CIS 6930 Topics in Computing for Data Science
Week 8: Transformers (1)+(2)

10/21/2021

Yoshihiko (Yoshi) Suhara

2pm-3:20pm

Week 8: Transformers

- **Week 8: Transformers**
- **Week 9: Pre-trained Language Models**
- Week 10: More Machine Learning Techniques
- Week 11: More Deep Learning Techniques for NLP (Text generation, Text summarization, Information Extraction etc.)
- Week 12: Advanced Techniques and Challenges
- Week 13: Final project presentations

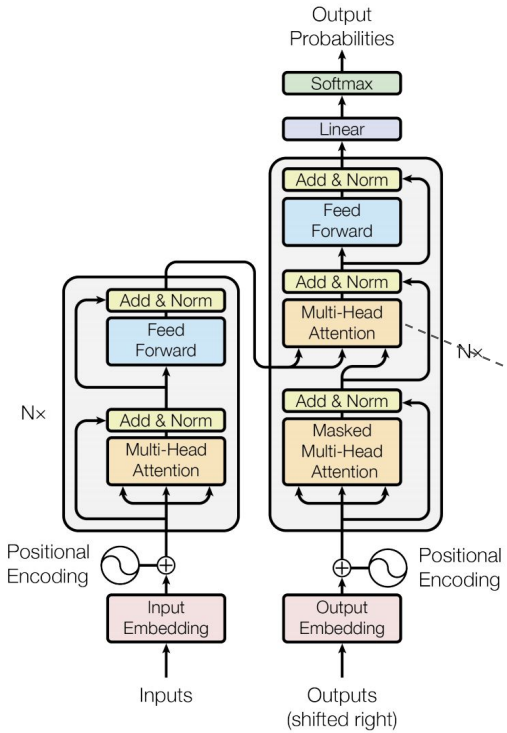
Attention!

- Today’s content is extremely important for understanding modern Deep Learning techniques

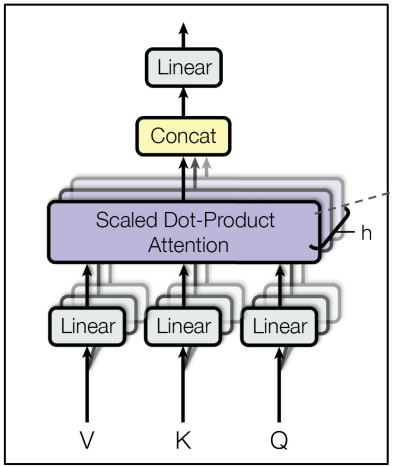
- The content might be technically difficult and complicated 
  - Don’t worry. The concept is relatively simple

- Please make sure that you follow the explanation of each step
  - Don’t hesitate to interrupt and ask questions if you don’t

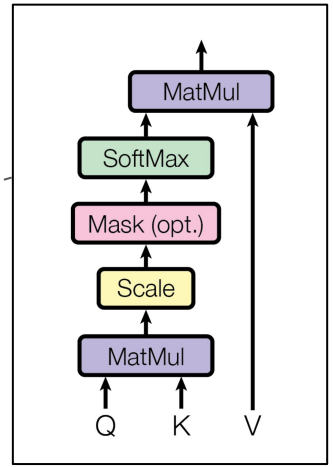
This Week's Goal:
Understanding the Transformer Architecture



Model Architecture



Multi-Head Attention

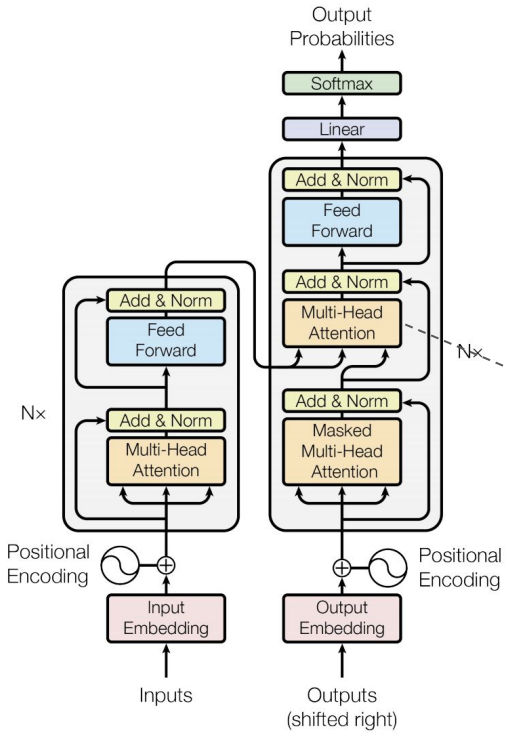


Scaled Dot-Product Attention

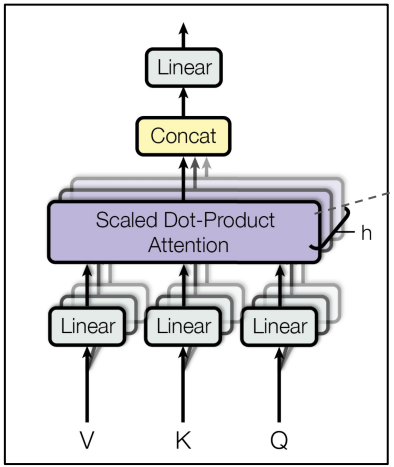
The Transformer: Basic Concepts

The figures and explanations are from the following excellent blog article (highly recommended)
The Illustrated Transformer – Jay Alammar – Visualizing machine learning one concept at a time.

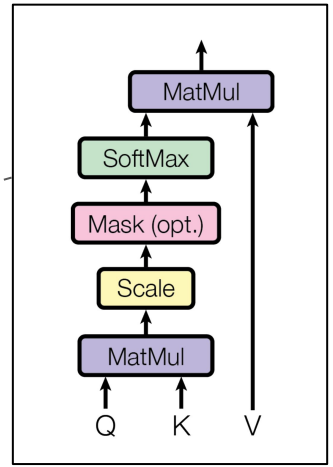
The Transformer



Model Architecture



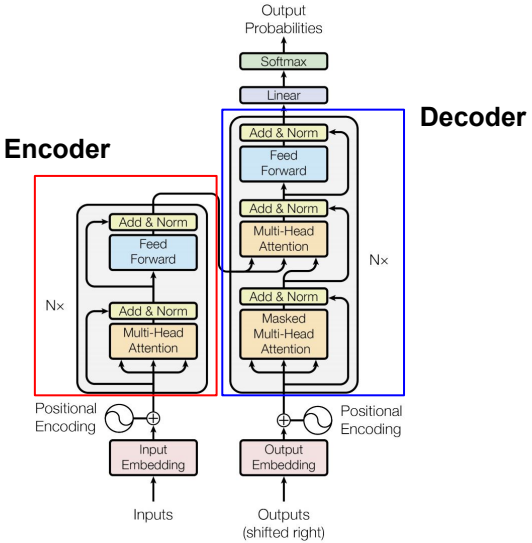
Multi-Head Attention



Scaled Dot-Product Attention

What is the Transformer?

- Transformer is an **Encoder-Decoder model!**
  - Can be used for any sequence-to-sequence generation tasks
- Transformer = **Transformer Encoder** + **Transformer Decoder**



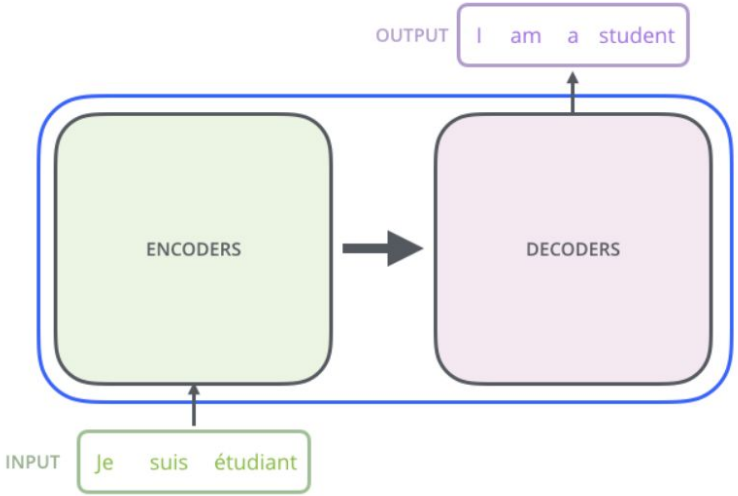
Transformer

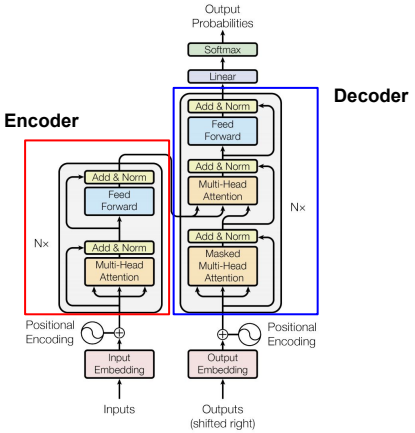
100%

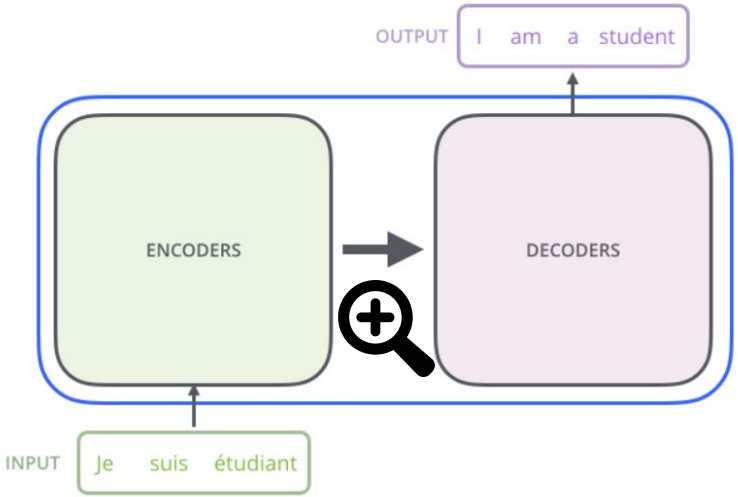
Transformer

100%

150%

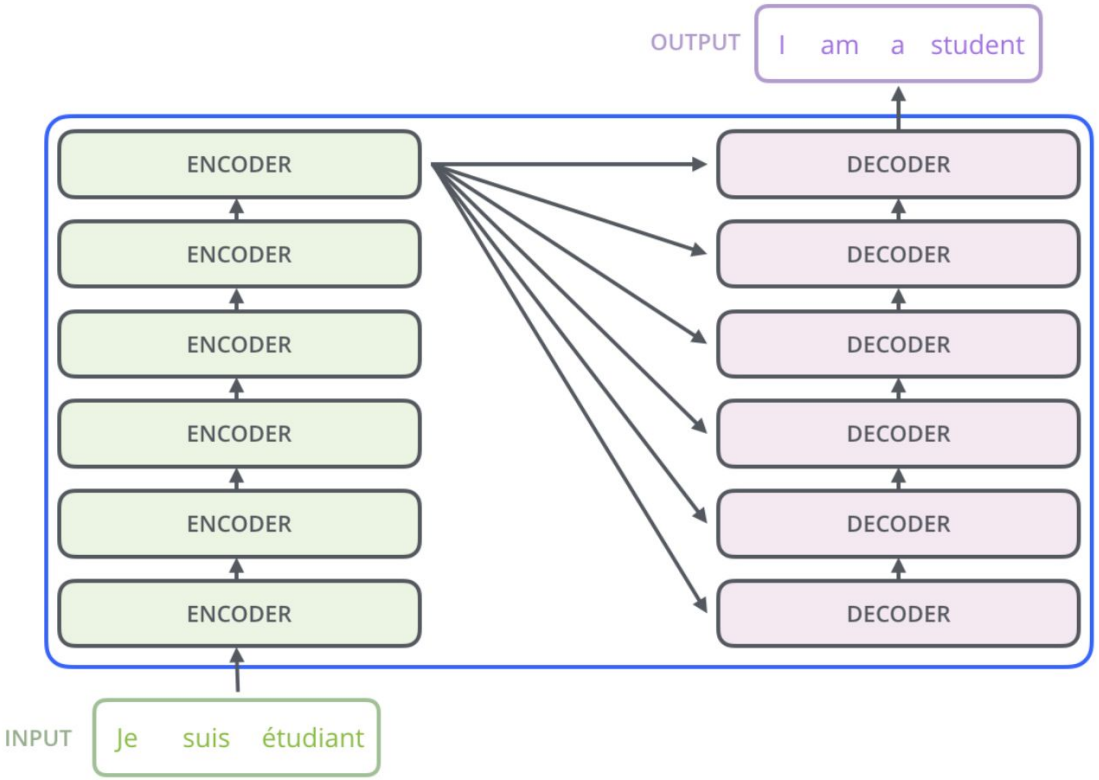

OUTPUT
I am a student
ENCODERS
DECODERS
INPUT
Je suis étudiant

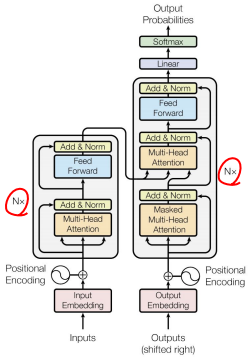

Output Probabilities
Softmax
Linear
Encoder
Decoder
Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
Add & Norm
Masked Multi-Head Attention
Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
N×
N×
Positional Encoding
Positional Encoding
Input Embedding
Output Embedding
Inputs
Outputs (shifted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


