K},\mathbf{V}$: The querry, key and value matrices
$\mathbf{W_A}$: The attention weight matrix$x$: An element of the self-attention input
$\mathbf{M_P}$: Padding Mask
$\mathbf{A}$: The scaled dot-product attention
x : An element of the self-attention input
q, k, v : An element of the querry, key or value matrix
$w^{[a]}$: An element of the attention weight matrix
m : The model or input embedding dimension, number of features
e : The attention dimension, number of features in $\mathbf{Q}$, $\mathbf{K}$ and $\mathbf{V}$
s : Sequence Dimension, number of tokens in the sequence
d_k : The key dimension (in this example $d_k = e$)
S : Softmax activation function

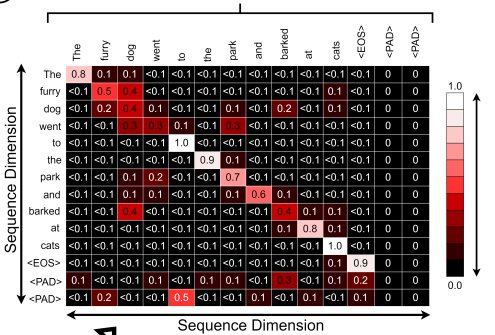
$\vec{z}$: Input vector
e^{z_i} : Stanard exponential function for input vector
e^{z_j} : Stanard exponential function for output vector
K : Length of the input vector (number of tokens in sequence)



$$\mathbf{W_A} = S\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_k}} + \mathbf{M}\right) = S\left(\frac{1}{\sqrt{d_k}}\begin{bmatrix} q_{1,1} & q_{1,2} & \cdots & q_{1,e} \\ q_{2,1} & q_{2,2} & \cdots & q_{2,e} \\ q_{3,1} & q_{3,2} & \cdots & q_{3,e} \\ \vdots & \vdots & \ddots & \vdots \\ q_{s,1} & q_{s,2} & \cdots & q_{s,e} \end{bmatrix}_{\text{Sequence dimension } s,\ \text{Attention embedding dimension } e} \begin{bmatrix} k_{1,1} & k_{2,1} & k_{3,1} & \cdots & k_{s,1} \\ k_{1,2} & k_{2,2} & k_{3,2} & \cdots & k_{s,2} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ k_{1,e} & k_{2,e} & k_{3,e} & \cdots & k_{s,e} \end{bmatrix}_{\text{Attention embedding dimension } e,\ \text{Sequence dimension } s} + \mathbf{M}\right) = \begin{bmatrix} w^{[a]}_{1,1} & w^{[a]}_{1,2} & w^{[a]}_{1,3} & \cdots & w^{[a]}_{1,s} \\ w^{[a]}_{2,1} & w^{[a]}_{2,2} & w^{[a]}_{2,3} & \cdots & w^{[a]}_{2,s} \\ w^{[a]}_{3,1} & w^{[a]}_{3,2} & w^{[a]}_{3,3} & \cdots & w^{[a]}_{3,s} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ w^{[a]}_{s,1} & w^{[a]}_{s,2} & w^{[a]}_{s,3} & \cdots & w^{[a]}_{s,s} \end{bmatrix}_{\text{Sequence dimension } s,\ \text{Sequence dimension } s}$$

Note that these values are completely fictional

Attention Weights $\mathbf{W_A}$



$$\mathbf{W_A} = S\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_k}} + \mathbf{M_P}\right)$$

