Transformers

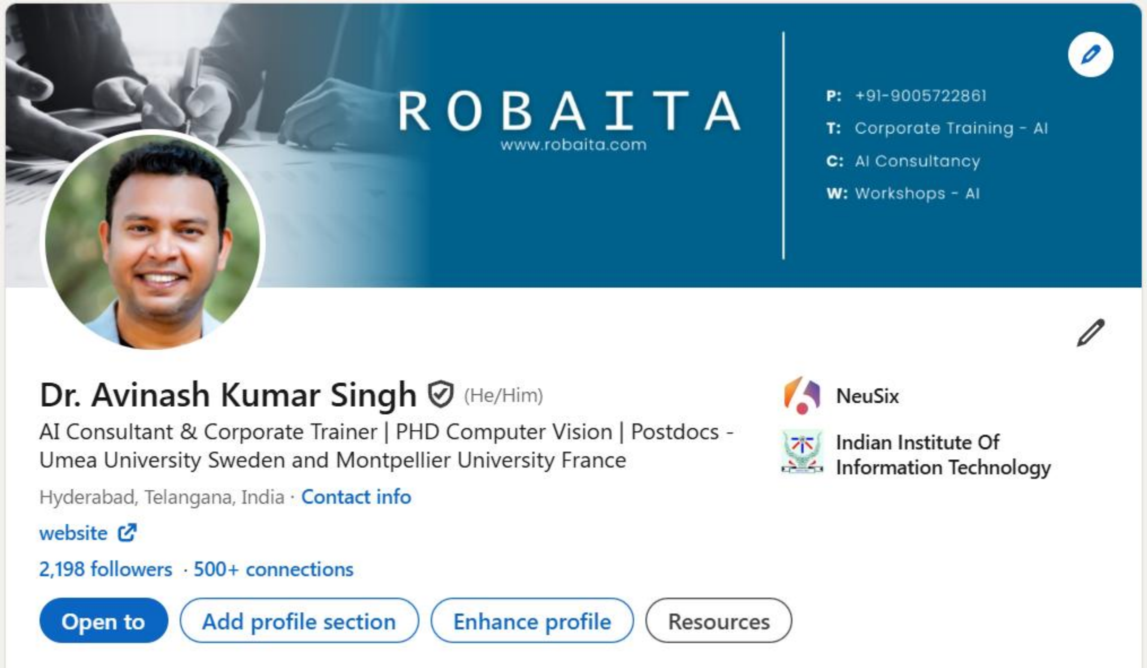
Attention and it’s different types

Dr. Avinash Kumar Singh

AI Consultant and Coach, Robaita

Dr. Avinash Kumar Singh

- **Possess** 15+ years of **hands-on expertise** in Machine Learning, Computer Vision, NLP, IoT, Robotics, and Generative AI.
- **Founded** Robaita—an initiative **empowering** individuals and organizations to **build, educate, and implement** AI solutions.
- **Earned** a Ph.D. in Human-Robot Interaction from IIIT Allahabad in 2016.
- **Received** postdoctoral fellowships at Umeå University, Sweden (2020) and Montpellier University, France (2021).
- **Authored** 30+ research papers in **high-impact** SCI journals and international conferences.
- Unlearning, learning, making mistakes …

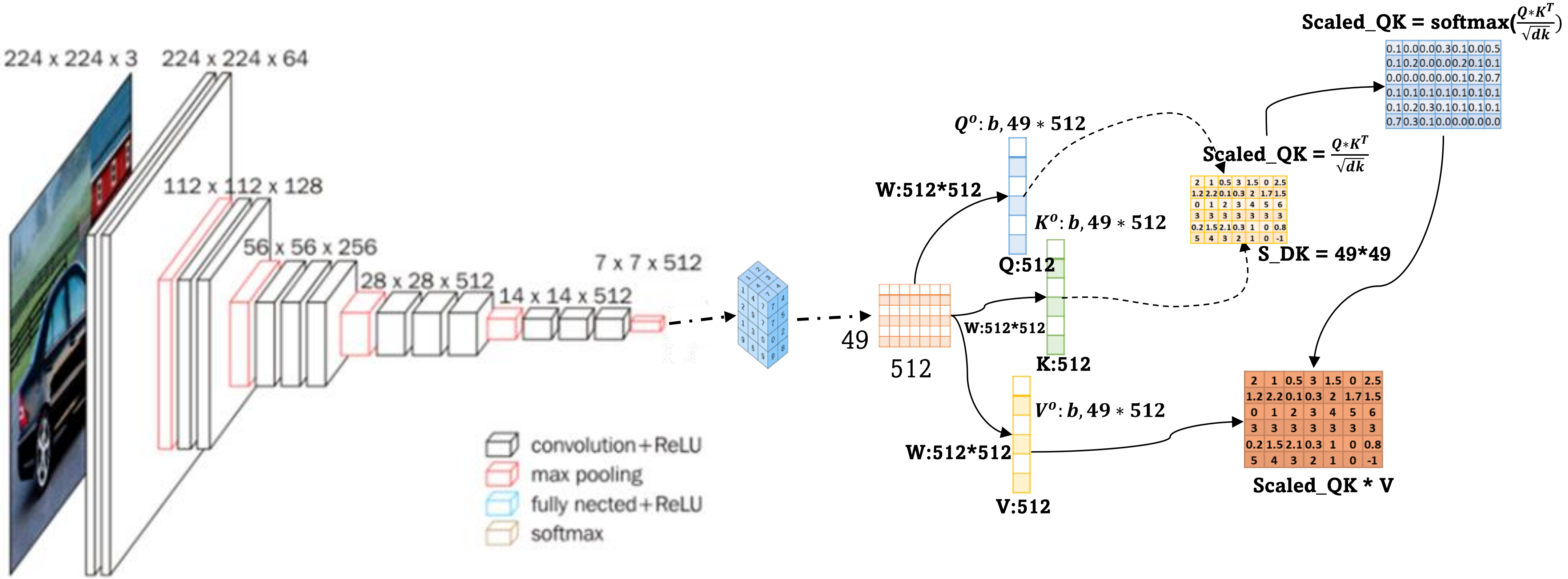


https://www.linkedin.com/in/dr-avinash-kumar-singh-2a570a31/

Discussion Points

- Attention
  - Self Attention
  - Multi Head Attention
  - Cross Attention
- Transformers
  - Basic idea
  - Decoder Only Network (GPT-2)
  - Encode Only Network (BERT)
  - Encode-Decoder Network (Machine Translation)

Self Attention



Self Attention: Example

Let's take an example, sentence and find out how attention works

"The cat sat on the mat"

Let's assume that every words is represented in R^7

Q = [0, 1, 0, 1, 1, 0, 0] # "cat"

Dot Products:

- *with The:* $0\times1 + 1\times0 + 0\times0 + 1\times1 + 1\times0 + 0\times0 + 0\times1 = 1$
- *with Cat:* *self-dot* $= 0\times0 + 1\times1 + 0\times0 + 1\times1 + 1\times1 + 0\times0 + 0\times0 = 3$
- *with Sat:* $0\times0 + 1\times0 + 0\times1 + 1\times0 + 1\times1 + 0\times1 + 0\times0 = 1$
- *with On:* $0\times1 + 1\times0 + 0\times1 + 1\times0 + 1\times0 + 0\times1 + 0\times0 = 0$
- *with The:* *same as before* $= 1$
- *with Mat:* $0\times0 + 1\times1 + 0\times1 + 1\times0 + 1\times0 + 0\times1 + 0\times0 = 1$

Row attention Score: [1, 3, 1, 0, 1, 1], softmax([1, 3, 1, 0, 1, 1]) ≈ [0.089, 0.659, 0.089, 0.033, 0.089, 0.089]

<u>New "cat" representation = 0.089×The + 0.659×Cat + 0.089×Sat + 0.033×On + 0.089×The + 0.089×Mat</u>

Word	**7-D Embedding (Vector)**
The (1)	[1, 0, 0, 1, 0, 0, 1]
Cat	[0, 1, 0, 1, 1, 0, 0]
Sat	[0, 0, 1, 0, 1, 1, 0]
On	[1, 0, 1, 0, 0, 1, 0]
The (2)	[1, 0, 0, 1, 0, 0, 1] (same as The (1))
Mat	[0, 1, 1, 0, 0, 1, 0]

What Does Dot Product do?