OUTPUT
I am a student
ENCODERS
DECODERS
INPUT
Je suis étudiant

Encoder/Decoder
= Stack of Encoder/Decoder Blocks

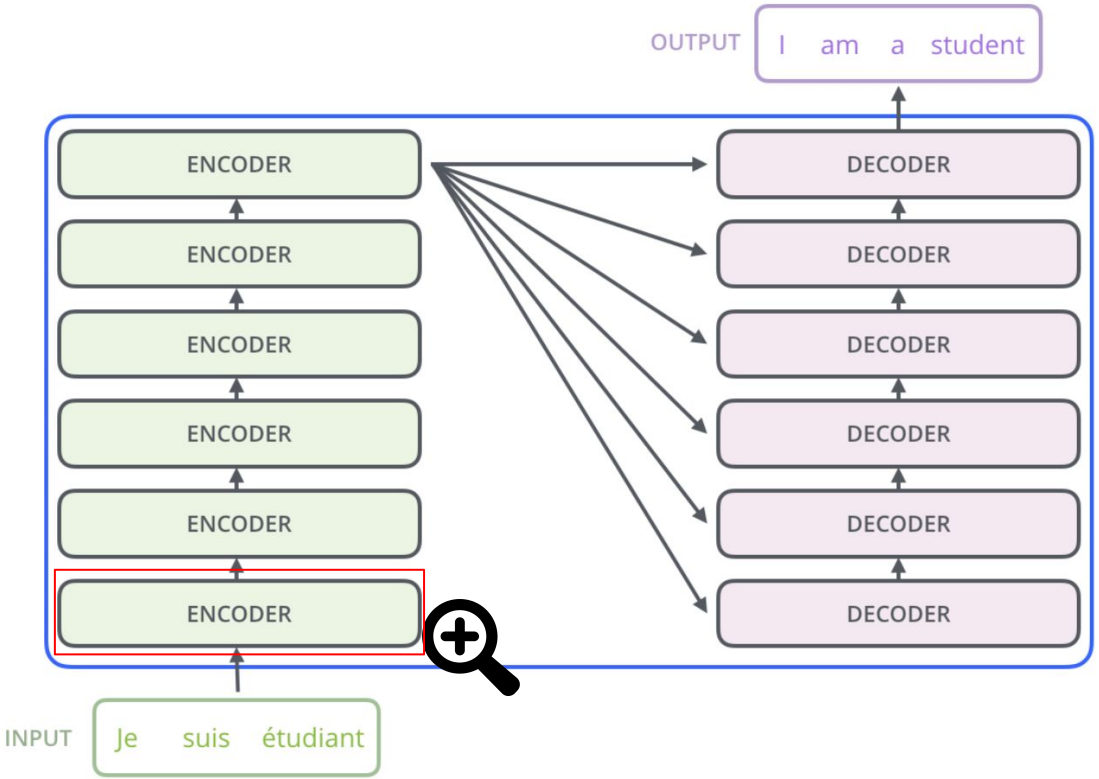
300%



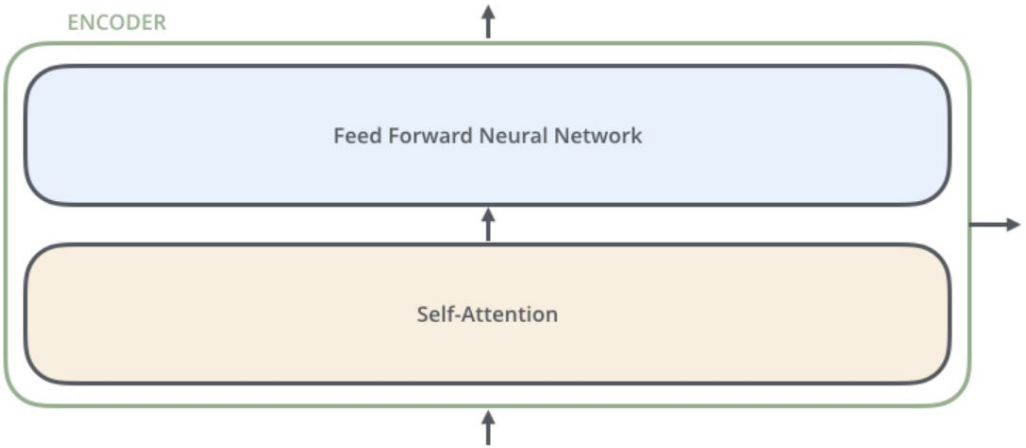


Encoder/Decoder
= Stack of Encoder/Decoder Blocks

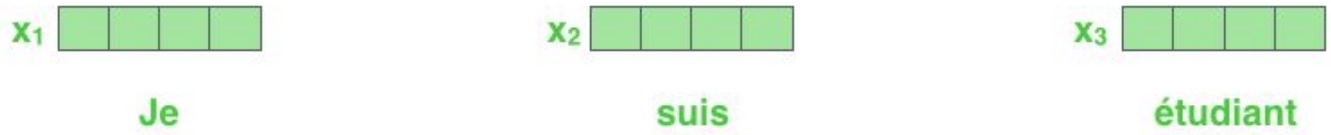
300%



Encoder Block
= Self-Attention + Feed Forward NN

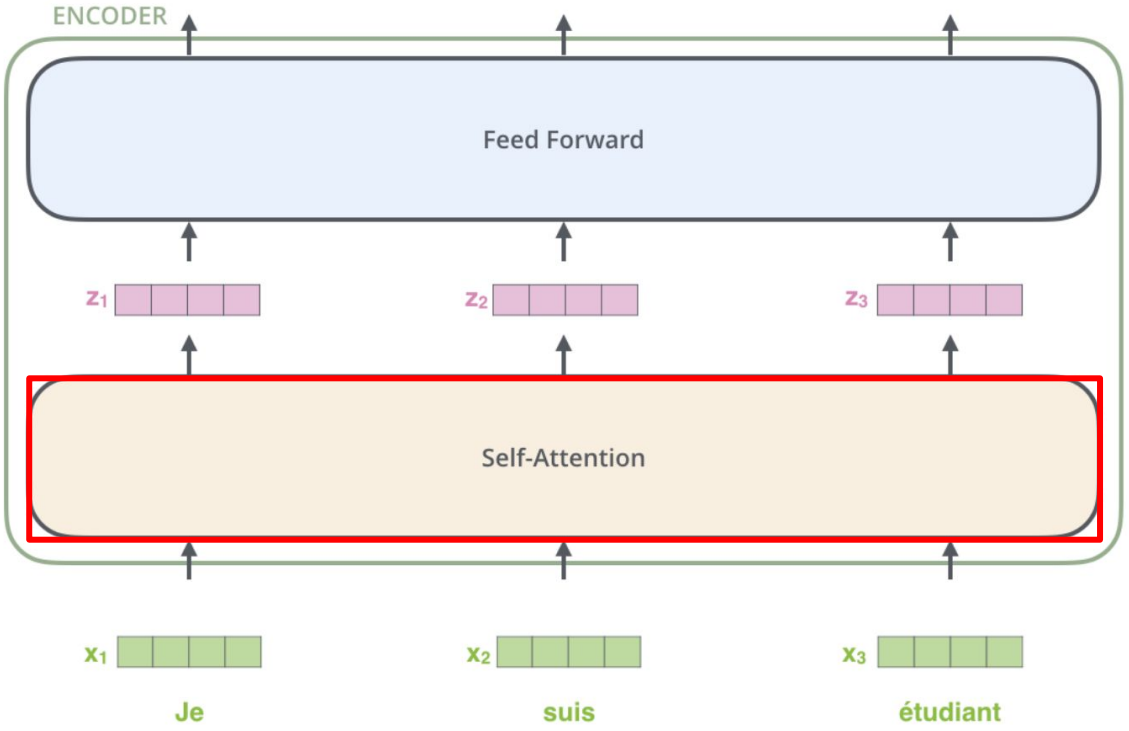


Step 1: Input Tokens → Token Embeddings



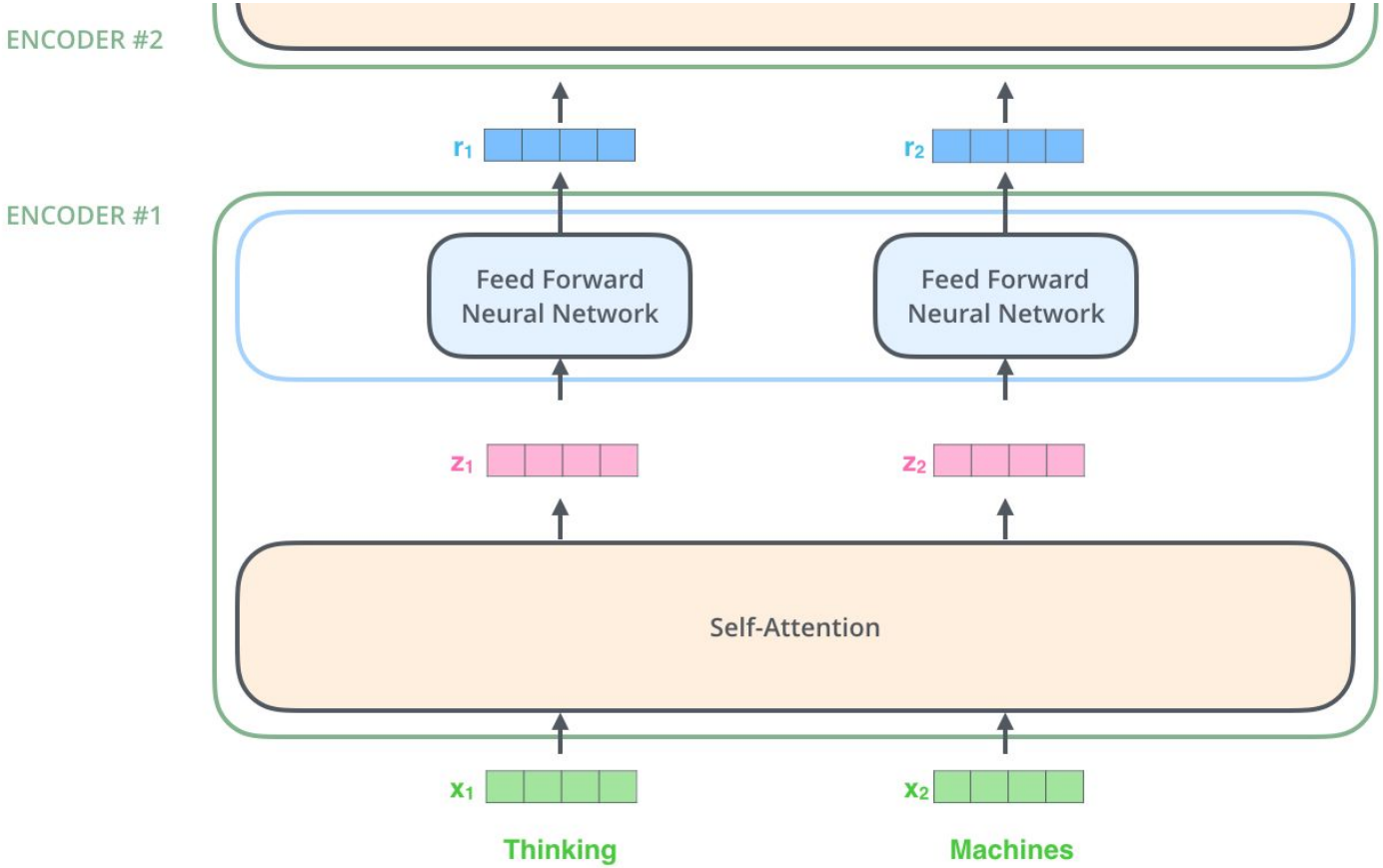
Will come back to the positional encoding later

Step 2: Token Embeddings → Attention Vectors

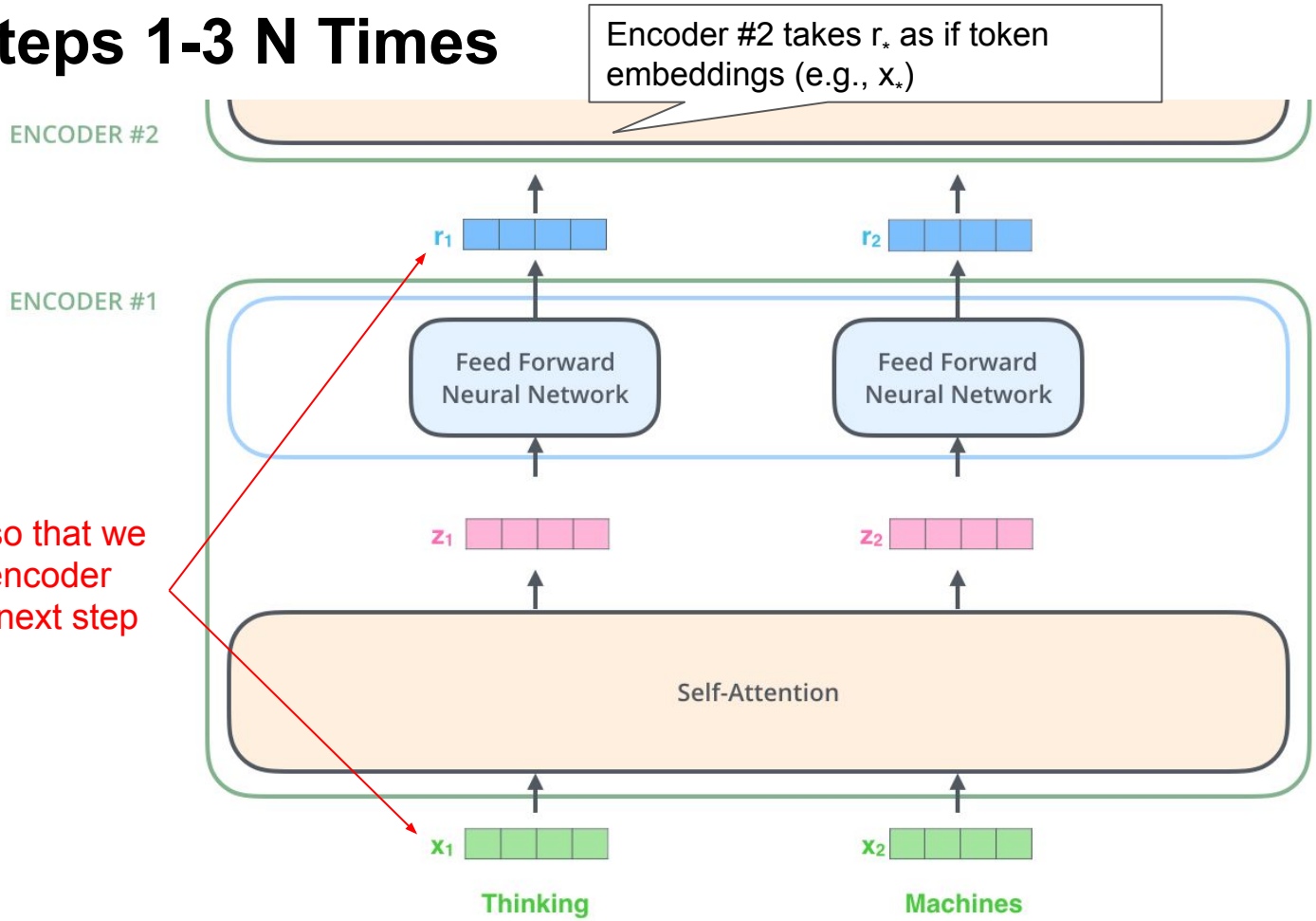


← Attention vectors

Step 3: Attention Vectors → Output Vectors



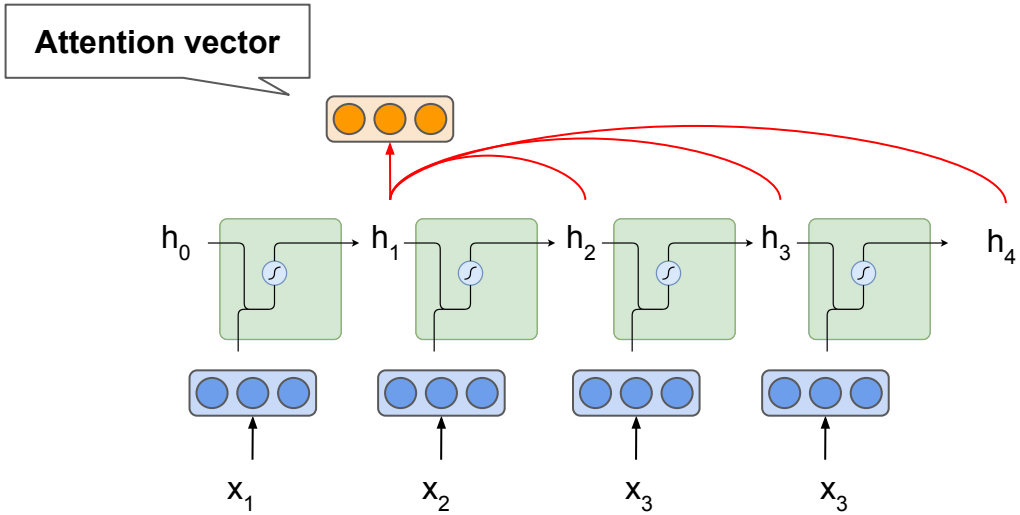
Repeat Steps 1-3 N Times



Same dimensions so that we can use the same encoder architecture as the next step

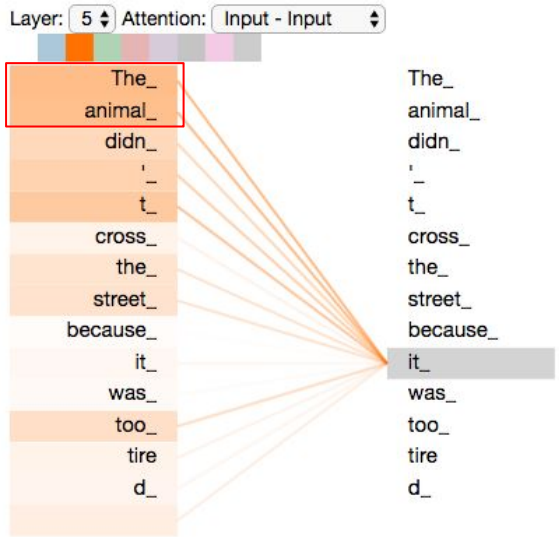
Self-Attention in Detail

Recap: Self-Attention for RNNs



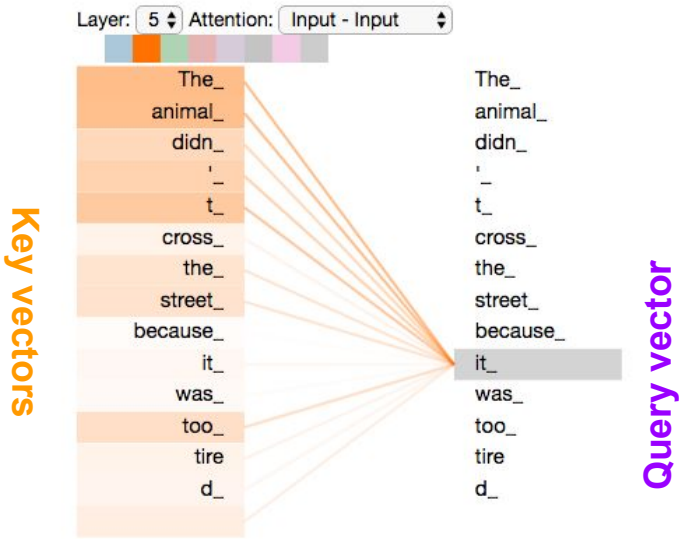
Self-Attention: Key Idea

- Self-Attention looks at **all tokens** in the input sequence to calculate **the target token embedding** based on the context
- Example: “The animal didn’t cross the street because **it** was too tired.”



Self-Attention: Key, Query, Value Vectors

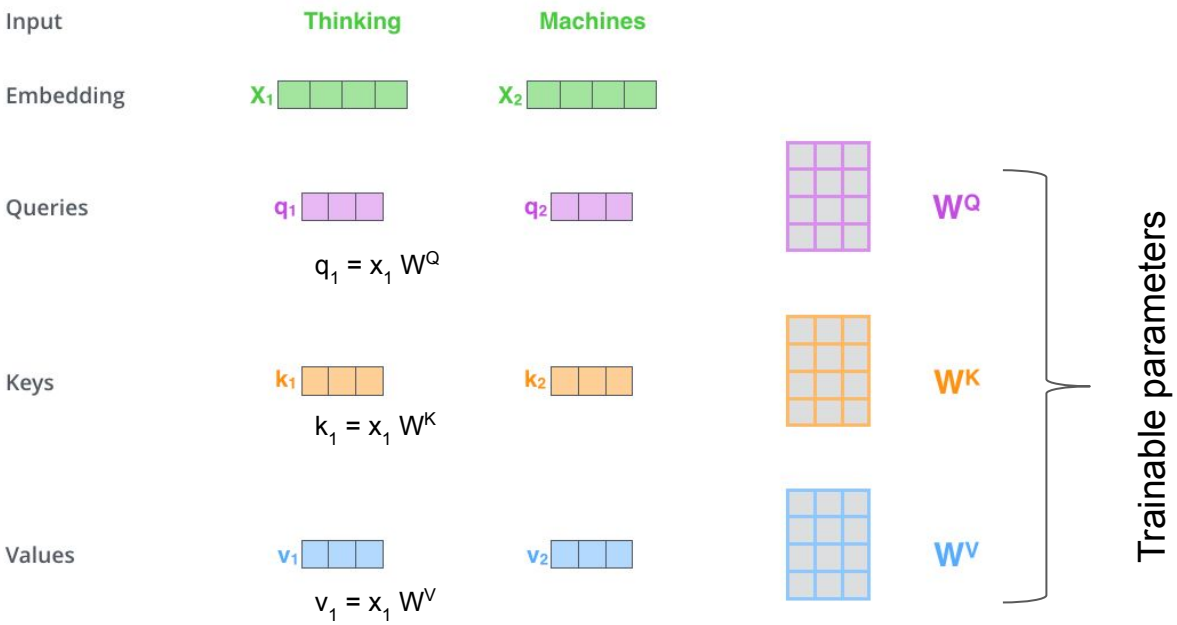
- The Self-Attention Layer encodes token embeddings by averaging **Value vectors** of input tokens based on the similarities between **Key vectors** of input tokens and **Query vector** of the target token



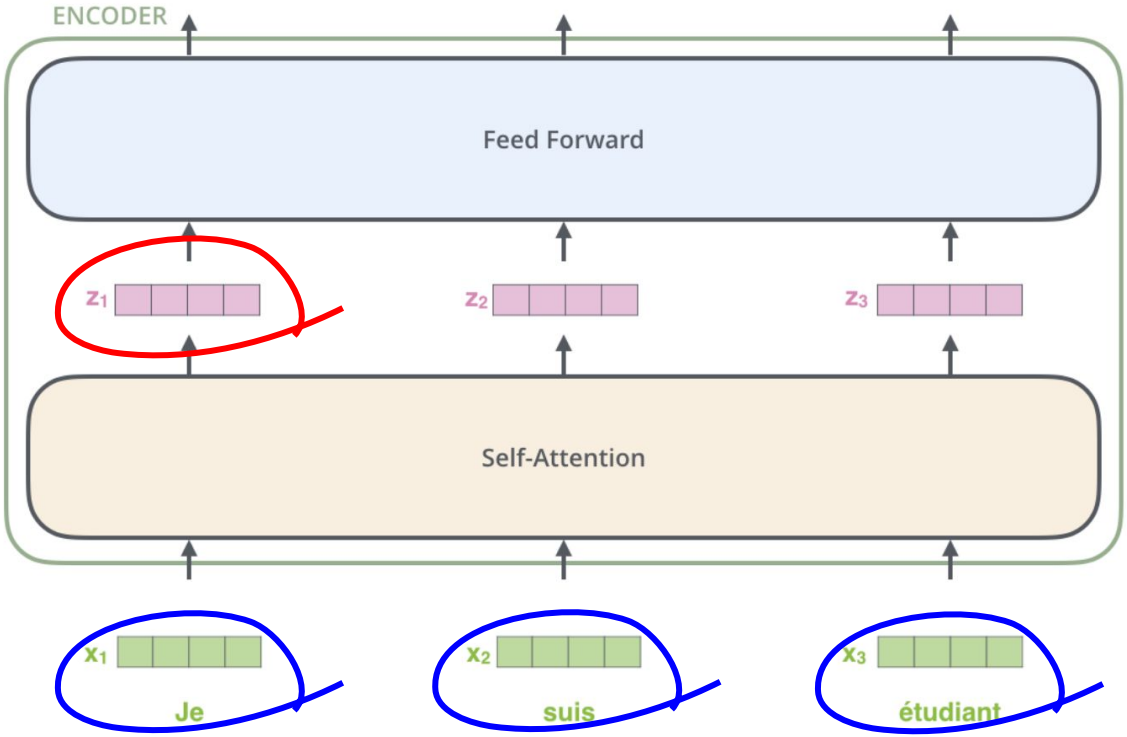
x **Value vectors** **= Attention vector**