Note that these values are completely fictional

We don't want these values to contribute to attention

$\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_k}}$

	The	furry	dog	went	to	the	park	and	barked	at	cats	<EOS>	<PAD>	<PAD>
The	5.3	0.11	-0.1	-0.4	0.1	0.0	-1.1	0.01	0.0	-0.3	0.1	0.0	-0.1	-0.2
furry	0.1	2.2	1.9	0.0	-0.2	0.0	-0.3	0.0	0.15	0.0	0.19	0.0	0.0	0.0
dog	-0.1	1.1	3.2	0.0	0.0	-0.2	0.2	-1.1	0.2	-0.3	0.5	0.0	-0.2	-0.1
went	0.1	0.2	0.6	1.1	0.5	0.0	0.4	0.1	0.0	0.0	0.0	-0.2	-0.6	0.0
to	0.0	0.0	0.0	-0.1	4.5	0.1	0.0	0.0	0.0	0.1	0.0	0.0	0.0	0.7
the	-0.3	0.0	0.0	0.0	0.4	3.5	0.6	0.0	0.0	-0.3	0.0	0.0	-0.1	0.0
park	0.0	0.0	0.2	0.1	0.0	0.2	4.4	0.0	0.2	0.0	0.1	0.0	-0.1	0.0
and	-0.4	0.0	0.1	0.3	0.0	0.0	0.3	5.5	0.3	0.0	0.0	-0.4	0.0	0.0
barked	-0.2	0.1	1.2	0.0	0.0	0.0	0.0	0.0	2.3	0.0	0.0	0.1	-0.8	0.2
at	-0.6	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.4	2.2	0.5	0.1	1.1	0.0
cats	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	10e9	0.0	0.0	0.0
<EOS>	0.2	-0.2	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.5	1.5	2.2	0.0	1.2
<PAD>	0.5	0.1	0.0	1.5	0.5	-1.1	0.0	-0.2	0.6	-0.3	0.2	1.2	2.1	1.1
<PAD>	0.6	0.0	0.2	0.1	1.1	0.8	-0.4	0.0	-0.9	0.0	0.0	0.7	0.5	3.5

Sequence Dimension

Sequence Dimension

+

Padding Mask $\mathbf{M_P}$

	The	furry	dog	went	to	the	park	and	barked	at	cats	<EOS>	<PAD>	<PAD>
The	0	0	0	0	0	0	0	0	0	0	0	0	$-\infty$	$-\infty$
furry	0	0	0	0	0	0	0	0	0	0	0	0	$-\infty$	$-\infty$
dog	0	0	0	0	0	0	0	0	0	0	0	0	$-\infty$	$-\infty$
went	0	0	0	0	0	0	0	0	0	0	0	0	$-\infty$	$-\infty$
to	0	0	0	0	0	0	0	0	0	0	0	0	$-\infty$	$-\infty$
the	0	0	0	0	0	0	0	0	0	0	0	0	$-\infty$	$-\infty$
park	0	0	0	0	0	0	0	0	0	0	0	0	$-\infty$	$-\infty$
and	0	0	0	0	0	0	0	0	0	0	0	0	$-\infty$	$-\infty$
barked	0	0	0	0	0	0	0	0	0	0	0	0	$-\infty$	$-\infty$
at	0	0	0	0	0	0	0	0	0	0	0	0	$-\infty$	$-\infty$
cats	0	0	0	0	0	0	0	0	0	0	0	0	$-\infty$	$-\infty$
<EOS>	0	0	0	0	0	0	0	0	0	0	0	0	$-\infty$	$-\infty$
<PAD>	0	0	0	0	0	0	0	0	0	0	0	0	$-\infty$	$-\infty$
<PAD>	0	0	0	0	0	0	0	0	0	0	0	0	$-\infty$	$-\infty$

