L08: Introduction to Transformers

FICT, UTAR

Table of Contents

1. **Introduction to Transformers**
2. Positional Encoding
3. The Encoder Network
   - Multi-Head Self-Attention
   - Layer Normalization
   - Feed-Forward
4. The Decoder Network
   - Masked Multi-Head Self-Attention
   - Multi-Head Cross-Attention
5. Types of Transformer Network

Limitations of RNNs and LSTMs

RNN and LSTM have difficulty modeling long-range dependencies. RNN suffer from vanishing / exploding gradient. LSTM helps to some extent but still struggle for very long sequences.

Transformer has better long-range dependency modeling. Processes whole sequence with self-attention.

In practice, LSTM and RNN focuses more on recent tokens, often forgetting earlier ones.

Transformer uses self-attention to learn relationships between tokens regardless of their distance.

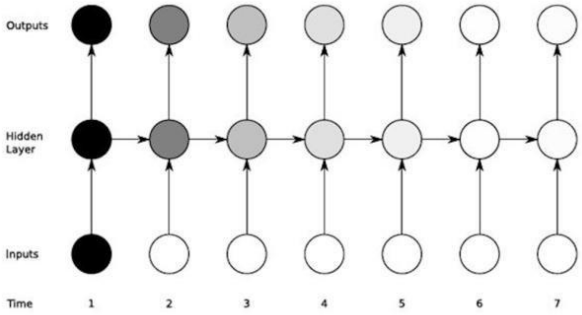
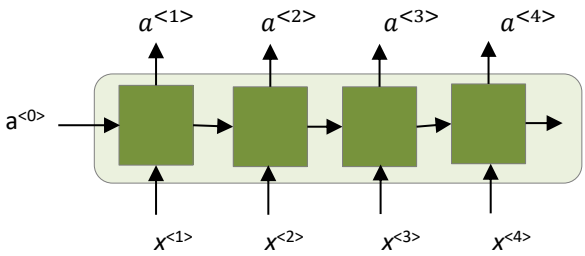


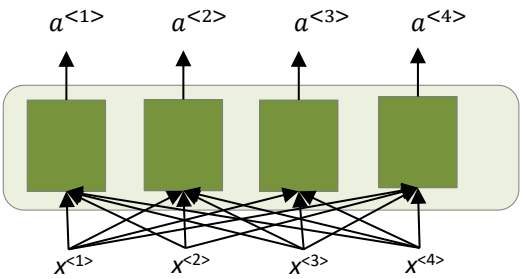
Figure shows an unrolled RNN. The darker the shade, the more the information remembers x at t=1. New inputs overwrite the activation of hidden unit and the network 'forgets' the first input after 7 time steps.

Limitations of RNNs and LSTMs

RNN and LSTM process tokens one by one → not parallelizable and slow

Transformer applies self-attention on the whole sequence → parallelizable and fast training. Easier to train with modern hardware (like GPUs and TPUs).

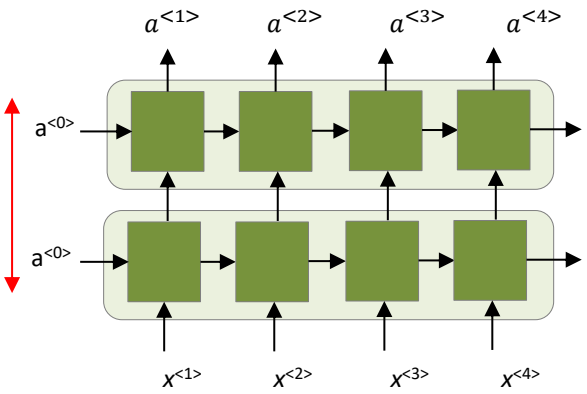




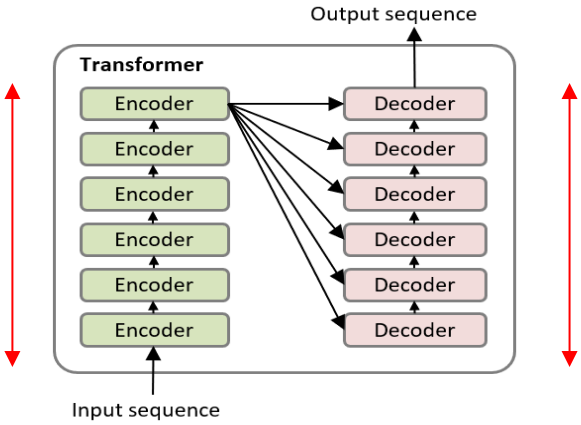
Limitations of RNNs and LSTMs

RNN and LSTM have shallow representation although their computational graph can be deep along the time dimension.

Transformer models are deep in representation. Simpler architecture (no recurrence). Easily adapted to large architectures with more layers and attention heads.

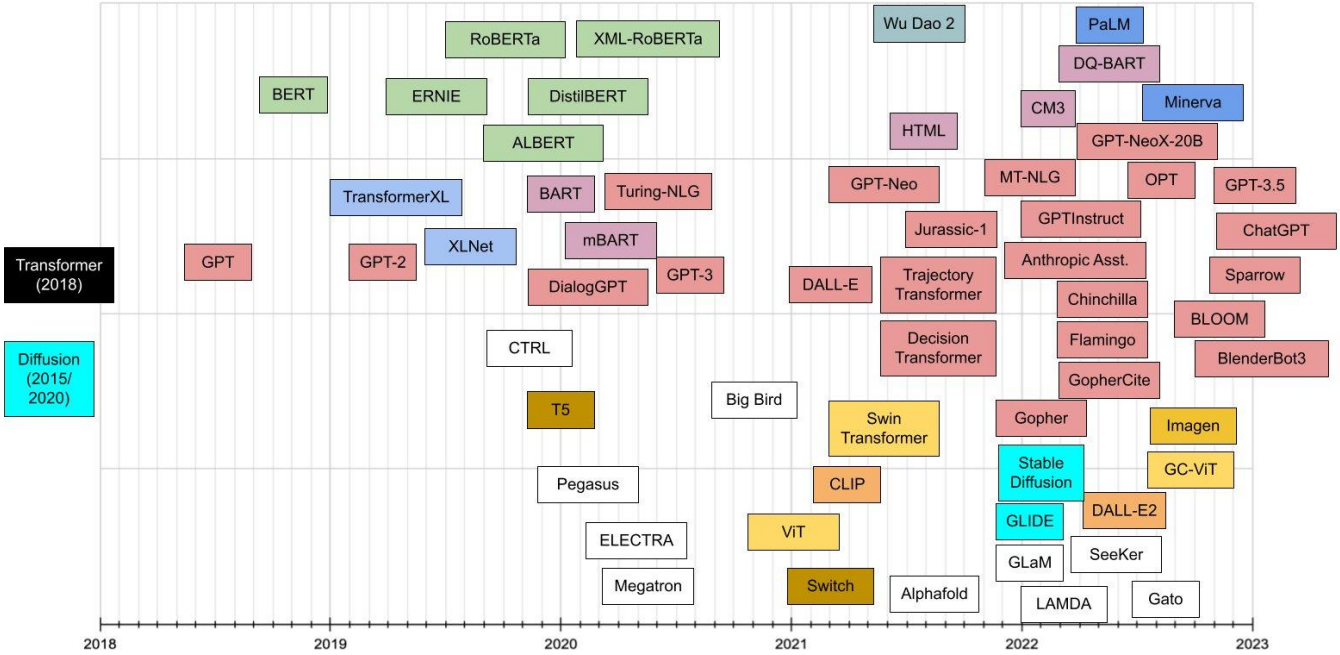


Typically 1-3 layers



GPT-3 Small : 12 layers (125M)
GPT-3 175B : 96 layers (175B)

Transformer Timeline

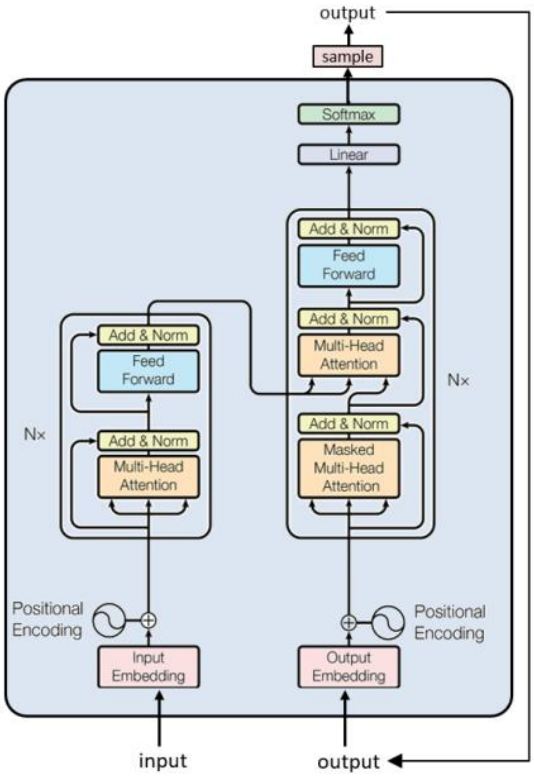


Different color describe different Transformer family

[Xavier Amatrian, et al., "Transformer models: an introduction and catalog" 2023]

Transformer

- A non-recursive sequence neural network by Google
- Follows an encoder-decoder structure
- No recurrence → easily parallelizable
- No convolution → less inductive bias
- Resilient architecture



Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, Ł. and Polosukhin, I., 2017. Attention is all you need. *Advances in neural information processing systems*, *30*.

Transformer Network

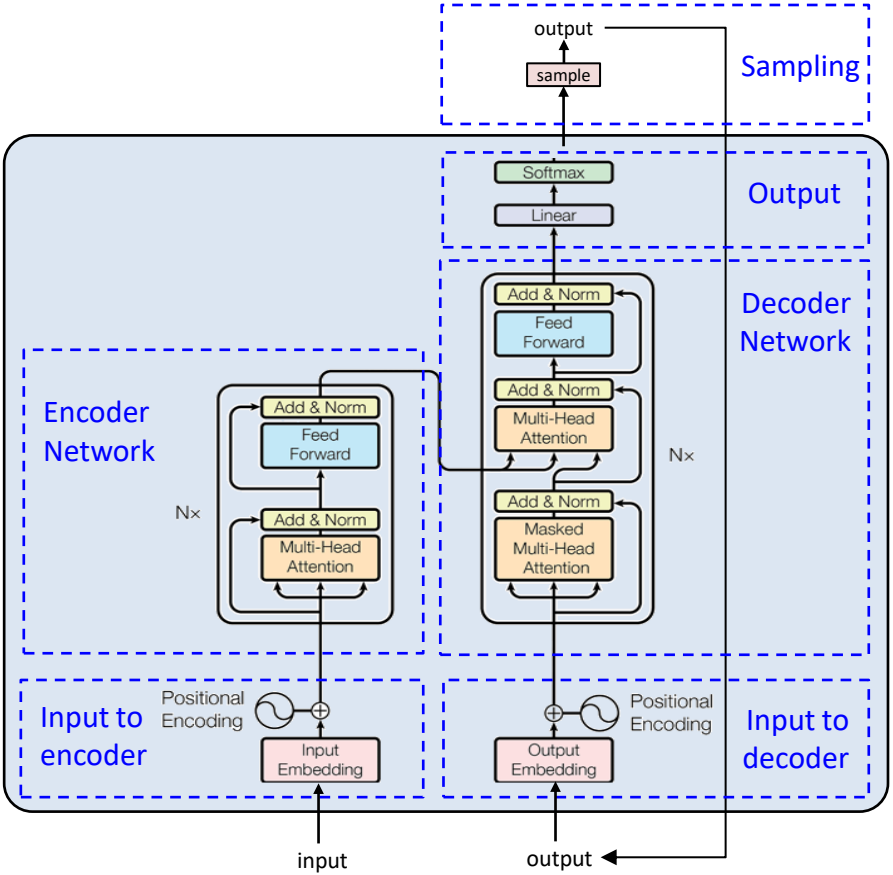
Sample to get different output each time