Token Embeddings → Query, Key, Value Vectors

- Self-attention layer has **transformation matrices** for Query, Key, and Value vectors



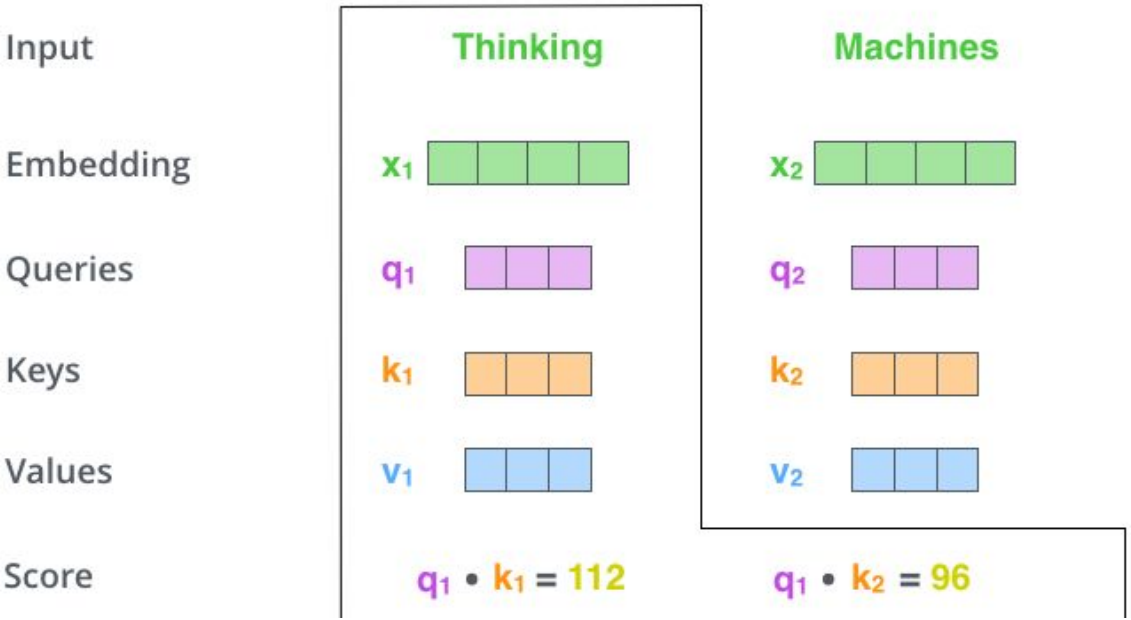
Calculating z_1 from Input Token Embeddings x_{1-3}



Example: Calculating Attention Vector for “Thinking”

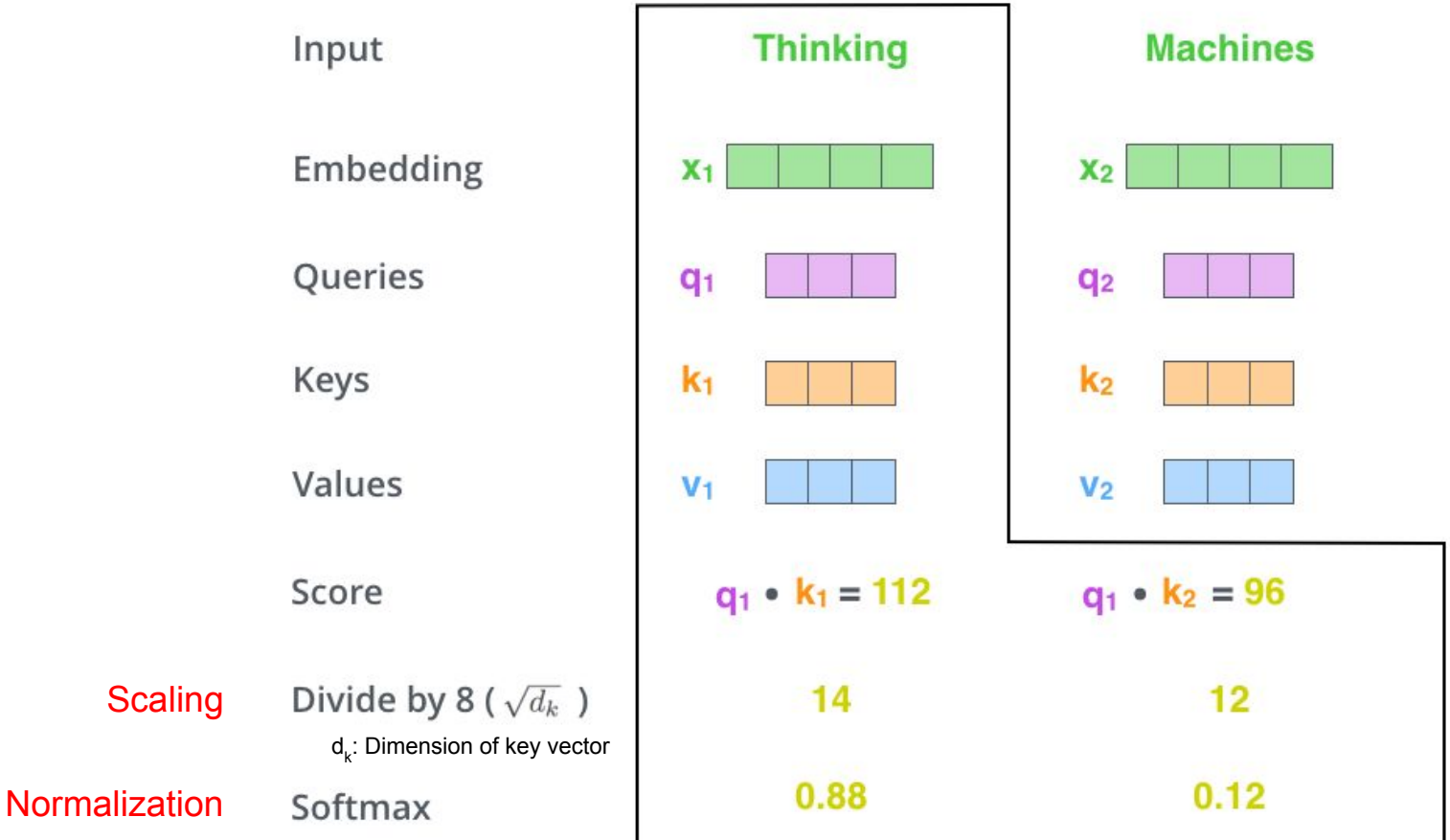
Step 1: Query-Key Similarity Calculation

- The inner product of q_1 and k_*



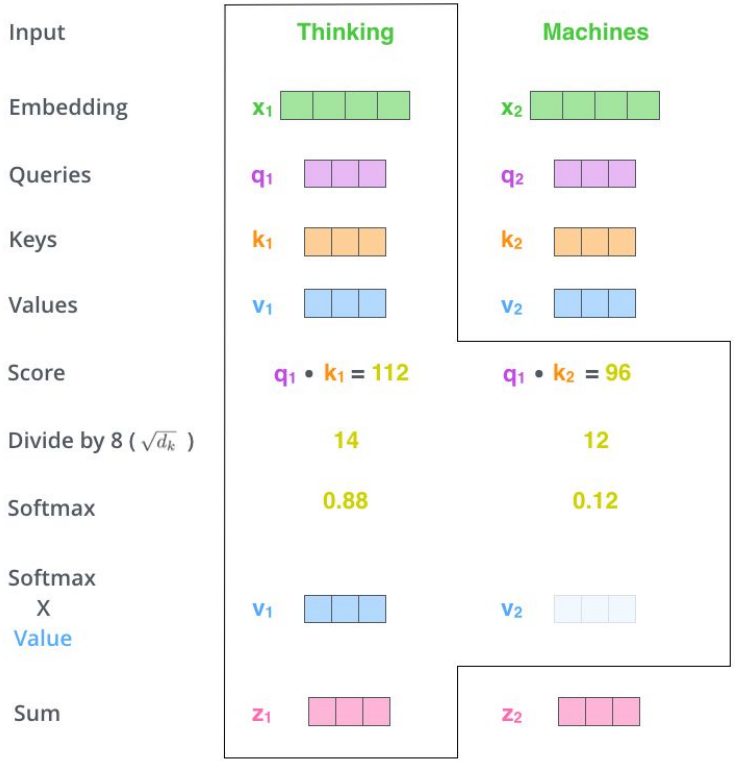
Example: Calculating Attention Vector for “Thinking”

Step 2: Scaling and normalization



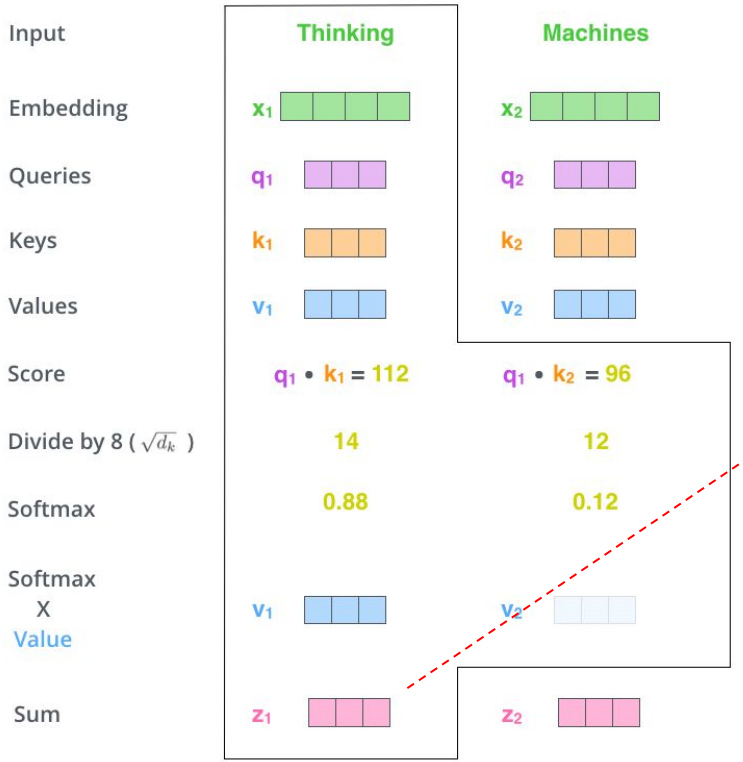
Example: Calculating Attention Vector for “Thinking”

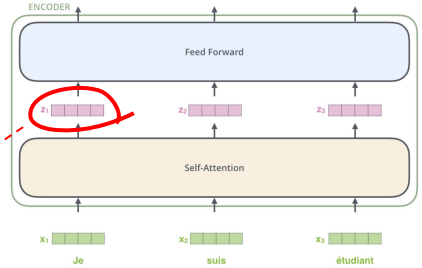
Step 3: Weighted Average of Value Vectors



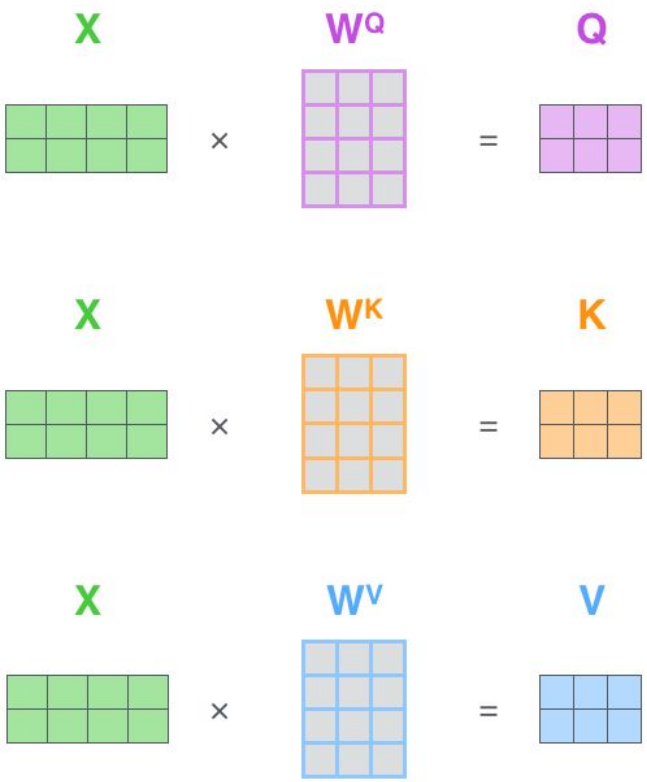
Example: Calculating Attention Vector for “Thinking”

Step 3: Weighted Average of Value Vectors



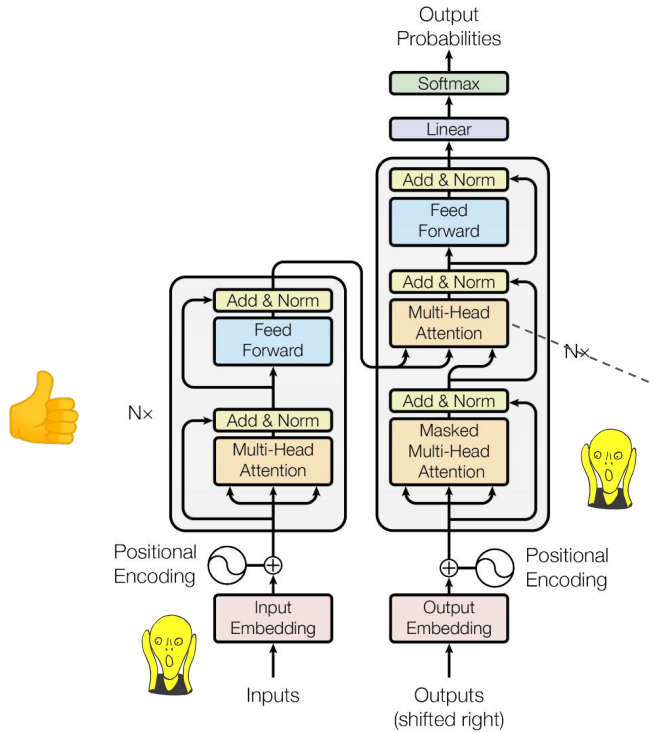


Understanding Self-Attention in Matrix Calculation

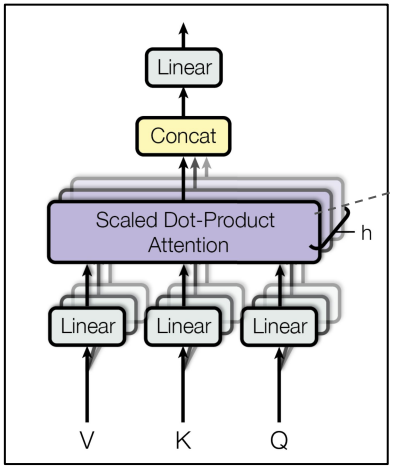


$$\text{softmax}\left(\frac{Q \times K^T}{\sqrt{d_k}}\right) V = Z$$

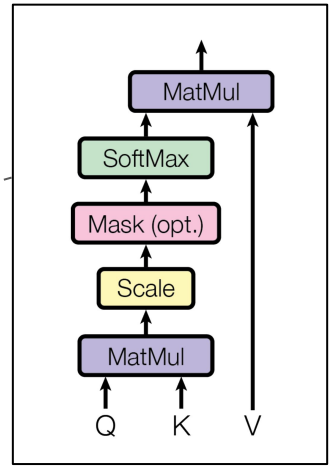
The Transformer



Model Architecture



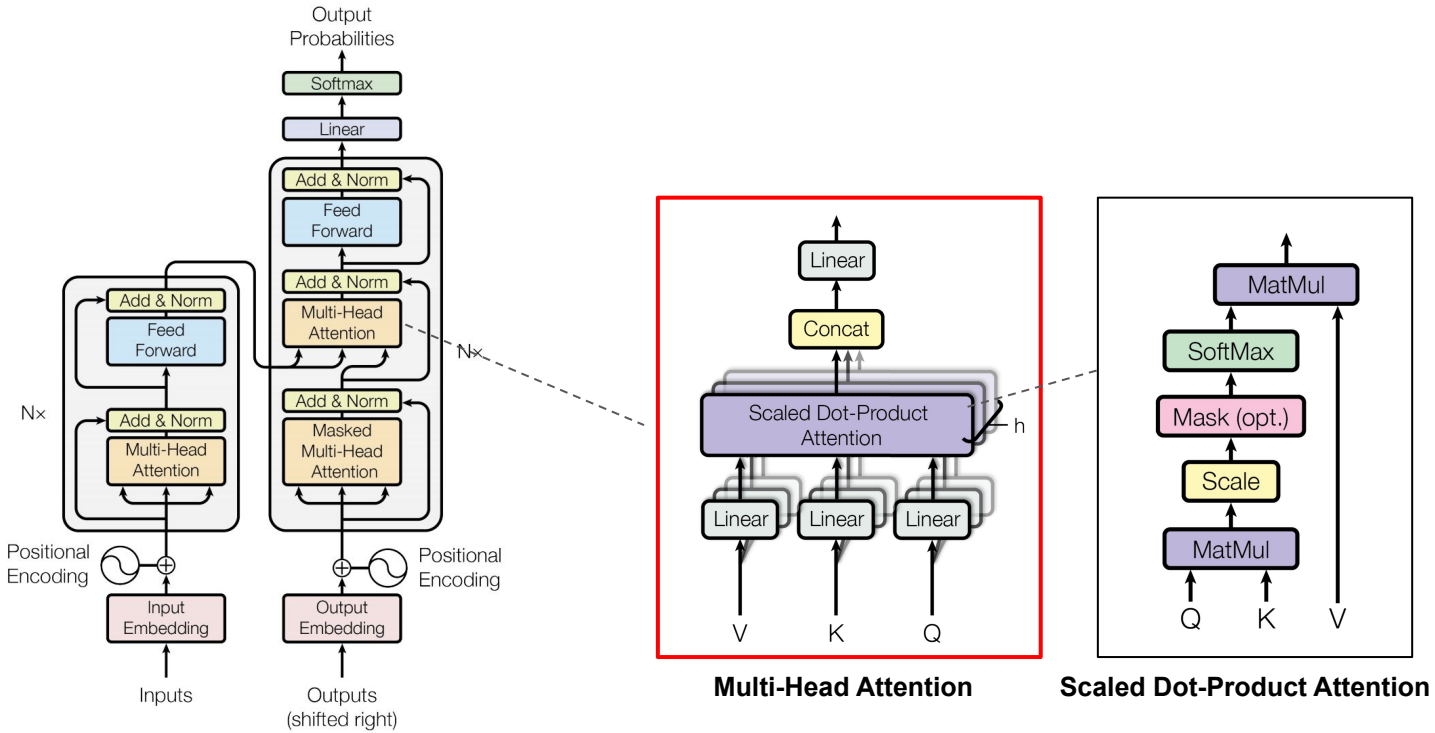
Multi-Head Attention



Scaled Dot-Product Attention

Multi-head Attention!