Sequence Dimension

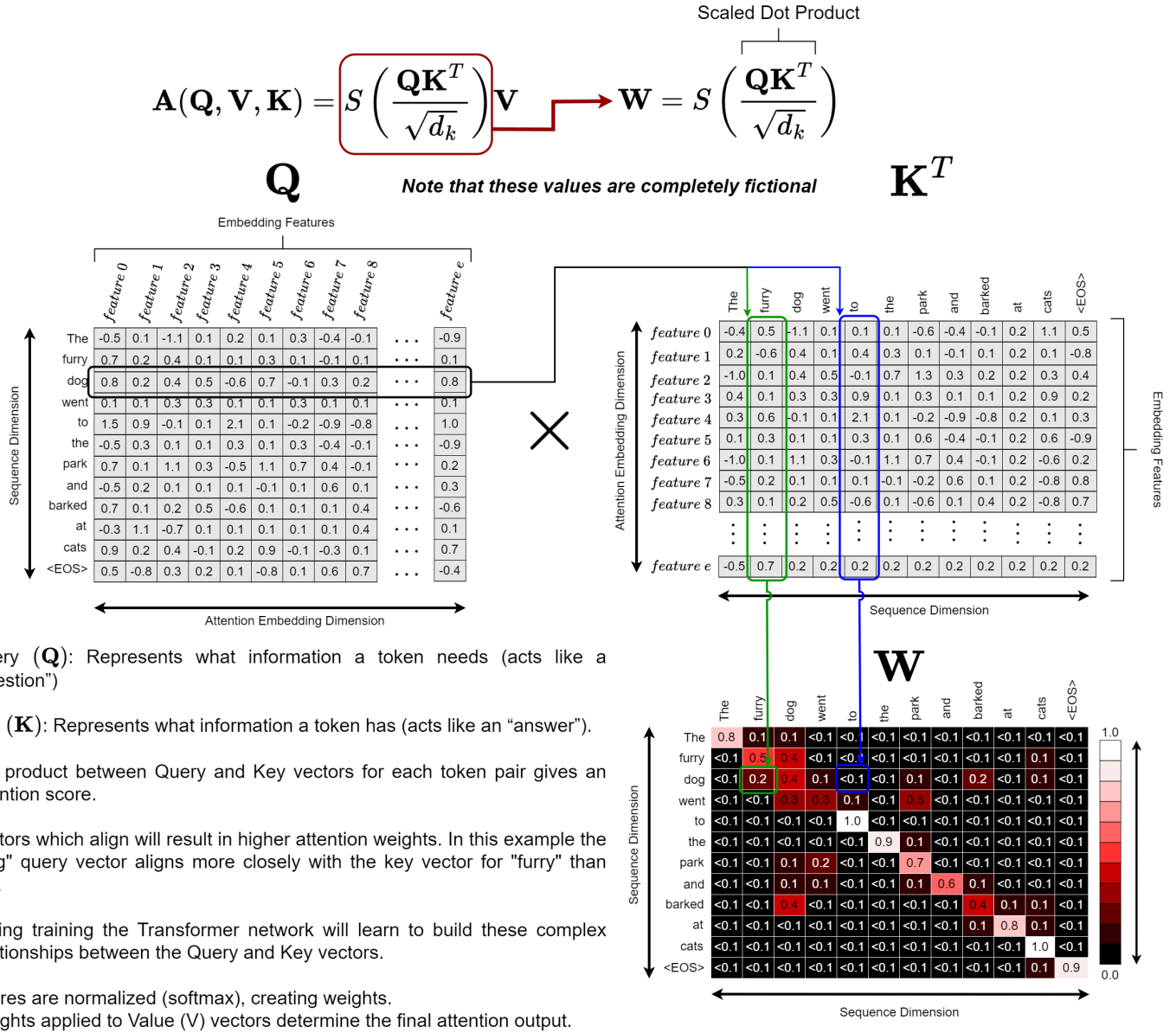
Sequence Dimension

$$S(\vec{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

$$e^0 = 1$$

$$e^{-\infty} \approx 0$$

Scaled Dot Product Intuition



Query $(\mathbf{Q})$: Represents what information a token needs (acts like a "question")

Key $(\mathbf{K})$: Represents what information a token has (acts like an "answer").

Dot product between Query and Key vectors for each token pair gives an attention score.

Vectors which align will result in higher attention weights. In this example the "dog" query vector aligns more closely with the key vector for "furry" than "to".

During training the Transformer network will learn to build these complex relationships between the Query and Key vectors.

Scores are normalized (softmax), creating weights.
Weights applied to Value (V) vectors determine the final attention output.

Self-Attention Calculation-Step 3: Attention

We now multiply the attention weights by the value matrix V to obtain the output of a **single attention head**.

$$\mathbf{A}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = S\left(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d_k}}\right)\mathbf{V}$$

- $S(\cdot)$: row-wise softmax
- Output shape: $\mathbb{R}^{L \times d_v}$

Matrix Form

$$\mathbf{W} = S\left(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d_k}}\right)$$