Output block generates class probability at each time step

Decoder generates the features for text generation based on the complete encoded input and current output. The decoder is auto-regressive - predicts based on past values

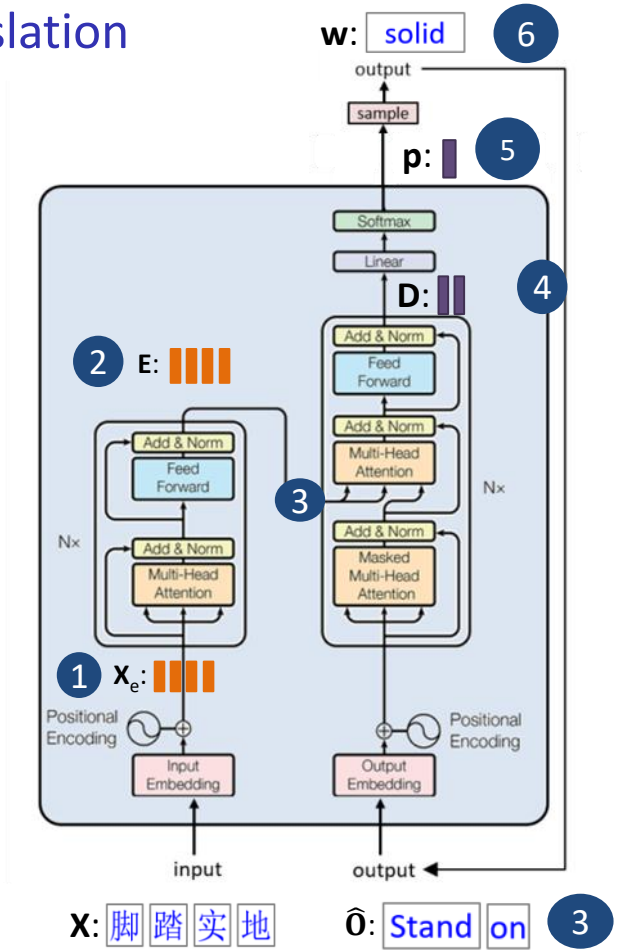
Encoder extracts high-level temporal features to represent the input

Input block performs input embedding and inserts positional information to generate the input feature

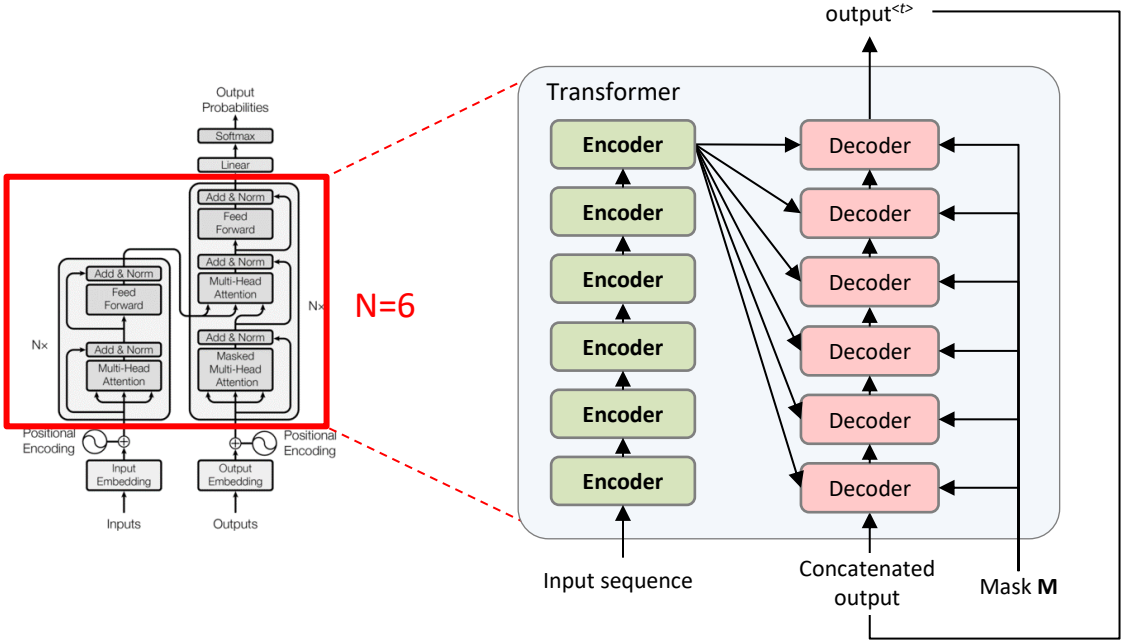


Example: Chinese to English Translation

1. Preprocessing: Infuse positional information to get $\mathbf{X}_e$.
2. Encoder: Extract encoder (temporal and high level) features **E**.
3. Input to decoder is the concatenated current output sequence $\hat{\mathbf{O}}$ (generated English) and the complete encoded input signals (Chinese sentence)
4. Decoder: Extract (temporal and high level) decoder features **D**.
5. Output: Computes the probability of the next token
6. Output: Sample the next word



The Transformer Model: Expanded view



- The encoder network stacks encoder layers to encode the input sequence
- The decoder network stacks decoder layers to generate the output sequence
- Default: `num_layers` = 6

PyTorch Implementation

Transformer Layers:

nn.Transformer	A transformer model.
nn.TransformerEncoder	TransformerEncoder is a stack of N encoder layers
nn.TransformerDecoder	TransformerDecoder is a stack of N decoder layers
nn.TransformerEncoderLayer	TransformerEncoderLayer is made up of self-attention and feedforward network.
nn.TransformerDecoderLayer	TransformerDecoderLayer is made up of self-attn, multi-head-attn and feedforward network.

PyTorch Implementation

nn.Transformer

TRANSFORMER

CLASS `torch.nn.Transformer(d_model=512, nhead=8, num_encoder_layers=6, num_decoder_layers=6, dim_feedforward=2048, dropout=0.1, activation=<function relu>, custom_encoder=None, custom_decoder=None, layer_norm_eps=1e-05, batch_first=False, norm_first=False, device=None, dtype=None)` [SOURCE]

A transformer model. User is able to modify the attributes as needed. The architecture is based on the paper "Attentio Is All You Need". Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Lukasz Kais and Illia Polosukhin. 2017. Attention is all you need. In Advances in Neural Information Processing Systems, pages 6000-6010. Users can build the BERT(https://arxiv.org/abs/1810.04805) model with corresponding parameters.

Parameters

- **d_model** – the number of expected features in the encoder/decoder inputs (default=512).
- **nhead** – the number of heads in the multiheadattention models (default=8).
- **num_encoder_layers** – the number of sub-encoder-layers in the encoder (default=6).
- **num_decoder_layers** – the number of sub-decoder-layers in the decoder (default=6).
- **dim_feedforward** – the dimension of the feedforward network model (default=2048).
- **dropout** – the dropout value (default=0.1).
- **activation** – the activation function of encoder/decoder intermediate layer, can be a string ("relu" or "gelu") or a unary callable. Default: relu
- **custom_encoder** – custom encoder (default=None).
- **custom_decoder** – custom decoder (default=None).
- **layer_norm_eps** – the eps value in layer normalization components (default=1e-5).
- **batch_first** – If `True`, then the input and output tensors are provided as (batch, seq, feature). Default: `False` (seq, batch, feature).
- **norm_first** – if `True`, encoder and decoder layers will perform LayerNorms before other attention and feedforward operations, otherwise after. Default: `False` (after).

Take in and process masked source/target sequences.

Parameters

- **src** – the sequence to the encoder (required).
- **tgt** – the sequence to the decoder (required).
- **src_mask** – the additive mask for the src sequence (optional).
- **tgt_mask** – the additive mask for the tgt sequence (optional).
- **memory_mask** – the additive mask for the encoder output (optional).
- **src_key_padding_mask** – the ByteTensor mask for src keys per batch (optional).
- **tgt_key_padding_mask** – the ByteTensor mask for tgt keys per batch (optional).
- **memory_key_padding_mask** – the ByteTensor mask for memory keys per batch (optional).

Shape:

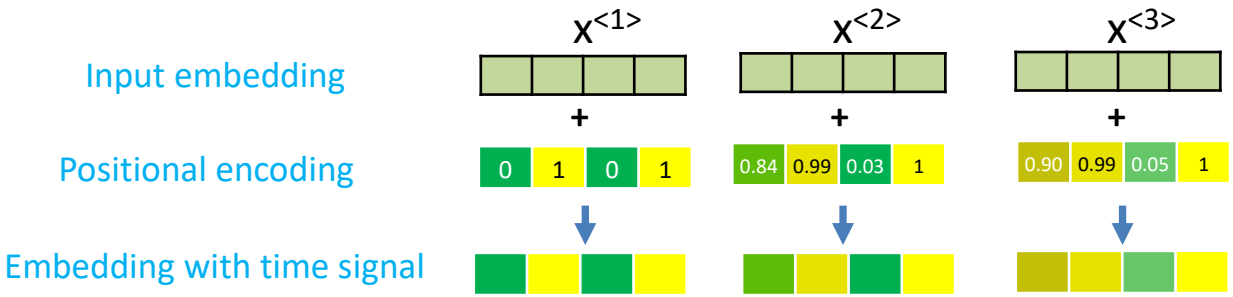
- src: (S, E) for unbatched input, (S, N, E) if *batch_first=False* or *(N, S, E)* if *batch_first=True*.
- tgt: (T, E) for unbatched input, (T, N, E) if *batch_first=False* or *(N, T, E)* if *batch_first=True*.
- src_mask: (S, S) or $(N \cdot \text{num_heads}, S, S)$.
- tgt_mask: (T, T) or $(N \cdot \text{num_heads}, T, T)$.
- memory_mask: (T, S).
- src_key_padding_mask: (S) for unbatched input otherwise (N, S).
- tgt_key_padding_mask: (T) for unbatched input otherwise (N, T).
- memory_key_padding_mask: (S) for unbatched input otherwise (N, S).

Table of Contents

1. Introduction to Transformers
2. **Positional Encoding**
3. The Encoder Network
   - Multi-Head Self-Attention
   - Layer Normalization
   - Feed-Forward
4. The Decoder Network
   - Masked Multi-Head Self-Attention
   - Multi-Head Cross-Attention
5. Types of Transformer Network

Positional Encoding

- Non-recursive models encodes positional information at the feature level. This makes it parallelable.
- Positional encoding (*PE*)
  - encodes the time stamp of the input
  - has the same dimensionality as the input embedding
  - Added to the input embeddings to encode time stamp
- Example: for $T_x = 3$



Source: https://jalammar.github.io/illustrated-transformer/

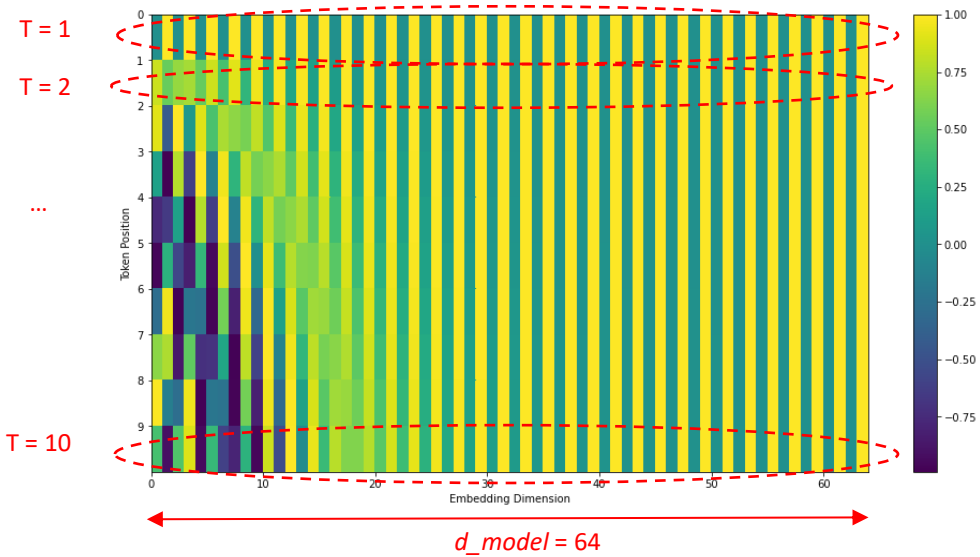
Positional Encoding

- Fixed positional encoding with sinusoidal signals:

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

Use sin when *i* is even, and cos when *i* is odd

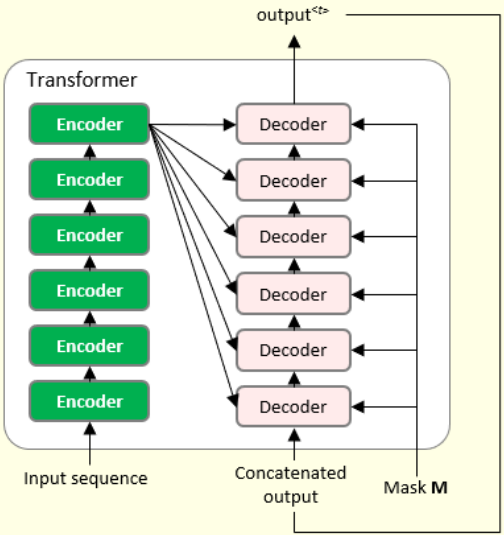
- Example: *d_model* = 64 and *T* = 10



Source: https://jalammar.github.io/illustrated-transformer/

Table of Contents

1. Introduction to Transformers
2. Positional Encoding
3. **The Encoder Network**
   - Multi-Head Self-Attention
   - Layer Normalization
   - Feed-Forward
4. The Decoder Network
   - Masked Multi-Head Self-Attention
   - Multi-Head Cross-Attention
5. Types of Transformer Network



PyTorch Implementation

[torch.nn.TransformerEncoder]

TRANSFORMERENCODER

CLASS torch.nn.TransformerEncoder(*encoder_layer*, *num_layers*, *norm=None*, *enable_nested_tensor=True*, *mask_check=True*) [SOURCE]

TransformerEncoder is a stack of N encoder layers.

Users can build the BERT(https://arxiv.org/abs/1810.04805) model with corresponding parameters.

Parameters

- **encoder_layer** – an instance of the TransformerEncoderLayer() class (required).
- **num_layers** – the number of sub-encoder-layers in the encoder (required).
- **norm** – the layer normalization component (optional).
- **enable_nested_tensor** – if True, input will automatically convert to nested tensor (and convert back on output). This will improve the overall performance of TransformerEncoder when padding rate is high. Default: `True` (enabled).

forward(*src*, *mask=None*, *src_key_padding_mask=None*, *is_causal=None*) [SOURCE]

Pass the input through the encoder layers in turn.

Parameters

- **src** (*Tensor*) – the sequence to the encoder (required).
- **mask** (*Optional*[*Tensor*]) – the mask for the src sequence (optional).
- **src_key_padding_mask** (*Optional*[*Tensor*]) – the mask for the src keys per batch (optional).
- **is_causal** (*Optional*[*bool*]) – If specified, applies a causal mask as `mask`. Default: `None`; try to detect a causal mask. Warning: `is_causal` provides a hint that `mask` is the causal mask. Providing incorrect hints can result in incorrect execution, including forward and backward compatibility.

Return type

Tensor

Examples::

```
>>> encoder_layer = nn.TransformerEncoderLayer(d_model=512, nhead=8)
>>> transformer_encoder = nn.TransformerEncoder(encoder_layer, num_layers=6)
>>> src = torch.rand(10, 32, 512)
>>> out = transformer_encoder(src)
```

PyTorch Implementation

torch.nn.TransformerEncoderLayer

TRANSFORMERENCODERLAYER

CLASS torch.nn.TransformerEncoderLayer(*d_model, nhead, dim_feedforward=2048, dropout=0.1, activation=<function relu>, layer_norm_eps=1e-05, batch_first=False, norm_first=False, bias=True, device=None, dtype=None*) [SOURCE]

Parameters

- **d_model** (*int*) – the number of expected features in the input (required).
- **nhead** (*int*) – the number of heads in the multiheadattention models (required).
- **dim_feedforward** (*int*) – the dimension of the feedforward network model (default=2048).
- **dropout** (*float*) – the dropout value (default=0.1).
- **activation** (*Union[str, Callable[[Tensor], Tensor]]*) – the activation function of the intermediate layer, can be a string ("relu" or "gelu") or a unary callable. Default: relu
- **layer_norm_eps** (*float*) – the eps value in layer normalization components (default=1e-5).
- **batch_first** (*bool*) – If `True`, then the input and output tensors are provided as (batch, seq, feature). Default: `False` (seq, batch, feature).
- **norm_first** (*bool*) – if `True`, layer norm is done prior to attention and feedforward operations, respectively. Otherwise it's done after. Default: `False` (after).
- **bias** (*bool*) – If set to `False`, `Linear` and `LayerNorm` layers will not learn an additive bias. Default: `True`.

forward(*src, src_mask=None, src_key_padding_mask=None, is_causal=False*) [SOURCE]

Pass the input through the encoder layer.

Parameters

- **src** (*Tensor*) – the sequence to the encoder layer (required).
- **src_mask** (*Optional[Tensor]*) – the mask for the src sequence (optional).
- **src_key_padding_mask** (*Optional[Tensor]*) – the mask for the src keys per batch (optional).
- **is_causal** (*bool*) – If specified, applies a causal mask as `src mask`. Default: `False`. Warning: `is_causal` provides a hint that `src_mask` is the causal mask. Providing incorrect hints can result in incorrect execution, including forward and backward compatibility.

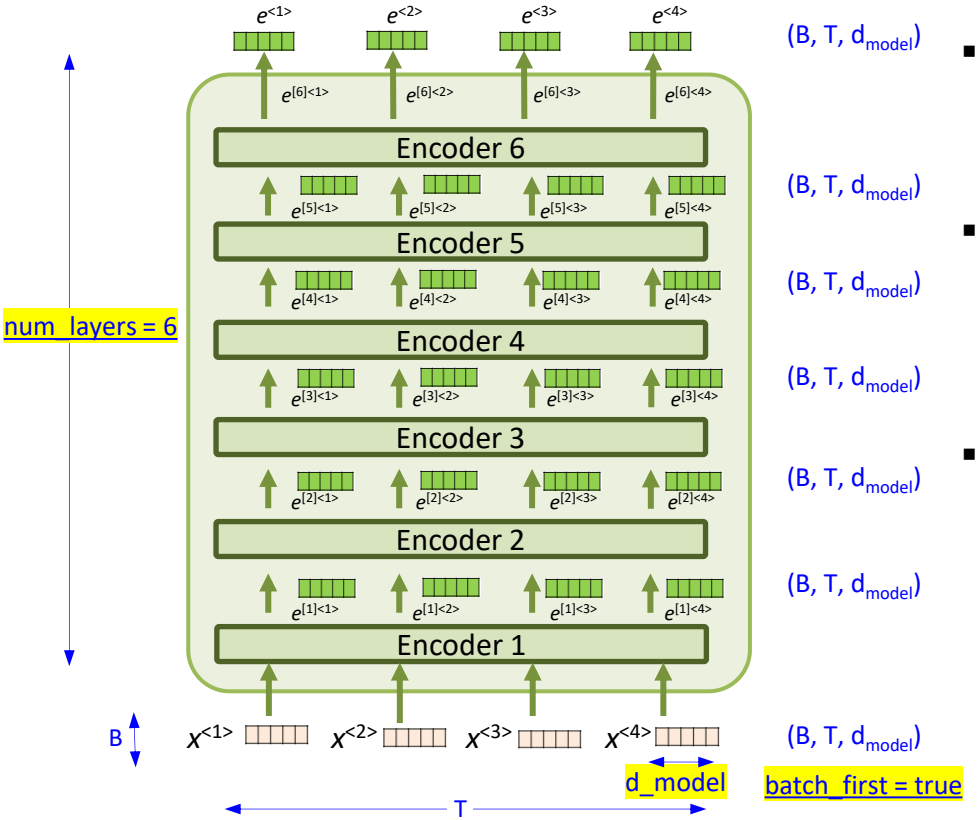
Return type

Tensor

Examples::

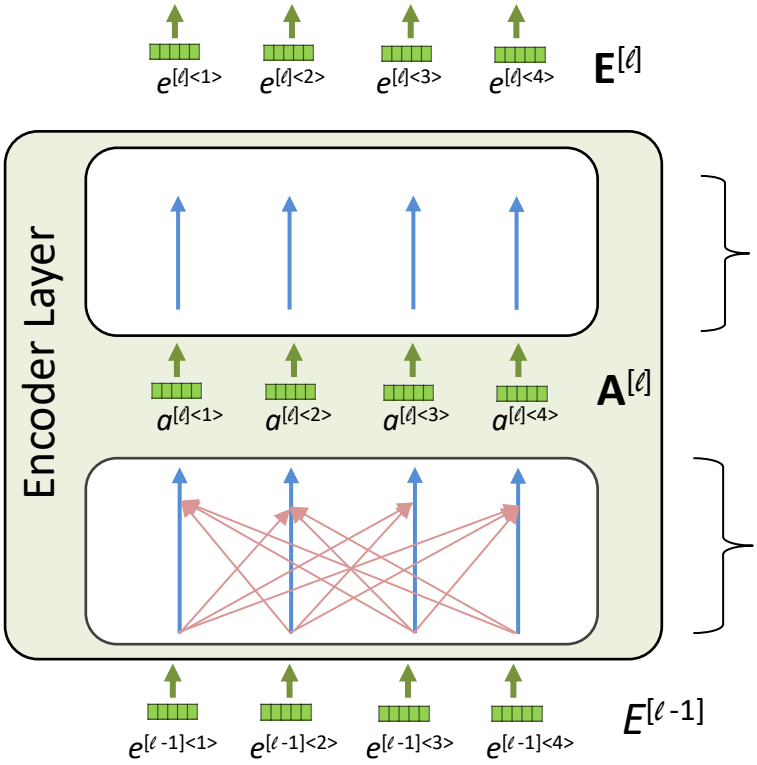
```
>>> encoder_layer = nn.TransformerEncoderLayer(d_model=512, nhead=8)
>>> src = torch.rand(10, 32, 512)
>>> out = encoder_layer(src)
```

The Encoder Network



- Data format:
  - batch_first: (B, T, d_{model})
  - time_first: (T, B, d_{model})
- Each layer successively enriches each token with its neighboring tokens in the sequence
- The shape of all layers are the same

The Encoder Layer

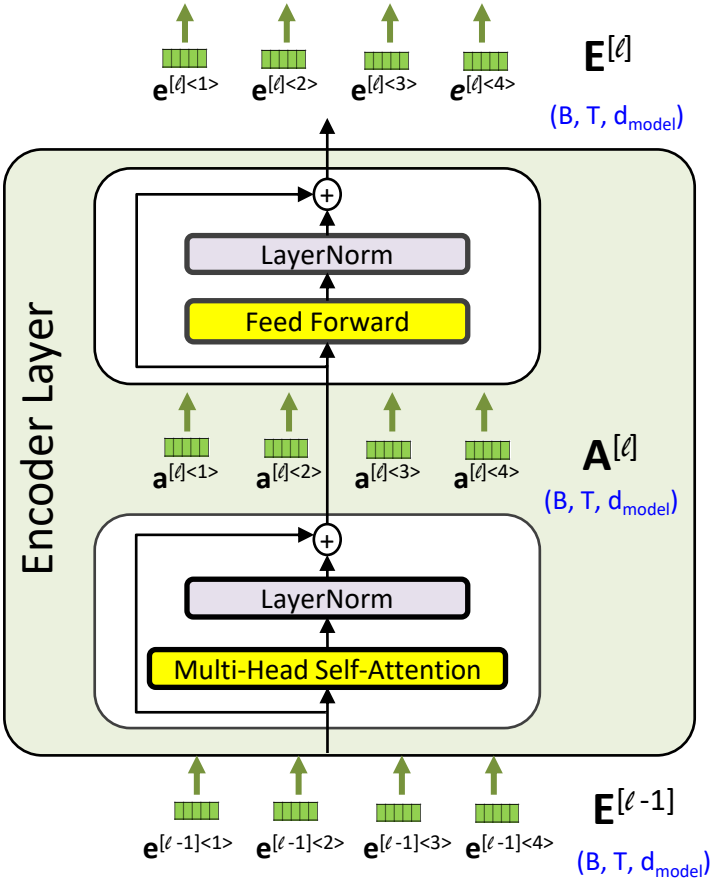


The encoder block consists of two blocks:

2 Feed-forward Block extracts higher-level features from the attention features

1 Multi-head self-attention Block enriches each word with other words in the *same* sentence to get the attention features

The Encoder Layer



All layers preserve the shape of the input tensor

Feed-forward Block:
Fully Connected(FC) + LayerNorm

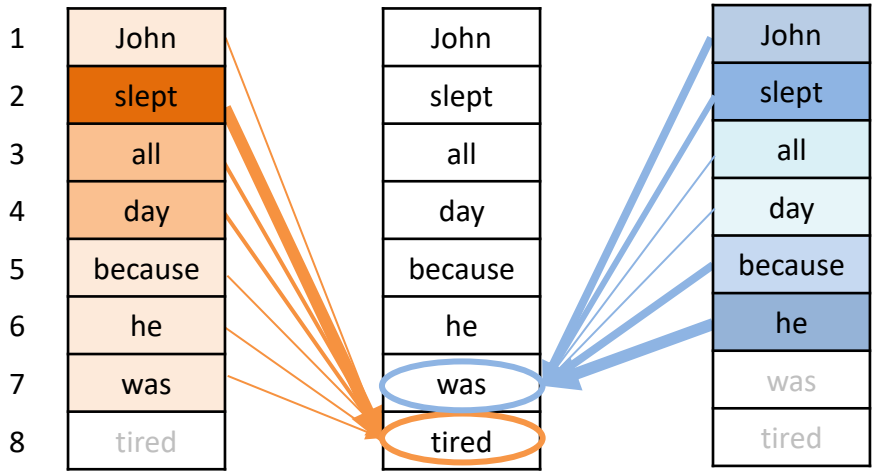
Multi-Head Self-Attention Block:
Multi-Head Self-Attention (MHSA) + LayerNorm

Table of Contents

1. Introduction to Transformers
2. Positional Encoding
3. The Encoder Network
   - **Multi-Head Self Attention**
   - Layer Normalization
   - Feed-Forward
4. The Decoder Network
   - Masked Multi-Head Self-Attention
   - Multi-Head Cross-Attention
5. Types of Transformer Network

What is Self-Attention?

Given an input sequence, enrich the representation of each word semantically or syntactically with its words in the same sentence



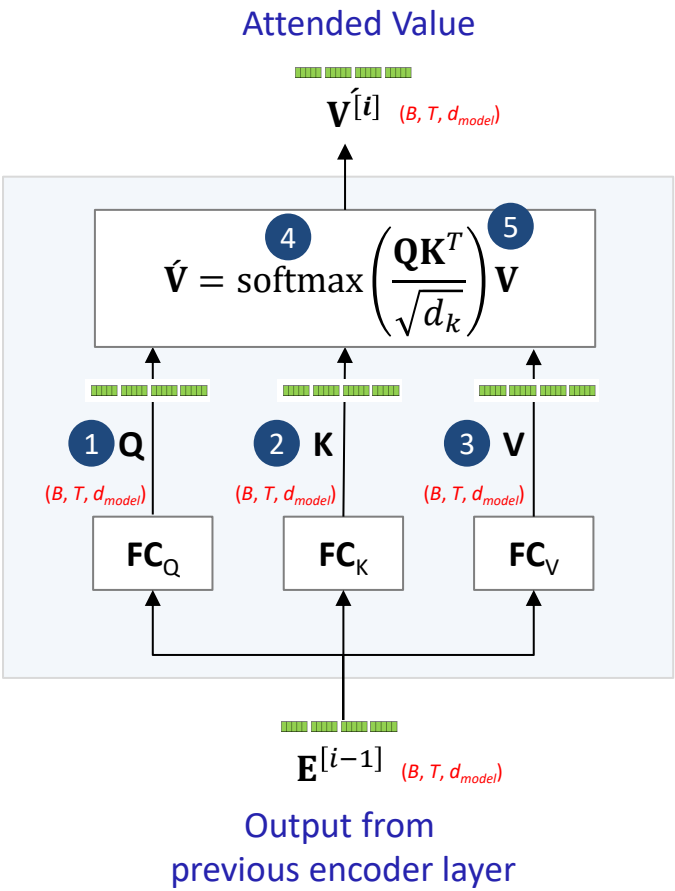
Enriching token at t=8 semantically with "*slept*" is useful for the model to correctly predict "*tired*".

Enriching token at t=6 syntactically with "he" is useful for the model to correctly predict "*was*".

The Scaled Dot-Product Attention Layer

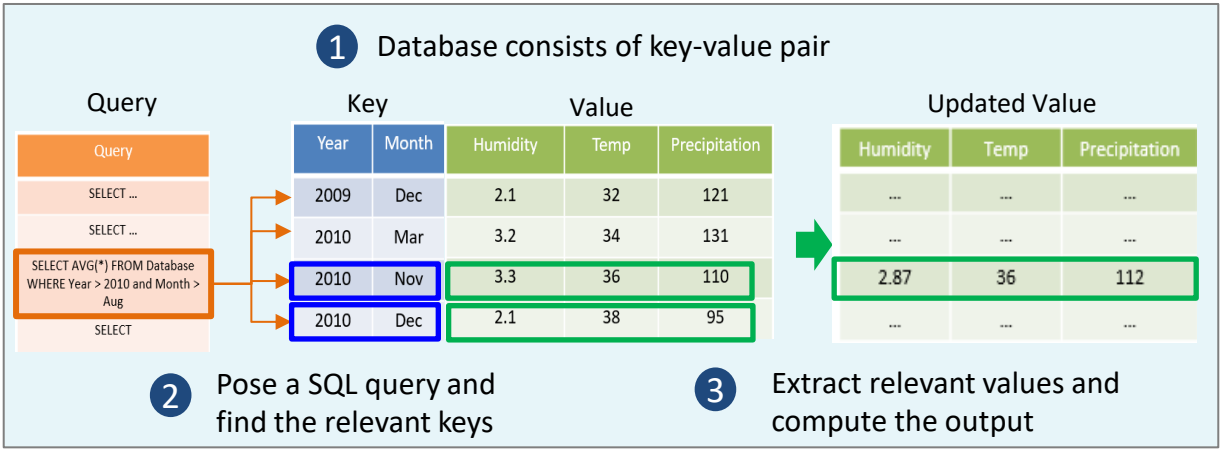
Commences in two steps:

1. Compute **QKV**
   - Input: $\mathbf{E}^{[i-1]}$
   - Uses Linear layer to generate:
     1. Query **Q**
     2. Key **K**
     3. Value **V**
   - Highly parallelizable across batch and time
2. Compute **A** (attended values)
   - Use **query** to find relevant tokens by comparing with the **keys**. 4
   - Enrich the token $\mathbf{V}^{[i]}$ by combining the values of relevant tokens. 5
   - Quadratic complexity



Analogy to Database Query

- Analogy to using SQL query to update records



1. The layer extract **values** (features) to represent each token. For each value, it also generates a **key** to index the token.

2. For each token, generate a **query** to specify what it is looking for. Then, use the dot product to compute similarity (**attention**) to the **keys** of all tokens

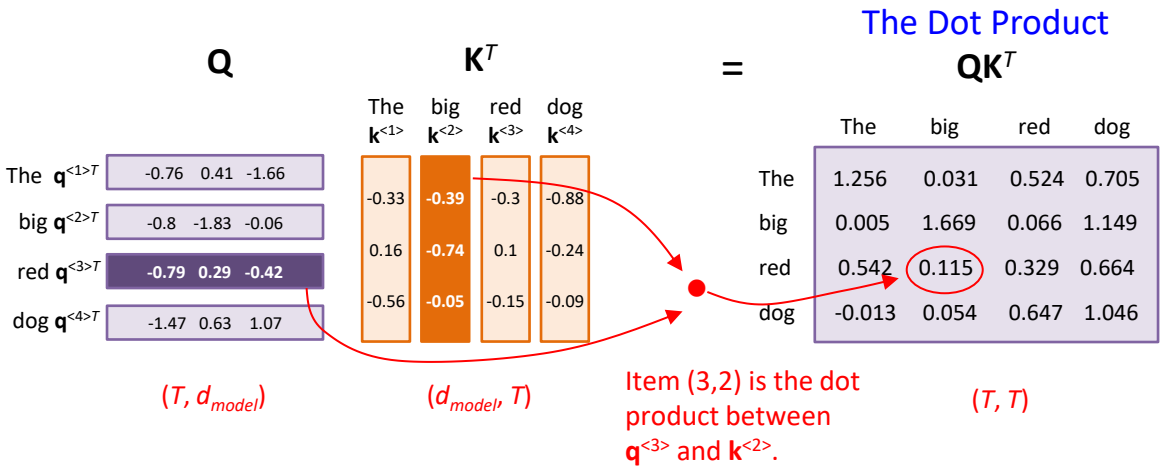
3. For each token, linearly combine the values of the **tokens** weighted by the **attention**. Tokens with higher attention value has higher weights.

Dot Product between Q and K → Attention scores

Attention scores

$$\acute{\mathbf{V}} = \text{attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}\left(\frac{\boxed{\mathbf{QK}^T}}{\sqrt{d_k}}\right)\mathbf{V}$$

- $\mathbf{QK}^T$ computes the dot product to get the attention scores (similarity) between query and key pairs.



Scaled Dot Product → Scaled Attention Scores

$$\acute{\mathbf{V}} = \text{attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}\left(\boxed{\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_k}}}\right)\mathbf{V}$$

Attention(Q,K) = Scaled attention scores

- Divide by the square root of the dimensionality of the key vector, $\sqrt{d_k}$ to get the scaled attention scores.
- This stabilizes the gradients and ensure that the dot products do not grow too large in magnitude.
- Without it, softmax output is more perky or sharp (nearly binary probability) → some tokens receive excessively high attention while others receive none → poor performance and difficulty in learning token relationship.

Weighted sum of values → Attention Probabilities

Attention probabilities (or weights)

$$\acute{\mathbf{V}} = \text{attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \boxed{\text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_k}}\right)}\mathbf{V}$$

- Softmax converts each row into a probability distribution.

