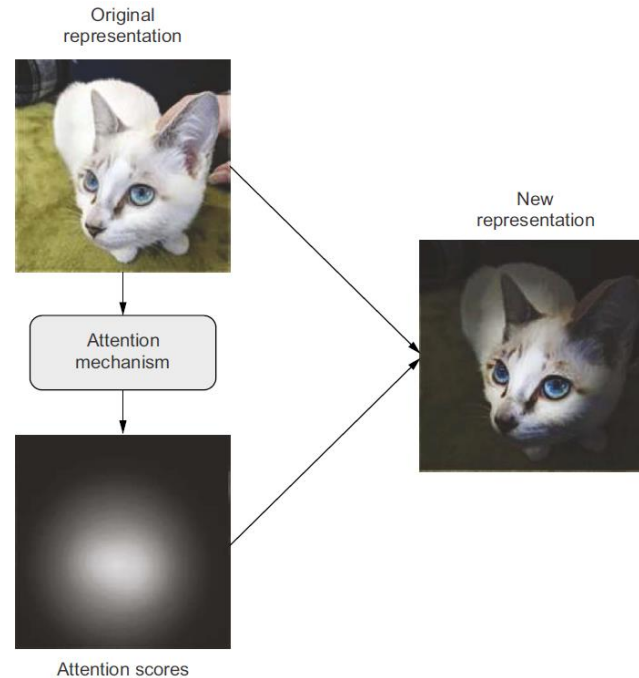


# Module 7 – Part 1

## Transformers prerequisites

### Why Attention is **ALL** you need?

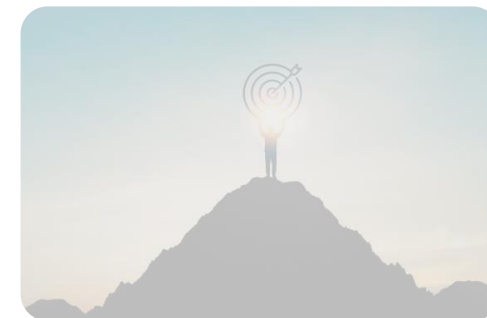
---





# Road map!

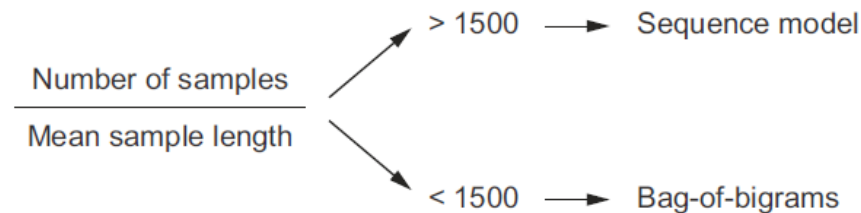
- Module 1- Introduction to Deep Forecasting
- Module 2- Setting up Deep Forecasting Environment
- Module 3- Exponential Smoothing
- Module 4- ARIMA models
- Module 5- Machine Learning for Time series Forecasting
- Module 6- Deep Neural Networks
- Module 7- Deep Sequence Modeling (RNN, LSTM)
- **Module 8- Transformers (Attention is all you need!)**
- Module 9- Prophet and Neural Prophet



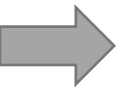
# → Transformers for NLP

- Starting in 2017 (**Attention is all you need!**), transformers started overtaking RNN across most NLP tasks.
- NLP architecture depends on word representation method
  - **Discard order** and treat text as an unordered set of words → bag-of-words models
  - **Respect order** and treat words one at a time (steps in timeseries) → recurrent models
- When to use sequence model over bag-of-words?

- For text classification →



- For any other NLP task → Transformers



# Transformers vs other sequence models

- Transformer architecture is technically **order-agnostic**, yet it **injects word-position** information into the representations it processes (**hybrid approach**)
- Transformers simultaneously look at different parts of a sentence (unlike RNNs) while still being order-aware.

The cat, sat on the mat.

NLP Models	Word order awareness	Context awareness (cross-word interactions)
Bag of unigrams	No	No
Bag of Bigrams	Very limited	No
RNN	Yes	No
Transformer	Yes	Yes

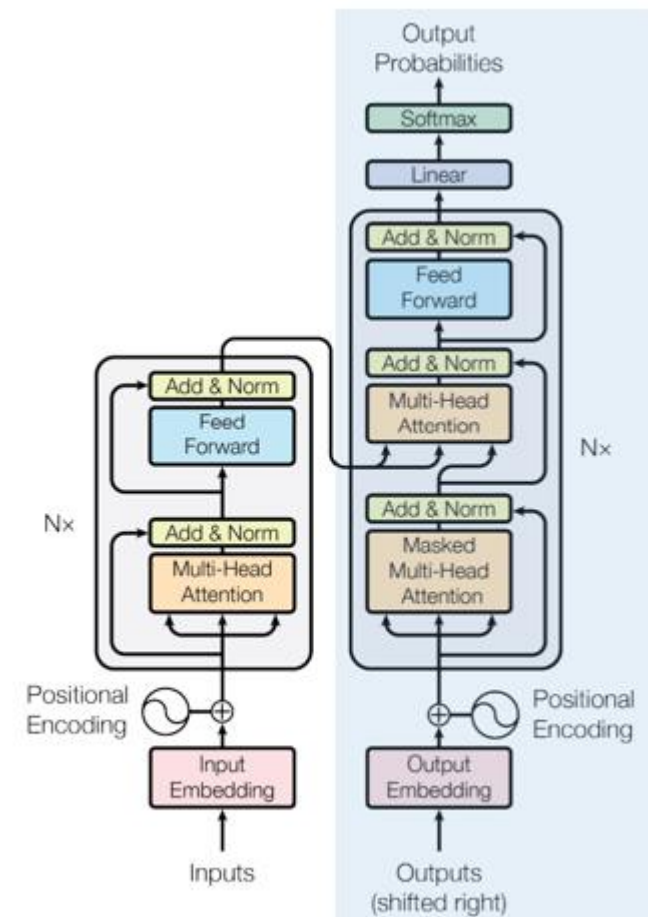
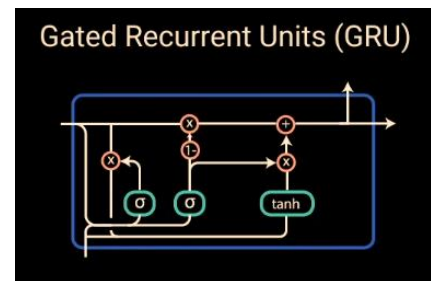
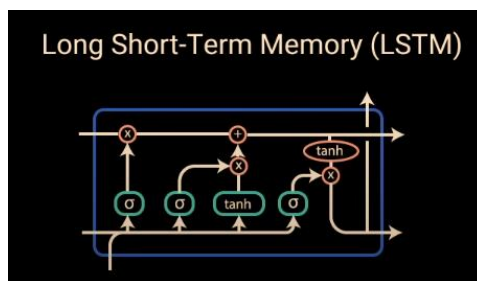
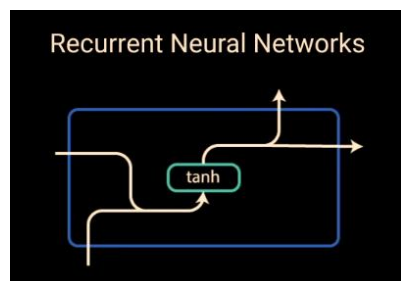




# Sequence Modeling Design Criteria

To model sequence data efficiently, we need an architecture that:

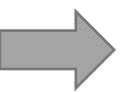
- Preserve the **order**
- Account for **long-term dependencies**
- Handle different **input-length**
- **Share parameters** across the sequence



# ➔ Applications of Transformers

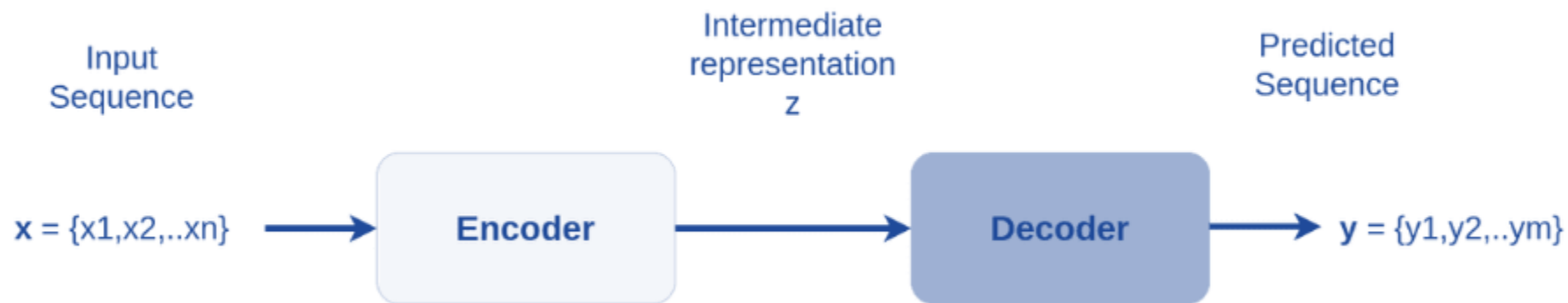
- Transformers are taking NLP, Computer vision and reinforcement learning by storm.
- NLP applications:
  - Machine translation, text generation, text summarization, text classification, chatbots, questions answering etc.
  - BERT, GPT
- Computer vision applications:
  - Image captioning, object detection and segmentation
  - ViT (vision transformers)
- Reinforcement learning applications:
  - Game playing, robotics and autonomous driving





# Sequence-to-sequence modeling

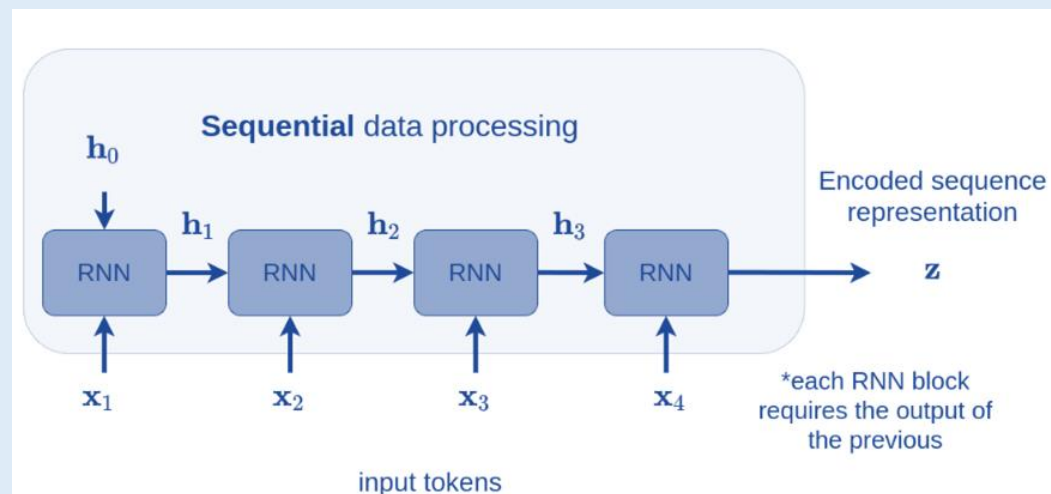
- The goal is to **transform** an input sequence (**source**) to a new one (**target**).



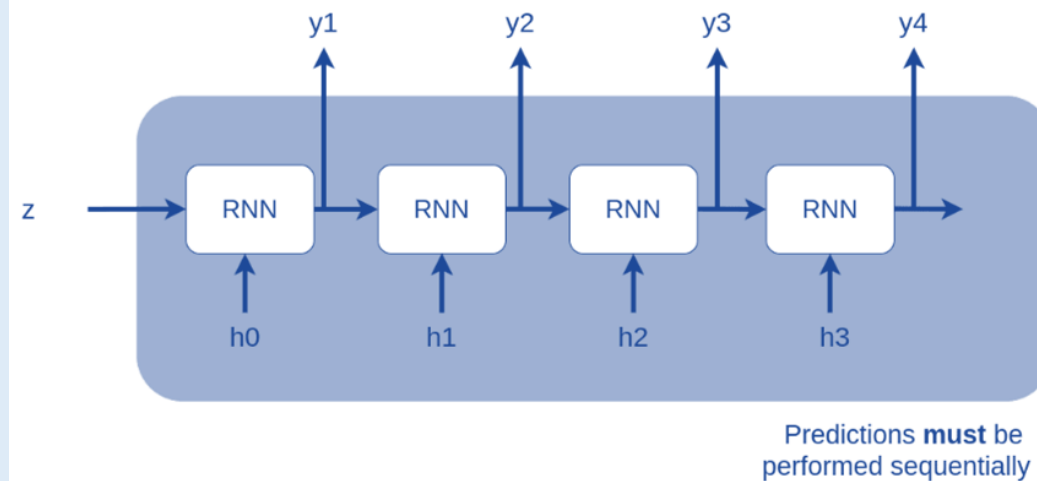
# Encoder – Decoder

- Recurrent Neural Networks (RNNs) were the prevailing method for sequence-to-sequence learning until Transformers demonstrated superior performance.