Multi-head Attention for Multiple K, Q, V Representations

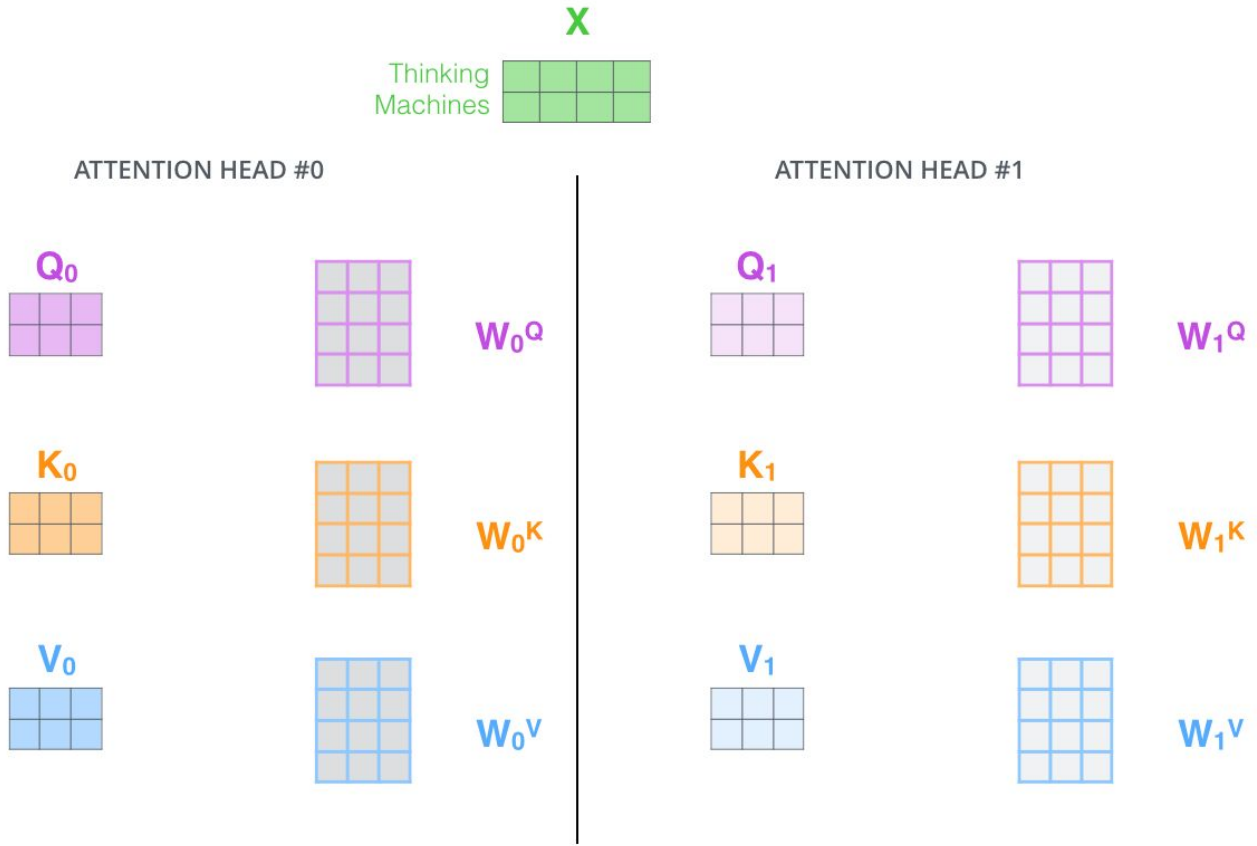


Model Architecture

Multi-Head Attention

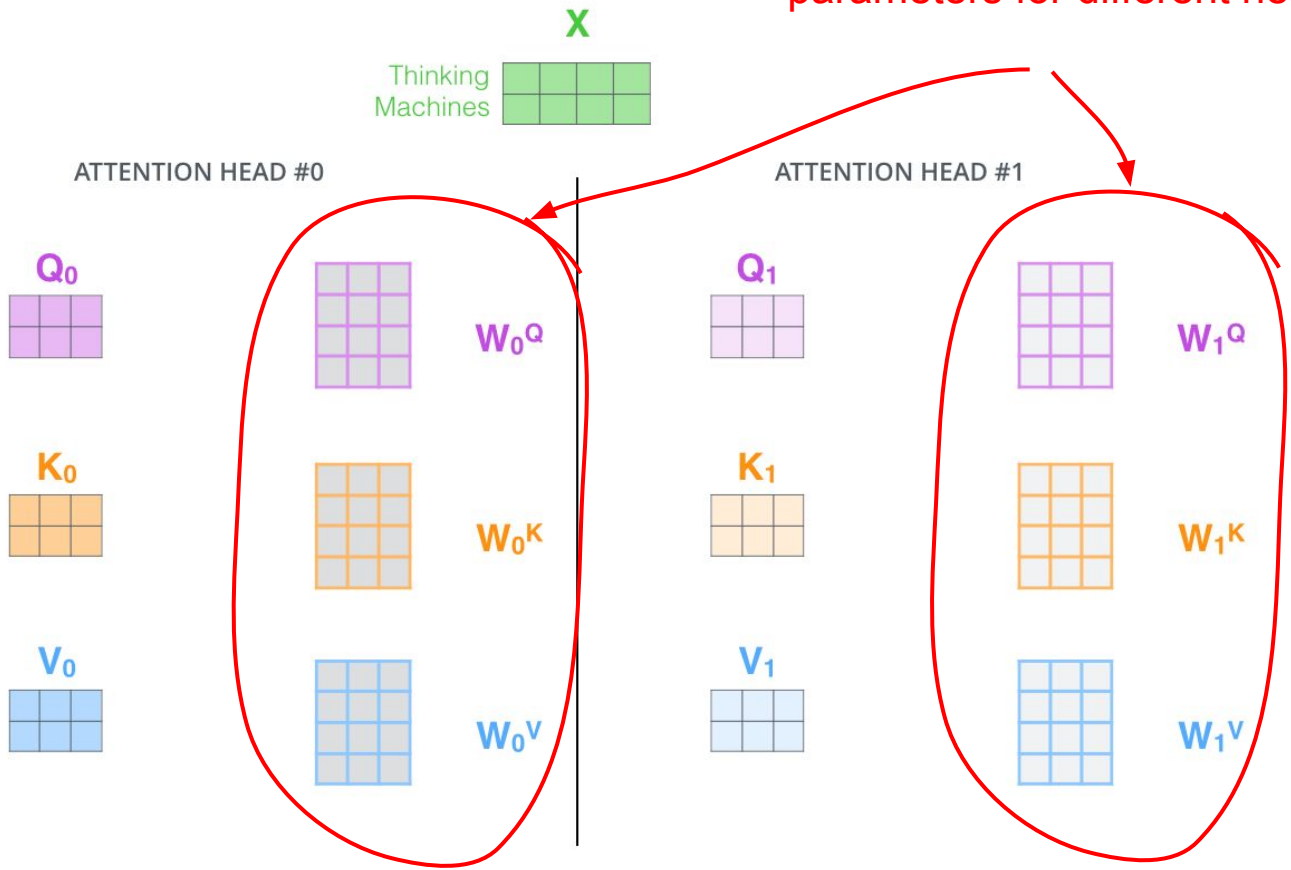
Scaled Dot-Product Attention

2-head Self-Attention

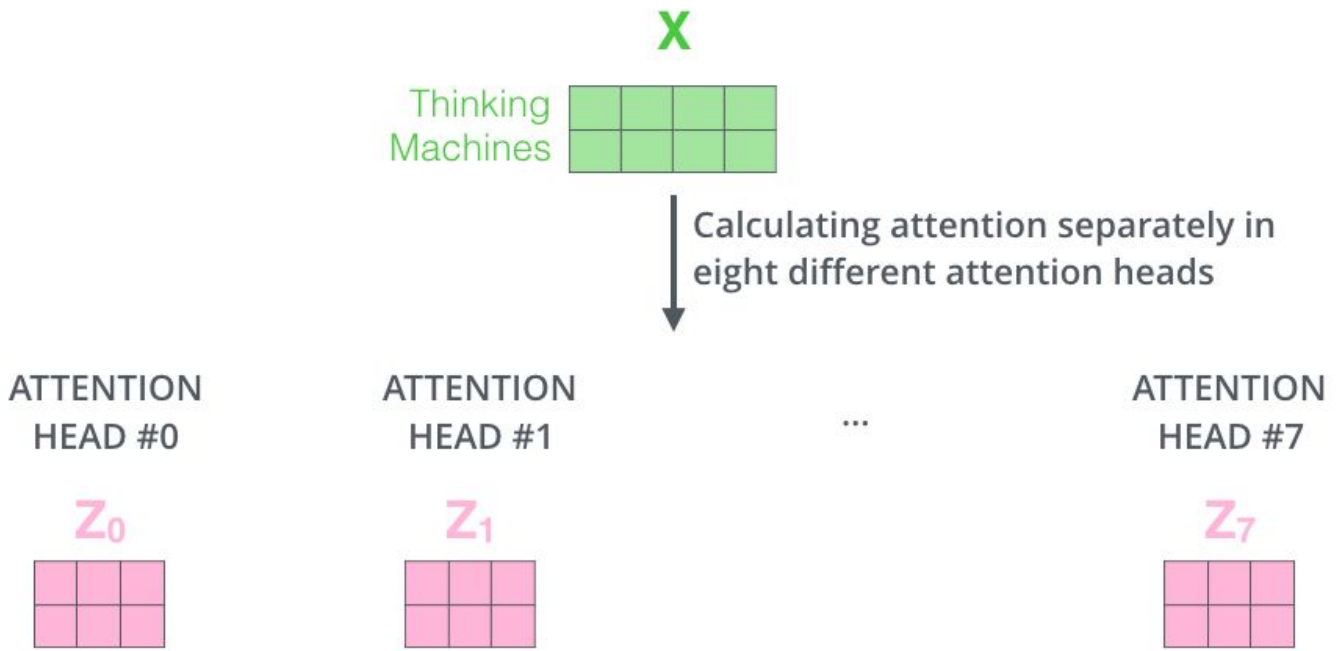


2-head Self-Attention

[Important!] Different trainable parameters for different heads

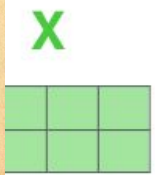


8-head Self-Attention = 8 Attention Vectors



8 Attention Vectors

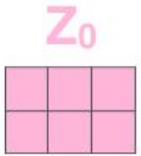
Yamata no orochi (やまたのおろち八岐大蛇)
A legendary 8-head 8-tail Japanese dragon
https://en.wikipedia.org/wiki/Yamata_no_Orochi

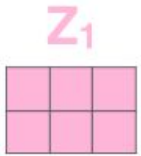


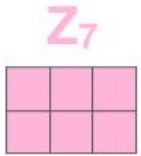
Calculating attention separately in eight different attention heads

...

ATTENTION HEAD #7







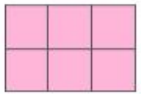
8

A

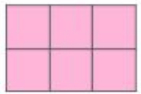
Yamata no Orochi (やまたのおろち 八岐大蛇)
A legendary 8-head 8-tail Japanese dragon
https://en.wikipedia.org/wiki/Yamata_no_Orochi

cf. Hydra (Greek myth)
Multi-head serpent
https://en.wikipedia.org/wiki/Lernaean_Hydra

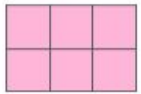
Z_0



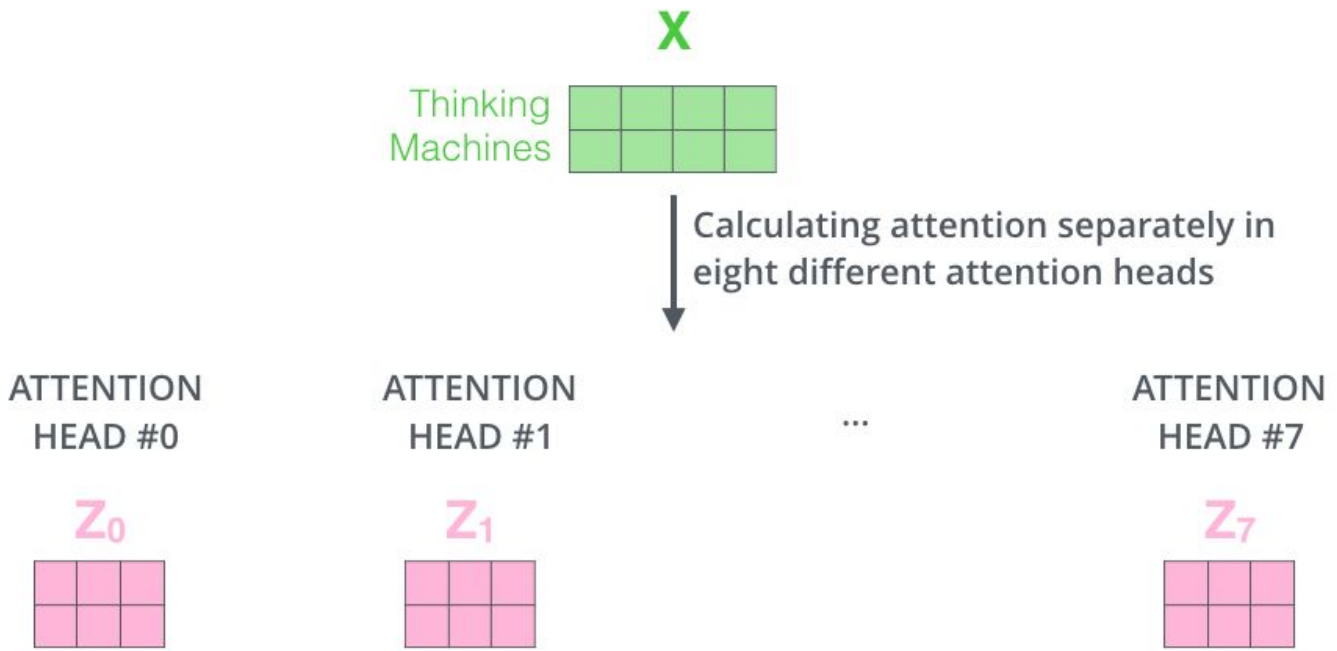
Z_1



Z_7



8-head Self-Attention = 8 Attention Vectors



Q. How do we aggregate them to get a single z_1?

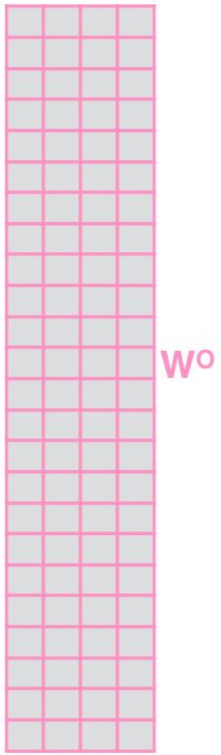
Concatenation & Linear-Transformation

1) Concatenate all the attention heads

(num_tokens, attention_dim*num_heads)

2) Multiply with a weight matrix W^O that was trained jointly with the model

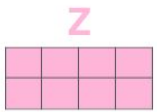
X



(num_heads*attention_dim, attention_dim)

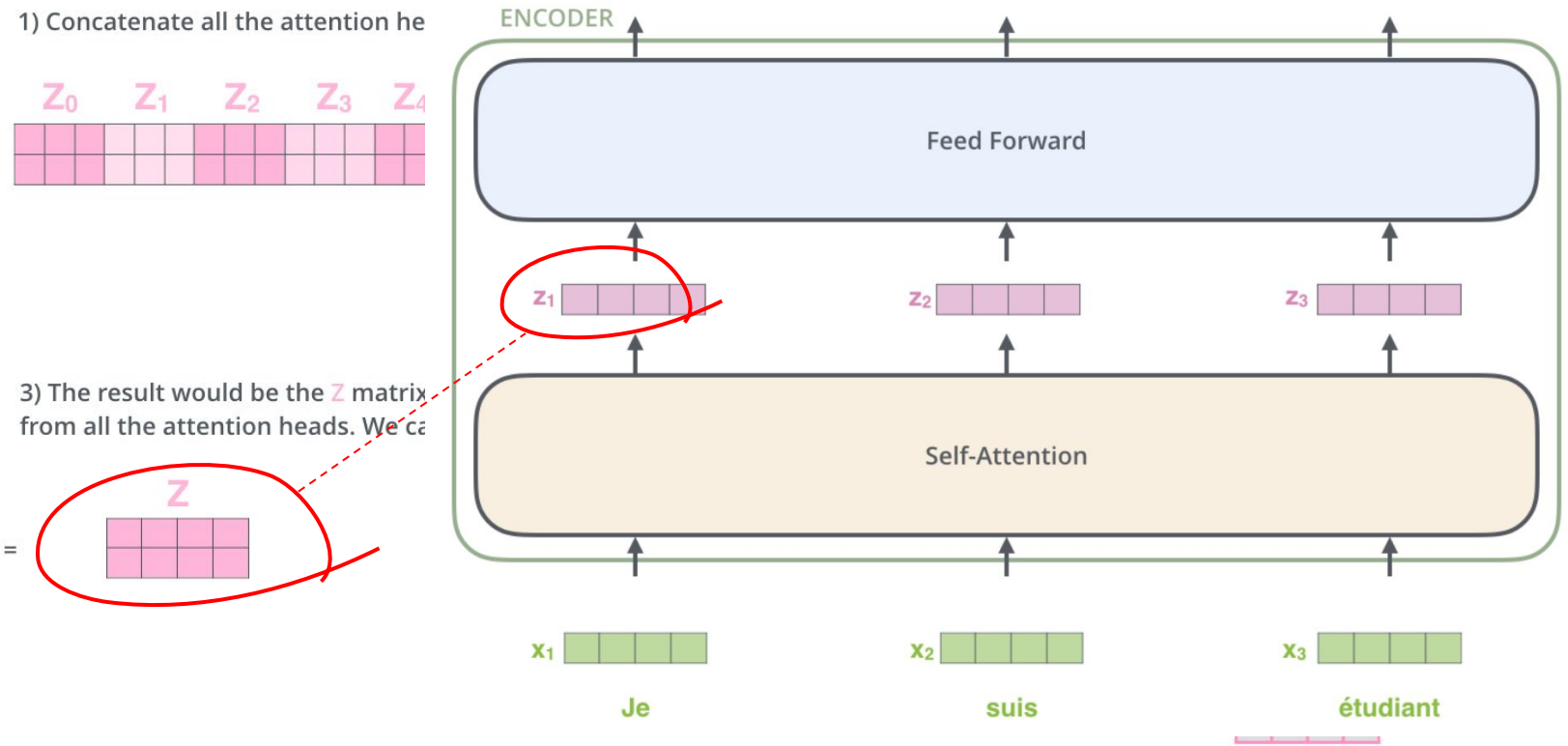
3) The result would be the Z matrix that captures information from all the attention heads. We can send this forward to the FFNN