QK^T

	The	big	red	dog
The	1.256	0.031	0.524	0.705
big	0.005	1.669	0.066	1.149
red	0.542	0.115	0.329	0.664
dog	-0.013	0.054	0.647	1.046

div & softmax →

Attention probabilities

softmax(**QK**$^T/\sqrt{d_k}$)

	The	big	red	dog	
The	0.3048	0.2026	0.2388	0.2537	→ ∑=1
big	0.1913	0.3332	0.1953	0.2801	→ ∑=1
red	0.2604	0.2258	0.2426	0.2712	→ ∑=1
dog	0.2187	0.2236	0.2463	0.3113	→ ∑=1

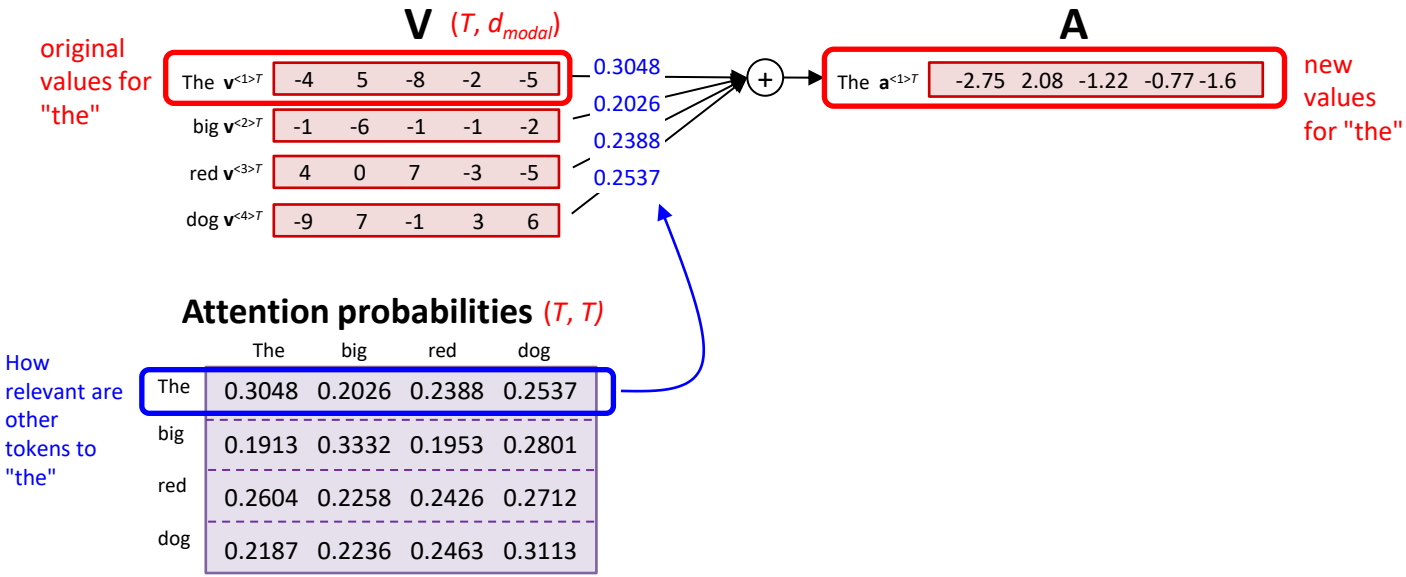
Row *t* represents the probability of the tokens in the sentence to be relevant to the token at time step *t*.

Computing the Attended Values (1)

The Attended Values

$$\acute{\mathbf{V}} = \text{attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \boxed{\text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_k}}\right)\mathbf{V}}$$

- The attended values is computed by linearly adding the **vectors** for all tokens weighted by the scaled dot product.

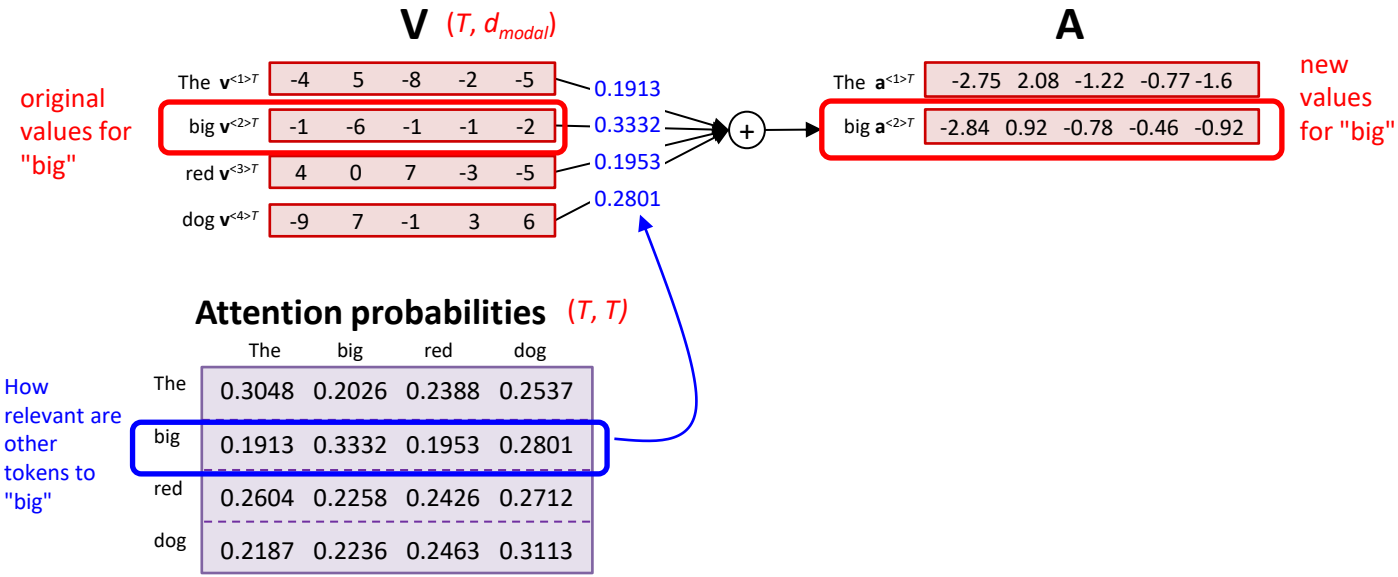


Computing the Attended Values (2)

The Attended Values

$$\acute{\mathbf{V}} = \text{attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \boxed{\text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_k}}\right)\mathbf{V}}$$

- The attended values is computed by linearly adding the **vectors** for all tokens weighted by the scaled dot product.

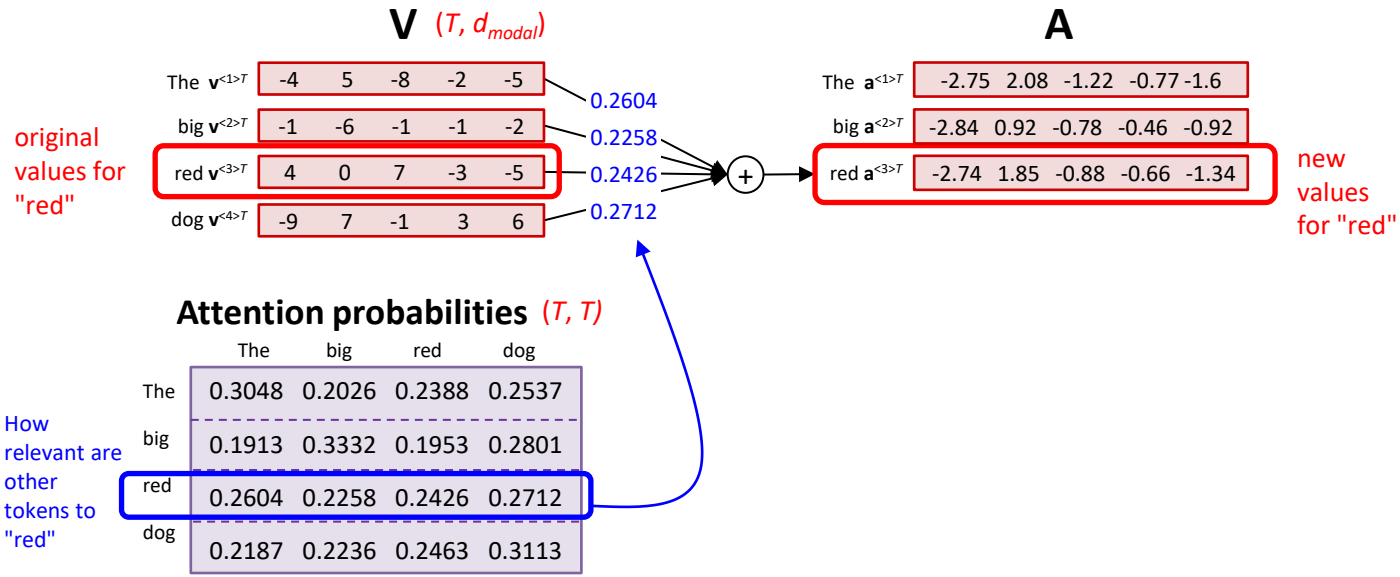


Computing the Attended Values (3)

The Attended Values

$$\acute{\mathbf{V}} = \text{attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \boxed{\text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_k}}\right)\mathbf{V}}$$

- The attended values is computed by linearly adding the **vectors** for all tokens weighted by the scaled dot product.



Computing the Attended Values (4)

The Attended Values

$$\acute{\mathbf{V}} = \text{attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \boxed{\text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_k}}\right)\mathbf{V}}$$

- The attended values is computed by linearly adding the **vectors** for all tokens weighted by the scaled dot product.

V (T, d_{modal})

The $\mathbf{v}^{<1>T}$	-4	5	-8	-2	-5
big $\mathbf{v}^{<2>T}$	-1	-6	-1	-1	-2
red $\mathbf{v}^{<3>T}$	4	0	7	-3	-5
dog $\mathbf{v}^{<4>T}$	-9	7	-1	3	6

original values for "dog"

0.2187
0.2236
0.2463
0.3113

+

A (T, d_{modal})

The $\mathbf{a}^{<1>T}$	-2.75	2.08	-1.22	-0.77	-1.6
big $\mathbf{a}^{<2>T}$	-2.84	0.92	-0.78	-0.46	-0.92
red $\mathbf{a}^{<3>T}$	-2.74	1.85	-0.88	-0.66	-1.34
dog $\mathbf{a}^{<4>T}$	-2.92	1.93	-0.56	-0.47	-0.90

new values for "dog"

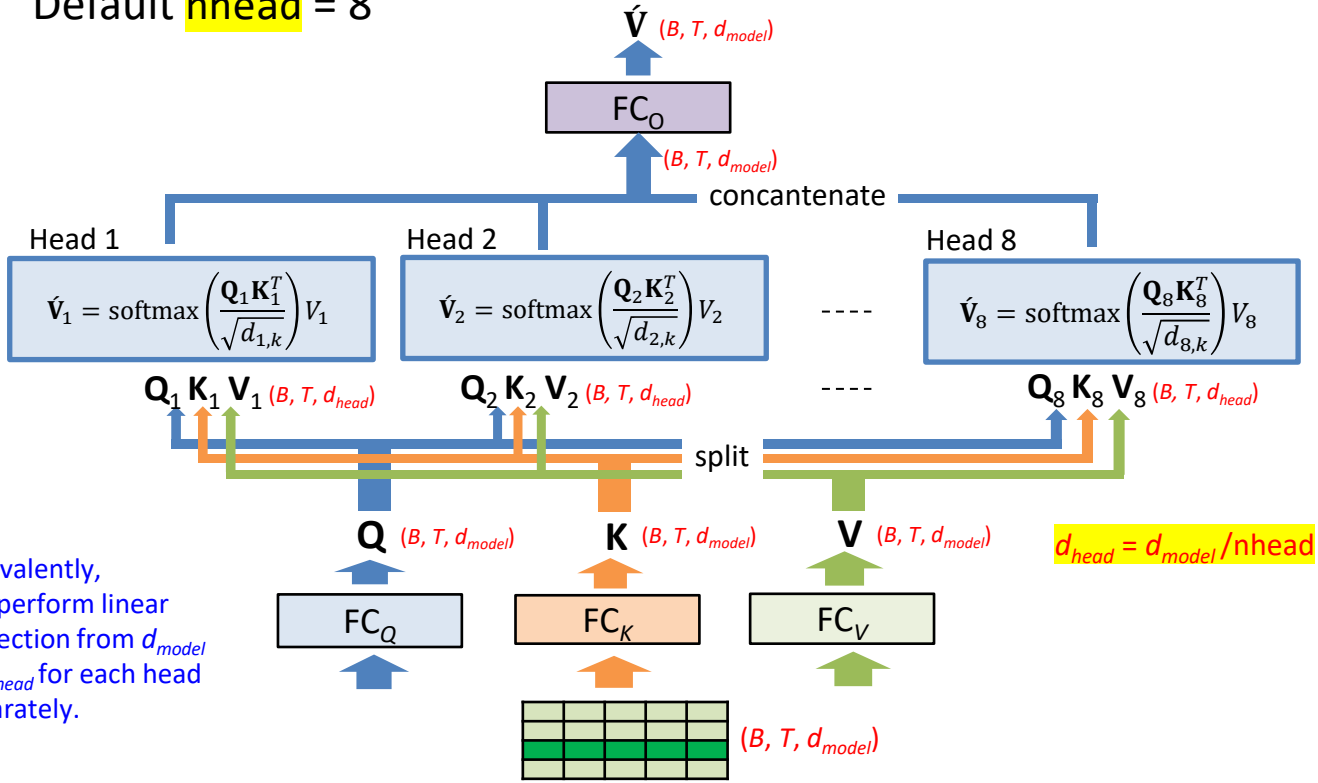
R (T, T)