At its core, attention answers this question:

“For each word (or token), which other words in the sequence are important to look at?”

To do this, each token (say, “cat”) becomes a query vector, and it compares itself to all other tokens (which are key vectors) — the more similar a key is to the query, the more the query "attends" to that token.

Dot Product: $Q * K^T$

If a query vector and a key vector point in the same direction, their dot product is large → the query "likes" that key.

Why

- *It's fast, differentiable, and scales with vector similarity.*
- *Higher dot product → higher alignment → more attention paid to that token.*

What Does $\sqrt{dk}$ and Softmax, do?

The effect of Normalization

If the vectors are high-dimensional, their dot product values can get large, causing softmax to **saturate** (outputs close to 0 or 1). That leads to:

- **Vanishing gradients**
- **Unstable training**

So, we **normalize** the dot product: $\frac{Q*K^T}{\sqrt{dk}}$

- This keeps values in a range where softmax gradients are useful.

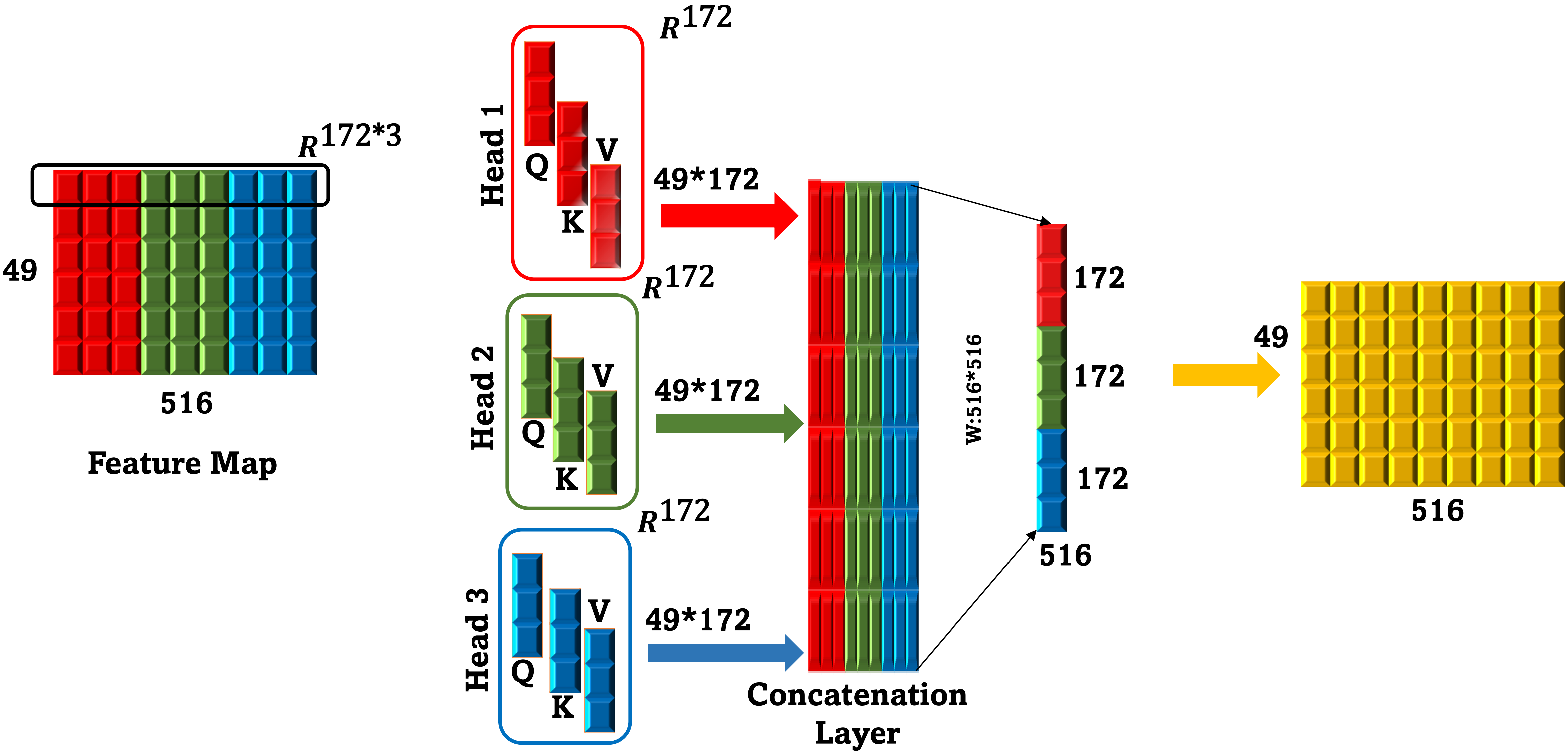
Softmax: attention weights = softmax($\frac{Q*K^T}{\sqrt{dk}}$)

Softmax turns the "similarity" numbers into how much focus a token gives to each other token.

Why

- *Converts raw scores into probabilities*
- *Ensures the weights:*
  - *are non-negative*
  - *sum to 1*
- *Let's each token compute a weighted average of value vectors*

Mult Head Attention



Transformers

- Input Embedding
- Output Embedding
- Position Encoding
- Add & Norm
- Feed forward
- Cross Attention
- Multi Head Attention
- Masked Multi Head Attention
- Linear Layer