=

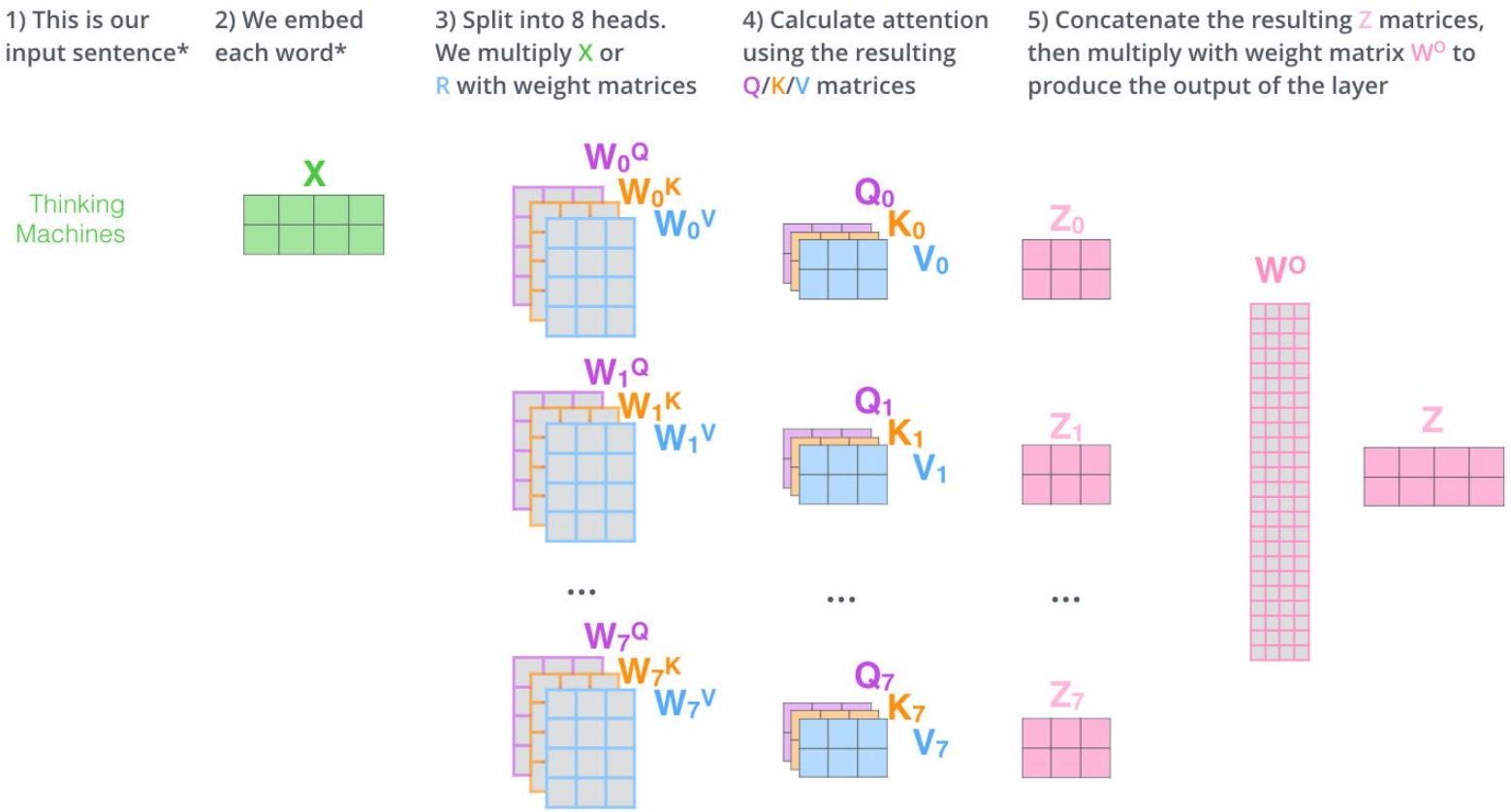


(num_tokens, attention_dim)

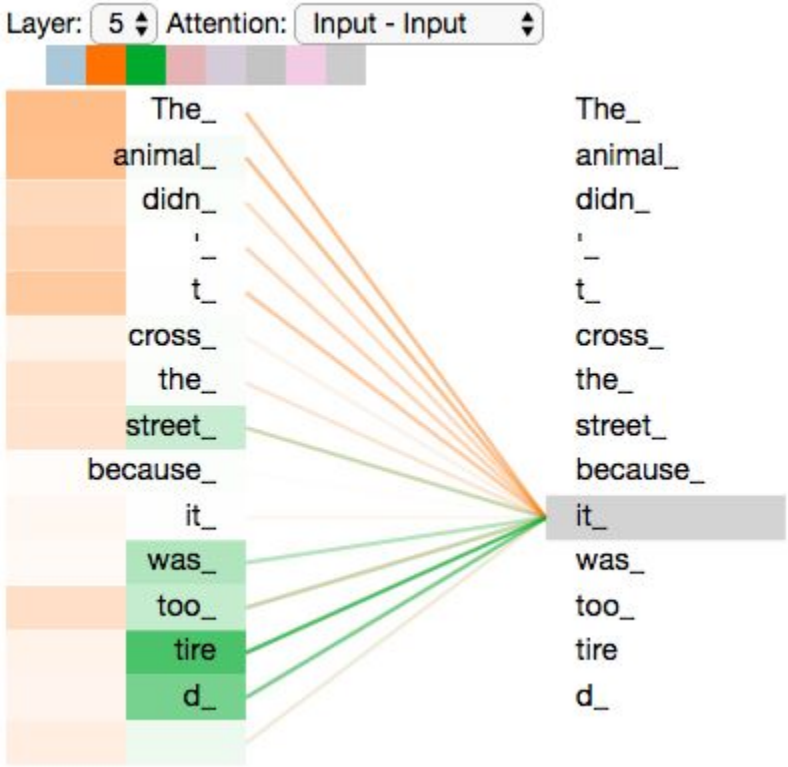
Concatenation & Linear-Transformation



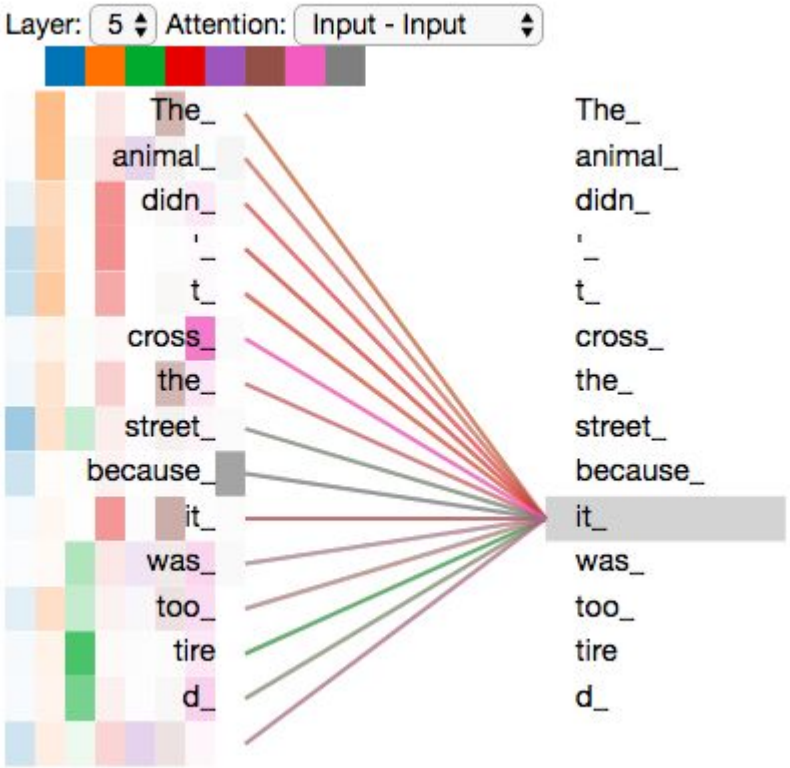
Recap: How Multi-head Self-Attention Works



In Theory: Different Heads Capture Different Types of Contextual Information



In Reality: Multiple Attention Heads are Hard to Interpret



???

Attention is not Explanation

Sarthak Jain
Northeastern University
jain.sar@husky.neu.edu

Byron C. Wallace
Northeastern University
b.wallace@northeastern.edu

Attention is not Explanation

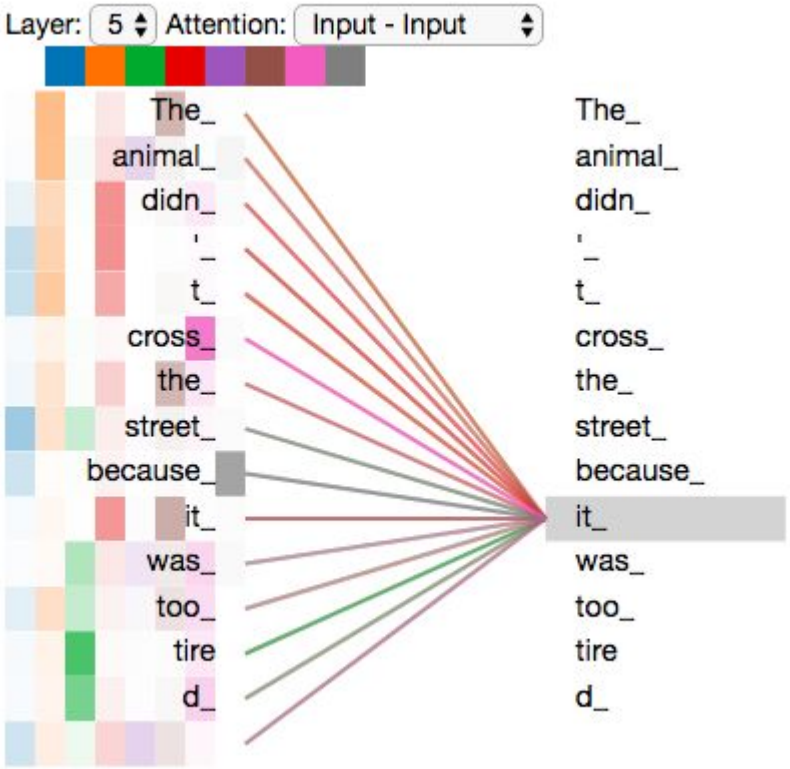
Sarthak Jain
Northeastern University
jain.sar@husky.neu.edu

Attention is not not Explanation

Sarah Wiegreffe*
School of Interactive Computing
Georgia Institute of Technology
saw@gatech.edu

Yuval Pinter*
School of Interactive Computing
Georgia Institute of Technology
uvp@gatech.edu

In Reality: Multiple Attention Heads are Hard to Interpret



???

Q. Do we really need so many attention heads?

Q. Do we really need so many attention heads?

- Short answer is **No**

ACL ‘19

Analyzing Multi-Head Self-Attention:
Specialized Heads Do the Heavy Lifting, the Rest Can Be Pruned

Elena Voita[1,2] **David Talbot**[1] **Fedor Moiseev**[1,5] **Rico Sennrich**[3,4] **Ivan Titov**[3,2]

[1]Yandex, Russia [2]University of Amsterdam, Netherlands
[3]University of Edinburgh, Scotland [4]University of Zurich, Switzerland
[5]Moscow Institute of Physics and Technology, Russia
{lena-voita, talbot, femoiseev}@yandex-team.ru
rico.sennrich@ed.ac.uk ititov@inf.ed.ac.uk

NeurIPS ‘19

Are Sixteen Heads Really Better than One?

Paul Michel
Language Technologies Institute
Carnegie Mellon University
Pittsburgh, PA
pmichel1@cs.cmu.edu

Omer Levy
Facebook Artificial Intelligence Research
Seattle, WA
omerlevy@fb.com

Graham Neubig
Language Technologies Institute
Carnegie Mellon University
Pittsburgh, PA
gneubig@cs.cmu.edu

BlackBoxNLP ‘19@ACL ‘19

What Does BERT Look At?
An Analysis of BERT’s Attention

Kevin Clark[†] **Urvashi Khandelwal**[†] **Omer Levy**[‡] **Christopher D. Manning**[†]
[†]Computer Science Department, Stanford University
[‡]Facebook AI Research
{kevclark,urvashik,manning}@cs.stanford.edu
omerlevy@fb.com

the simple weighted average. In this paper we make the surprising observation that even if models have been trained using multiple heads, in practice, a large percentage of attention heads can be removed at test time without significantly impacting performance. In fact, some layers can even be reduced to a single head. We further examine greedy algorithms for pruning down models, and

(My opinion)
We could tune the number of heads
But, overshooting may not be that harmful

Note: Conventional Attention Techniques for RNN

- To obtain **context vector c_t** as auxiliary information for the decoder

Similarity

$$\mathbf{c}_t = \sum_{i=1} \alpha_{t,i} \boldsymbol{h}_i \quad ; \text{Context vector for output } y_t$$

$$\alpha_{t,i} = \text{align}(y_t, x_i) \quad ; \text{How well two words } y_t \text{ and } x_i \text{ are aligned.}$$

$$= \frac{\exp(\text{score}(\boldsymbol{s}_{t-1}, \boldsymbol{h}_i))}{\sum_{i'=1}^{n} \exp(\text{score}(\boldsymbol{s}_{t-1}, \boldsymbol{h}_{i'}))} \quad ; \text{Softmax of some predefined alignment score..}$$

Attention? Attention! by Lilian Weng https://lilianweng.github.io/lil-log/2018/06/24/attention-attention.html

Note: Conventional Attention Techniques for RNN

- To obtain **context vector c_t** as auxiliary information for the decoder

Similarity

Value = hidden state of the encoder

$$\mathbf{c}_t = \sum_{i=1} \alpha_{t,i} \boldsymbol{h}_i \quad ; \text{Context vector for output } y_t$$

Query

Key

$$\alpha_{t,i} = \text{align}(y_t, \ldots) \quad ; \text{How well two words } y_t \text{ and } x_i \text{ are aligned.}$$

$$= \frac{\exp(\text{score}(\boldsymbol{s}_{t-1}, \boldsymbol{h}_i))}{\sum_{i'=1}^{n} \exp(\text{score}(\boldsymbol{s}_{t-1}, \boldsymbol{h}_{i'}))} \quad ; \text{Softmax of some predefined alignment score..}$$

Attention? Attention! by Lilian Weng https://lilianweng.github.io/lil-log/2018/06/24/attention-attention.html

Note: Various Score Functions

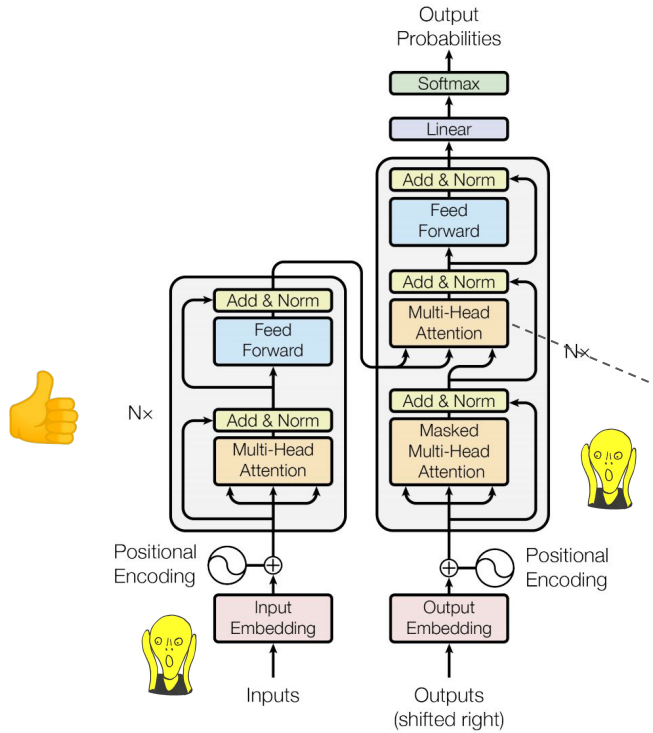
Query

Key

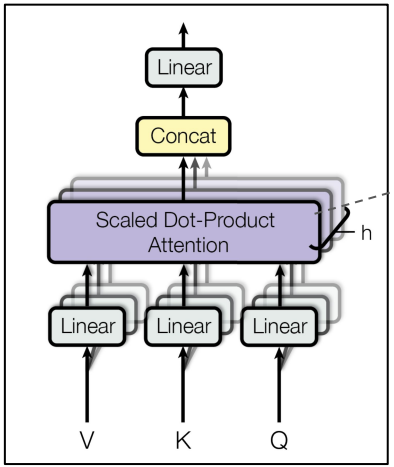
Name	Alignment score function	Citation
Content-base attention	$\text{score}(\boldsymbol{s}_t, \boldsymbol{h}_i) = \text{cosine}[\boldsymbol{s}_t, \boldsymbol{h}_i]$	Graves2014
Additive(*)	$\text{score}(\boldsymbol{s}_t, \boldsymbol{h}_i) = \mathbf{v}_a^\top \tanh(\mathbf{W}_a[\boldsymbol{s}_t; \boldsymbol{h}_i])$	Bahdanau2015
Location-Base	$\alpha_{t,i} = \text{softmax}(\mathbf{W}_a \boldsymbol{s}_t)$ Note: This simplifies the softmax alignment to only depend on the target position.	Luong2015
General	$\text{score}(\boldsymbol{s}_t, \boldsymbol{h}_i) = \boldsymbol{s}_t^\top \mathbf{W}_a \boldsymbol{h}_i$ where $\mathbf{W}_a$ is a trainable weight matrix in the attention layer.	Luong2015
Dot-Product	$\text{score}(\boldsymbol{s}_t, \boldsymbol{h}_i) = \boldsymbol{s}_t^\top \boldsymbol{h}_i$	Luong2015
Scaled Dot-Product(^)	$\text{score}(\boldsymbol{s}_t, \boldsymbol{h}_i) = \frac{\boldsymbol{s}_t^\top \boldsymbol{h}_i}{\sqrt{n}}$ Note: very similar to the dot-product attention except for a scaling factor; where n is the dimension of the source hidden state.	Vaswani2017