## Encoder



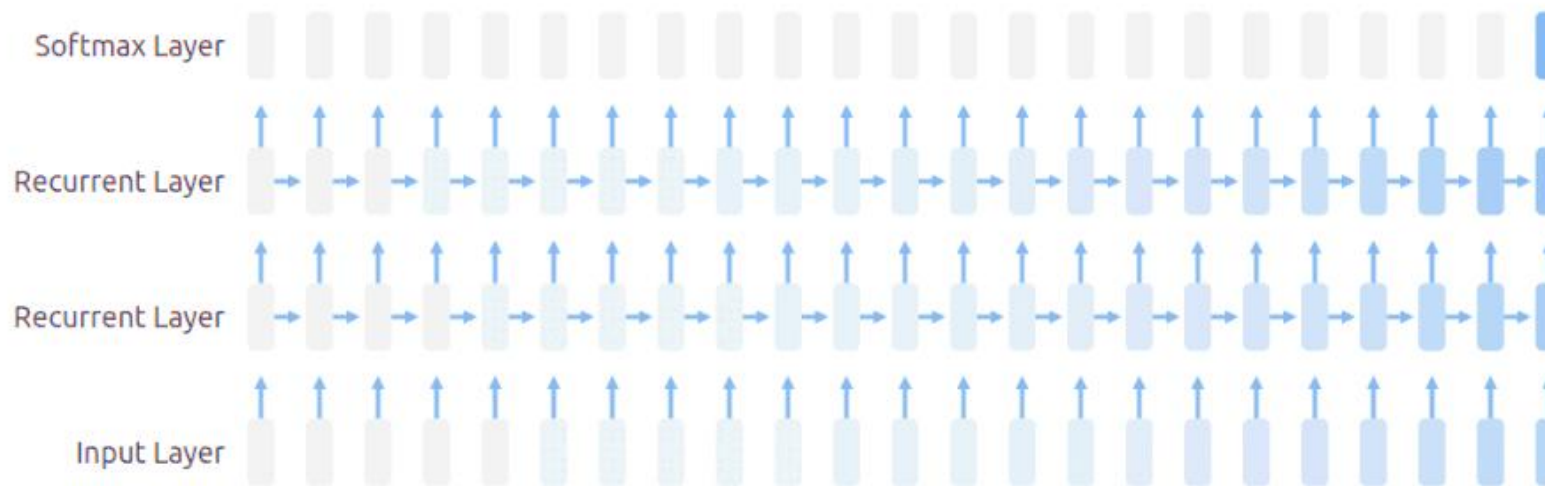
## Decoder





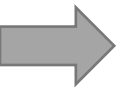
# → Limitations of RNN

- **Bottleneck problem:** the Encoder state vector(s) must store the entire input sequence representation → Significant **limitations on** translatable sentence **size** and **complexity**
- RNN tends to progressively **forget about the past** (~100 tokens) and eventually pays more attention to the **last parts** of the sequence.
- **Vanishing gradient** problem:



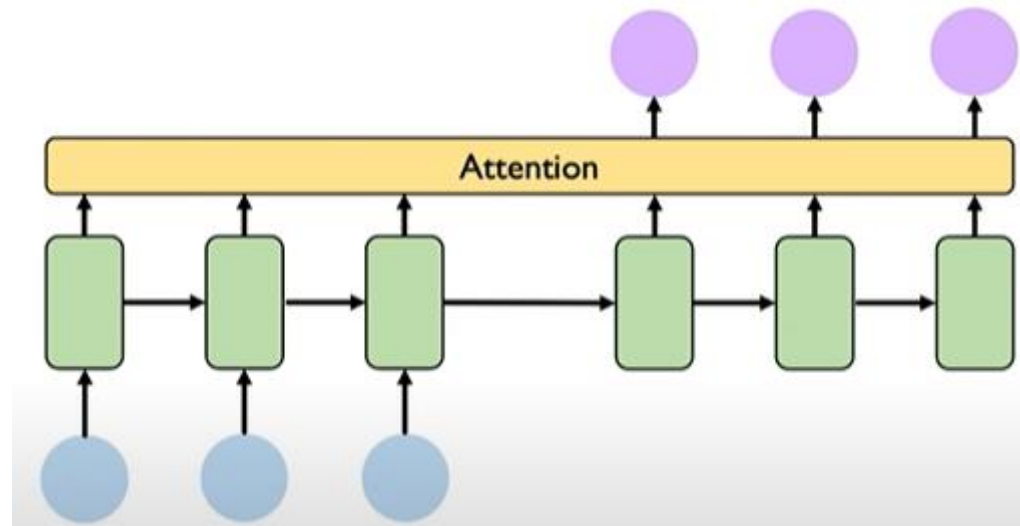
**Vanishing Gradient:** where the contribution from the earlier steps becomes insignificant in the gradient for the vanilla RNN unit. <https://distill.pub/2019/memorization-in-rnns/>

Attention is ALL you need!



# Attention

- To avoid **vanishing gradient**, we need to form a **direct connection** with each timestamp.
- By letting the decoder have an attention mechanism, we relieve the encoder from the burden of having to encode all information in the source sentence (**bottleneck problem**)
- Attention is a mechanism that allows the model to **weigh (score)** and focus on specific parts of the input when generating output.
- Attention mechanism can be applied to any encoder and decoder architecture (RNN, LSTM, GRU, CNN, etc)



# → Attention, deep dive

- What does it mean **mathematically**?
- **Vocabulary**: collection of symbols in each sequence
- Converting symbols to numbers: **One-hot encoding**
- Dot product can be used to measure **similarity**
- One-hot vectors can pull out a particular row of a matrix!

The diagram shows a matrix multiplication where a one-hot vector **A** is multiplied by a matrix **B** to produce a single row from **B**.

1	0	0	0
0	0	0	1
0	0	1	0

**A**

.2	.9
.7	0
.8	.3
.1	.4

**B**

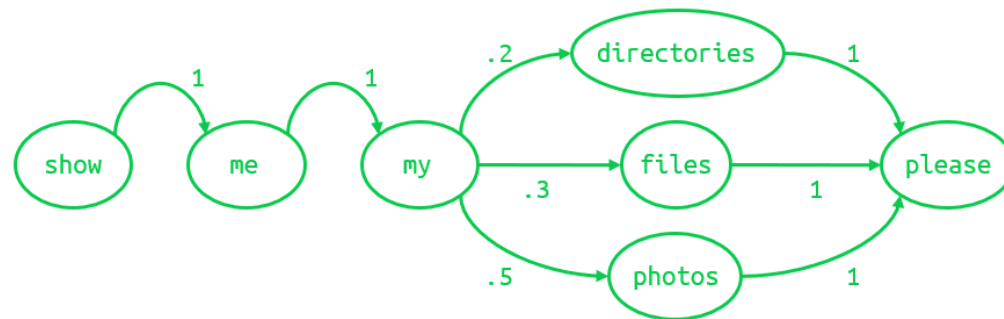
=

.2	.9
.1	.4
.8	.3

# → First order sequence model

- Example:

- Show me my **directories** please
- Show me my **files** please
- Show me my **photos** please



- **Vocabulary** size = 7 {directories, files, me, my, photos, please, show}.
- Markov chain transition model
- Matrix form

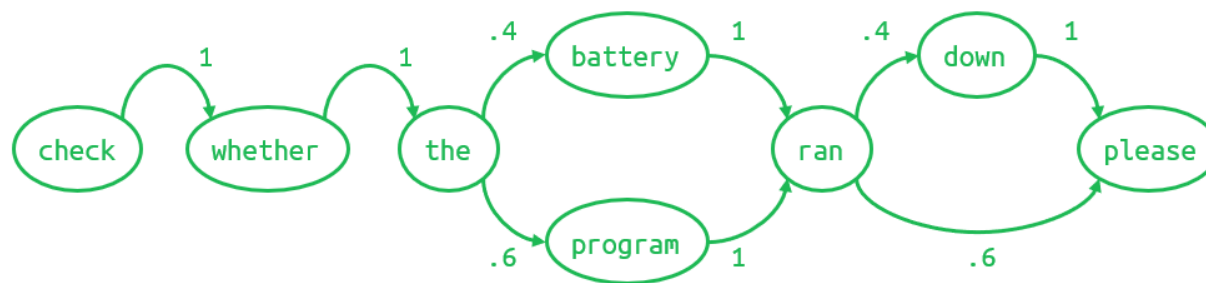
	directories	files	me	my	photos	please	show
directories	0	0	0	0	0	1	0
files	0	0	0	0	0	1	0
me	0	0	0	1	0	0	0
my	.2	.3	0	0	.5	0	0
photos	0	0	0	0	0	1	0
please	0	0	0	0	0	0	0
show	0	0	1	0	0	0	0

=

	directories	files	me	my	photos	please	show
directories	.2	.3	0	0	.5	0	0
files	.2	.3	0	0	.5	0	0
me	.2	.3	0	0	.5	0	0
my	.2	.3	0	0	.5	0	0
photos	.2	.3	0	0	.5	0	0
please	.2	.3	0	0	.5	0	0
show	.2	.3	0	0	.5	0	0

# → Second order sequence model

- First order Markov model only looks at the single most recent word.
- Predicting based on only one last word is hard! Let's consider the two most recent words!
- Example: (a 40/60 proportion)
  - Check whether the **battery** **ran** **down** please.
  - Check whether the **program** **ran** **please**.
- First order model:
- How can we remove the uncertainty after the word “ran”?





# Second order sequence model

- Check whether the **battery ran down** please.
- Check whether the **program ran** please.
- **Vocabulary**: {battery, check, down, please, program, ran, the, whether} **size = 8**

	battery	check	down	please	program	ran	the	whether
battery	0	0	0	0	0	1	0	0
check	0	0	0	0	0	0	0	1
down	0	0	0	1	0	0	0	0
please	0	0	0	0	0	0	0	0
program	0	0	0	0	0	1	0	0
ran	0	0	.4	.6	0	0	0	0
the	.4	0	0	0	.6	0	0	0
whether	0	0	0	0	0	0	1	0

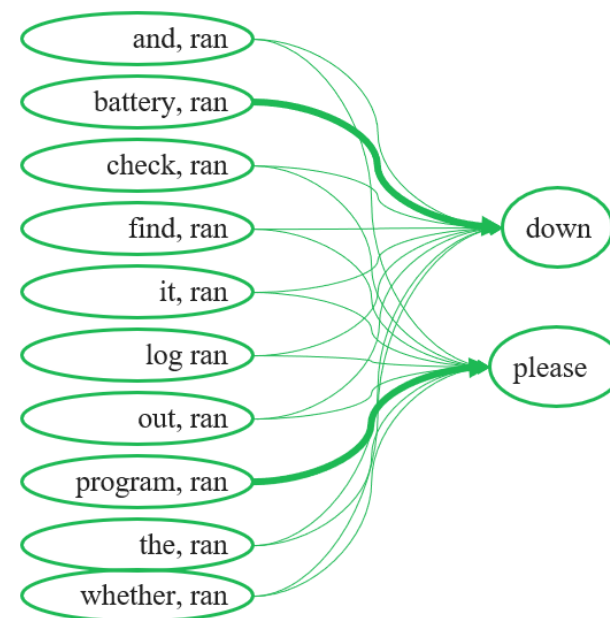
	battery	check	down	please	program	ran	the	whether
battery ran	0	0	1	0	0	0	0	0
check whether	0	0	0	0	0	0	1	0
program ran	0	0	0	1	0	0	0	0
the battery	0	0	0	0	0	1	0	0
the program	0	0	0	0	0	1	0	0
ran down	0	0	0	1	0	0	0	0
whether the	.4	0	0	0	.6	0	0	0
.	0	0	0	0	0	0	0	0
.	0	0	0	0	0	0	0	0



# Higher order sequence models?

- Check the **program** log and find out whether it **ran** please.
- Check the **battery** log and find out whether it **ran** down please.
- What comes after the word “**ran**”? It is unreasonable to investigate 9<sup>th</sup> order sequence model! (Vocab Size<sup>9</sup>) combinations!
- Solution: Second order sequence model **with skips**

	and	battery	check	down	find	it	log	out	please	program	ran	the	whether
and, ran			.5					.5					
battery, ran			1					0					
check, ran			.5					.5					
down, ran			0										
find, ran			.5					.5					
it, ran			.5					.5					
log, ran			.5					.5					
out, ran			.5					.5					
please, ran													
program, ran			0					1					
ran, ran													
the, ran			.5					.5					
whether, ran			.5					.5					





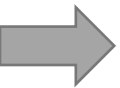


# Masking

- **Masking**: Crossing out all the uninformative feature votes
- The only important rows are *battery, ran* and *program, ran*. We could **mask** everything else!