$$H^{(\text{attn})} = \mathbf{W}\mathbf{V}$$

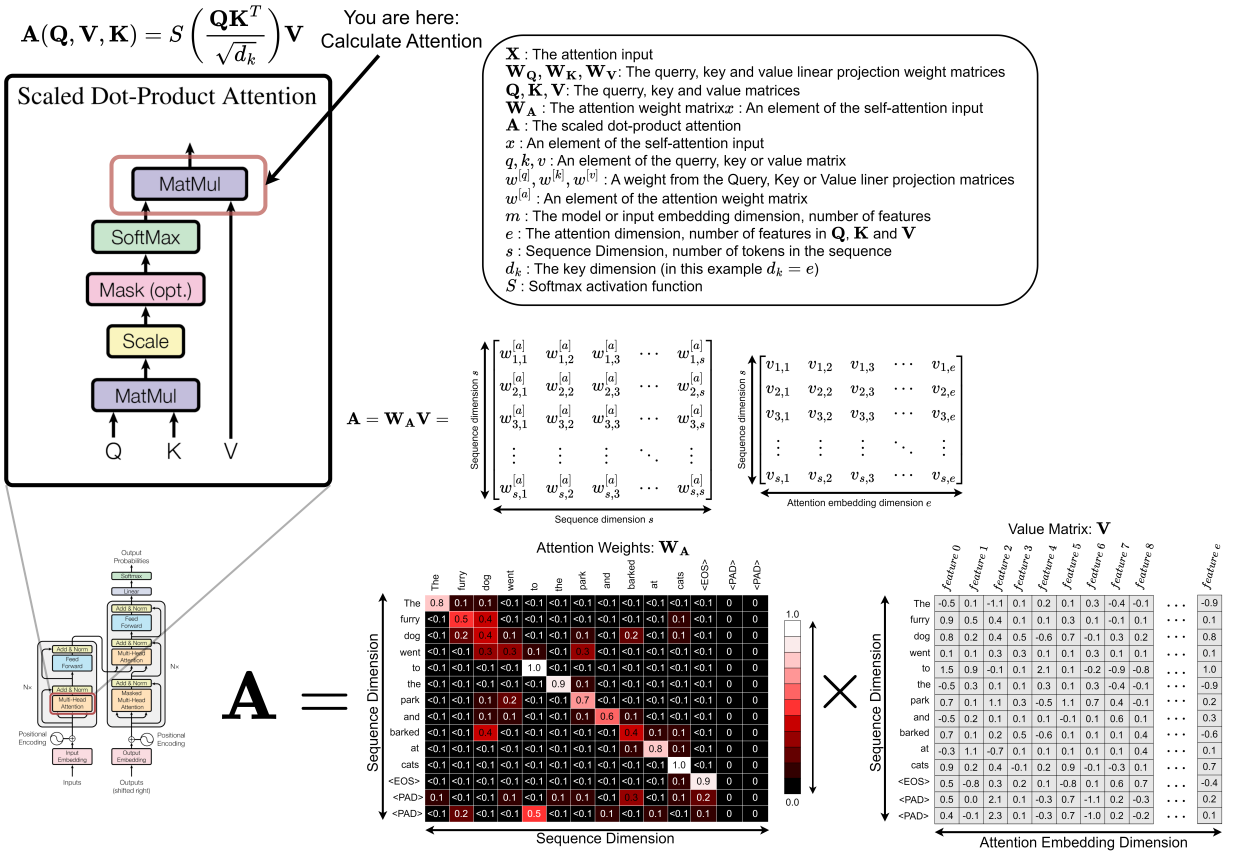
- $\mathbf{W} \in \mathbb{R}^{L \times L}$: attention weights
- $\mathbf{V} \in \mathbb{R}^{L \times d_v}$
- $H^{(\text{attn})} \in \mathbb{R}^{L \times d_v}$

Interpretation

- Each row of $\mathbf{W}$ defines how a token attends to all tokens
- Each output token is a **weighted combination of value vectors**
- This produces a **context-aware representation** of each token