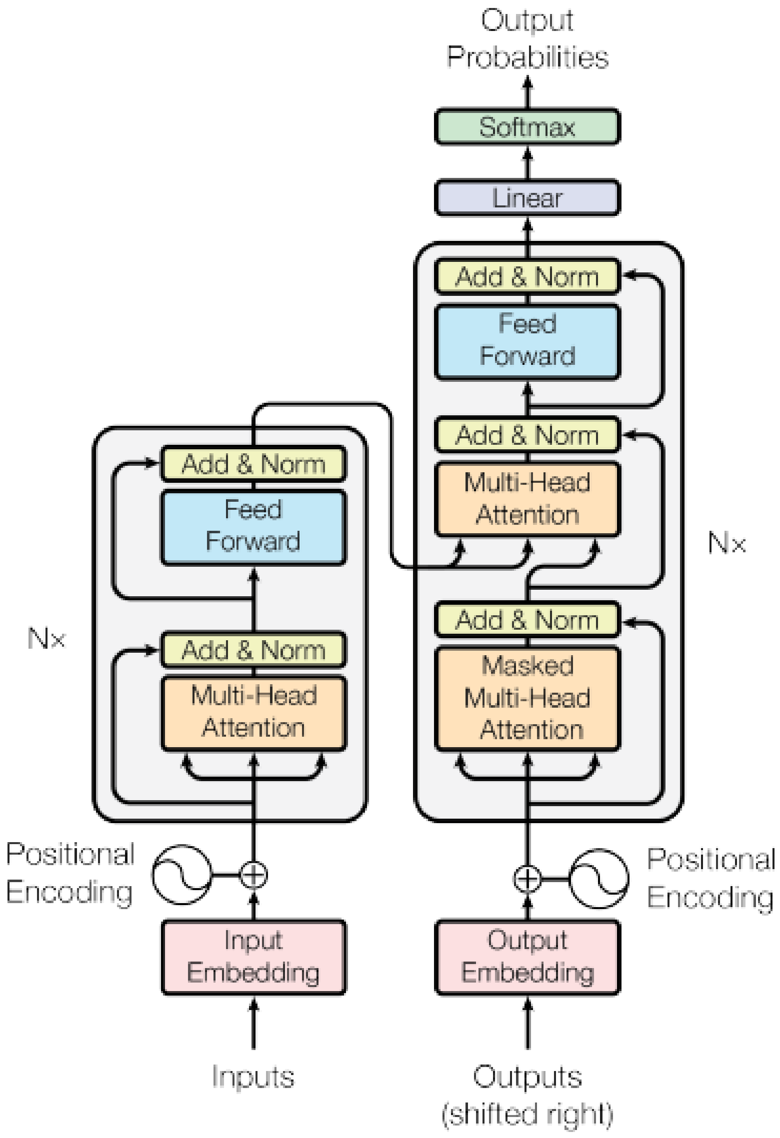


Figure 1: The Transformer - model architecture.

Vaswani, Ashish, et al. "Attention is all you need." *Advances in neural information processing systems* 30 (2017).

Transformers

Positional Embedding

- Transformers process inputs in parallel (no recurrence, no convolution).
- Therefore, they need positional information to understand the order of tokens in a sequence.

Without it

"I ate pizza" vs "Pizza ate I" would be indistinguishable to the model.

Even Indices

$$\mathrm{PE}(pos, 2i) = \sin\left(\frac{pos}{10000^{\frac{2i}{d_{\text{model}}}}}\right)$$

Odd Indices

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

- pos = position in sequence (e.g., 0, 1, 2, ...)
- i = dimension index in the vector (e.g., 0, 1, 2, ..., d_model - 1)
- d_model = embedding size (e.g., 8, 16, 512)

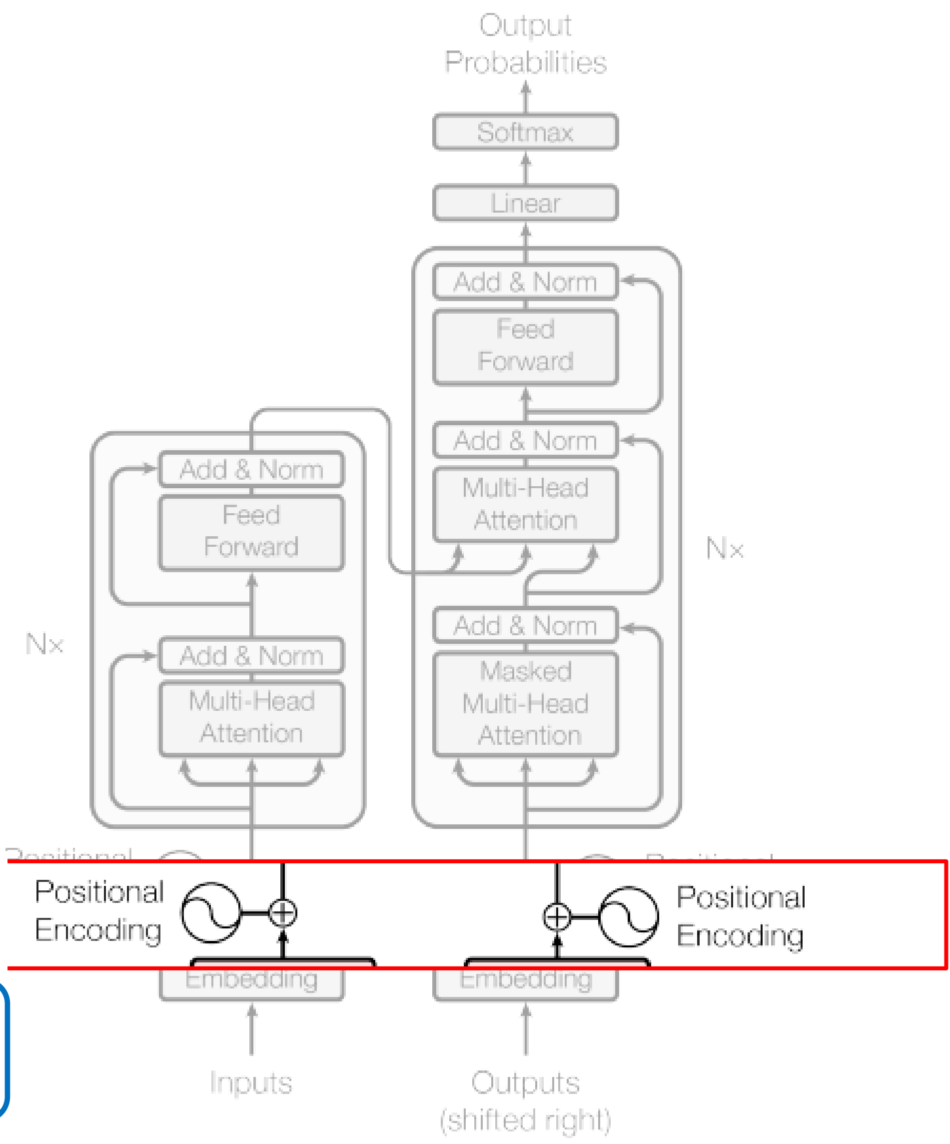


Figure 1: The Transformer - model architecture.

Vaswani, Ashish, et al. "Attention is all you need." *Advances in neural information processing systems* 30 (2017).

Transformers

Positional Embedding

Input: *"the cat sat on the mat"*

Let's calculate the encoding for "cat'

- Word: "cat"
- Position: pos = 1
- Model dimension: d_model = 4
- Indices: i = 0 to 3

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

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

$$\boxed{\mathrm{PE}(1,0) = \sin\left(\frac{1}{10000^{0/4}}\right) = \sin(1/1) = \sin(1) \approx 0.84147}$$

$$\boxed{\mathrm{PE}(1,1) = \cos\left(\frac{1}{10000^{0/4}}\right) = \cos(1/1) = \cos(1) \approx 0.54030}$$

$$\boxed{\mathrm{PE}(1,2) = \sin\left(\frac{1}{10000^{2/4}}\right) = \sin(1/100) = \sin(0.01) \approx 0.0099998}$$

$$\boxed{\mathrm{PE}(1,3) = \cos\left(\frac{1}{10000^{2/4}}\right) = \cos(1/100) = \cos(0.01) \approx 0.99995}$$