Check the *program* log and find out whether it *ran* please.

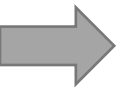
	and, ran	<b>battery, ran</b>	check, ran	down, ran	find, ran	it, ran	log, ran	out, ran	please, ran	<b>program, ran</b>	ran, ran	the, ran	whether, ran
feature activities	1	0	1	0	1	1	1	1	0	1	0	1	1
	*												
mask	0	1	0	0	0	0	0	0	0	1	0	0	0
	=												
masked feature activities	0	0	0	0	0	0	0	0	0	1	0	0	0



# Masking: selective second order model with skips

- The mask has the effect of **hiding** a lot of the transition matrix
- In this example, it hides the combination of **ran** with everything except **battery** and **program**, leaving **just the features that matter**
- This process of selective masking is the **attention** thing!

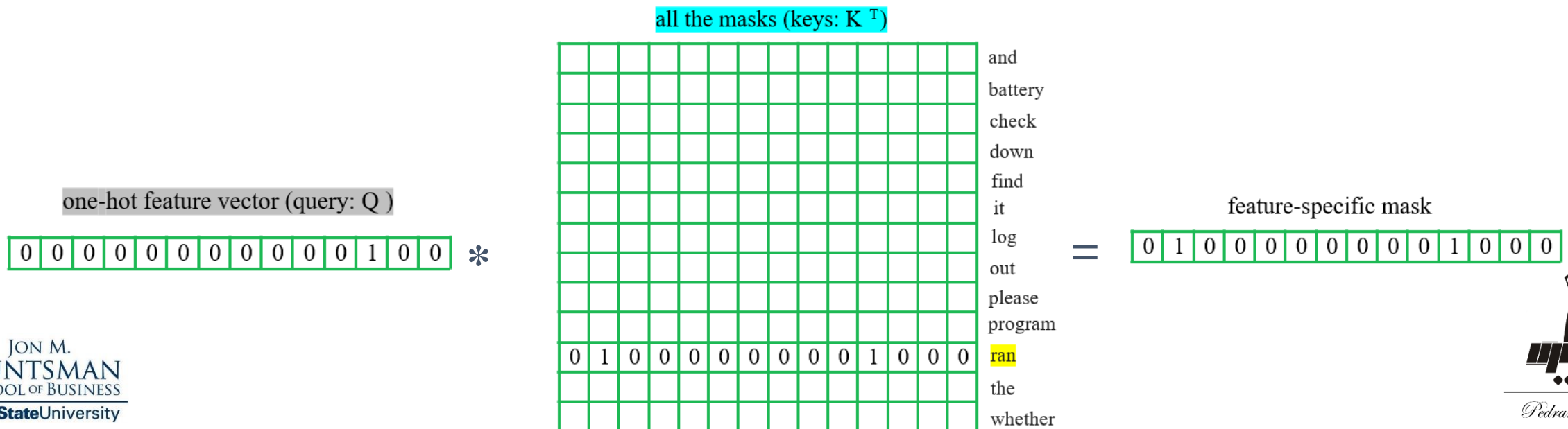
	and	battery	check	down	find	it	log	out	please	program	ran	the	whether
and, ran													
battery, ran			1					0					
check, ran													
down, ran													
find, ran													
it, ran													
log, ran													
out, ran													
please, ran													
program, ran			0					1					
ran, ran													
the, ran													
whether, ran													



# Attention as Matrix Multiplication

- Stack the mask vectors for every word into a matrix (**Keys**)
- Use one-hot representation of the most recent word (**Query**) to pull out the relevant mask (**attention score**)
- Wight the tokens in the input sequence (**Values**) by the attention scores to create the **context vector**.

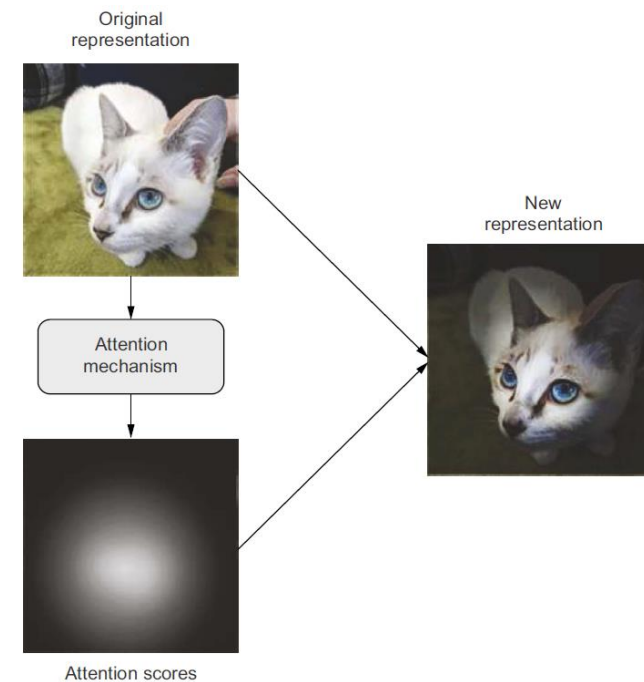
Check the program log and find out whether it *ran* please.

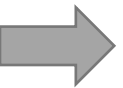




# Self-Attention

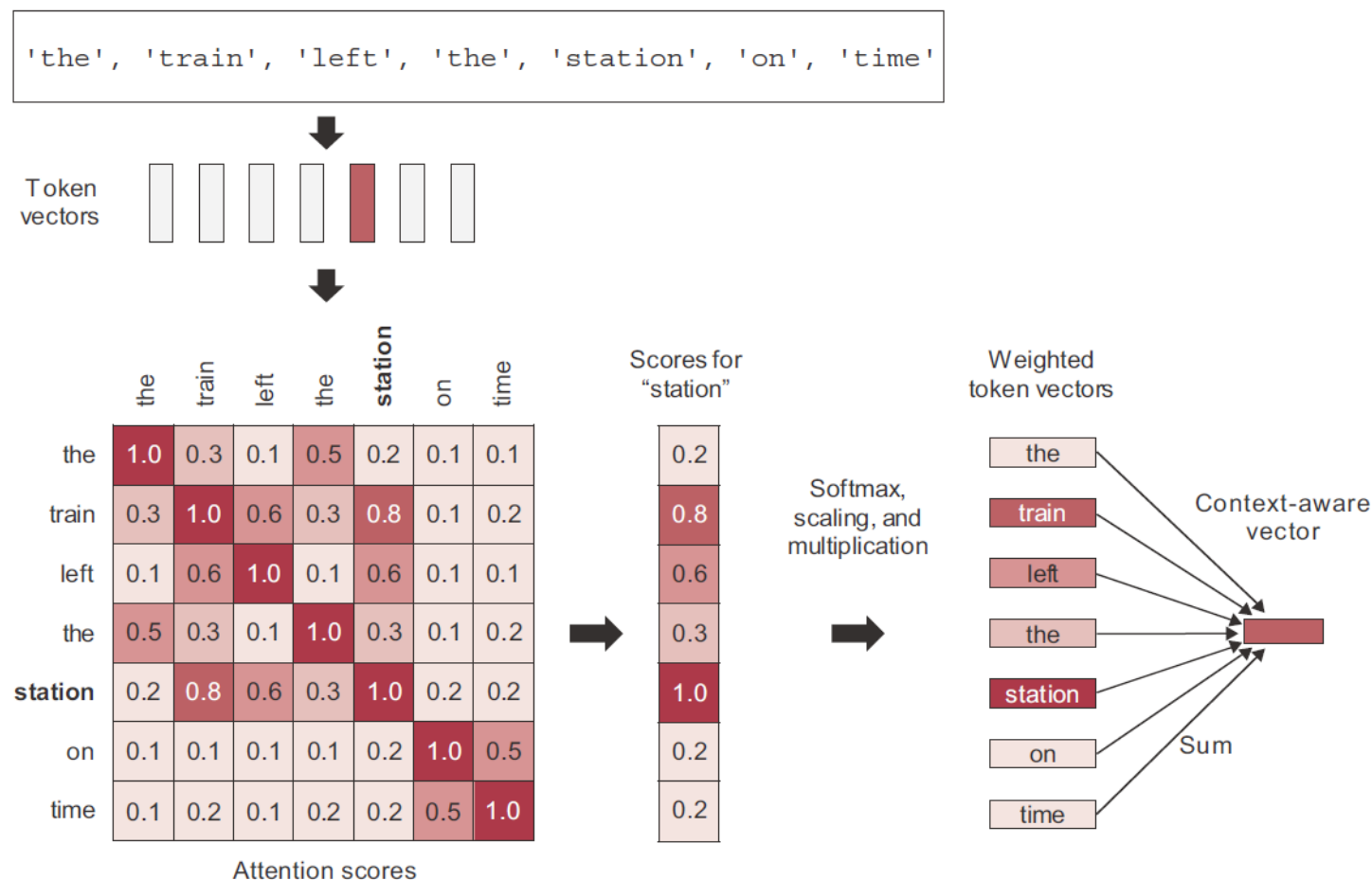
- **Self attention** is a variation of the attention mechanism where the **input and output sequences are the same**, meaning the model is attending to its own input.
- Self-Attention starts by computing **scores** for a set of features. High score → more relevant
- Self attention allows the model to **relate** different parts of the input sequence to each other, capturing dependencies and relationships within the sequence itself
- Have we seen this idea before? Max Pooling and TF-IDF





# Self-Attention (context-aware representation)

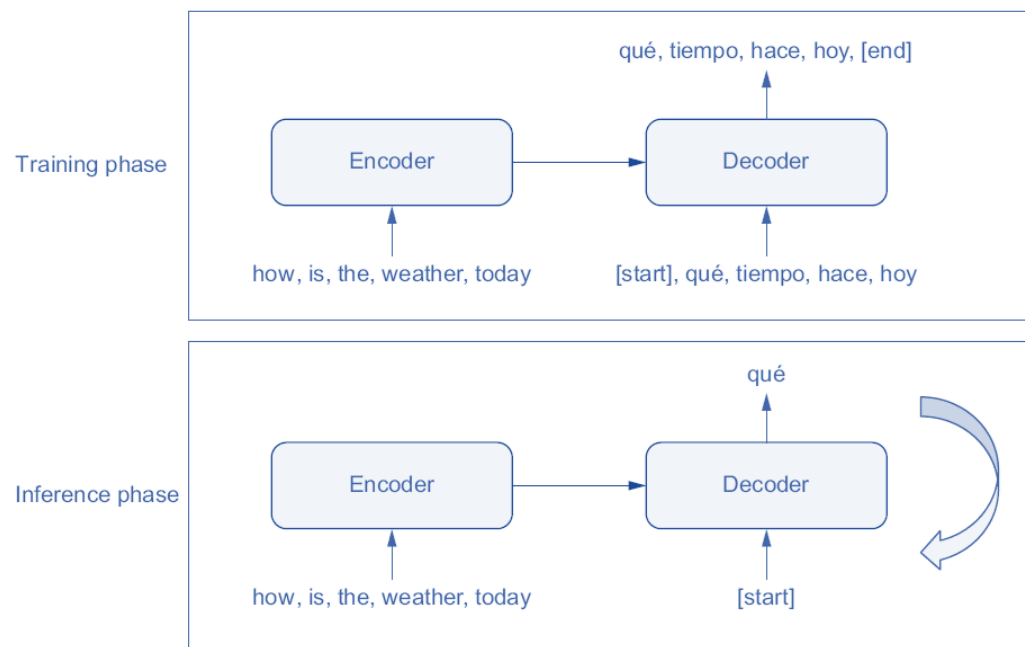
- Self-attention helps to **adjust the representation of a token** by considering the information from **related tokens** in the input sequence → **context awareness**