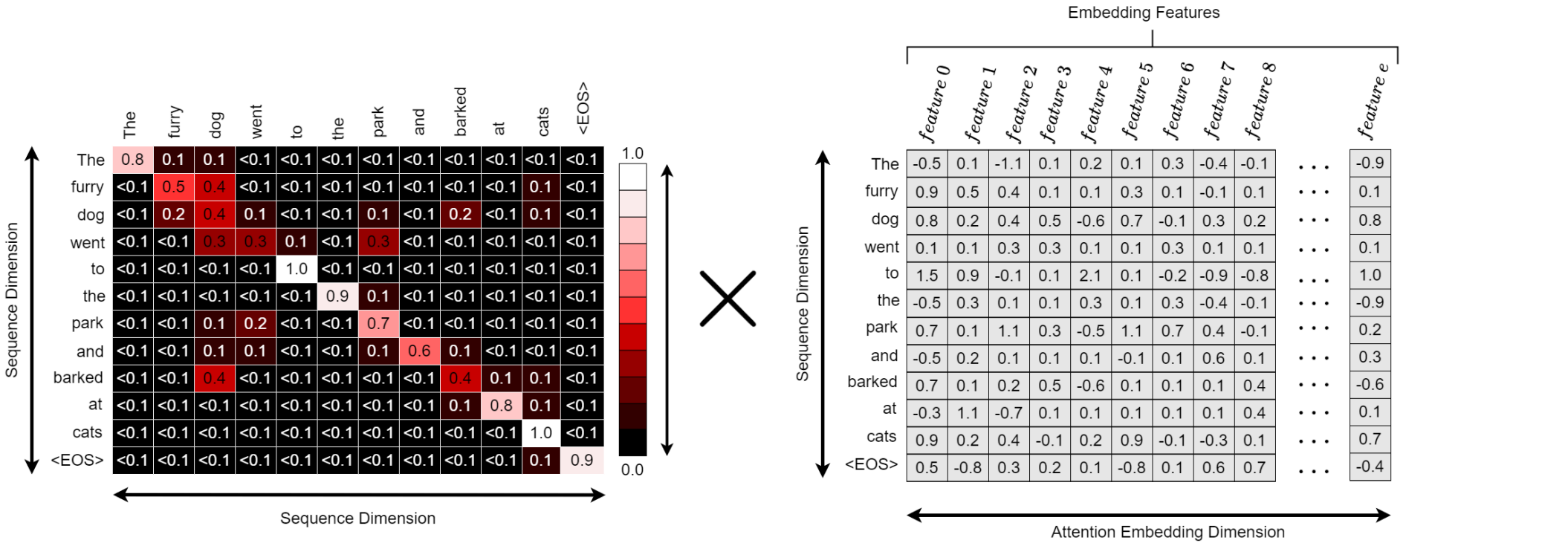
Encoder Self-Attention Step 3

> Multiplying to form a **weighted combination of value vectors**, allowing each token to aggregate the most relevant information from other tokens into its contextual representation.



Self Attention Concept

- Mathematically speaking, the whole purpose of attention is to “weight” token embeddings/features by attention weights
- Allows contextually relevant tokens to influence the embedding feature values
- Below we see that the token “barked” allows the token “dog” to “strongly” influence its embedding values



Final step: Output projection

This is handled differently for single-head and multi-head attention.

Single-Head Attention

$$\text{Output} = \mathbf{A}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = S\left(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d_k}}\right)\mathbf{V}$$

Which can also be written as

$$\text{Output} = H^{(\text{attn})} W_O \in \mathbb{R}^{L \times d}$$

- Projects from $d_v \rightarrow d$
- Produces **contextualized representations per token**

Multi-Head Attention

For each head $h = 1, \ldots, H$:

$$\text{head}_h = \text{Attention}(Q_h, K_h, V_h) \in \mathbb{R}^{L \times d_v}$$

Concatenate all heads:

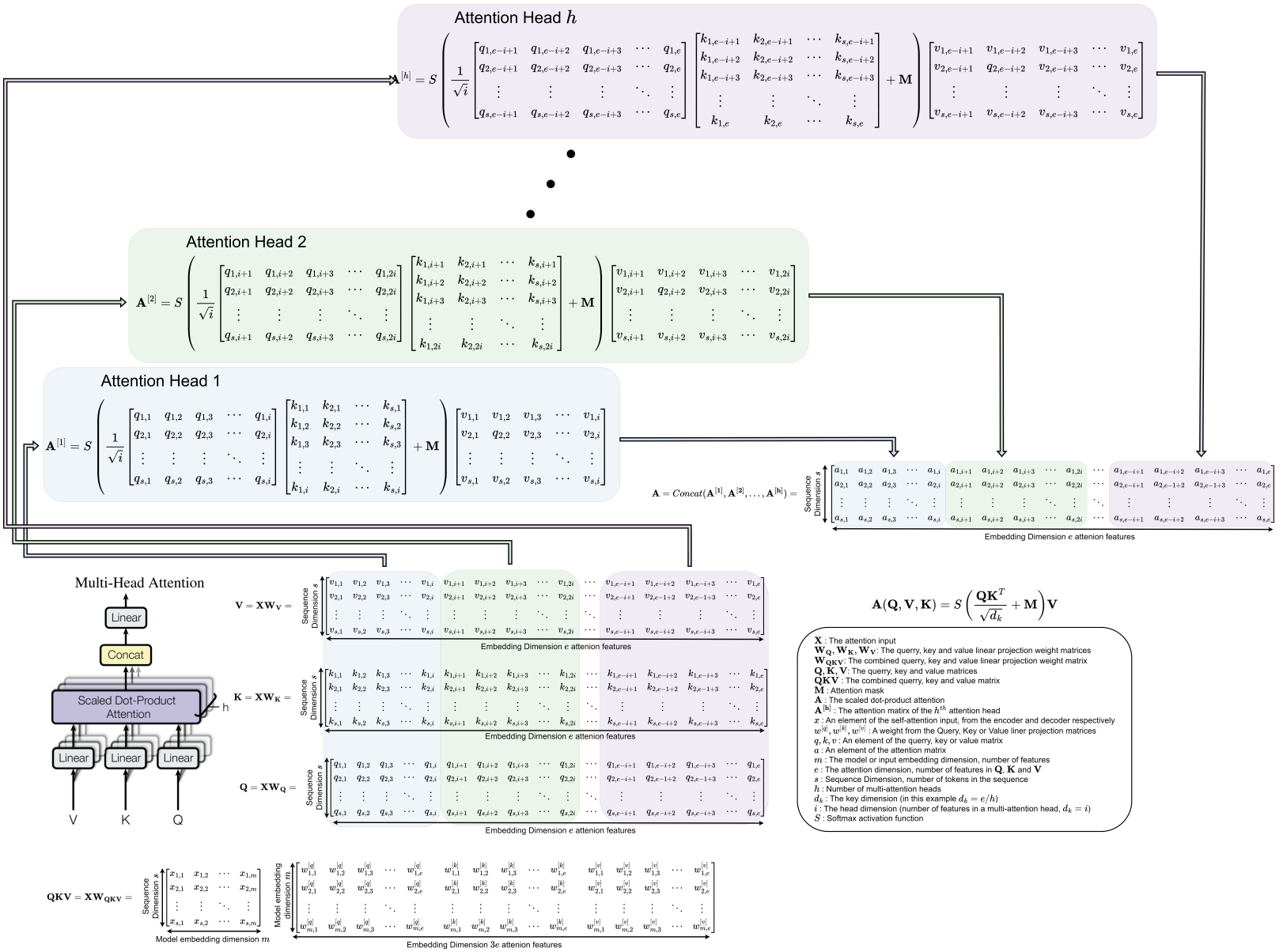
$$\text{Concat}(\text{head}_1, \ldots, \text{head}_H) \in \mathbb{R}^{L \times (H \cdot d_v)}$$

Final projection:

$$\text{Output} = \text{Concat}(\cdot)\, W_O \in \mathbb{R}^{L \times d}$$

- Each head learns a **different attention pattern / subspace**
- Concatenation aggregates these perspectives
- W_O mixes them back into the model dimension d

Multi-headed Attention



$\mathbf{X}$: The attention input
$\mathbf{W_Q}$, $\mathbf{W_K}$, $\mathbf{W_V}$: The query, key and value linear projection weight matrices
$\mathbf{W_{QKV}}$: The combined query, key and value linear projection weight matrix
$\mathbf{Q}, \mathbf{K}, \mathbf{V}$: The query, key and value matrices
$\mathbf{QKV}$: The combined query, key and value matrix
$\mathbf{M}$: Attention mask
$\mathbf{A}$: The scaled dot-product attention
$\mathbf{A^{[h]}}$: The attention matirx of the h^{th} attention head
x : An element of the self-attention input, from the encoder and decoder respectively
$w^{[q]}, w^{[k]}, w^{[v]}$: A weight from the Query, Key or Value liner projection matrices
q, k, v : An element of the querry, key or value matrix
a : An element of the attention matrix
m : The model or input embedding dimension, number of features
e : The attention dimension, number of features in $\mathbf{Q}$, $\mathbf{K}$ and $\mathbf{V}$
s : Sequence Dimension, number of tokens in the sequence
h : Number of multi-attention heads
d_k : The key dimension (in this example $d_k = e/h$)
i : The head dimension (number of features in a multi-attention head, $d_k = i$)
S : Softmax activation function