	The	big	red	dog
The	0.3048	0.2026	0.2388	0.2537
big	0.1913	0.3332	0.1953	0.2801
red	0.2604	0.2258	0.2426	0.2712
dog	0.2187	0.2236	0.2463	0.3113

How relevant are other tokens to "dog"

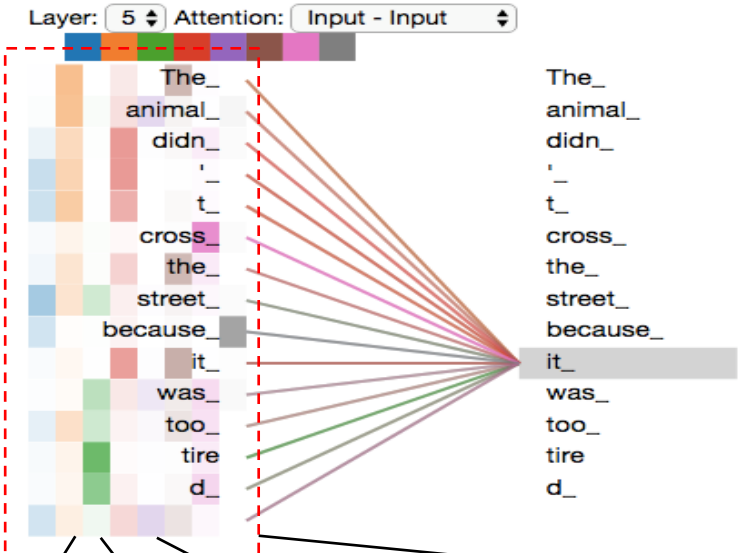
Multi-mode learning with multiple heads

- Can use multiple head to learn different modes of attention
- Default nhead = 8



Equivalently, can perform linear projection from d_{model} to d_{head} for each head separately.

Why multiple heads?



- Each head focuses or specializes on different types of relationships or patterns in the input data.
- E.g., one head on short-range dependencies, another on long-range dependencies, and another on syntactic or semantic patterns.

Total 7 columns here. Column i represents the attention A_i generated by head *i*

A_2 focuses the words before "it"

A_3 focuses on words after "it"

A_5 focuses on the subject "animal"

Source: https://jalammar.github.io/illustrated-transformer/

Multi-Head Self-Attention (MHSA)

- The Multi-Head Self-Attention (MHSA) module is shown below:

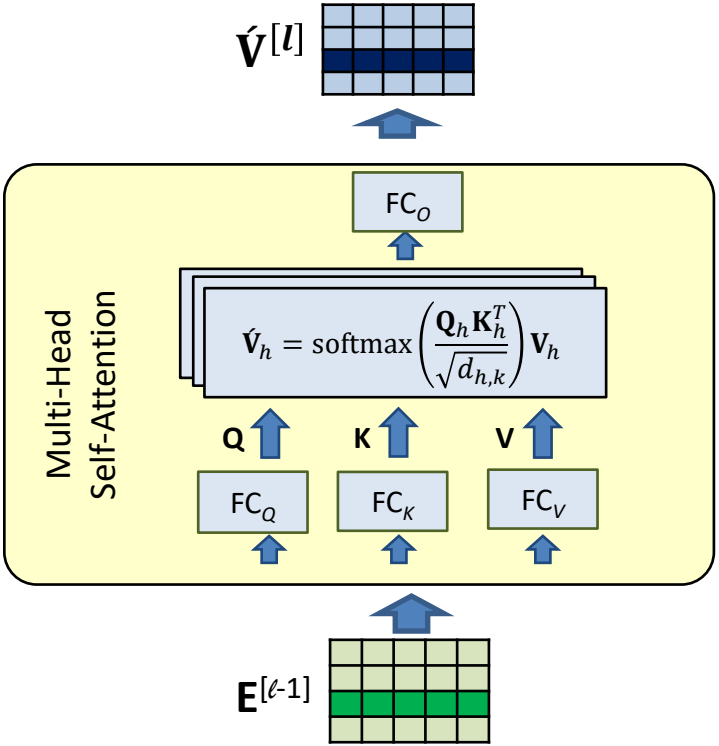
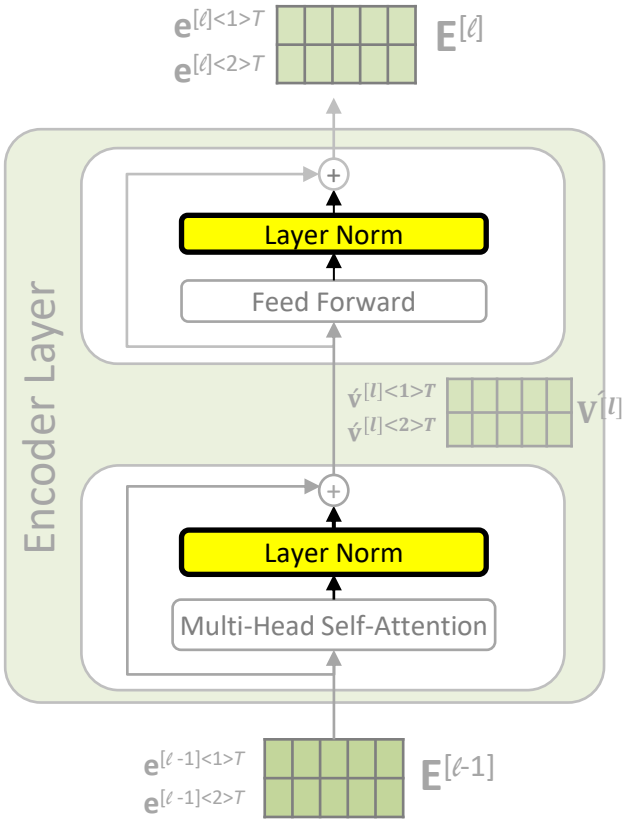


Table of Contents

1. Introduction to Transformers
2. Positional Encoding
3. The Encoder Network
   - Multi-Head Self-Attention
   - **Layer Normalization**
   - Feed-Forward
4. The Decoder Network
   - Masked Multi-Head Self-Attention
   - Multi-Head Cross-Attention
5. Types of Transformer Network

Layer Normalization



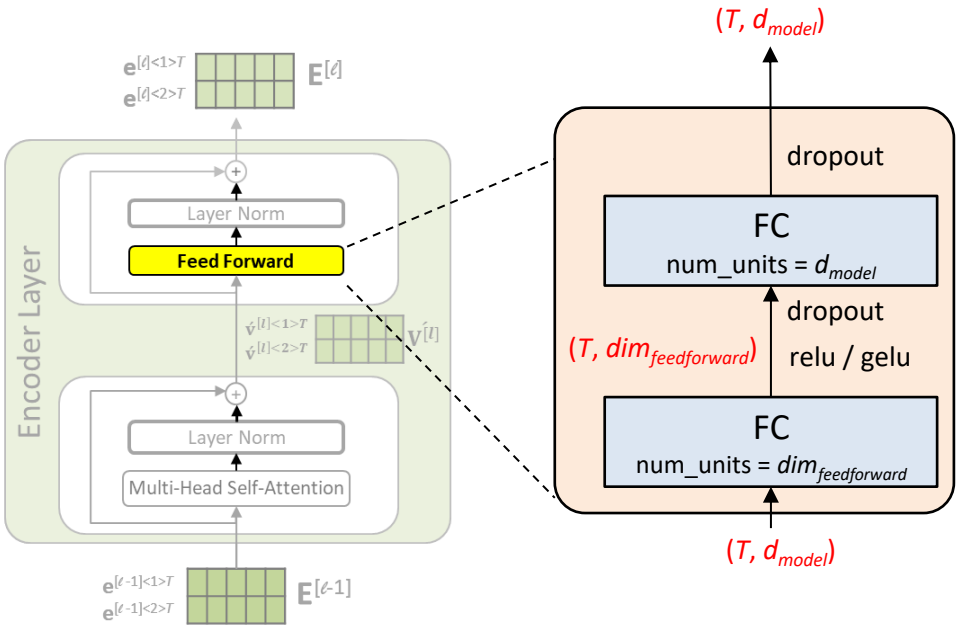
- Layer Norm standardizes each batch sample and time step across the fature dimension.
- Can be placed before or after FFN
  - ✓ Default: `norm_first=False`
  - ✓ Likely better gradient if placed before MHSA and Feedforward

Table of Contents

1. Introduction to Transformers
2. Positional Encoding
3. The Encoder Network
   - Multi-Head Self-Attention
   - Layer Normalization
   - **Feed-Forward**
4. The Decoder Network
   - Masked Multi-Head Self-Attention
   - Multi-Head Cross-Attention
5. Types of Transformer Network

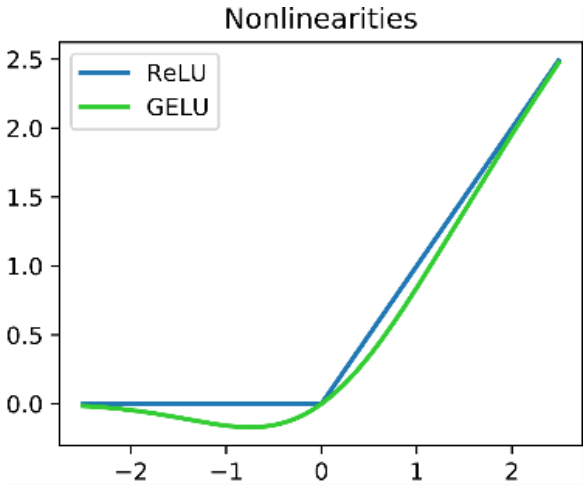
The Feed-Forward Layer

- Uses two fully connected layers to extract higher-level features.
  - First FC layer is configurable: $dim_{feedforward}$ (default = 2048)
  - Second FC layer preserves the input dimension (num_units = d_{model})
  - Uses dropout for regularization: dropout (default = 0.1)



The Feed-Forward Layer

- Uses two fully connected layers to extract higher-level features.
  - First FC layer is configurable: $dim_{feedforward}$ (default = 2048)
  - Second FC layer preserves the input dimension (num_units = d_{model})
  - Uses dropout for regularization: dropout (default = 0.1)
  - Activation choice: activation = relu / gelu (default = relu)



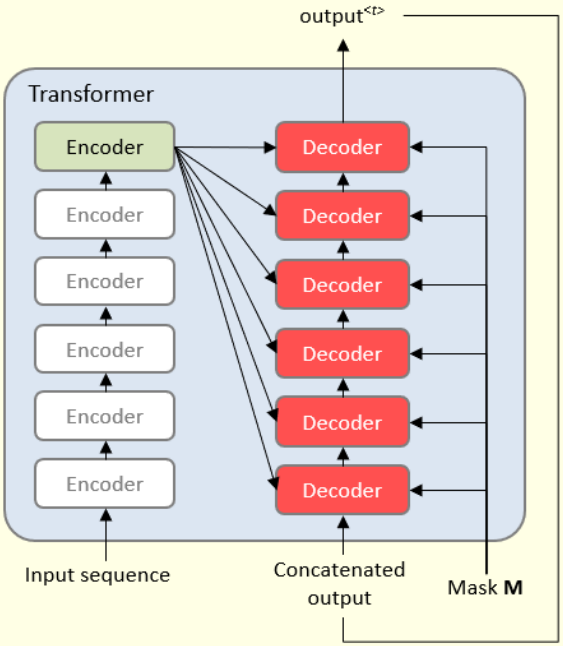
$$\mathrm{GELU}(x) = x \times \frac{1}{2}\left(1 + \mathrm{erf}\left(\frac{x}{\sqrt{2}}\right)\right)$$

$$\mathrm{RELU}(x) = \max(0, x)$$

- Choice of activation functions is task dependent.