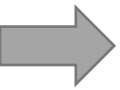




# Encoder–Decoder (Machine Translation)

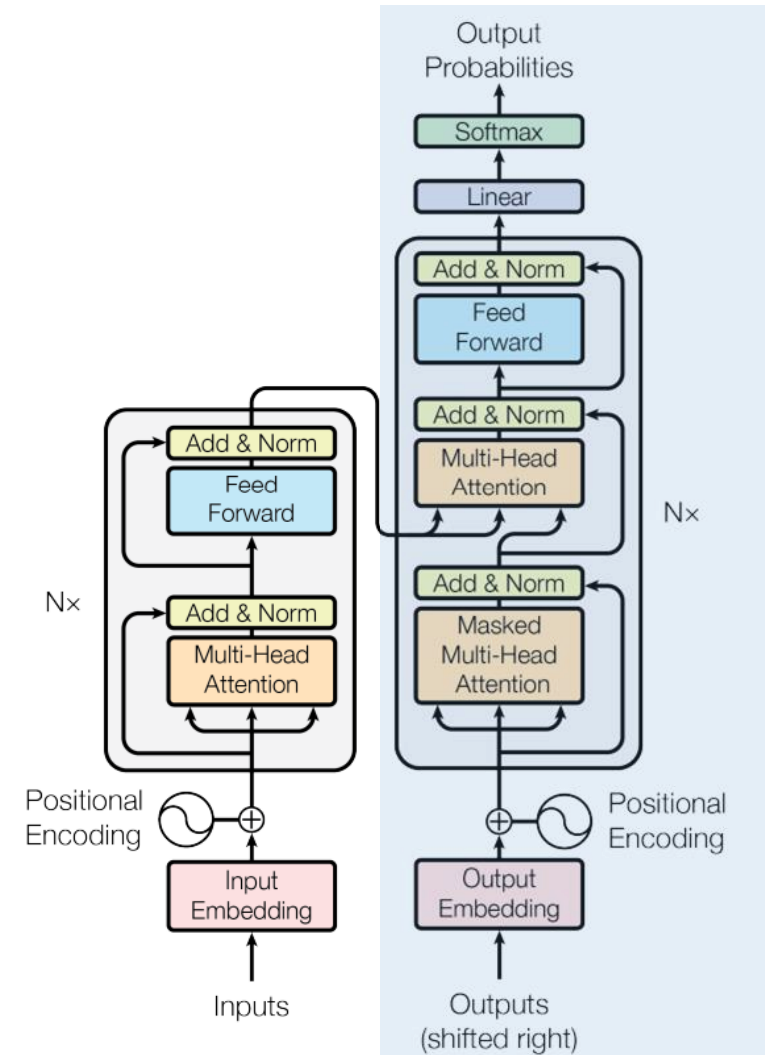
- An **encoder model** turns the source sequence into an **intermediate representation**.
- A **decoder model** is trained to predict the next token in the target sequence by looking at both previous tokens and the encoded source sequence.
- During **inference**, we don't have access to the target sequence (predict it from scratch) → must generate it one token at a time





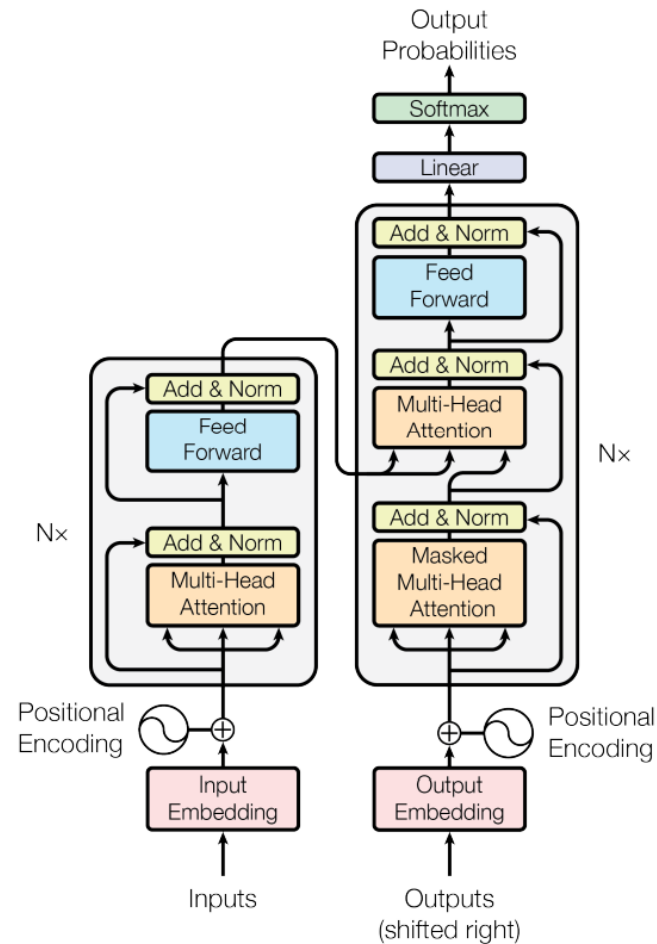
# Self-Attention and Transformers

- Transformers **primarily** use **self-attention** mechanisms.
- Both the **encoder** and **decoder** layers apply self-attention to capture relationships and dependencies within the input sequence itself.
- By using self-attention, transformers can:
  - Process inputs in parallel
  - Identify long-range dependencies
  - Model complex relationships more efficiently compared to RNN and CNN.



# Module 7 – Part 2

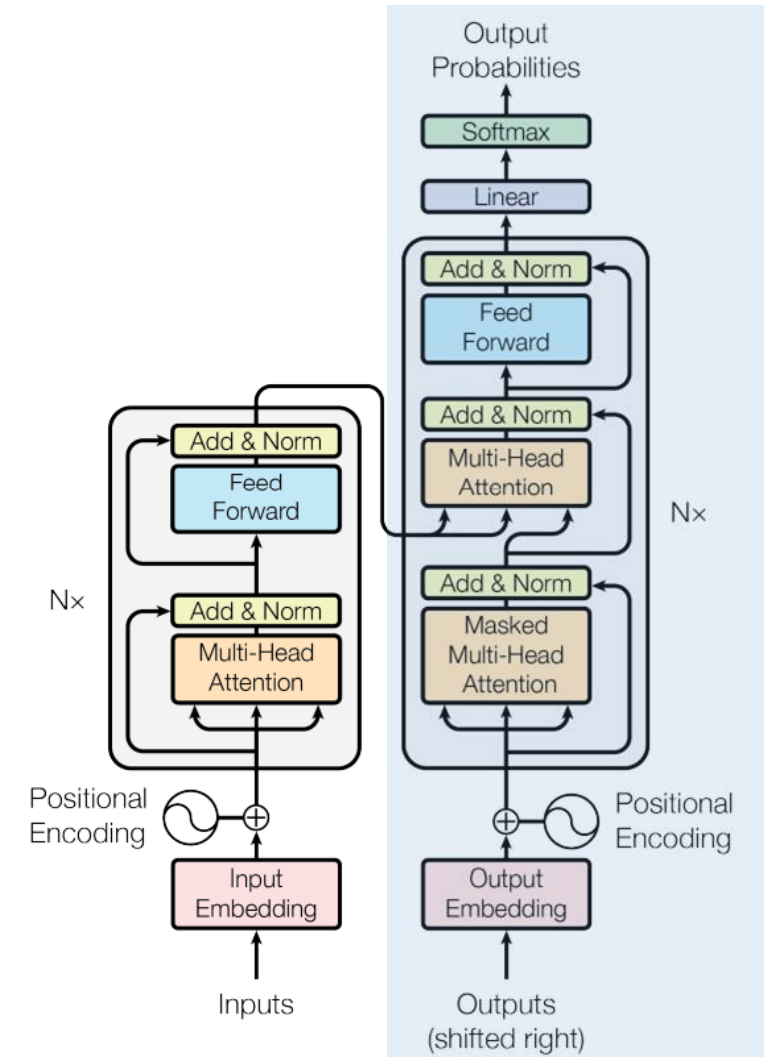
## Transformers Architecture





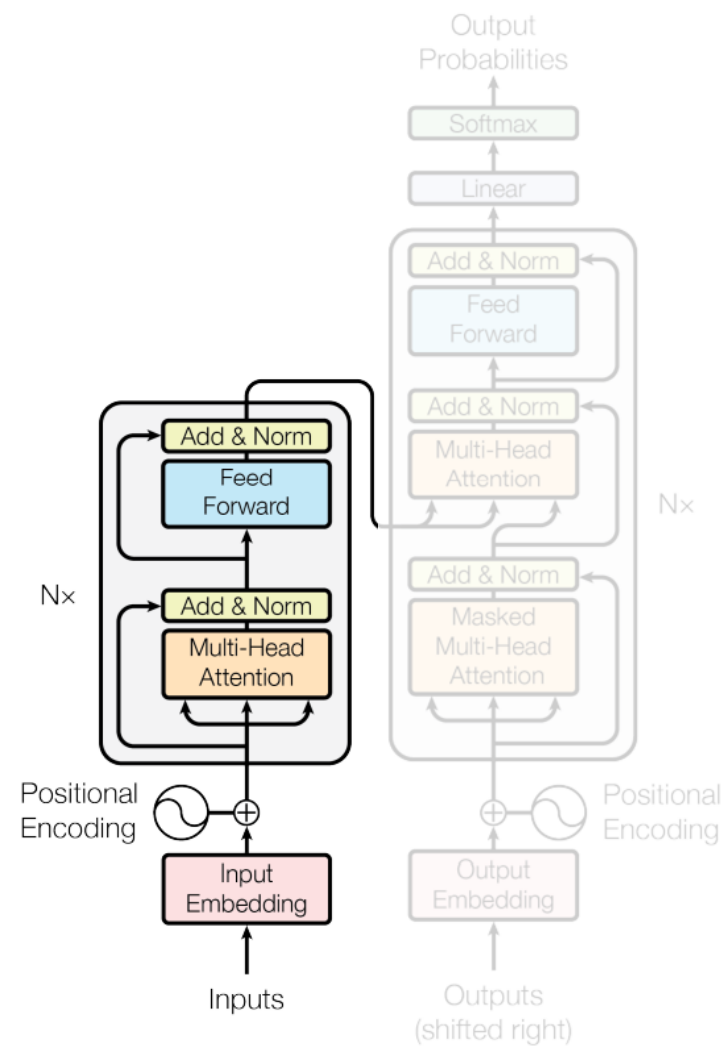
# Transformers outline

- Encoder – Decoder architecture
  - Embedding and Positional Encoding
  - Self-Attention (scaled dot product attention)
  - Query-Key-Value model
  - (Masked) Multi-head attention
  - Encoder-Decoder attention
  - Residual connections
  - Layer normalization
  - Feed Forward
  - Softmax layer