$$[0.84147,\ 0.54030,\ 0.0099998,\ 0.99995]$$

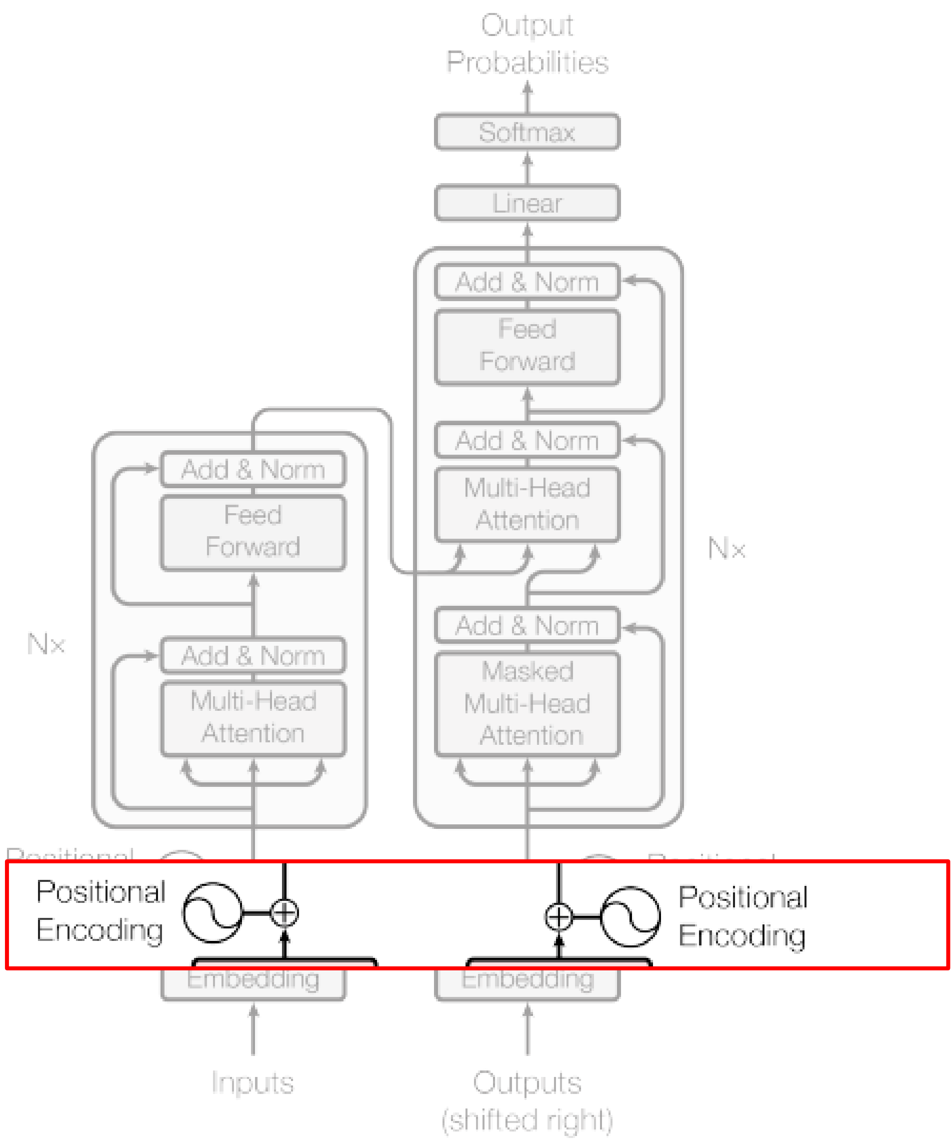


Figure 1: The Transformer - model architecture.

Vaswani, Ashish, et al. "Attention is all you need." *Advances in neural information processing systems* 30 (2017).

Transformers

Input Embedding

= Word Embedding + Positional Encoding

Word Embedding

Each word is converted into a dense vector using a learned embedding layer.

Word → Index → Vector

Token	Token ID	Embedding Vector (d_model=4)
the	5	[0.1, 0.3, 0.5, 0.2]
cat	42	[0.6, 0.4, 0.2, 0.9]
sat	33	[0.3, 0.8, 0.6, 0.1]
on	14	[0.9, 0.2, 0.1, 0.5]
mat	71	[0.7, 0.1, 0.3, 0.4]

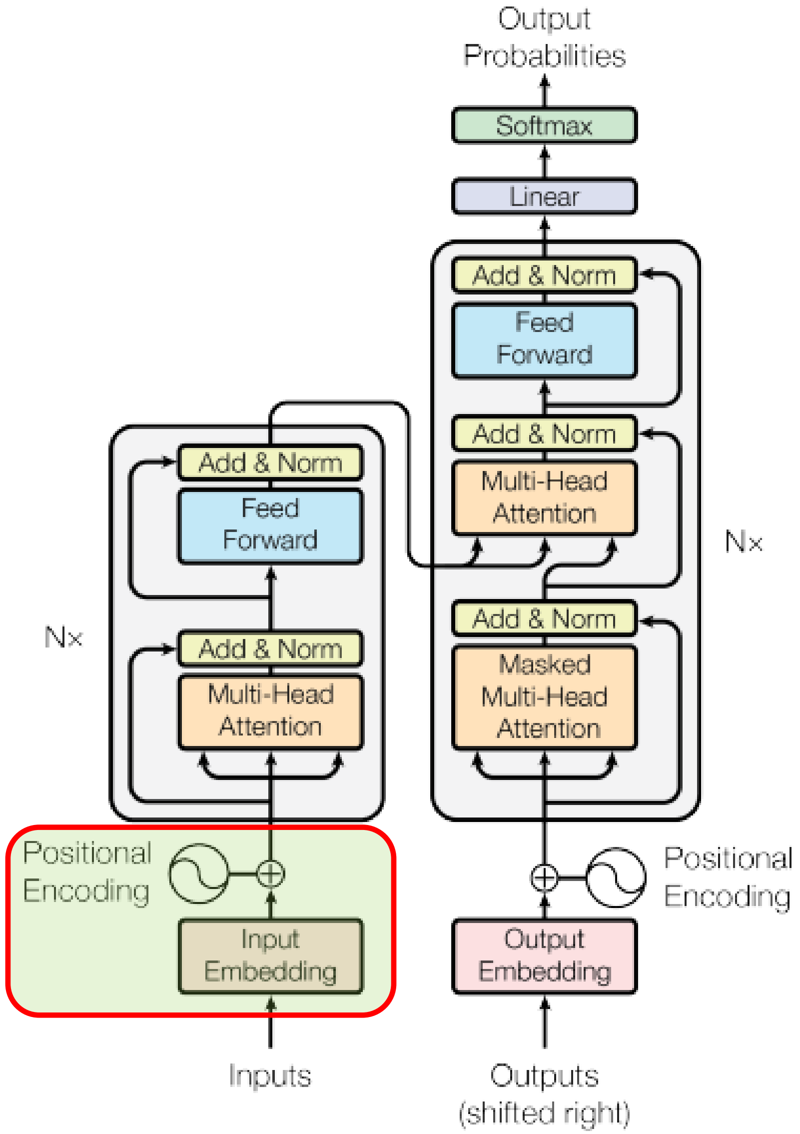


Figure 1: The Transformer - model architecture.

Vaswani, Ashish, et al. "Attention is all you need." *Advances in neural information processing systems* 30 (2017).