Attention? Attention! by Lilian Weng https://lilianweng.github.io/lil-log/2018/06/24/attention-attention.html

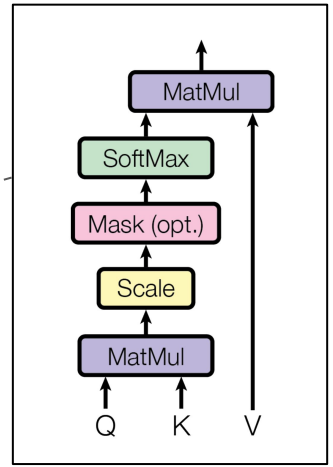
The Transformer



Model Architecture



Multi-Head Attention



Scaled Dot-Product Attention

Remaining Topics

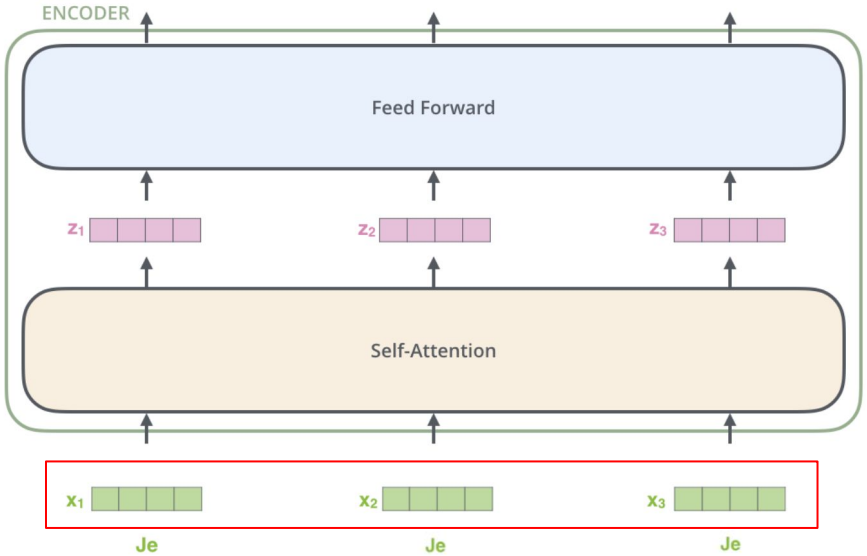
- Positional Encoding
- Residual Connection & Layer normalization
- Transformer Decoder

Remaining Topics

- **Positional encoding**
- **Residual Connection & Layer normalization**
- Transformer Decoder

Why Positional Encoding?

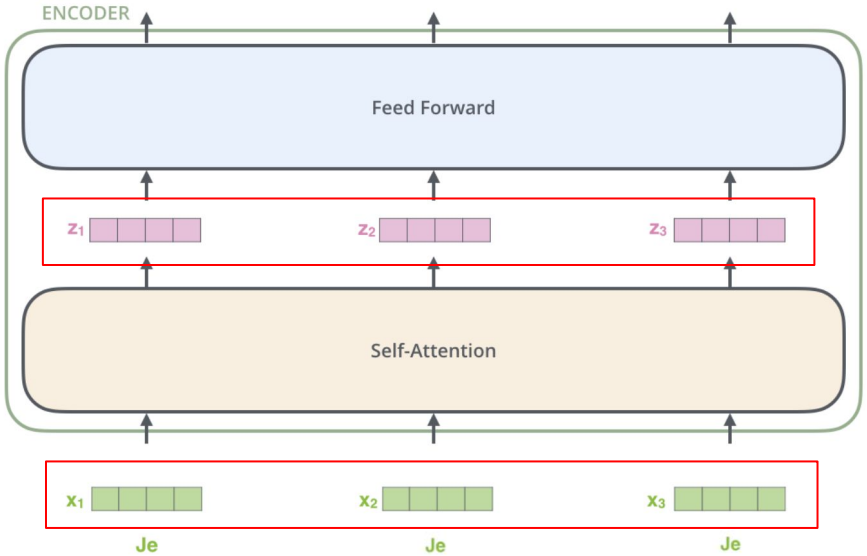
- Transformer Encoder (i.e., self-attention) does **NOT consider any positional information**



If the input token embeddings were the same

Why Positional Encoding?

- Transformer Encoder (i.e., self-attention) does **NOT consider any positional information**
  - In one sense, it can be considered as **"contextualized" bag-of-words** representations

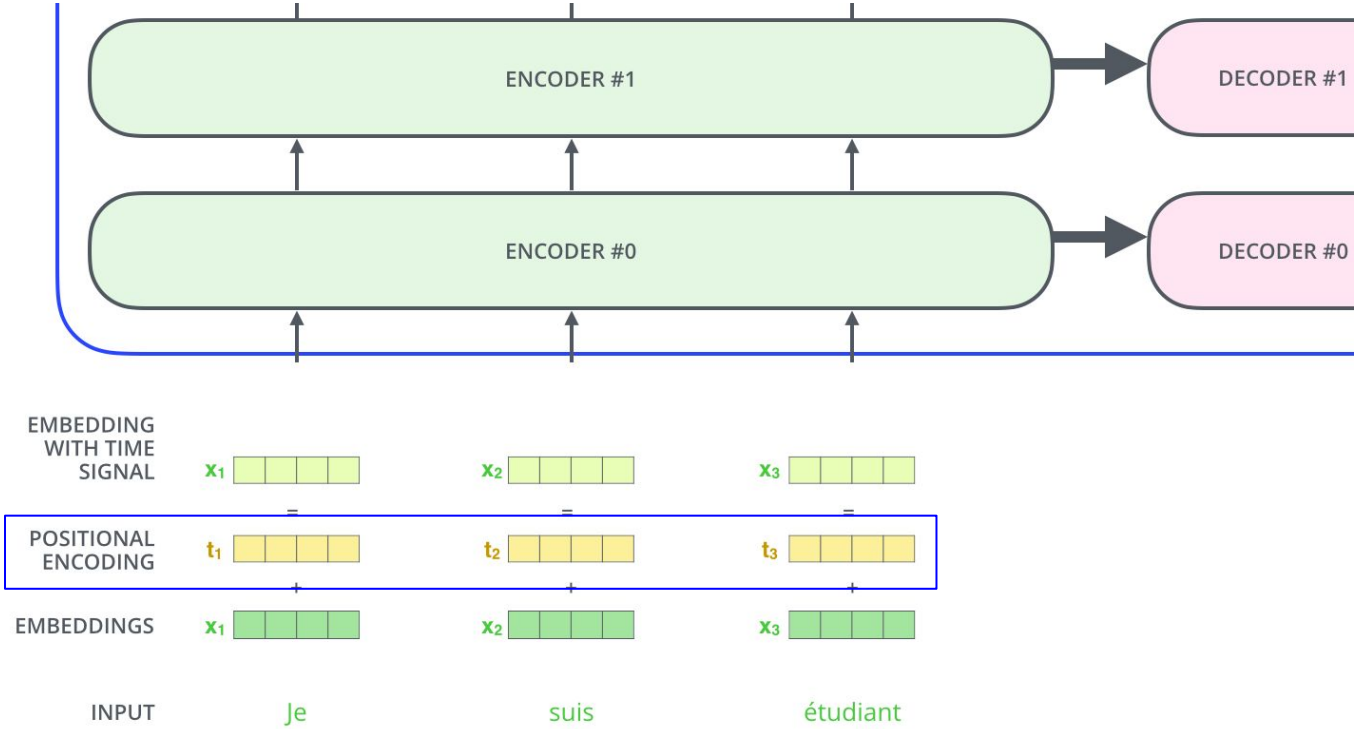


The corresponding attention vectors would become identical

If the input token embeddings were the same

Adding Positional Encoding to Make Position-aware

- Just adding **position information** to input word embeddings

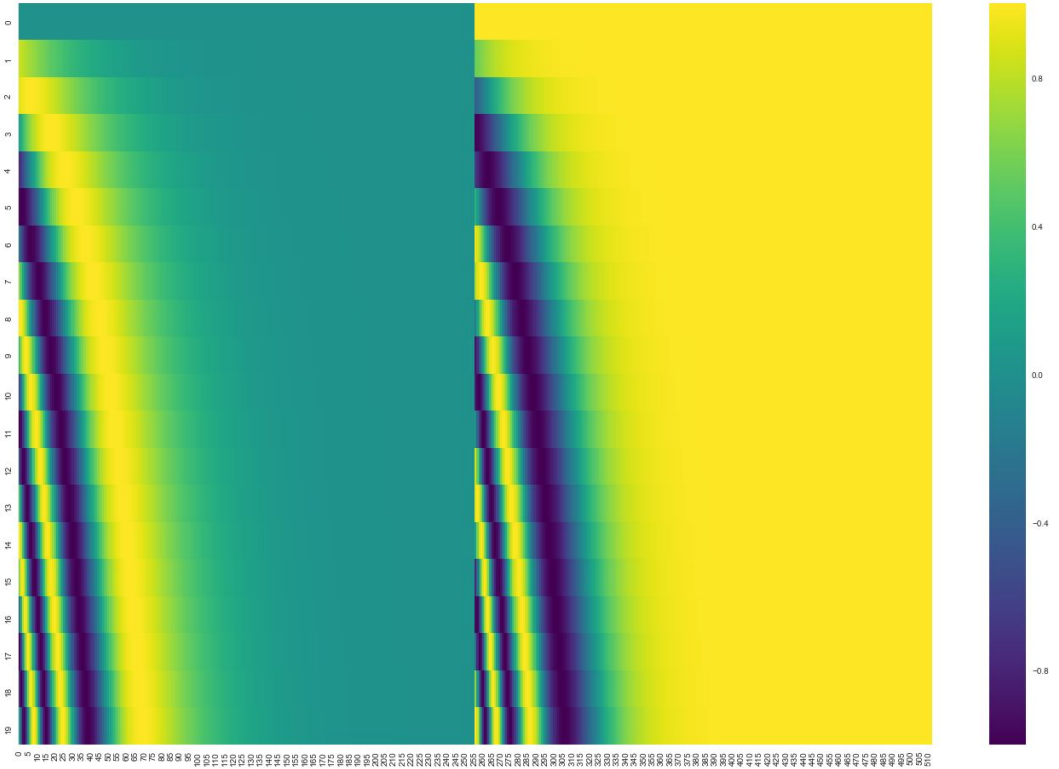


Positional Encoding in Transformer

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

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

How Actual Positional Encoding Values Look Like
20 tokens & embedding size of 512



Input Embedding Layer: Pytorch Example

```python
class TransformerModel(nn.Module):

    def __init__(self, ntoken: int, d_model: int, nhead: int, d_hid: int,
                 nlayers: int, dropout: float = 0.5):
        super().__init__()
        self.model_type = 'Transformer'
        self.pos_encoder = PositionalEncoding(d_model, dropout)
        encoder_layers = TransformerEncoderLayer(d_model, nhead, d_hid, dropout)
        self.transformer_encoder = TransformerEncoder(encoder_layers, nlayers)
        self.encoder = nn.Embedding(ntoken, d_model)
        self.d_model = d_model
        self.decoder = nn.Linear(d_model, ntoken)

        self.init_weights()
```

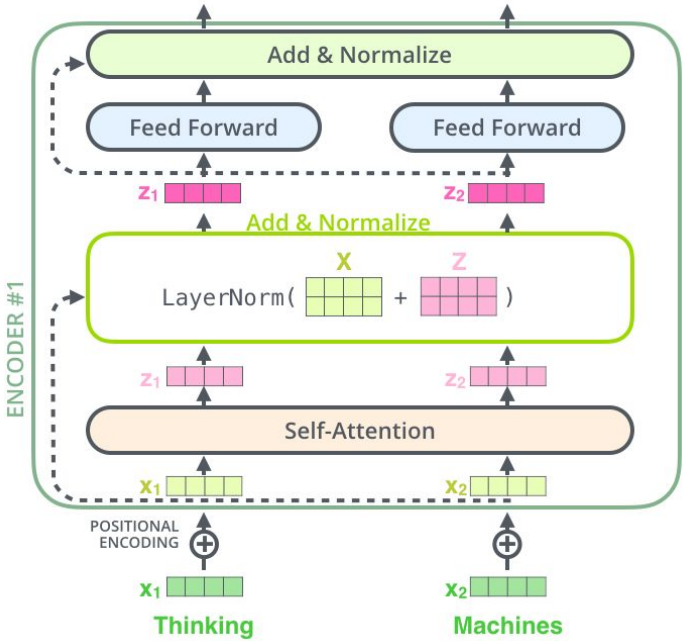
```python
    def forward(self, src: Tensor, src_mask: Tensor) -> Tensor:
        """
        Args:
            src: Tensor, shape [seq_len, batch_size]
            src_mask: Tensor, shape [seq_len, seq_len]

        Returns:
            output Tensor of shape [seq_len, batch_size, ntoken]
        """
        src = self.encoder(src) * math.sqrt(self.d_model)
        src = self.pos_encoder(src)
        output = self.transformer_encoder(src, src_mask)
```

```python
class PositionalEncoding(nn.Module):

    def __init__(self, d_model: int, dropout: float = 0.1, max_len: int = 5000):
        super().__init__()
        self.dropout = nn.Dropout(p=dropout)

        position = torch.arange(max_len).unsqueeze(1)
        div_term = torch.exp(torch.arange(0, d_model, 2) * (-math.log(10000.0) / d_model))
        pe = torch.zeros(max_len, 1, d_model)
        pe[:, 0, 0::2] = torch.sin(position * div_term)
        pe[:, 0, 1::2] = torch.cos(position * div_term)
        self.register_buffer('pe', pe)

    def forward(self, x: Tensor) -> Tensor:
        """
        Args:
            x: Tensor, shape [seq_len, batch_size, embedding_dim]
        """
        x = x + self.pe[:x.size(0)]
        return self.dropout(x)
```

[Language Modeling with nn.Transformer and TorchText — PyTorch Tutorials 1.9.1+cu102 documentation]

Residual Connection & Layer Normalization

- Apply **Residual Connection** (i.e., adding $\mathbf{x}_*$) before **Layer Normalization** (*1)



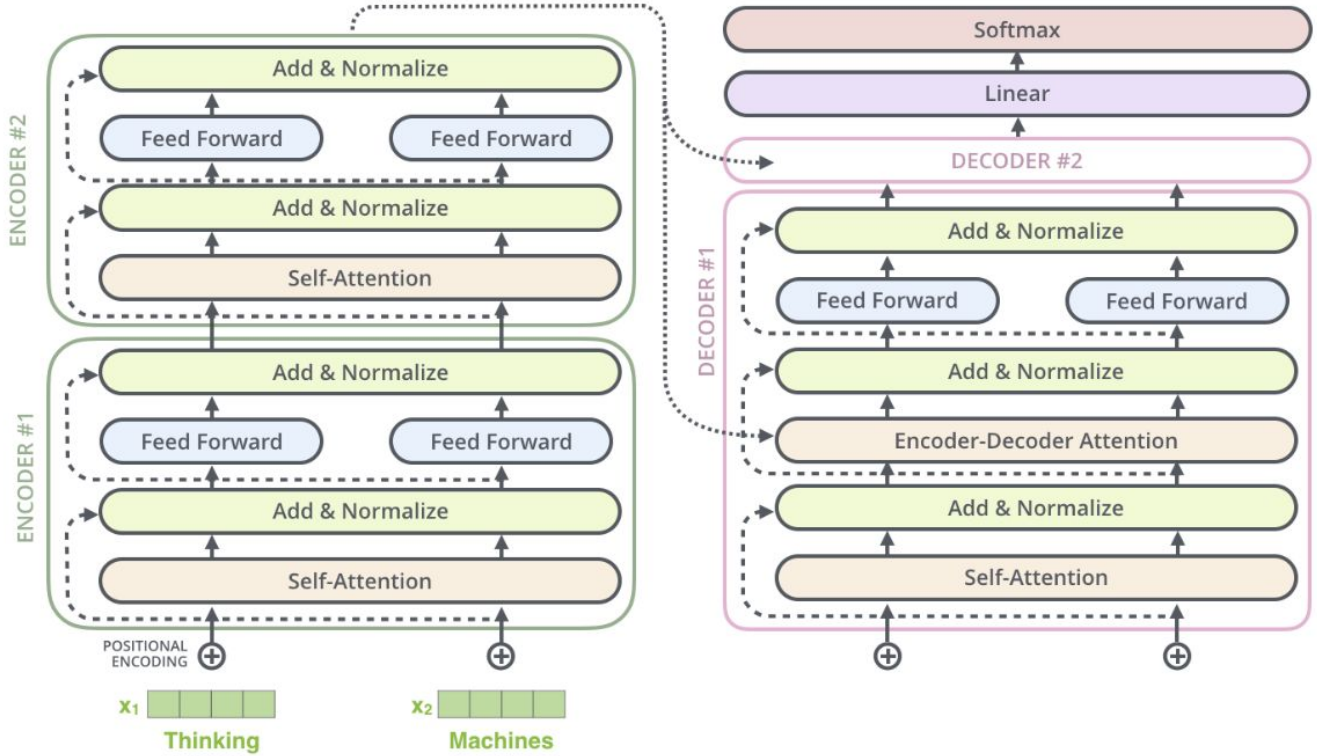
(*1) Layer normalization: Normalization over input tokens (cf. Batch Normalization)

Questions?