# Transformer architecture

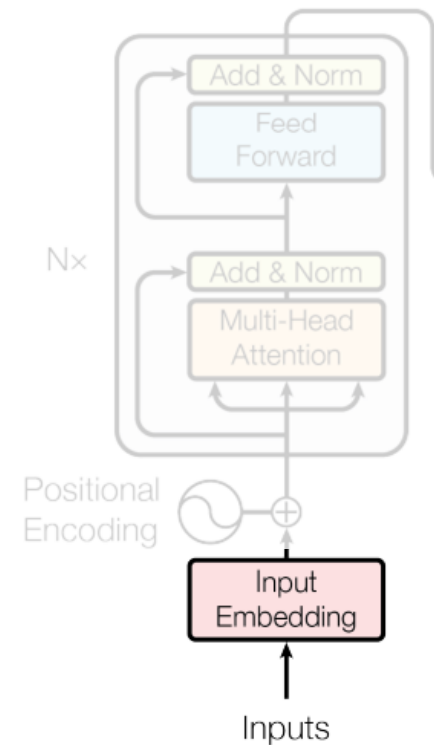
## Encoder





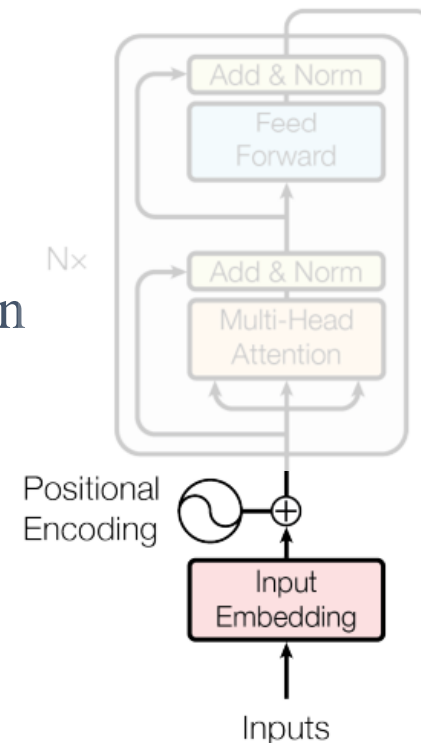
# Input Embedding

- **Word embedding** is a numerical representation of words as vectors in a continuous, relatively low-dimensional space compared to the size of the vocabulary, which captures their semantic meaning and **relationships with other words**.
- Word embedding is **learned** through a neural network
- Notion of order is **lost**!
- Example: “ I love Transformers!” with embedding dim=10
  - "I" : [-0.2, 0.1, 0.3, -0.4, 0.5, -0.6, -0.1, -0.3, -0.2, 0.4]
  - "love" : [0.3, -0.2, 0.1, 0.5, -0.4, 0.2, -0.6, -0.1, -0.3, 0.2]
  - "transformers" : [-0.4, -0.3, 0.6, -0.1, 0.2, 0.1, 0.4, 0.5, 0.2, 0.3]
  - "!" : [0.1, -0.4, -0.2, 0.3, 0.2, -0.1, 0.5, -0.3, -0.5, -0.4]



# → Positional Encoding

- Positional encoding **reinject** order information
- positional encoding is a set of small constants, which are added to the word embedding vector before the first self-attention layer.
- If the same word appears in a different position, the actual representation will be slightly different, depending on where it appears in the input sentence
- The model **will figure out** how to leverage this additional information.
- Naïve solution: My name is Pedram → {0,1,2,3}
- Better solution: Add a circular wiggle



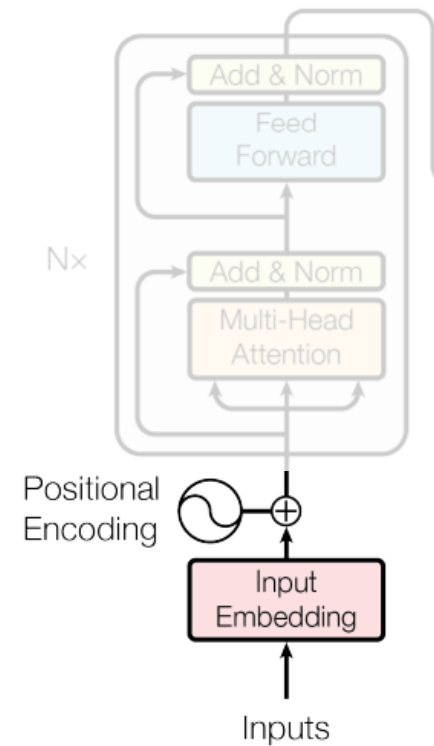
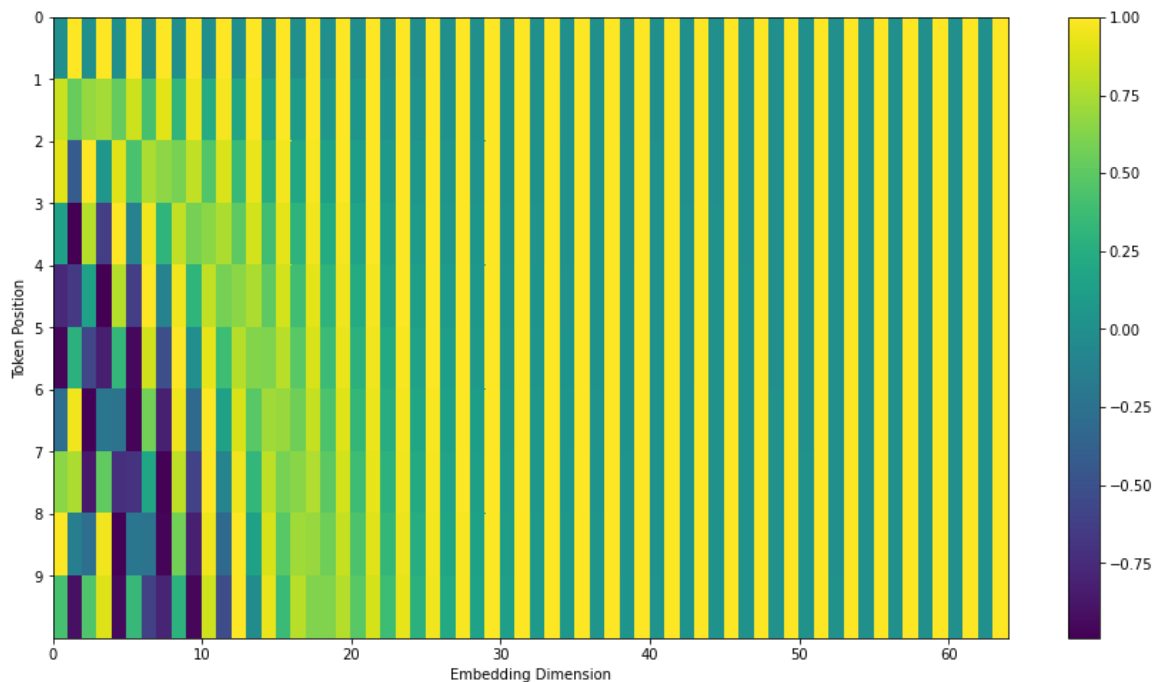
Word embedding + positional encoding =  
positional embedding

# → Positional Encoding

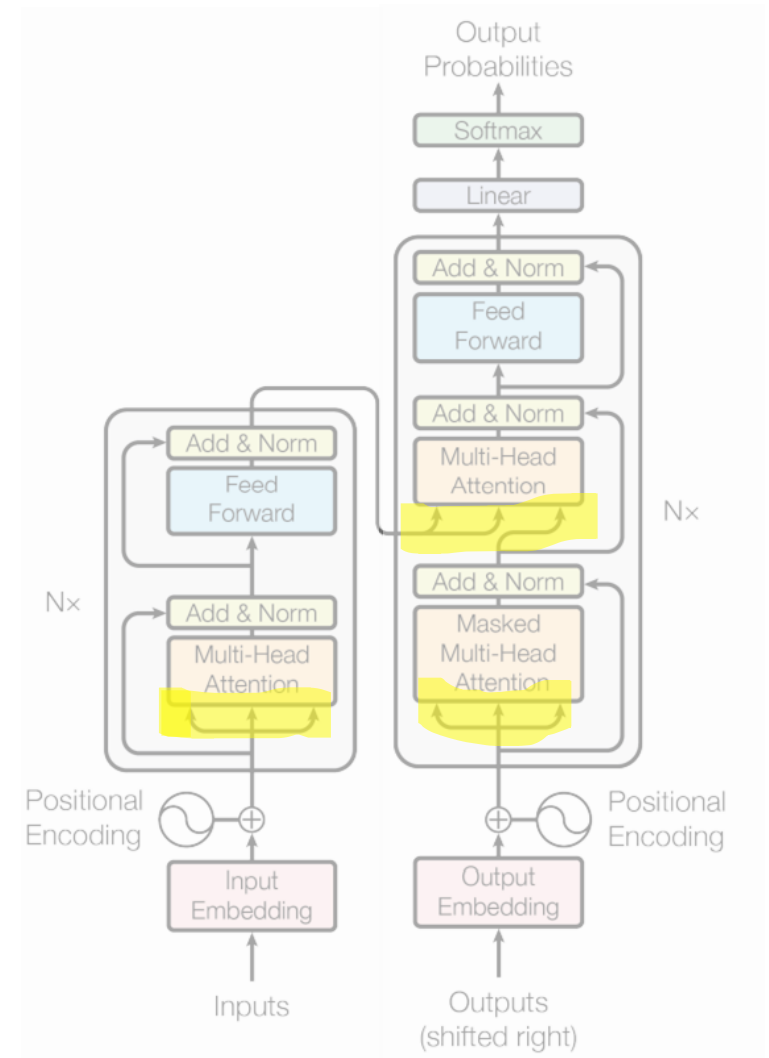
- Better solution: Sine – Cosine function

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

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




# Self-Attention mechanism Query-Key-Value



# ➔ The Query-Key-Value

- This terminology comes from **search engines**
- **Query**: What we are looking for?
- **Value**: Body of knowledge that we are trying to extract information from (database)
- **Key**: Set of “keywords” that describes the value in a format that can be readily compared to a query.
- The “query” is **compared** to a set of “keys,” and the match scores are used to **rank** “values” (images in this example).

Query

 “dogs on the beach”

Keys

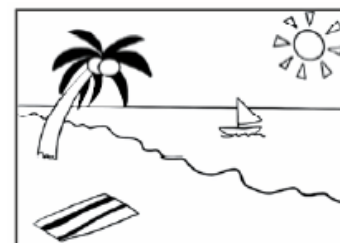
Values

match: 0.5

Beach

Tree

Boat

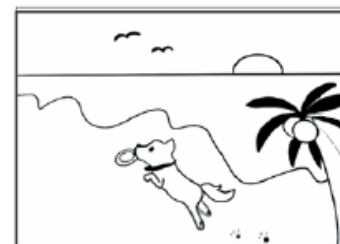


match: 1.0

Beach

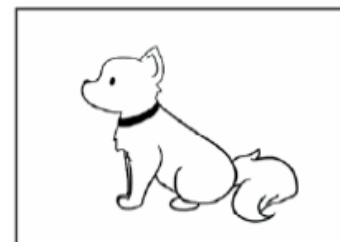
Dog

Tree



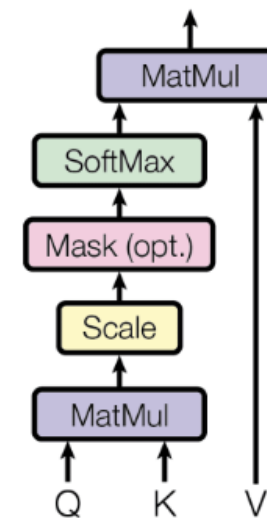
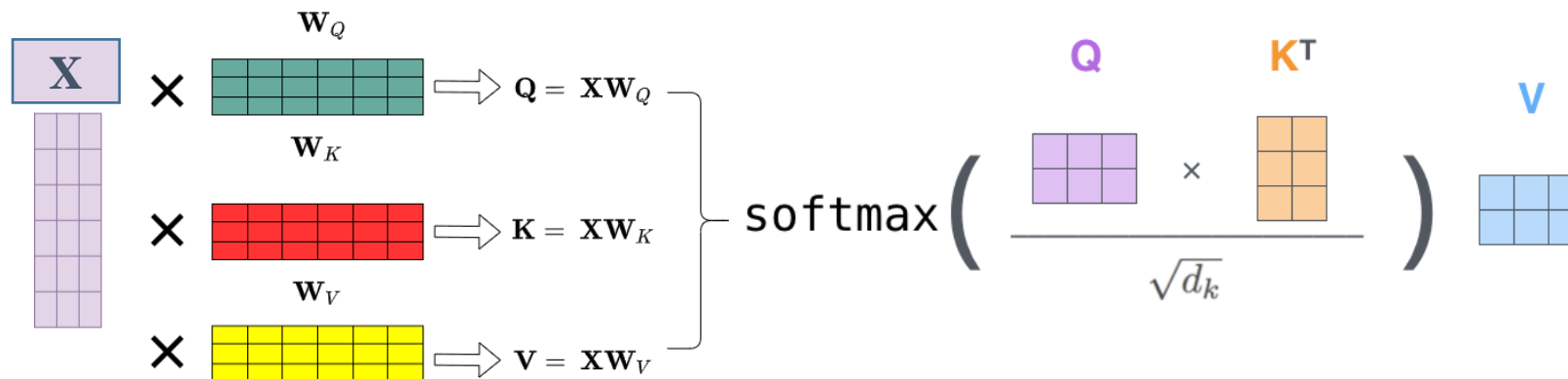
match: 0.5

Dog



# → Scaled dot product attention

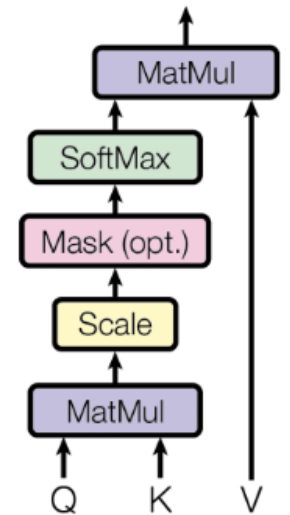
- **Goal:** Identify which part to attend to and Extracting features with high attention
- Scaled dot product attention utilizes three weight matrices, referred to as  $W_Q$ ,  $W_K$ , and  $W_V$  which are **learned** as model parameters during training.
- These matrices serve to project the inputs into query, key, and value components of the sequence.