Transformers

Input Embedding

Token	Token ID	Embedding Vector (d_model=4)
the	5	[0.1, 0.3, 0.5, 0.2]
cat	42	[0.6, 0.4, 0.2, 0.9]
sat	33	[0.3, 0.8, 0.6, 0.1]
on	14	[0.9, 0.2, 0.1, 0.5]
mat	71	[0.7, 0.1, 0.3, 0.4]

Positional Embedding Vector (d_model=4)
[0.375 0.951 0.732 0.599]
[0.841, 0.540, 0.010, 0.999]
[0.156 0.058 0.866 0.601]
[0.708 0.021 0.97 0.832]
[0.212, 0.183, 0.304, 0.525]

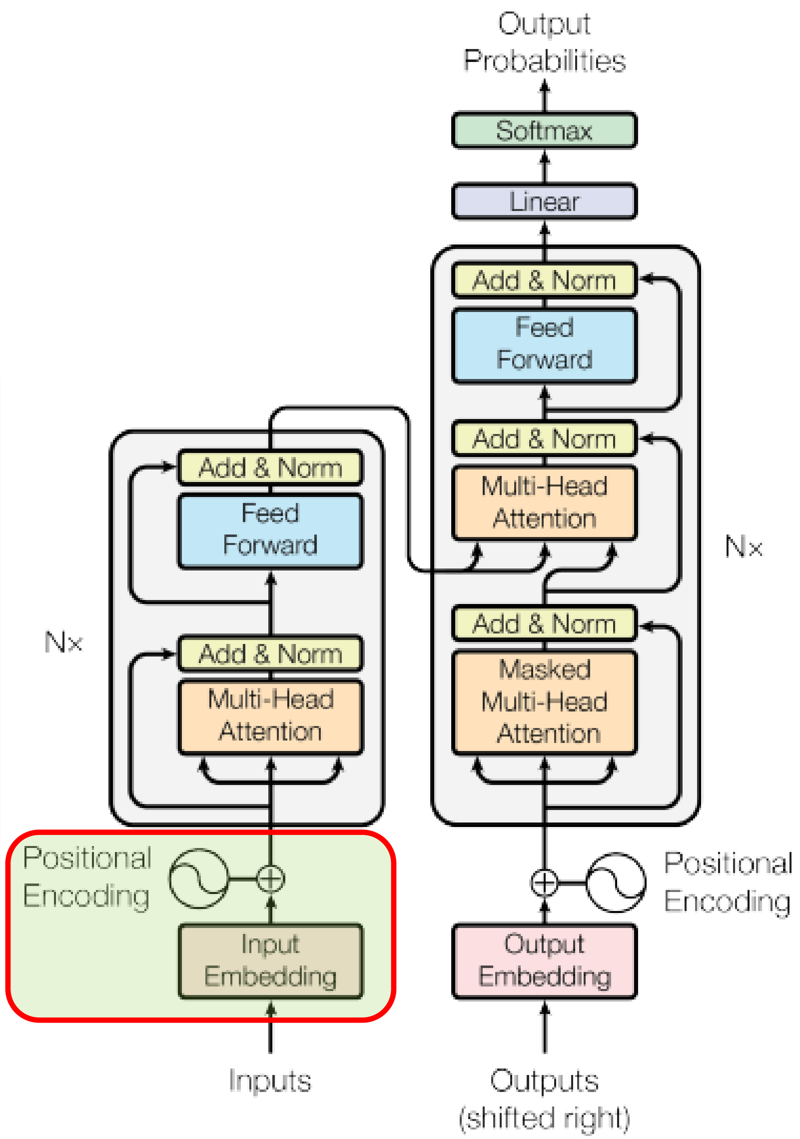


Figure 1: The Transformer - model architecture.

Vaswani, Ashish, et al. "Attention is all you need." *Advances in neural information processing systems* 30 (2017).

Transformers

Output Embedding = Word Embedding + Positional Encoding

In a Transformer, **output embedding** refers to the **embedding of the target tokens** (i.e., the tokens the model is supposed to generate), typically used in the **decoder** during training.

Input Embedding → for the input sequence (e.g., "The cat sat…")

Output Embedding → for the output sequence (e.g., "Le chat s'est…")

Word Embedding [French]

During **training**, the decoder receives the correct target tokens (i.e., ground truth). These are tokenized and then passed through an **embedding layer** (just like the input side).

- a token ID
- then embedded to a vector of dimension d_model

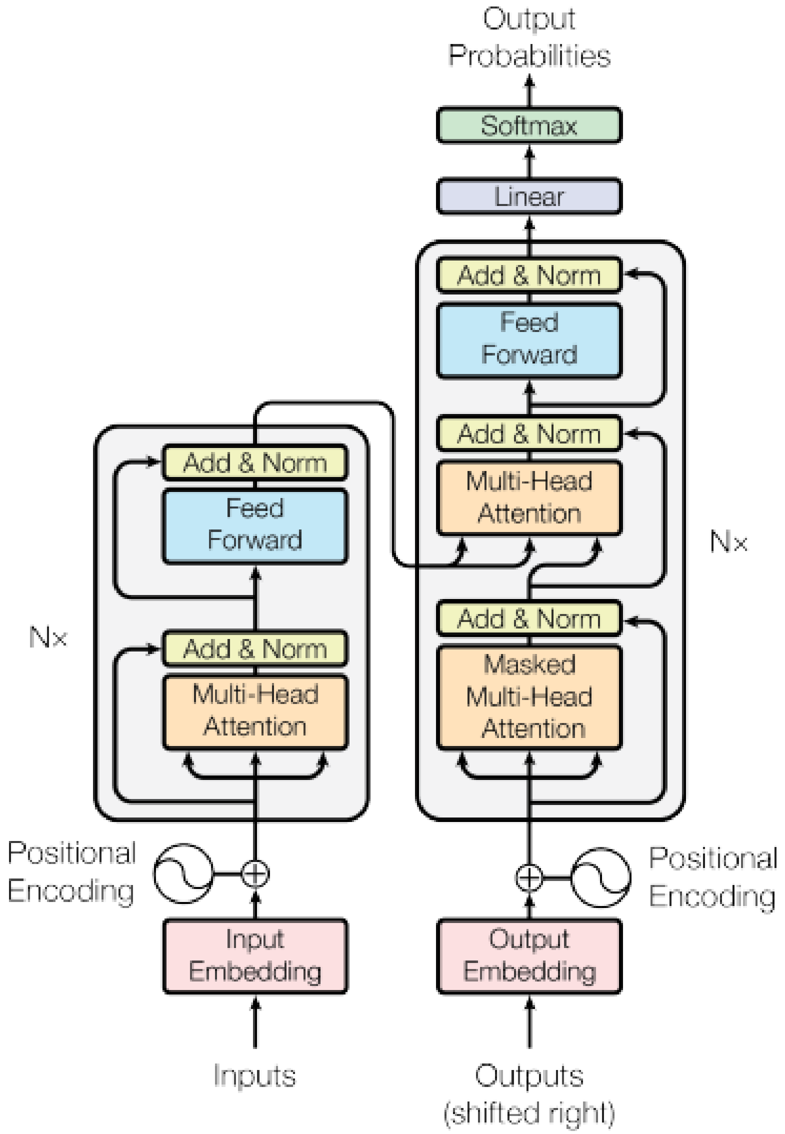


Figure 1: The Transformer - model architecture.

Vaswani, Ashish, et al. "Attention is all you need." *Advances in neural information processing systems* 30 (2017).