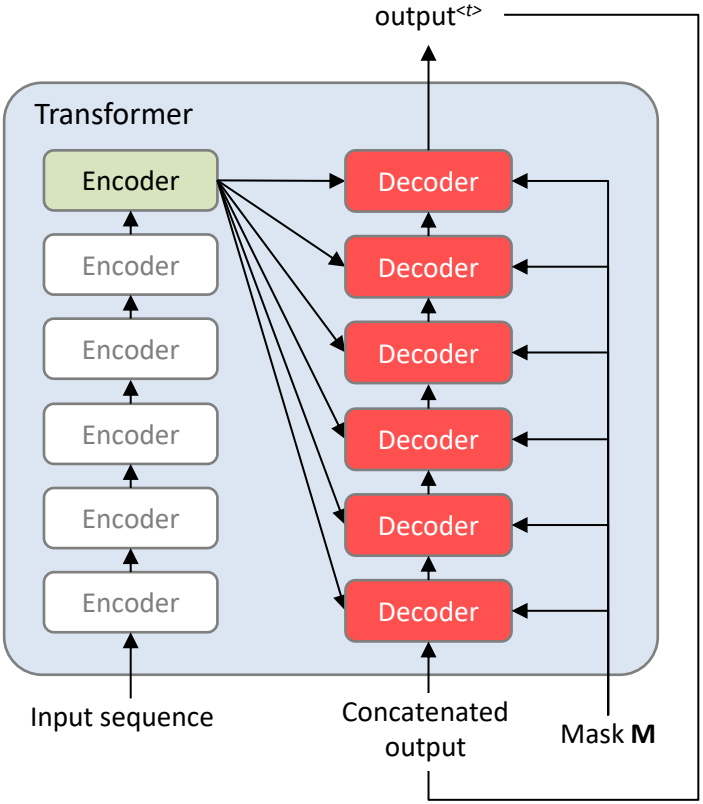
Table of Contents

1. Introduction to Transformers
2. Positional Encoding
3. The Encoder Network
   - Multi-Head Self-Attention
   - Layer Normalization
   - Feed-Forward
4. **The Decoder Network**
   - Masked Multi-Head Self-Attention
   - Multi-Head Cross-Attention
5. Types of Transformer Network

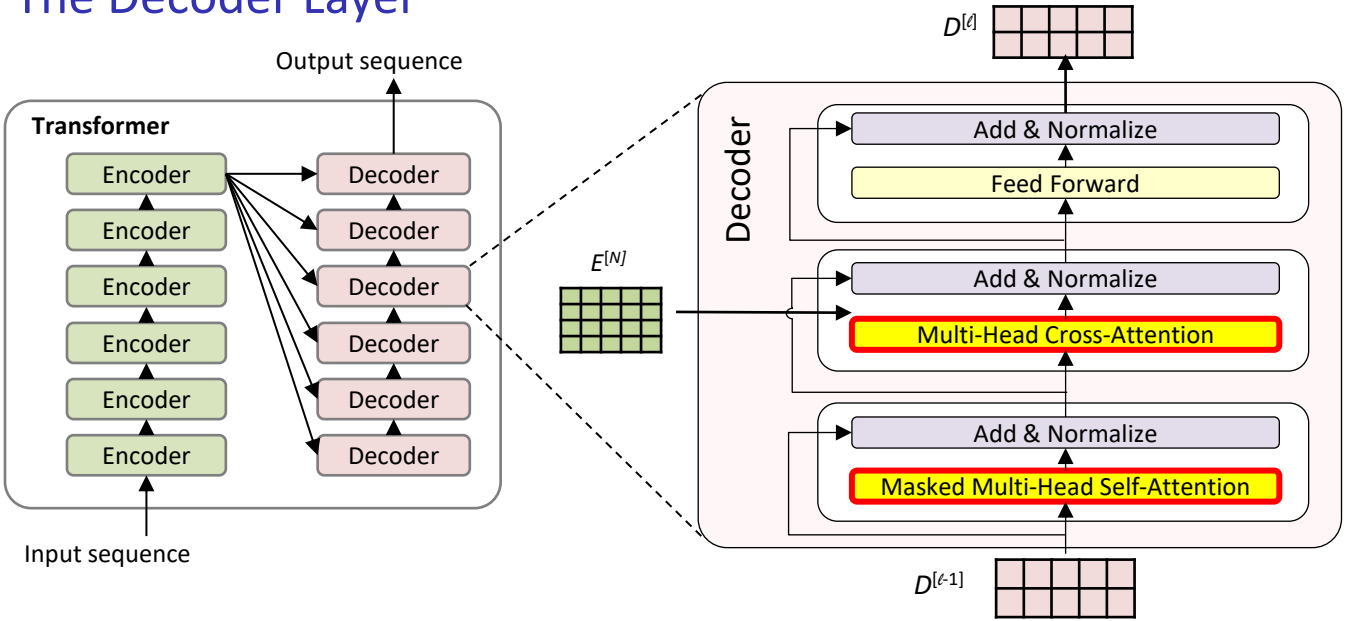


The Decoder Network

- The decoder network is a stack of decoder layers.
- The decoder network extracts features hierarchically to predict the next token.
- Input to a decoder layer:
  - The encoded input sequence (from last encoder layer in the encoder network)
  - The current decoded output sequence (from previous decoder layer in the decoder network)
  - A mask to ignore future tokens



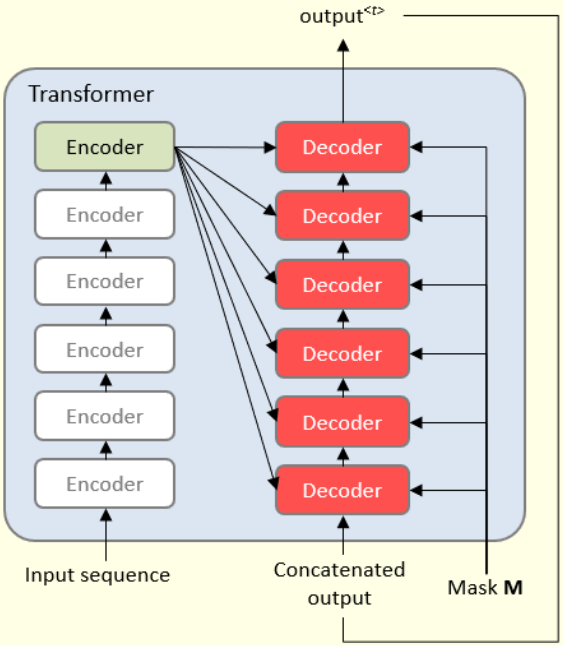
The Decoder Layer



- Similar to the encoder layer except:
  - Masked Multi-Head Self-Attention
    - Considers only past and current input tokens and ignore future ones
  - Multi-head Cross-Attention:
    - Query: current decoded output sequence
    - Value and Key: complete encoded input sequence as **values**

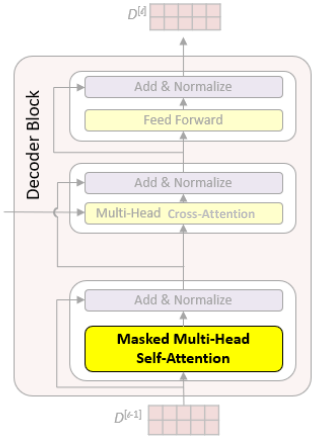
Table of Contents

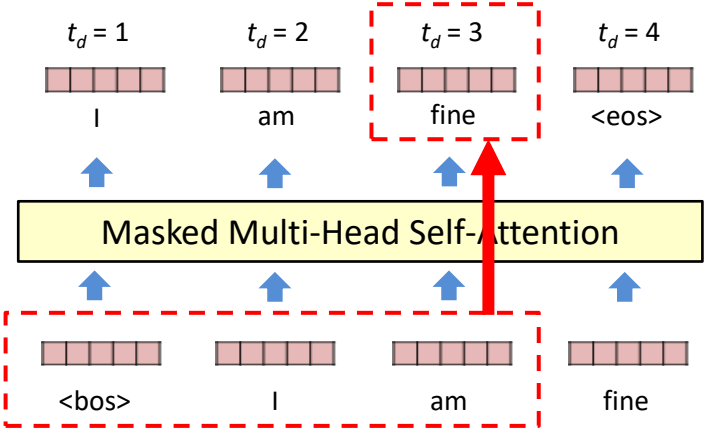
1. Introduction to Transformers
2. Positional Encoding
3. The Encoder Network
   - Multi-Head Self-Attention
   - Layer Normalization
   - Feed-Forward
4. The Decoder Network
   - **Masked Multi-Head Self-Attention**
   - Multi-Head Cross-Attention
5. Types of Transformer Network



Masking the MHSA

- For generative task, the MHSA must support masking.
- Masking makes the layer autoregressive at all time step, i.e., predicts based on past and current input only, but not future input tokens.





At t_d=1, no previous output. Generates 'I'

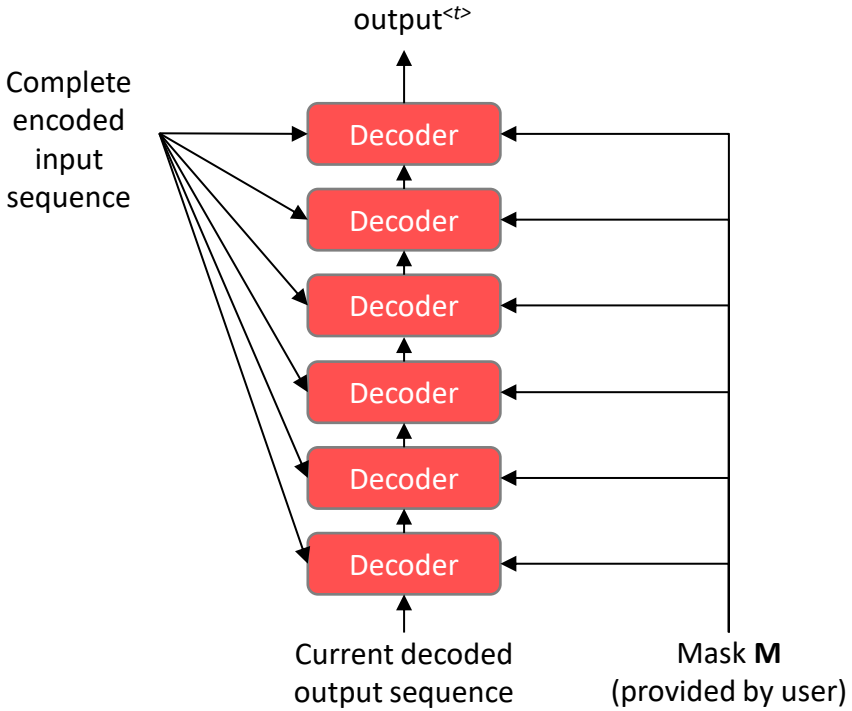
At t_d=2, consider only {'I'} to generate 'am'

At t_d=3, consider only {'I', 'am'} to generate 'fine'

At t_d=4, consider only {'I', 'am', 'fine'} to generate '<eos>'

Masking the MHSA

- In PyTorch, the mask is created by the user and input to the decoder layer



Masking the MHSA

- Add the mask to the dot product before softmax

$$\acute{\mathbf{V}}_m = \text{softmax}\left(\frac{\mathbf{QK}^{\mathrm{T}}+\mathbf{M}}{\sqrt{d_k}}\right)\mathbf{V}$$

Masking at each time step:

- set to -∞ to ignore
- set to 0 to consider

Mask **M**

<t1>	0	-∞	-∞	-∞
<t2>	0	0	-∞	-∞
<t3>	0	0	0	-∞
<t4>	0	0	0	0

Mask for time step <t>

- Example:

Dot Product $\mathbf{QK}^{\mathrm{T}}$

2.23	-0.64	1.24	0.69
0.24	0.52	-1.45	1.44
1.38	0.45	0.31	-0.4
-1.31	-1.85	-0.48	-1.31

no mask: $\text{softmax}\left(\frac{\mathbf{QK}^{\mathrm{T}}}{\sqrt{d_k}}\right)$ →

with mask: $\text{softmax}\left(\frac{\mathbf{QK}^{\mathrm{T}}+\mathrm{M}}{\sqrt{d_k}}\right)$ →

Scaled Dot Product

0.06	0.15	0.45	0.33
0.22	0.26	0.08	0.44
0.40	0.24	0.22	0.14
0.23	0.17	0.37	0.23

1.00	0.00	0.00	0.00
0.46	0.54	0.00	0.00
0.47	0.28	0.25	0.00
0.23	0.17	0.37	0.23

Masked Multi-Head Self-Attention

- The complete architecture of the masked multi-head attention module

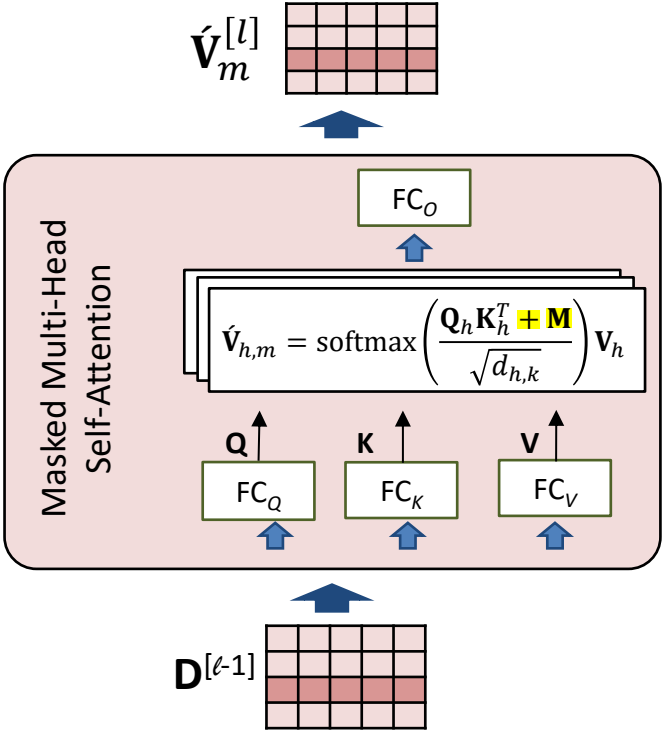
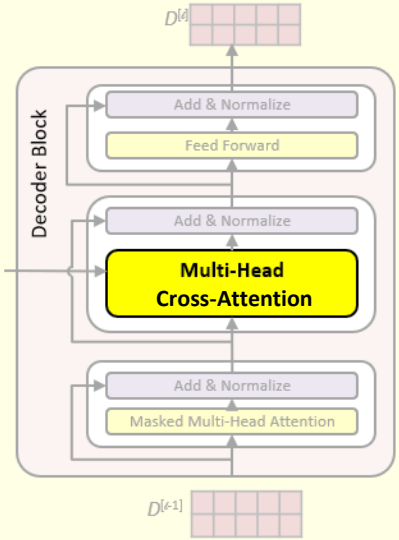
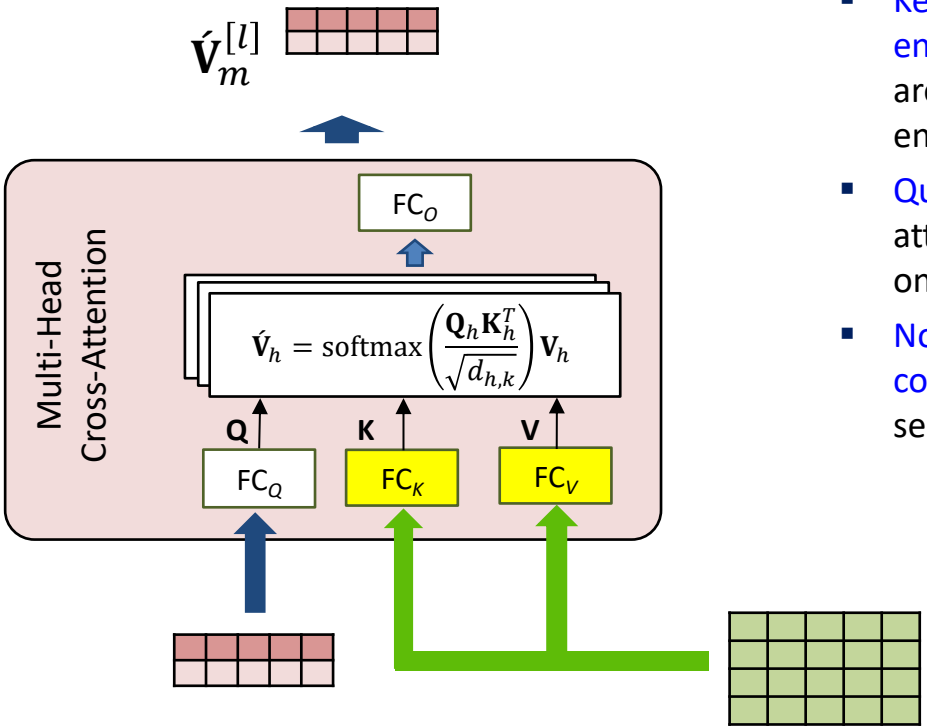


Table of Contents

1. Introduction to Transformers
2. Positional Encoding
3. The Encoder Network
   - Multi-Head Self-Attention
   - Layer Normalization
   - Feed-Forward
4. The Decoder Network
   - Masked Multi-Head Self-Attention
   - **Multi-Head Cross-Attention**
5. Types of Transformer Network



Multi-Head Cross-Attention



$\mathbf{D}^{[\ell-1]}$

Output Sequence (From previous decoder Layer)

$\mathbf{E}^{[N]}$

Input Sequence (Output of Encoder Network)

- Keys and Values are from the encoder: the decoder features are generated by combining encoder features.
- Query is from the decoder: attention is formulated based on output sequence
- No masking since the complete encoded input sequence is available

Table of Contents

1. Introduction to Transformers
2. Positional Encoding
3. The Encoder Network
   - Multi-Head Self-Attention
   - Layer Normalization
   - Feed-Forward
4. The Decoder Network
   - Masked Multi-Head Self-Attention
   - Multi-Head Cross-Attention
5. **Types of Transformer Network**

Size and architectures of GPT-3

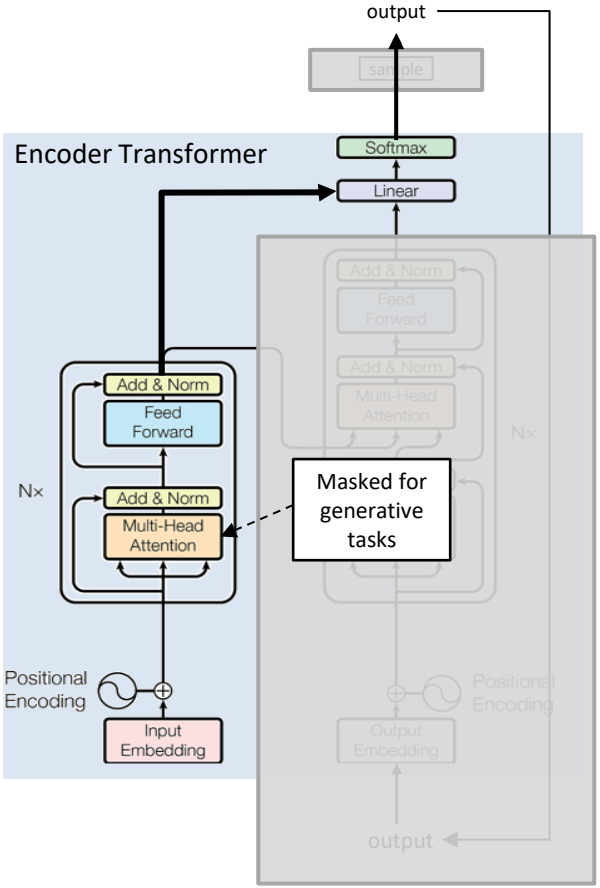
- Different variants of the model can be created by tuning the number of layers, embedding dimensions, number of head and the head dimension

Model Name	n_{params}	n_{layers}	d_{model}	n_{heads}	d_{head}	Batch Size	Learning Rate
GPT-3 Small	125M	12	768	12	64	0.5M	6.0×10^{-4}
GPT-3 Medium	350M	24	1024	16	64	0.5M	3.0×10^{-4}
GPT-3 Large	760M	24	1536	16	96	0.5M	2.5×10^{-4}
GPT-3 XL	1.3B	24	2048	24	128	1M	2.0×10^{-4}
GPT-3 2.7B	2.7B	32	2560	32	80	1M	1.6×10^{-4}
GPT-3 6.7B	6.7B	32	4096	32	128	2M	1.2×10^{-4}
GPT-3 13B	13.0B	40	5140	40	128	2M	1.0×10^{-4}
GPT-3 175B or "GPT-3"	175.0B	96	12288	96	128	3.2M	0.6×10^{-4}

Table 2.1: Sizes, architectures, and learning hyper-parameters (batch size in tokens and learning rate) of the models which we trained. All models were trained for a total of 300 billion tokens.

Encoder-only Transformers

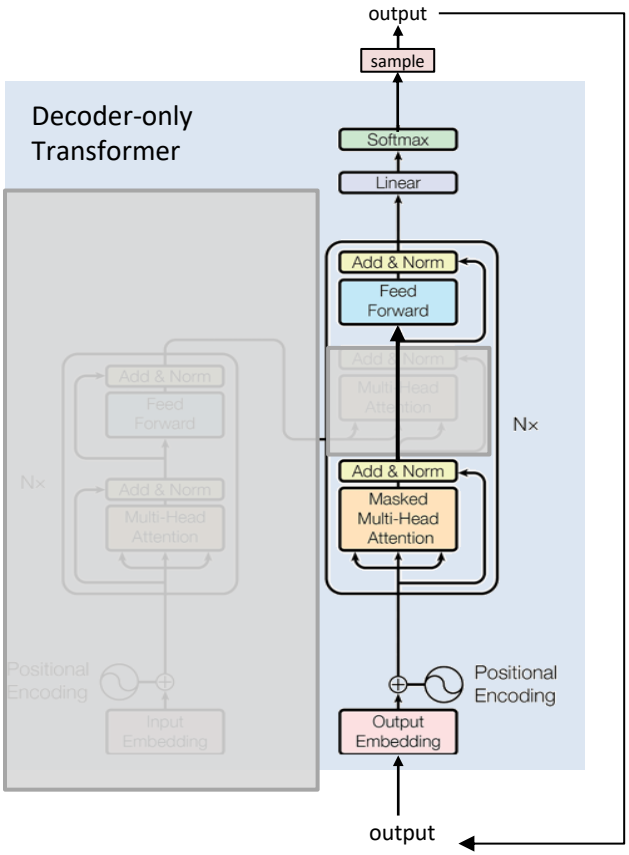
- Uses only the encoder of the network
- Good for classification tasks that require understanding of the full sentence, e.g.,
  - Named entity recognition (NER)
  - Sentiment classification
  - Sentence classification
  - Part of speech tagging
- Example
  - BERT, Albert, DistilBert, RoBERTa, ELECTRA
- Can be used for generative task if
  - MHSA supports masking
  - Feeds the concatenated output as the input



[Devlin, J. et al. (Google), "Bert: Pre-training of deep bidirectional transformers for language understanding." 2018]

Decoder-Only Transformer

- Uses only decoder of a Transformer model
- The cross-attention module is removed.
- Good for generative task, e.g., question answering, text completion, etc.
- Auto-regressive - only access previous and current input to predict current token
- Example:
  - GPT, GPT-2, GPT-3, GPT-3.5
  - PaLM



Thank you

for your attention