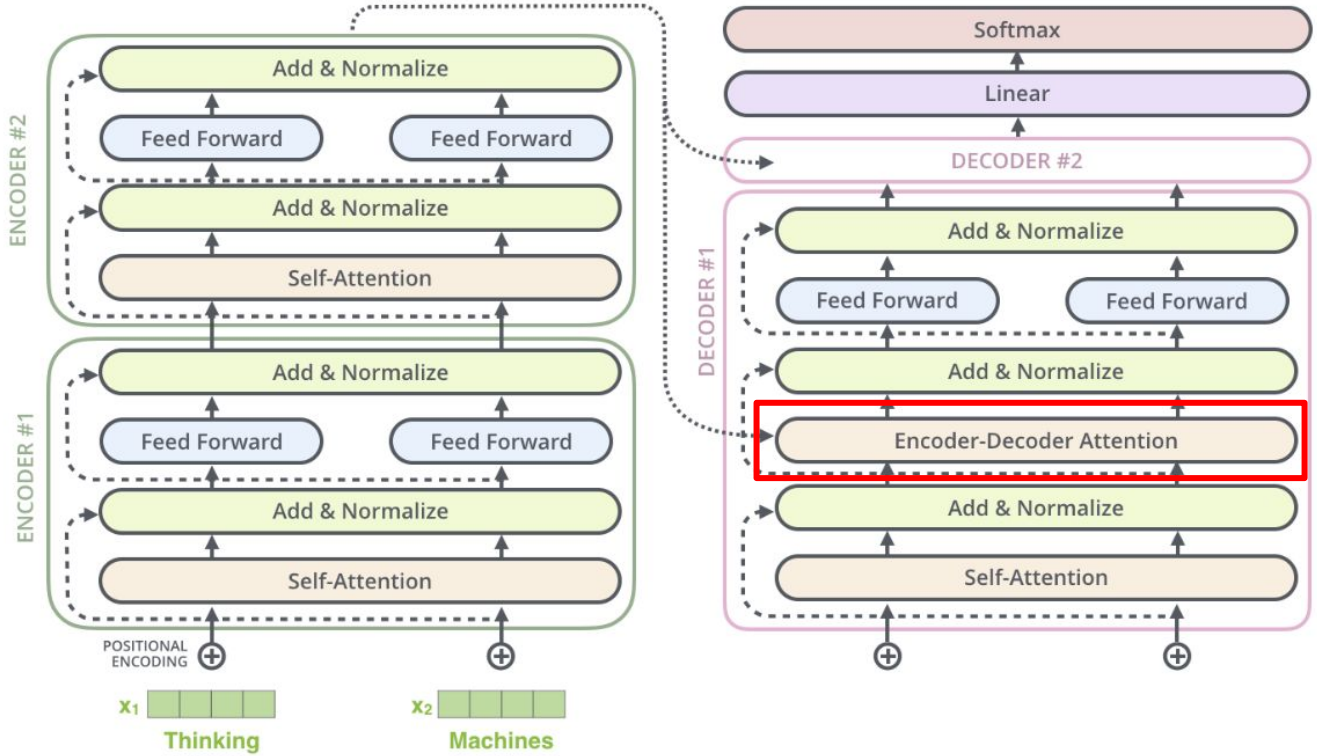
Remaining Topics

- Positional encoding
- Residual Connection & Layer normalization
- **Transformer Decoder**

Transformer (2 Encoders & 2 Decoders)

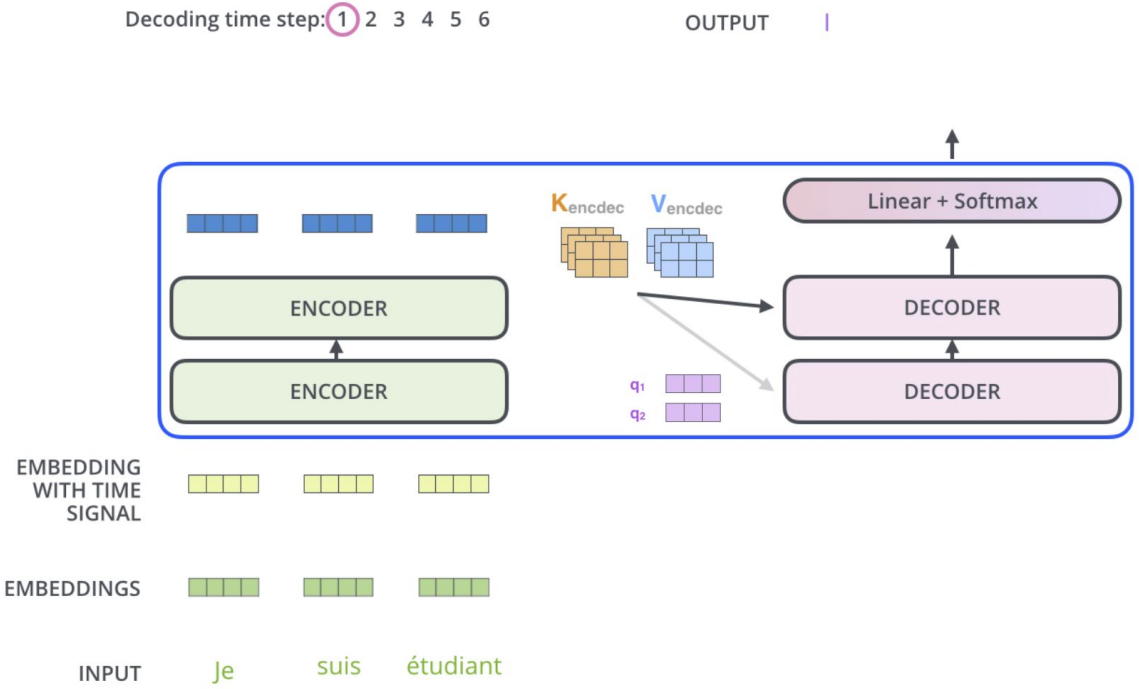


Transformer (2 Encoders & 2 Decoders)



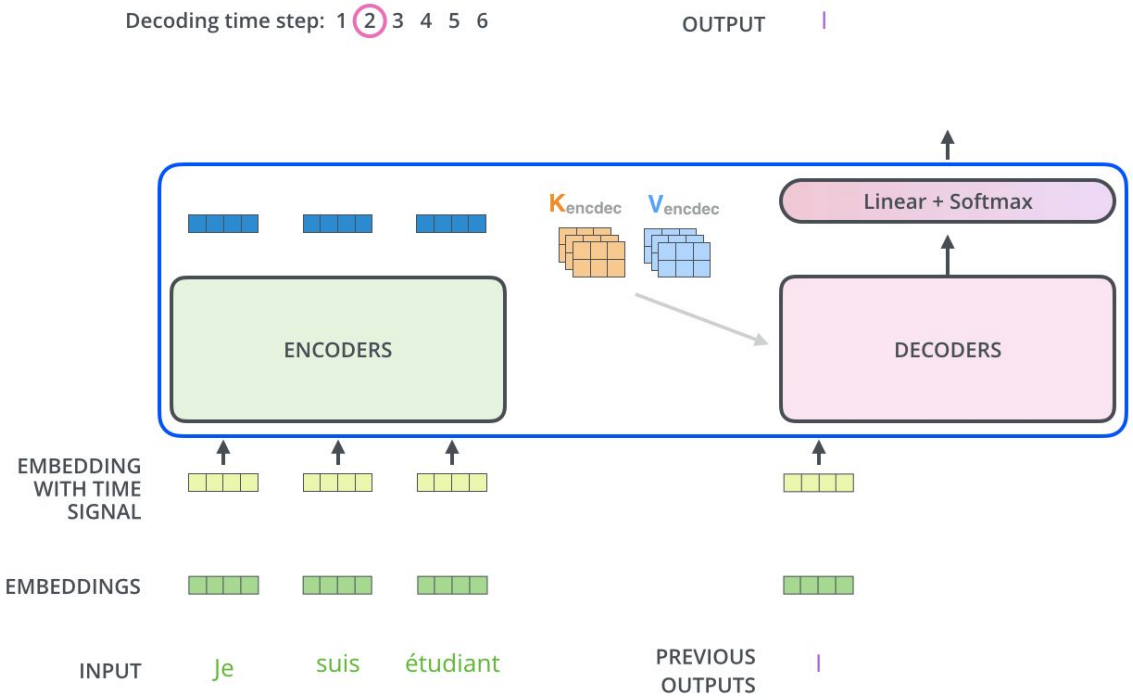
Decoder: Encoder-Decoder Attention

- The same “Query-Key-Value” style attention where **Key** and **Value vectors** are from Encoder and **Query vectors** are from Decoder

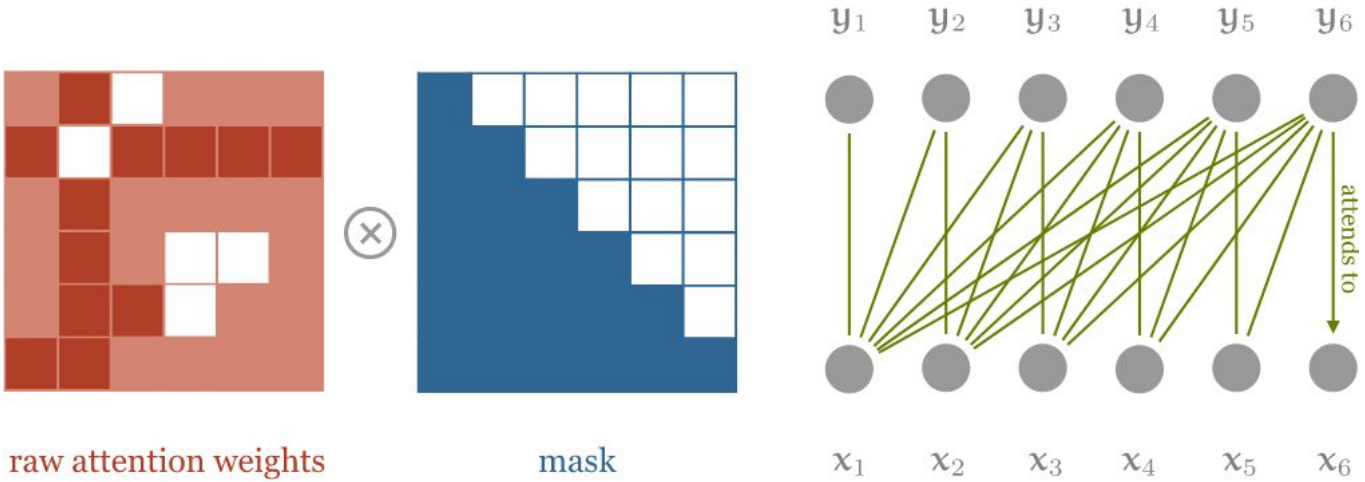


Decoder: “Masked” Self-Attention

- The self-attention layer (in the Decoder) is **only allowed to attend to earlier positions** in the output sequence



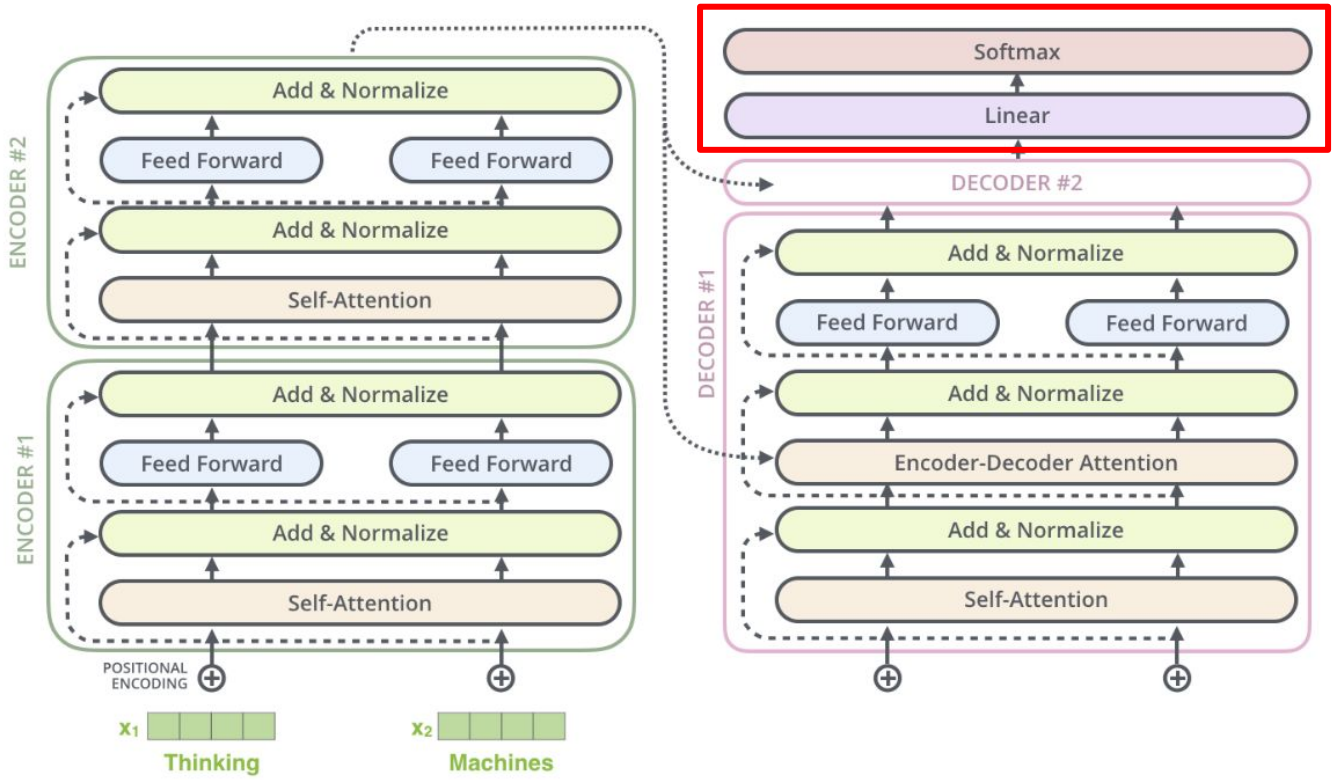
Masked Attention to Prevent “Previous” Step From Attending “Future” Steps



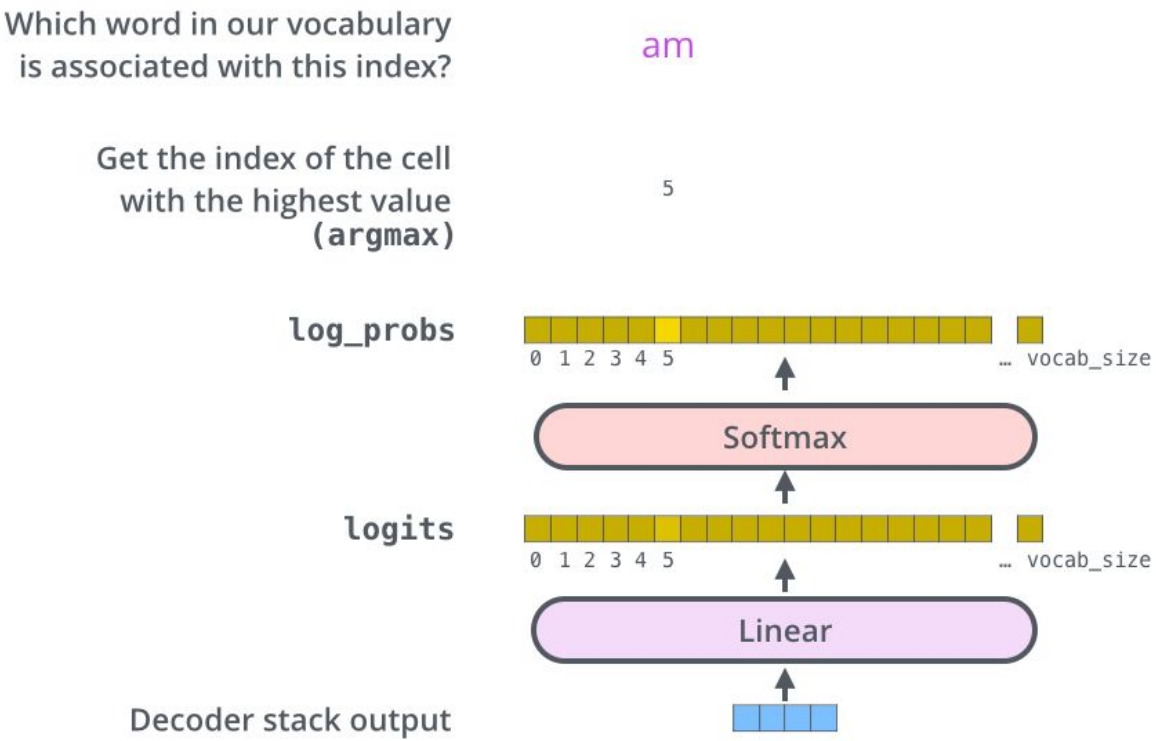
Decoder

[Transformers from scratch | peterbloem.nl]

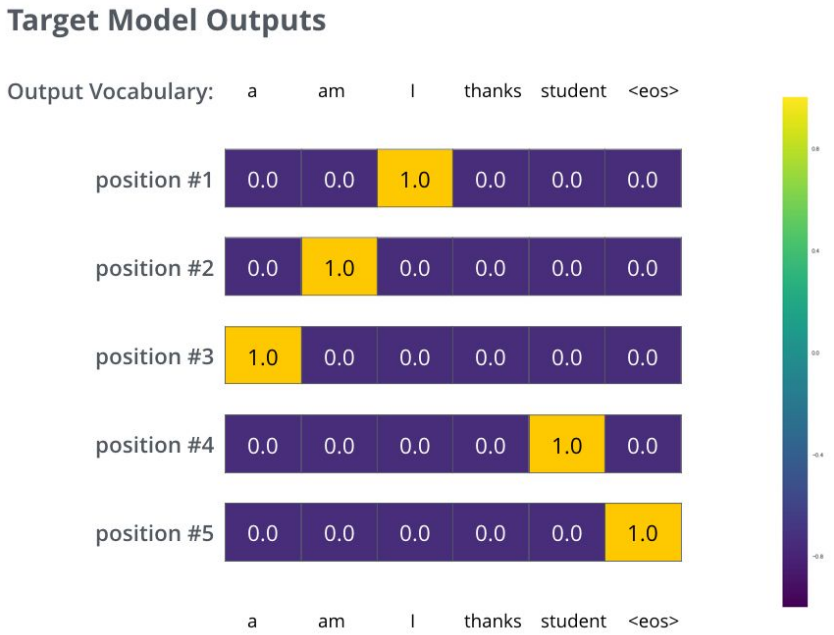
Transformer (2 Encoders & 2 Decoders)