Transformers

Encoder Block

“The cat sat on the mat”

- Number of tokens (sequence length) = 6
- Vector dimension (d_model) = 4
- Input matrix = $X \in \mathbb{R}^{6x4}$

1

Token	Vector [word+pos]			
the	0.1	0.3	0.5	0.2
cat	0.6	0.4	0.2	0.9
sat	0.3	0.8	0.6	0.1
on	0.9	0.2	0.1	0.5
the	0.1	0.3	0.5	0.2
mat	0.7	0.1	0.3	0.4

Multi Head Attention [H=2]

2

$\mathbb{R}^{6x4}$

Add & Norm

3

Token	Vector [word+pos]			
the	0.1	0.3	0.5	0.2
cat	0.6	0.4	0.2	0.9
sat	0.3	0.8	0.6	0.1
on	0.9	0.2	0.1	0.5
the	0.1	0.3	0.5	0.2
mat	0.7	0.1	0.3	0.4

+

Token	Vector [word+pos]			
the	0.1	0.3	0.5	0.2
cat	0.6	0.4	0.2	0.9
sat	0.3	0.8	0.6	0.1
on	0.9	0.2	0.1	0.5
the	0.1	0.3	0.5	0.2
mat	0.7	0.1	0.3	0.4

$\mathbb{R}^{6x4}$

4

Feed Forward

$\mathbb{R}^{6x4}$

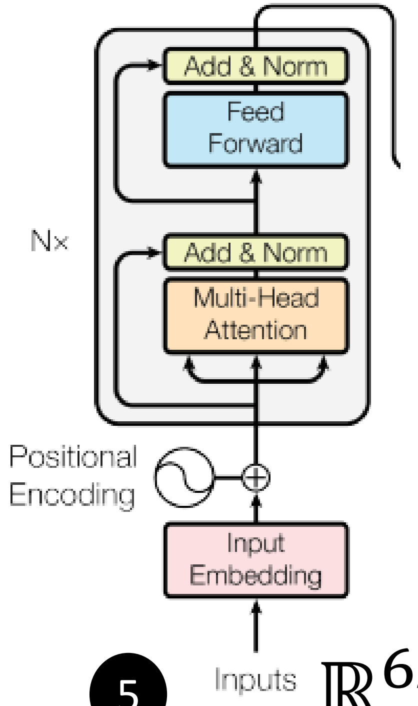
$$FFN(x) = ReLU(x \times W_1 + b_1) \times W_2 + b_2$$

- First Dense: $W_1 \in \mathbb{R}^{4x8}$ → expands to 8 dimensions
  - Second Dense: $W_2 \in \mathbb{R}^{8x4}$ → projects back to 4

Add & Norm

5

Token	Vector [word+pos]			
the	0.1	0.3	0.5	0.2
cat	0.6	0.4	0.2	0.9
sat	0.3	0.8	0.6	0.1
on	0.9	0.2	0.1	0.5
the	0.1	0.3	0.5	0.2
mat	0.7	0.1	0.3	0.4

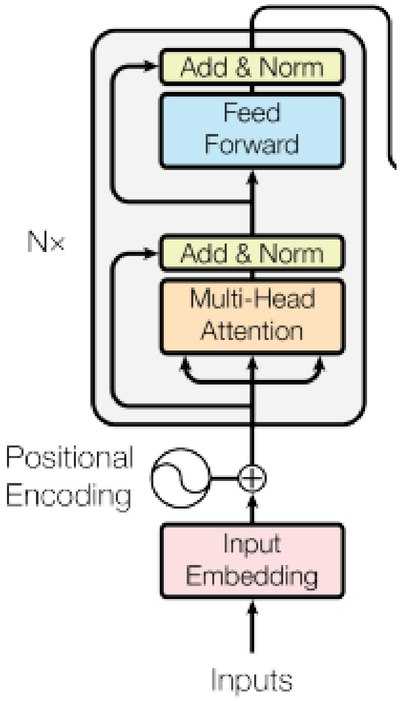


$\mathbb{R}^{6x4}$

Transformers

Encoder Block

Step	Shape	Description
Input	(6, 4)	6 tokens, 4-dim vectors
Q, K, V projection	(6, 4) each	Linear projections
Split into heads	(1, 2, 6, 2)	2 heads, each of depth 2
Attention per head	(6, 2)	Each head computes its attention output
Concatenate heads	(6, 4)	Join outputs of 2 heads
Dense after concat	(6, 4)	Output of MHA
Add & Norm	(6, 4)	Residual + LayerNorm
Feed Forward	(6, 4)	Dense → ReLU → Dense
Add & Norm again	(6, 4)	Residual + LayerNorm



Transformers

Masked Multi Head Attention

- Masked attention ensures that **each position in the decoder** can **only attend to earlier positions** (and itself).
- This is **essential during training** so the model **doesn't cheat** by looking ahead at future tokens.

Example

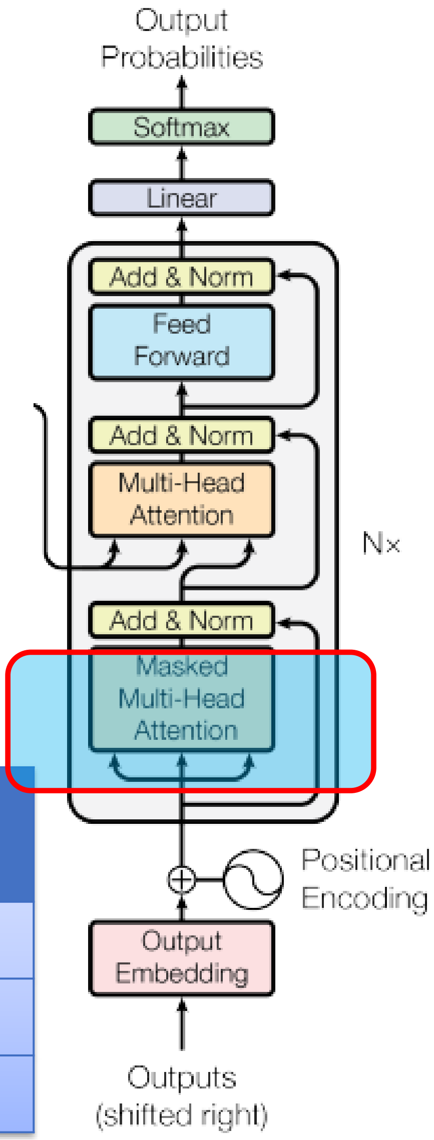
Let's say the **decoder has seen only the first 3 tokens of a sentence** during generation:

"Le chat dort" ("The cat sleeps")

- sequence length = 3
- d_model = 4
- num_heads = 1
- depth = 4

Q = K = V
[0.1, 0.0, 0.3, 0.7]
[0.4, 0.1, 0.2, 0.6]
[0.8, 0.2, 0.1, 0.5]

Token	Token ID	Embedding Vector (d_model=4)
le	5	[0.1, 0.3, 0.5, 0.2]
chat	16	[0.6, 0.4, 0.2, 0.9]
dort	23	[0.3, 0.8, 0.6, 0.1]



Transformers

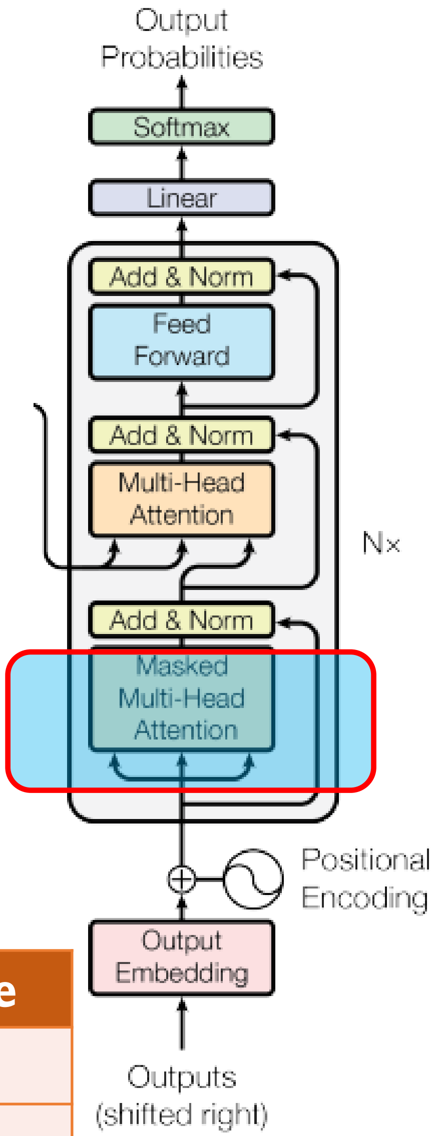
Masked Multi Head Attention

Example: *"Le chat dort" ("The cat sleeps")*

Scaled Dot Product (Q&K)

$$\text{score}_{i,j} = \frac{Q_i \cdot K_j^T}{\sqrt{d_k}}$$

	Token 1 ("Le")	Token 2 ("chat")	Token 3 ("dort")
Token 1	(0.1×0.1 + ...) = **0.63** → ÷2 = 0.315	**0.54**	**0.45**
Token 2	**0.54**	**0.61**	**0.56**
Token 3	**0.45**	**0.56**	**0.54**