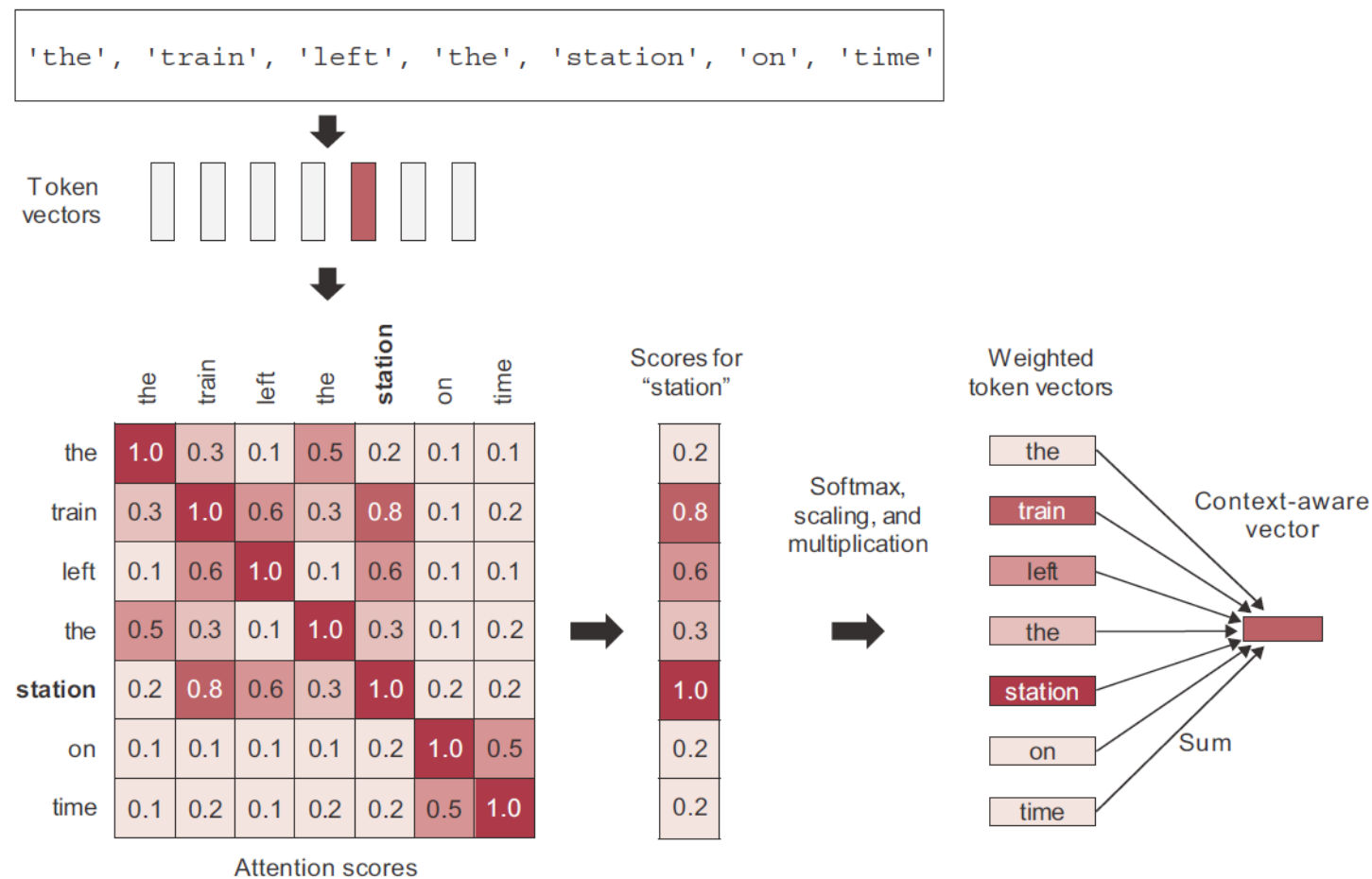
# Self-Attention in Language Models

1. The input embeddings are transformed into **query**, **key**, and **value** vectors using different **learned** weight matrices. Each of these vectors is used for different purposes:
  - **Query (Q)**: Represents the current token that the model is focusing on.
  - **Key (K)**: Represents all the tokens in the input sequence, which are compared to the query to compute **attention scores**.
  - **Value (V)**: Represents all the tokens in the input sequence, which are weighted by the attention scores to create the **context vector**.
2. **Attention scores** =  $\text{softmax}\left(\frac{Q \cdot K^T}{\sqrt{d_k}}\right)$
3. The attention scores are then used to weight the value vectors, and the weighted sum of these value vectors forms the **context vector** for the current word.
4. This context vector is then used to generate the output.



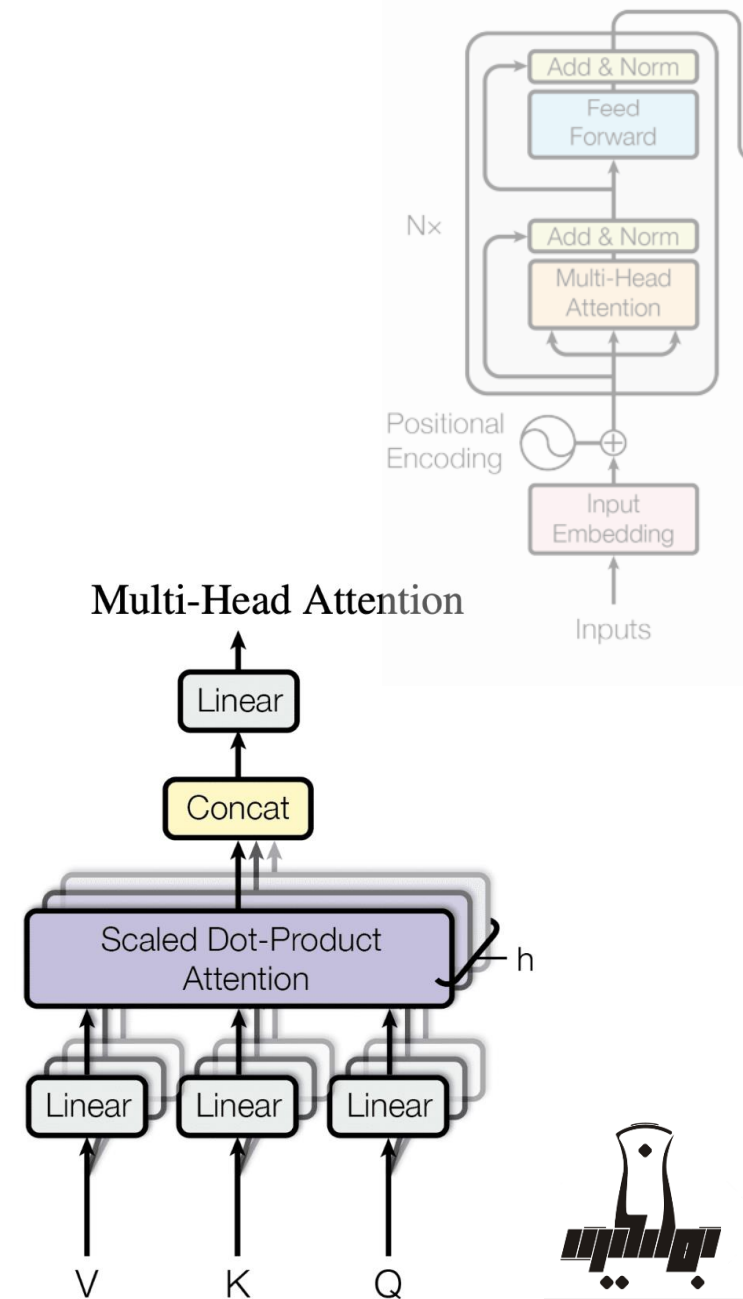


# Self-Attention in Language Models



# Multi-Head Attention

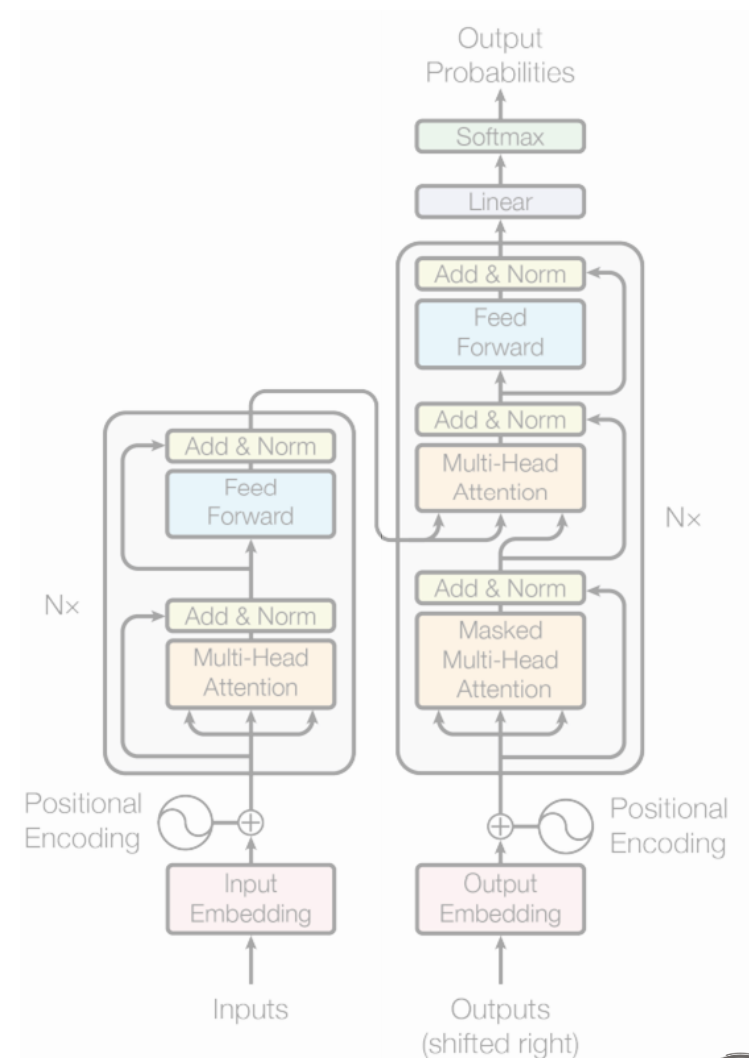
- **Intuition**: it allows the model to **attend to different parts** of the sequence **differently** each time
- Features **within** a self-attention head are **correlated** but mostly independent from those in other heads, similar to Depthwise separable convolutions treating each channel independently to **obtain more expressive representations**.
- Multi-head attention is simply the application of the same idea to self-attention.
- Independent heads help the layer learn different groups of features for each token.