Attention Score Matrix
[0.315, 0.270, 0.225]
[0.270, 0.305, 0.280]
[0.225, 0.280, 0.270]

Look-Ahead Mask →

Masked Score
[0.315, -∞, -∞]
[0.270, 0.305, -∞]
[0.225, 0.280, 0.270]

Softmax →

Attention Score
[1.0, 0.0, 0.0]
[0.491, 0.509, 0.0]
[0.326, 0.347, 0.327]

Transformers

Masked Multi Head Attention

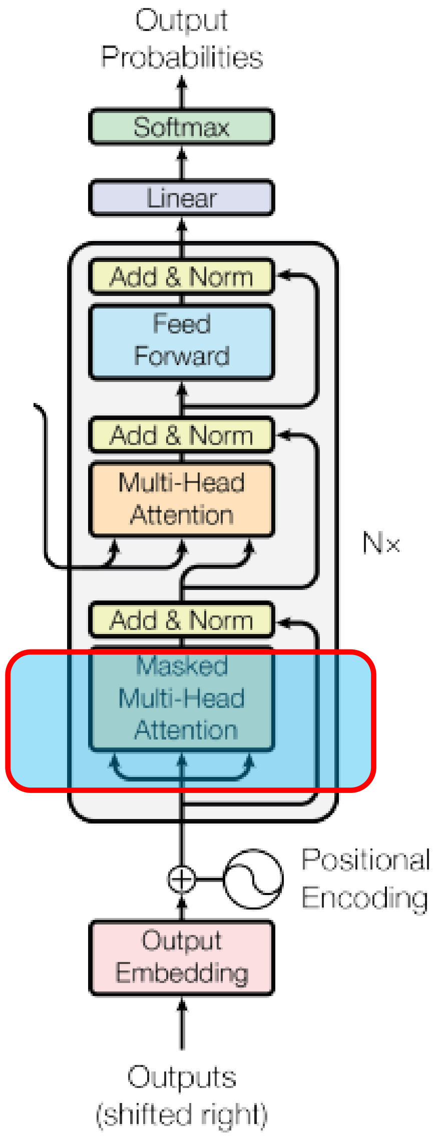
Example: *"Le chat dort" ("The cat sleeps")*

Final Attention Output

$\mathbb{R}^{3x4}$

Multiply Attention Scores with Value Vector
output[1] = 1.0 * V[0] = V[0]
output[2] = 0.491 * V[0] + 0.509 * V[1]
output[3] = 0.326 * V[0] + 0.347 * V[1] + 0.327 * V[2]

Step	Role
Dot Product (QK^T)	Measures similarity between tokens
Scaling (÷√d_k)	Prevents large softmax values
Masking	Ensures no peeking ahead
Softmax	Produces attention weights
MatMul with V	Weighted average of value vectors



Transformer

Cross Multi-Head Attention

Let decoder attend to encoder outputs (i.e., from "the cat sat on the mat")

Let:

- Encoder output = E ∈ (6, 4)

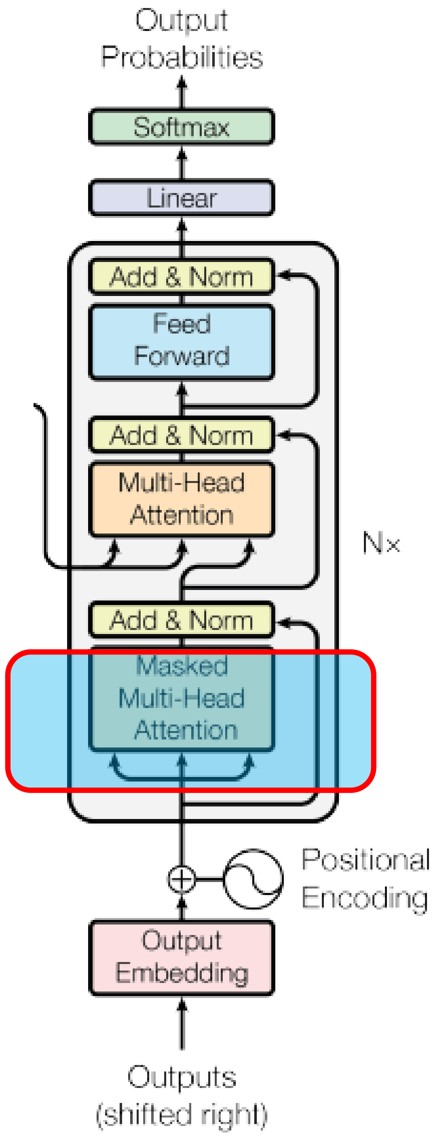
Here:

- Q = from decoder (shape = (3, 4))
- K, V = from encoder output E (shape = (6, 4))

→ Attention shape:

- Each head: (3, 2)
- Concatenated: (3, 4)

Output shape: (3, 4)



Transformer

Linear Layer (Dense Layer)

After the final decoder block:

- A tensor of shape: (batch_size, target_seq_len, d_model)

This tensor contains the decoder's output.

- A sequence of context-rich vectors, one per position in the output sentence (e.g., "Le chat dort...").

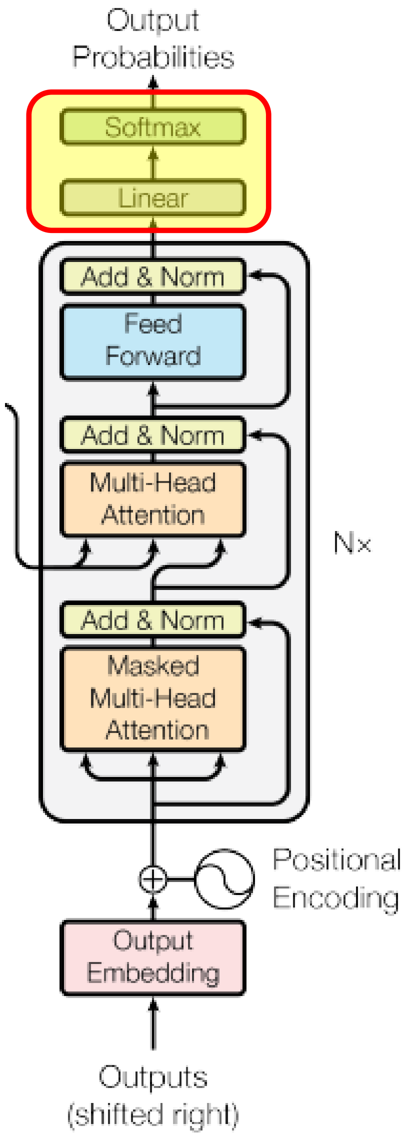
$\mathbb{R}^{3x4}$

We need to convert each of those vectors into probabilities over the vocabulary — to predict the next word.