# Add & Norm

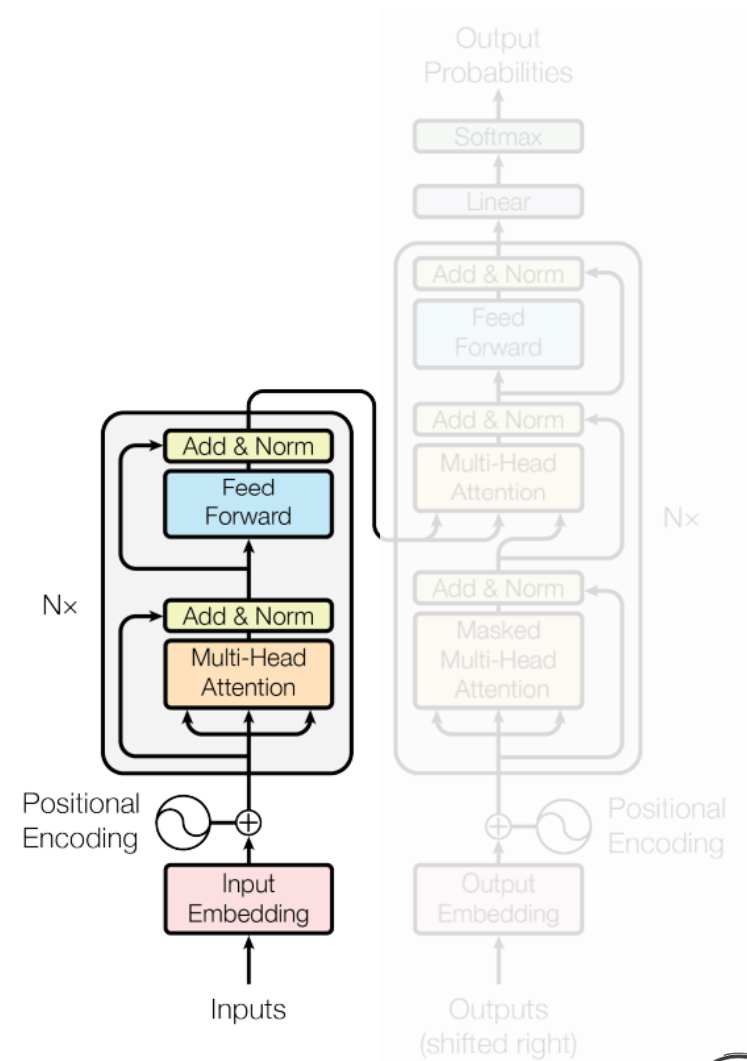
- Residual connections are added to preserve valuable information and prevent information loss during the training process.
- Normalization layers aid gradient flow during backpropagation to improve the training process.
  - Layer Normalization instead of Batch Normalization
  - Normalizing each sequence independently
- Leveraging **standard architectural** patterns such as factoring outputs into multiple independent spaces, adding residual connections, and incorporating normalization layers can enhance the performance of complex models.





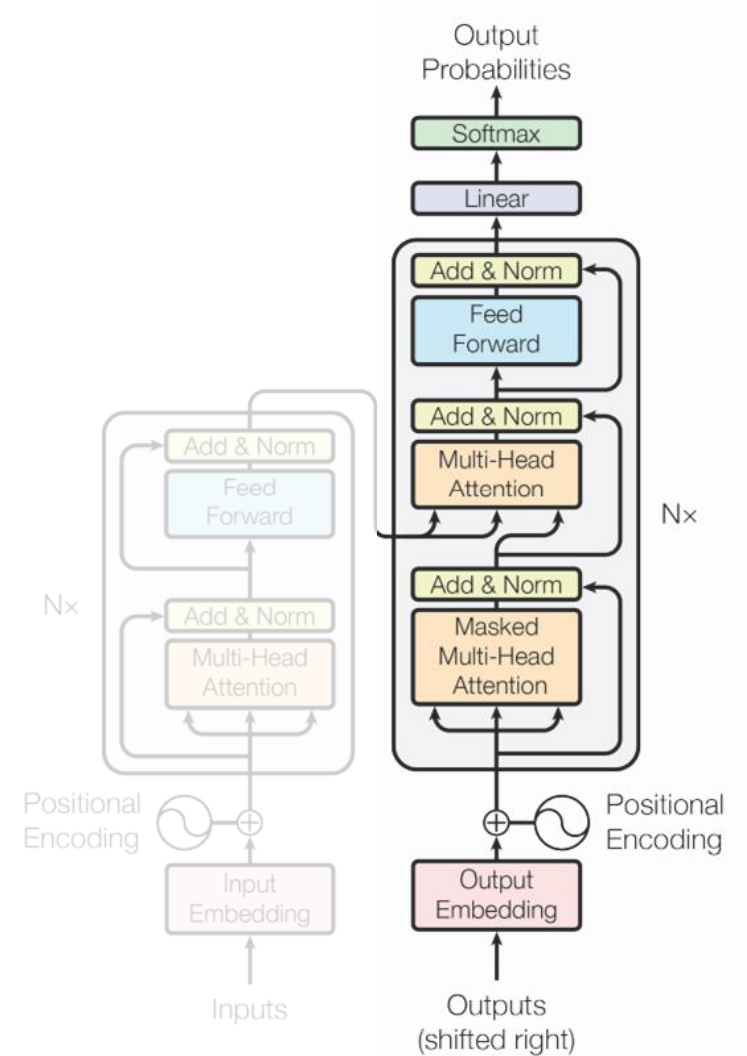
# Encoder summary

- Input embedding
- Positional encoding
- Encoder block:
  - Multi-head self-attention layer
  - Layer normalization
  - Residual connection
  - MLP (2 linear layers + RELU activation)
  - Second Layer normalization
  - Second residual connection
- Replicating N times (N=6 in the original paper)



# Transformer architecture

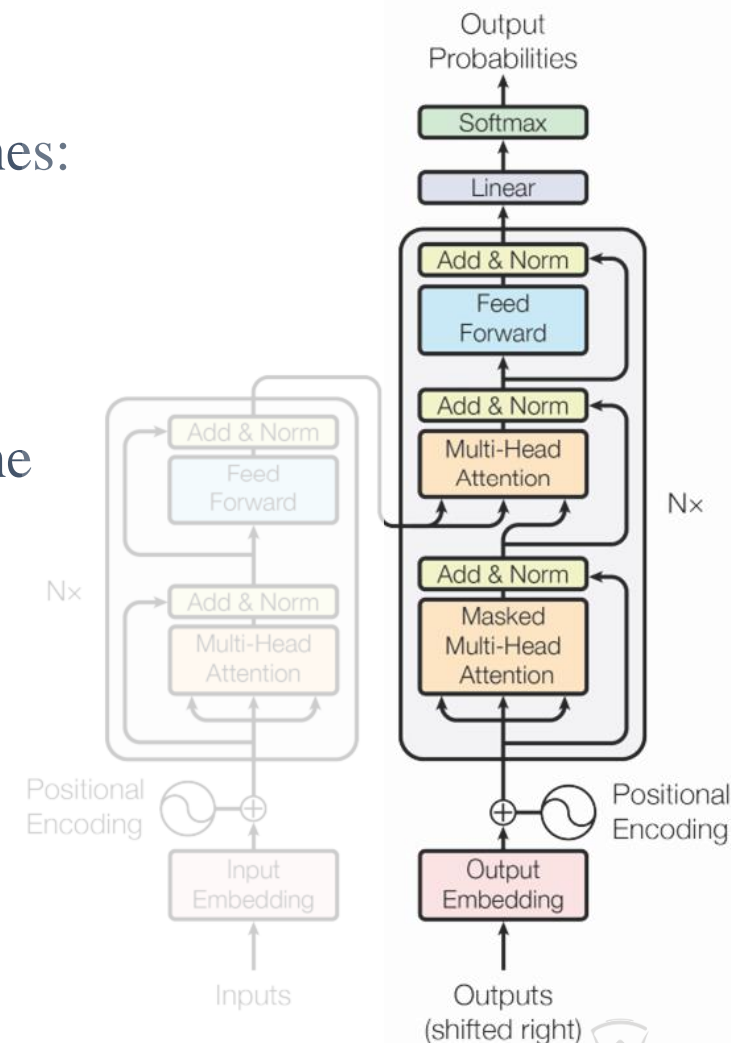
## Decoder

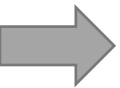




# Decoder

- Decoder consist all the components of encoder plus two novel ones:
  - **Masked** multi-head self-attention layer
  - **Encoder-Decoder** Attention layer
- Final decoder block output is passed through a **linear layer** and the probabilities are calculated with a standard **softmax** function
- Decoder is **autoregressive**:
  - Generate output one at a time
  - This is repeated until the **<end>** token is seen.





# Masked multi-head self-attention layer

- First Multi-headed attention computes the attention scores for the decoders input.
- For this Multi-headed attention layer, we need to apply **masking** to avoid cheating.

$$\text{MaskedAttention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax} \left( \frac{\mathbf{QK}^T + \mathbf{M}}{\sqrt{d_k}} \right) \mathbf{V}$$

0.7	0.1	0.1	0.1
0.1	0.6	0.2	0.1
0.1	0.3	0.6	0.1
0.1	0.3	0.3	0.3

+

0	-inf	-inf	-inf
0	0	-inf	-inf
0	0	0	-inf
0	0	0	0

=

0.7	-inf	-inf	-inf
0.1	0.6	-inf	-inf
0.1	0.3	0.6	-inf
0.1	0.3	0.3	0.3

⇒

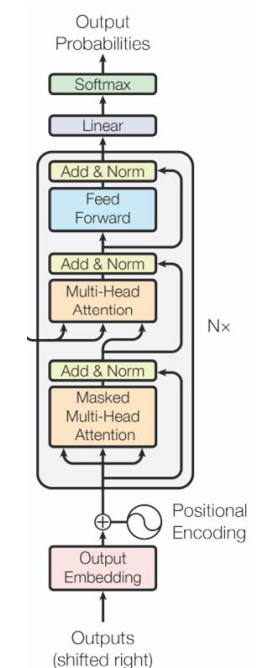
Softmax(

0.7	-inf	-inf	-inf
0.1	0.6	-inf	-inf
0.1	0.3	0.6	-inf
0.1	0.3	0.3	0.3

)