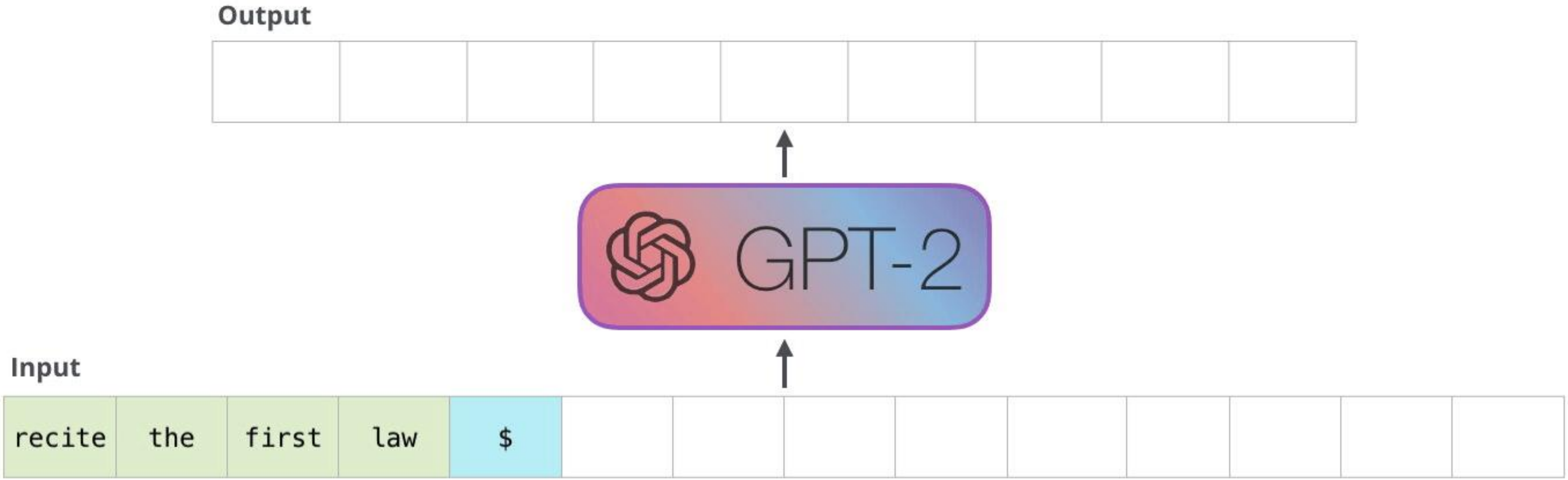
The linear layer acts as a projection from d_model to the vocabulary size (vocab_size):

$$\text{logits} = \text{decoder\_output} \times W_{\text{output}} + b$$

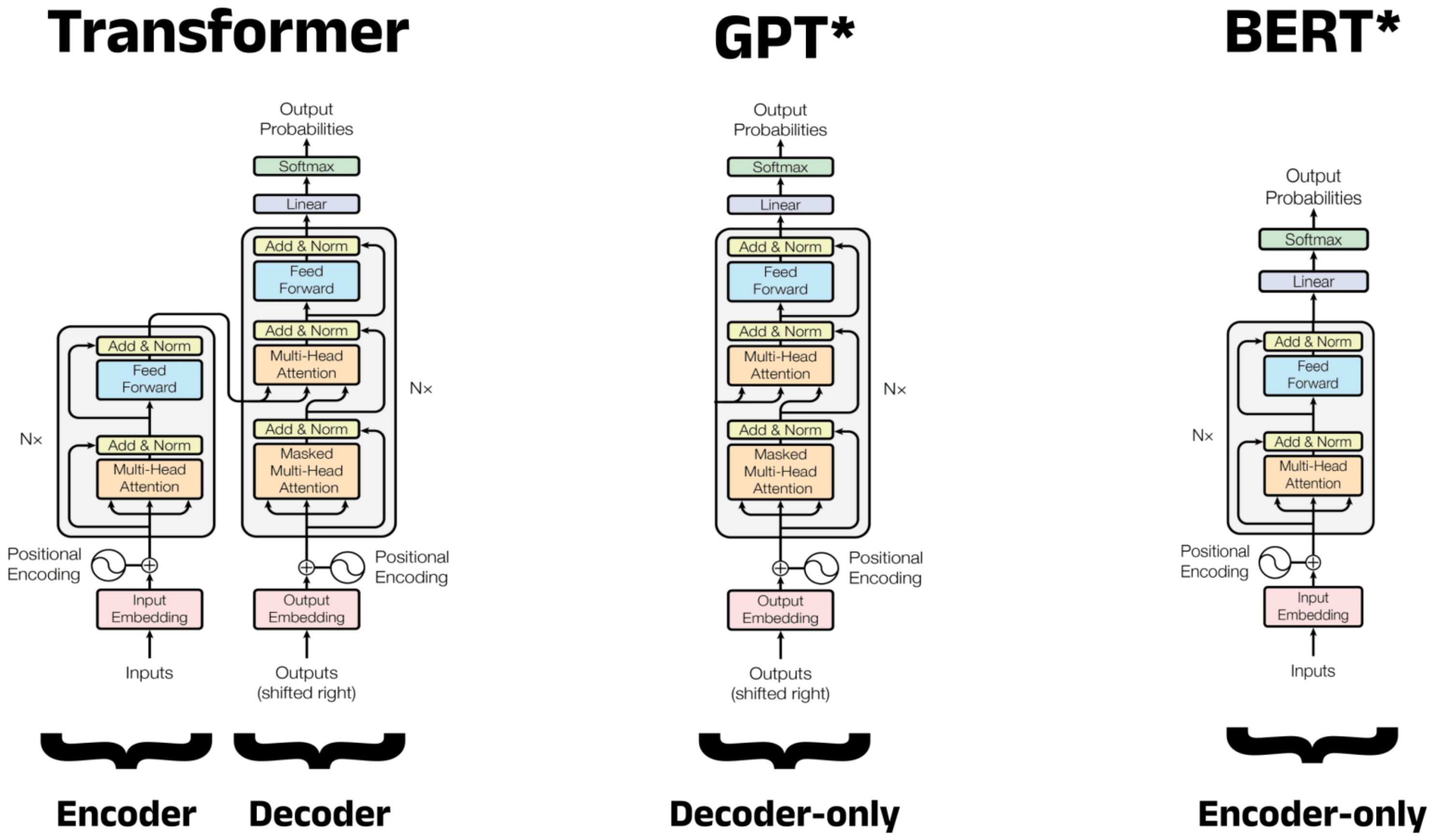
- Decoder final output
  - (batch_size, target_seq_len, d_model)
- Output Dense (Linear)
  - projects to → (batch_size, target_seq_len, vocab_size)



Generative Pre-trained Transformer Architecture



BERT (Bidirectional Encoder Representations from Transformers)



*Illustrative example, exact model architecture may vary slightly

References

- Vaswani, Ashish, et al. "Attention is all you need." Advances in neural information processing systems 30 (2017).
- “GPT-2” explanation, https://jalammar.github.io/illustrated-gpt2/
- “GPT” architecture animation, https://bbycroft.net/llm
- “Let’s Build GPT: from scratch, in code, spelled out.”, Andrej Karpathy, https://www.youtube.com/watch?v=kCc8FmEb1nY
- Devlin, Jacob, et al. "Bert: Pre-training of deep bidirectional transformers for language understanding." Proceedings of the 2019 conference of the North American chapter of the association for computational linguistics: human language technologies, volume 1 (long and short papers). 2019.
- https://huggingface.co/blog/bert-101?
- https://jalammar.github.io/illustrated-bert/

Thanks for your time