=

1	0	0	0
0.37	0.62	0	0
0.26	0.31	0.43	0
0.21	0.26	0.26	0.26

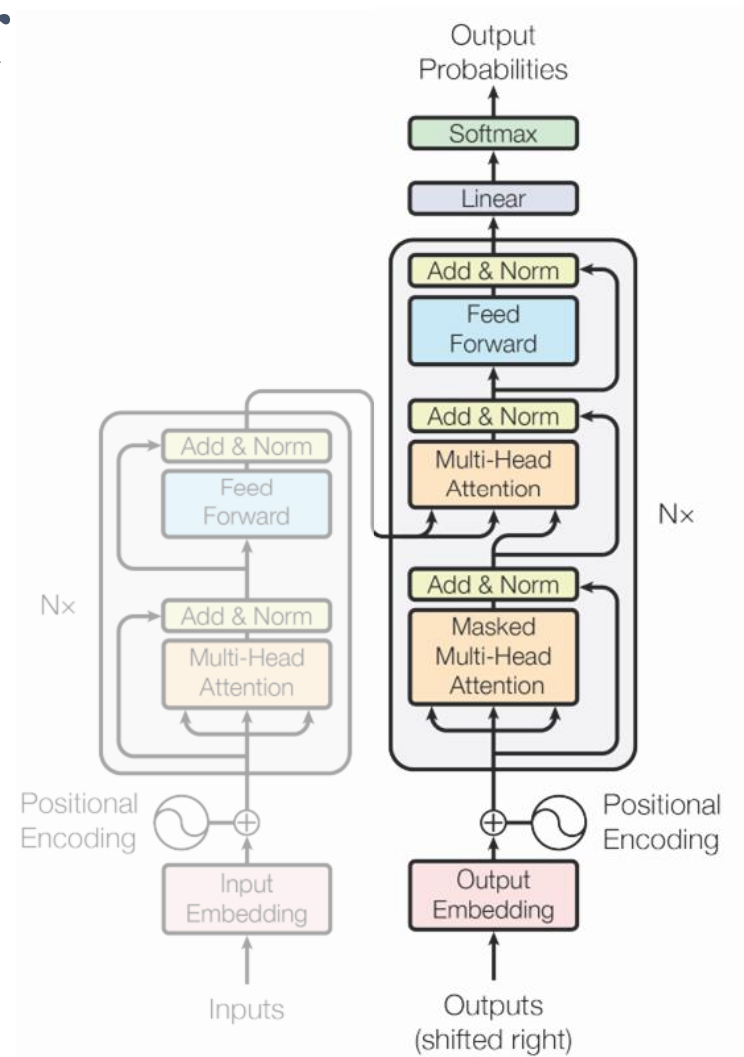






# Encoder-Decoder Attention Layer

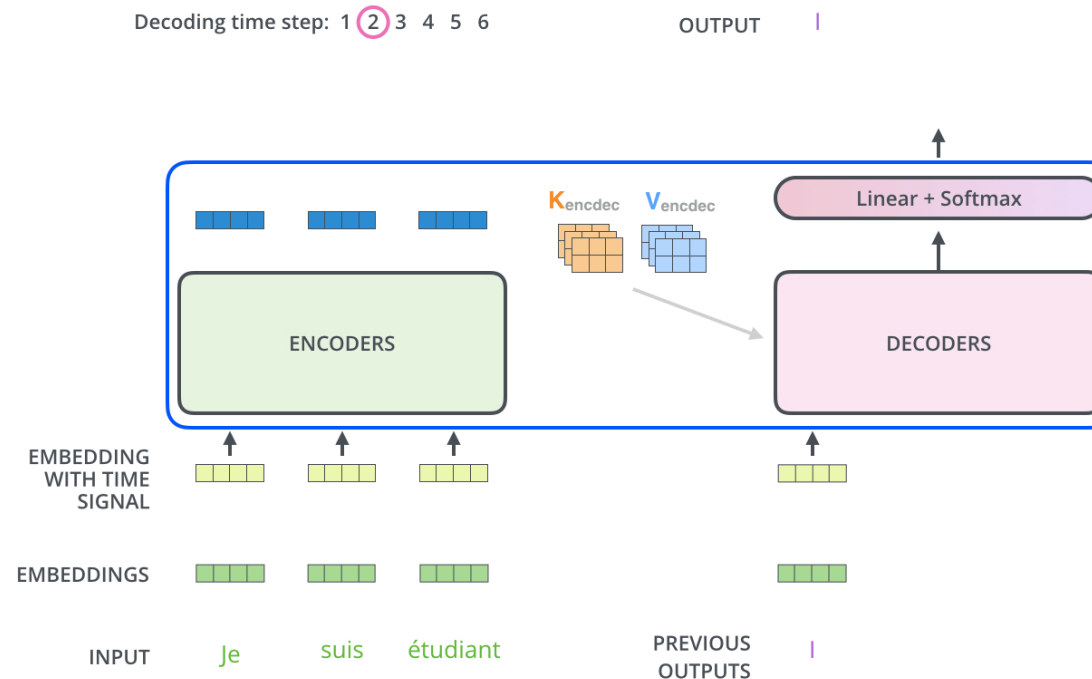
- Transformers also use a form of attention known as "encoder-decoder attention" in the **decoder** layers.
- This is **where the magic happens**, where the decoder processes the encoded representation (K,V)
- This attention mechanism allows the decoder to focus on specific parts of the **input sequence encoded by the encoder** when generating each output element.
- The encoder-decoder attention mechanism helps the model to **align the input and output sequences better**, which is particularly useful in tasks like machine translation.





# Encoder-Decoder Attention Layer (cross-attention)

- The encoder output is utilized to generate the **Key** and **Value** matrices.
- On the other hand, the Masked Multi-head attention block's output contains the newly generated sentence, represented as the **Query** matrix in the attention layer.
- This process matched the encoders input to the decoders input allowing the decoder to decide which encoder input is relevant to focus on.



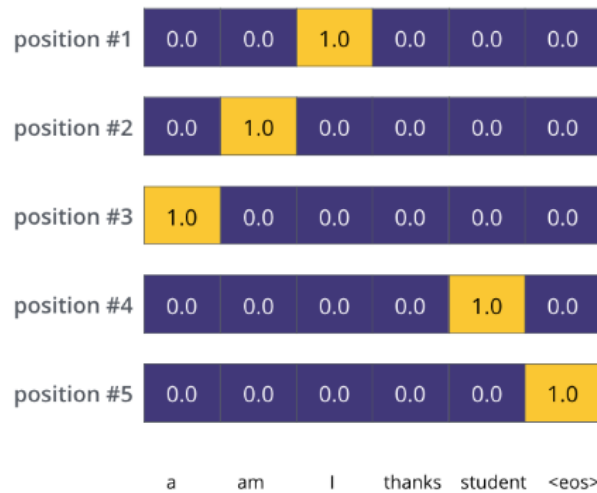


# Training

- Like any other deep learning model, training involves:
  - Feed forward, loss computation, backpropagation, parameter updates
- Cross-entropy compare two probability distributions.

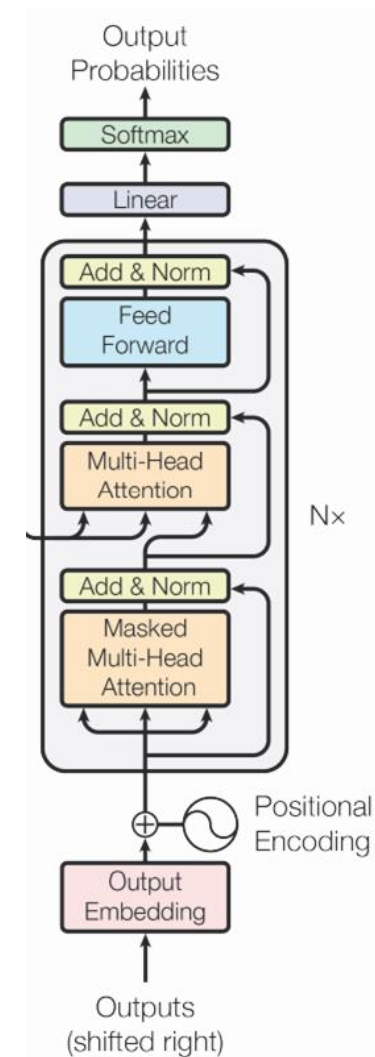
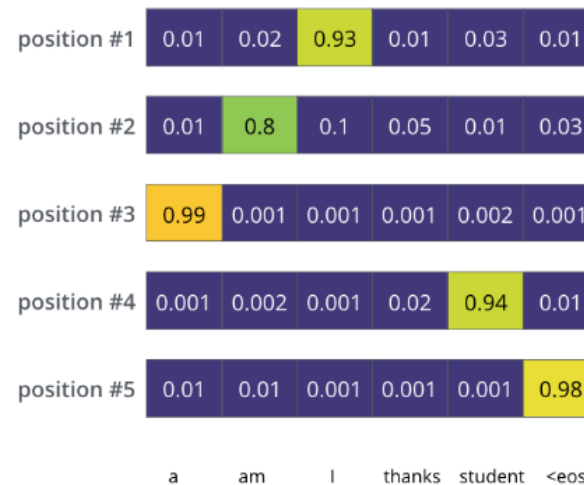
Target Model Outputs

Output Vocabulary: a am I thanks student <eos>



Trained Model Outputs

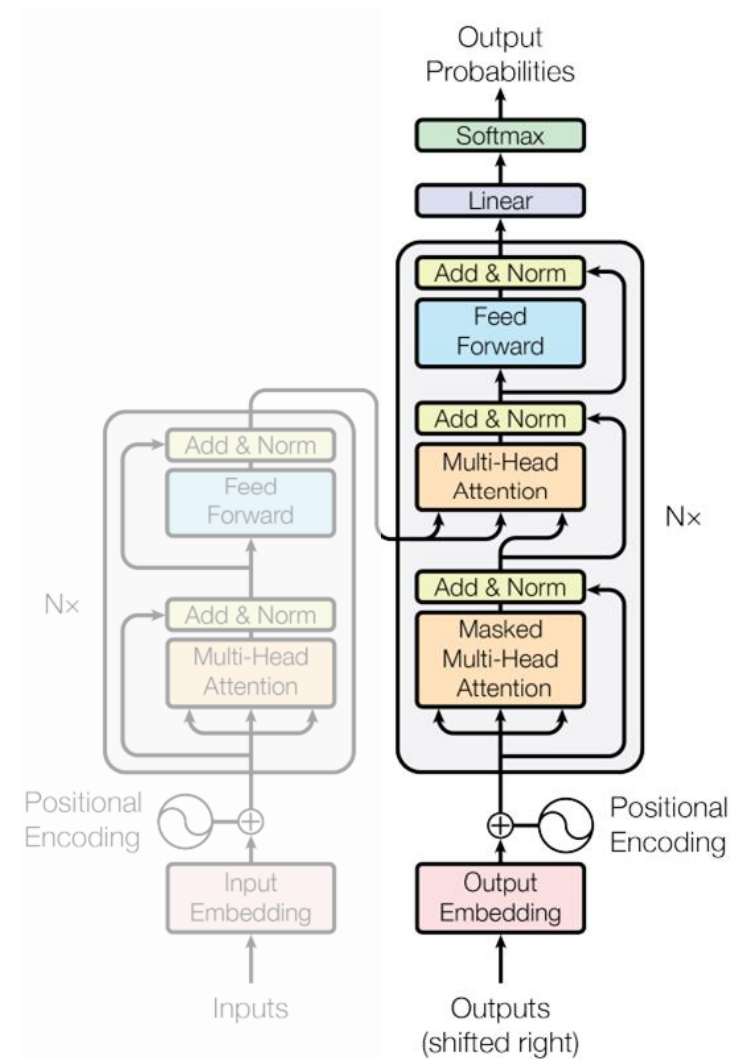
Output Vocabulary: a am I thanks student <eos>





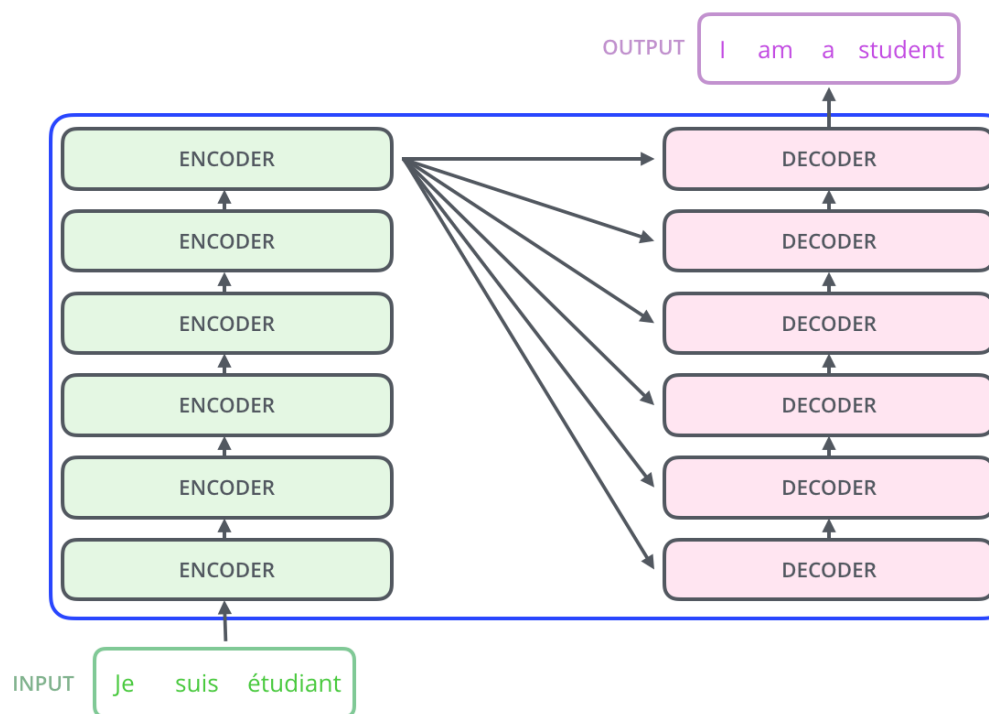
# Decoder summary

- Input embedding
- Positional encoding
- Decoder block:
  - Masked Multi-head self-attention layer
  - First Normalization and residual connection
  - Encoder-Decoder Multi-head attention
  - Second Normalization and residual connection
  - MLP (2 linear layers + RELU activation)
  - Third Normalization and residual connection
- Linear layer followed by a softmax function
- Replicating N times (N=6 in the original paper)



# ➔ Why transformers work?

- Multi-head attention (multiple representations of the same input)
- Context awareness through self-attention (data-dependent dynamic weights)
- Stacking multiple encoder and decoder blocks (transformer blocks are shape-invariant)





# References

---

- Attention is all you need! *Vaswani et al*
- Deep learning with Python, *Francois Chollet*
- How Transformers work in deep learning and NLP: an intuitive introduction, *Nikolas Adaloglou*
- Transformers from scratch, *Brandon Rohrer*
- The illustrated transformers, *Jay Alammar*

# ➔ Road map!

- ✓ Module 1- Introduction to Deep Forecasting
- ✓ Module 2- Setting up Deep Forecasting Environment
- ✓ Module 3- Exponential Smoothing
- ✓ Module 4- ARIMA models
- ✓ Module 5- Machine Learning for Time series Forecasting
- ✓ Module 6- Deep Neural Networks
- ✓ Module 7- Deep Sequence Modeling (RNN, LSTM)
- ✓ Module 8- Transformers (Attention is all you need!)
- Module 9- Prophet and Neural Prophet