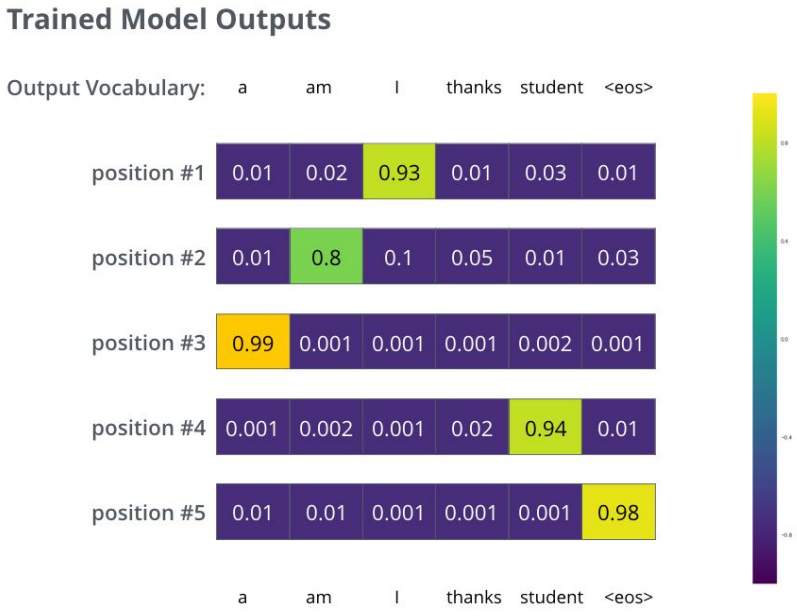


Output Layer



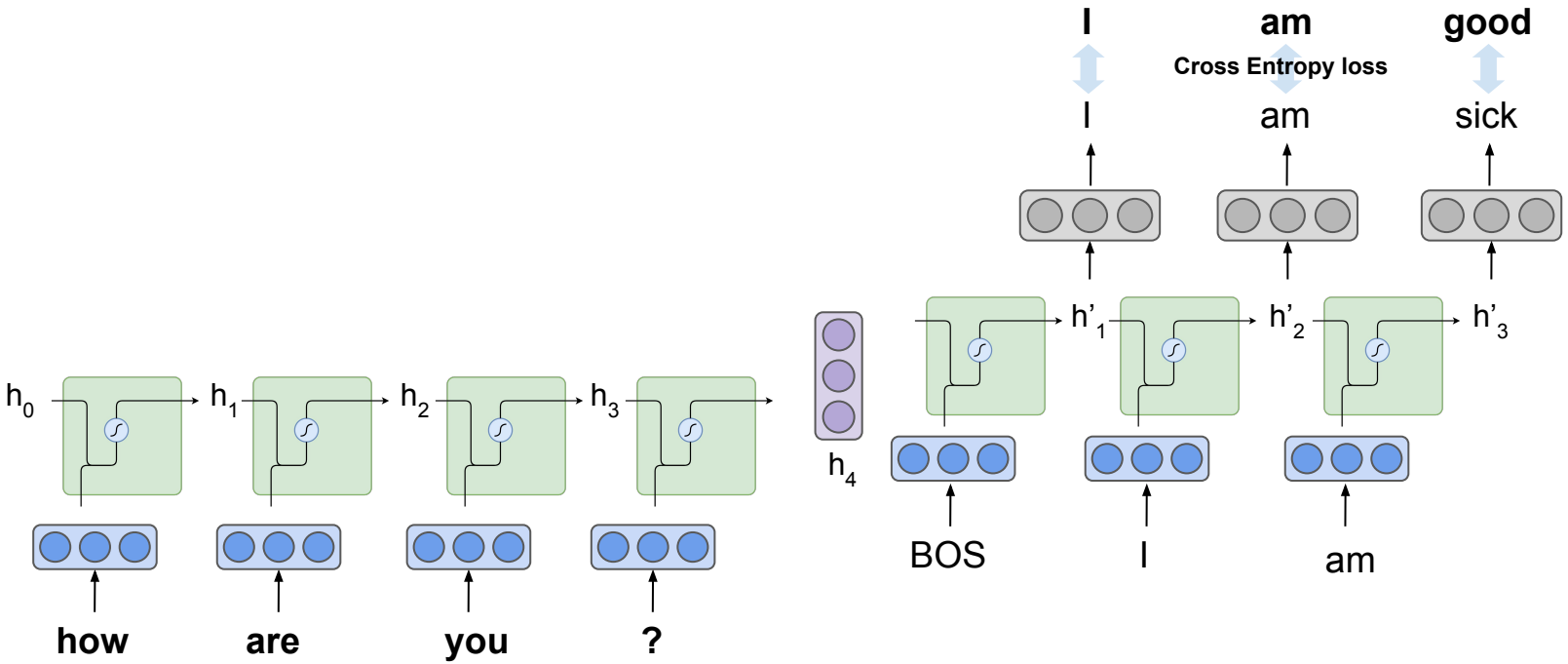
Output Layer: Training with Ground-truth Labels aka Teacher-forcing Training





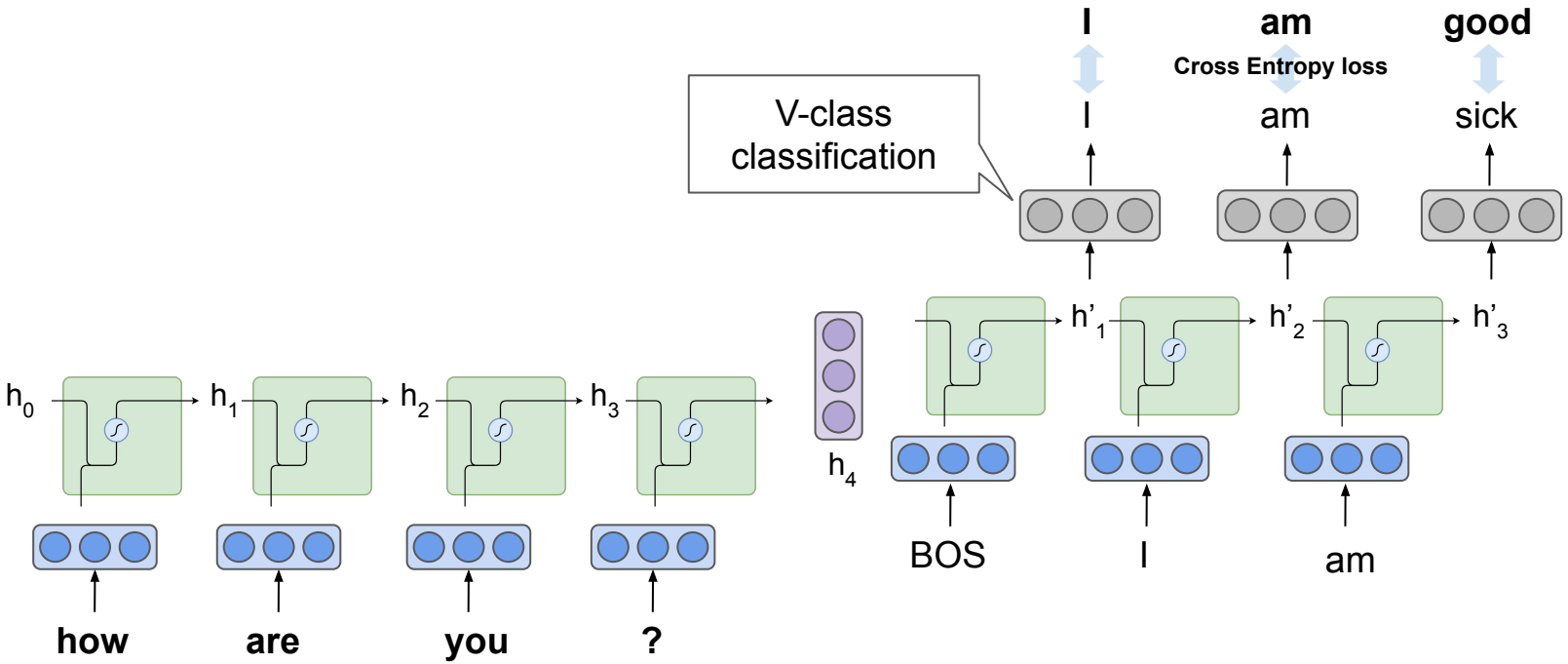
Training Encoder-Decoder Models: Training Data

- Training data: **Pairs of text sequences** (called a **parallel corpus)**
  - e.g., (“How are you?”, “I am good”)



Training Encoder-Decoder Models: “Teacher Forcing”

- Train the model to output **ground-truth outputs**



Note: Transformer Needs Decoding Algorithm

- Any decoding algorithm (e.g., Greedy/Beam Search) can be used
- Again, the Transformer is just an **encoder-decoder model**
- Although the Transformer does not have any recurrent architecture, the decoding procedure is still conducted in an iterative manner

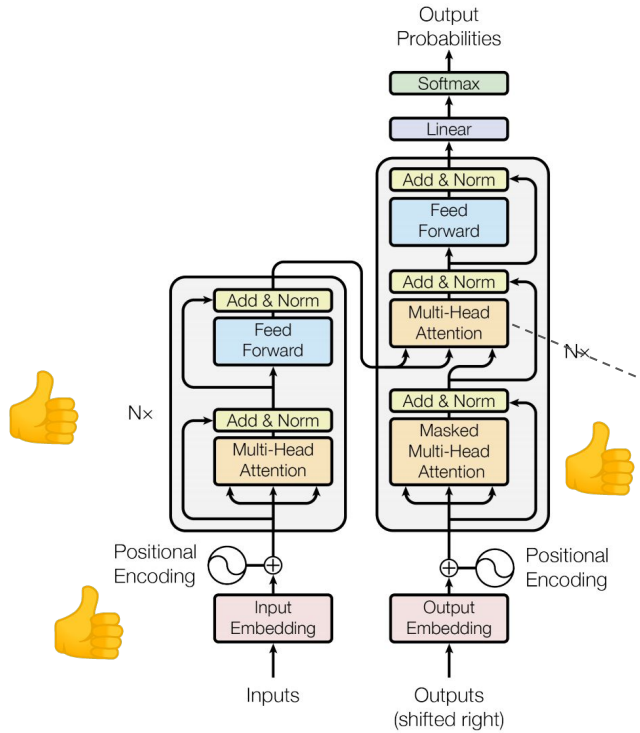
Parameter Initialization: Xavier Initialization

- Sampling initial parameter values from the Normal Distribution, where the standard deviation is **the number of parameters in the previous layer**
  - i.e., More parameters in the prev layer → Lower variance → Smaller absolute values

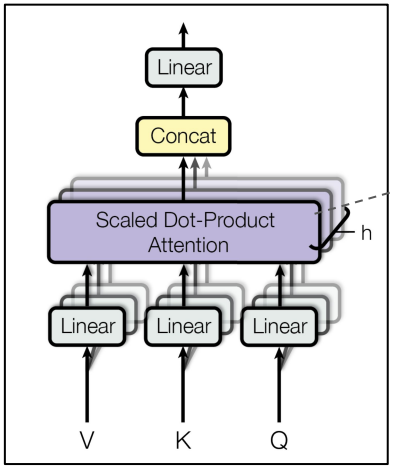
$$W_{i,j}^{[l]} = \mathcal{N}\left(0, \frac{1}{n^{[l-1]}}\right)$$

Detailed derivation: https://cs230.stanford.edu/section/4/

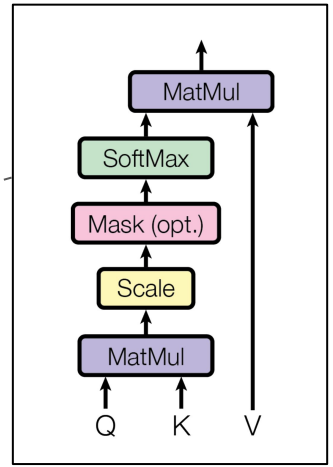
The Transformer 👍



Model Architecture



Multi-Head Attention



Scaled Dot-Product Attention

Key Takeaway

- Model Architecture
  - The Transformer is an **Encoder-Decoder model**
    - Transformer Encoder + Transformer Decoder (not explained yet)
  - Transformer Encoder = Stack of **Transformer Encoder Blocks**
  - Transformer Encoder Block = **Self-Attention Layer** + Feed Forward NN
- **Self-Attention Layer**
  - Token Embeddings → Query, Key, Value Vectors → Attention Vectors
  - Multi-head Attention
- Additional techniques
  - Positional Encoding
  - Residual Connection + Layer Normalization

Let’s Take A Look At The Original Paper

Attention Is All You Need

Ashish Vaswani* Google Brain avaswani@google.com

Noam Shazeer* Google Brain noam@google.com

Niki Parmar* Google Research nikip@google.com

Jakob Uszkoreit* Google Research usz@google.com

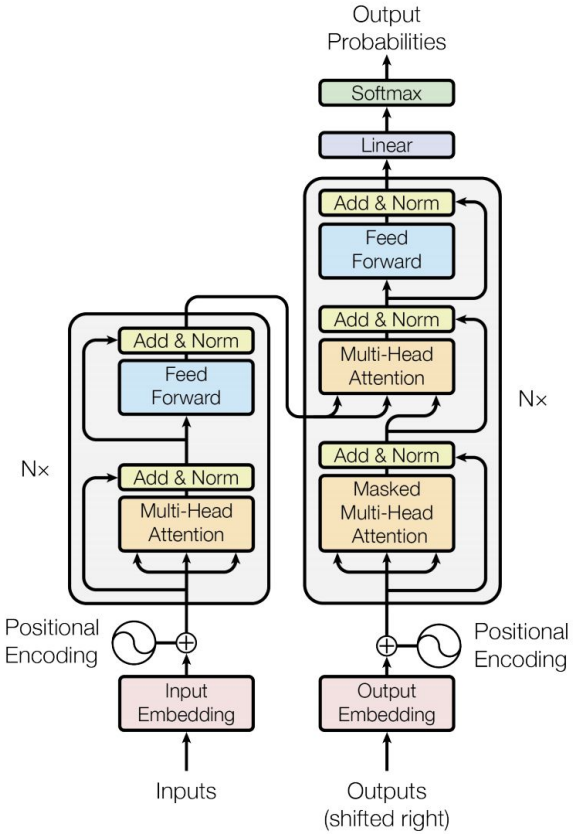
Llion Jones* Google Research llion@google.com

Aidan N. Gomez* † University of Toronto aidan@cs.toronto.edu

Łukasz Kaiser* Google Brain lukaszkaiser@google.com

Illia Polosukhin* ‡ illia.polosukhin@gmail.com

Model Descriptions in the Original Paper

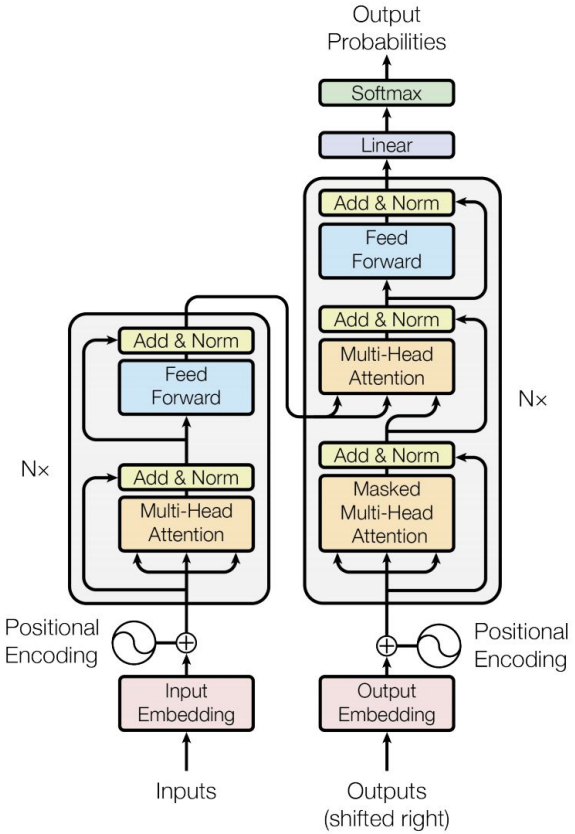


3.1 Encoder and Decoder Stacks

Encoder: The encoder is composed of a stack of $N = 6$ identical layers. Each layer has two sub-layers. The first is a multi-head self-attention mechanism, and the second is a simple, position-wise fully connected feed-forward network. We employ a residual connection [11] around each of the two sub-layers, followed by layer normalization [1]. That is, the output of each sub-layer is $\mathrm{LayerNorm}(x + \mathrm{Sublayer}(x))$, where $\mathrm{Sublayer}(x)$ is the function implemented by the sub-layer itself. To facilitate these residual connections, all sub-layers in the model, as well as the embedding layers, produce outputs of dimension $d_{\text{model}} = 512$.

Decoder: The decoder is also composed of a stack of $N = 6$ identical layers. In addition to the two sub-layers in each encoder layer, the decoder inserts a third sub-layer, which performs multi-head attention over the output of the encoder stack. Similar to the encoder, we employ residual connections around each of the sub-layers, followed by layer normalization. We also modify the self-attention sub-layer in the decoder stack to prevent positions from attending to subsequent positions. This masking, combined with fact that the output embeddings are offset by one position, ensures that the predictions for position i can depend only on the known outputs at positions less than i.

Model Descriptions in the Original Paper

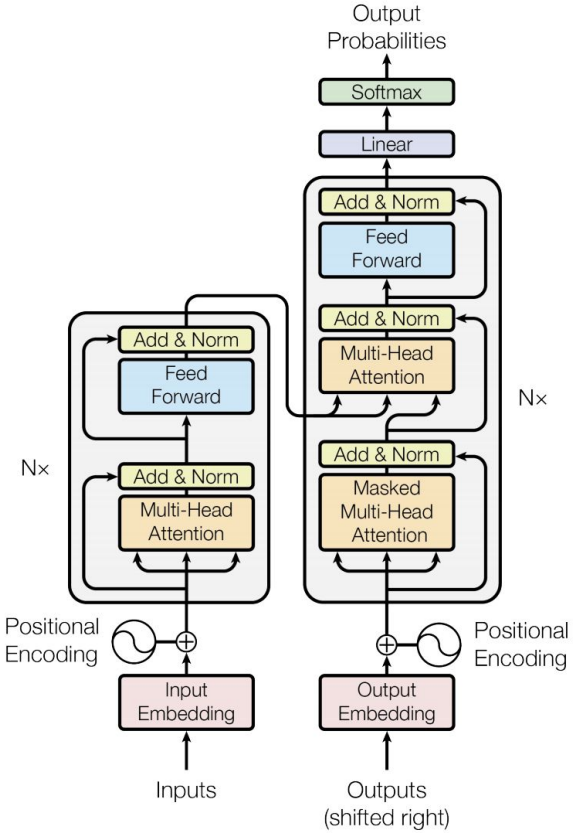


3.1 Encoder and Decoder Stacks

Encoder: The encoder is composed of a stack of $N = 6$ identical layers. Each layer has two sub-layers. The first is a multi-head self-attention mechanism, and the second is a simple, position-wise fully connected feed-forward network. We employ a residual connection [11] around each of the two sub-layers, followed by layer normalization [1]. That is, the output of each sub-layer is $\mathrm{LayerNorm}(x + \mathrm{Sublayer}(x))$, where $\mathrm{Sublayer}(x)$ is the function implemented by the sub-layer itself. To facilitate these residual connections, all sub-layers in the model, as well as the embedding layers, produce outputs of dimension $d_{\text{model}} = 512$.

Decoder: The decoder is also composed of a stack of $N = 6$ identical layers. In addition to the two sub-layers in each encoder layer, the decoder inserts a third sub-layer, which performs multi-head attention over the output of the encoder stack. Similar to the encoder, we employ residual connections around each of the sub-layers, followed by layer normalization. We also modify the self-attention sub-layer in the decoder stack to prevent positions from attending to subsequent positions. This masking, combined with fact that the output embeddings are offset by one position, ensures that the predictions for position i can depend only on the known outputs at positions less than i.

Model Descriptions in the Original Paper

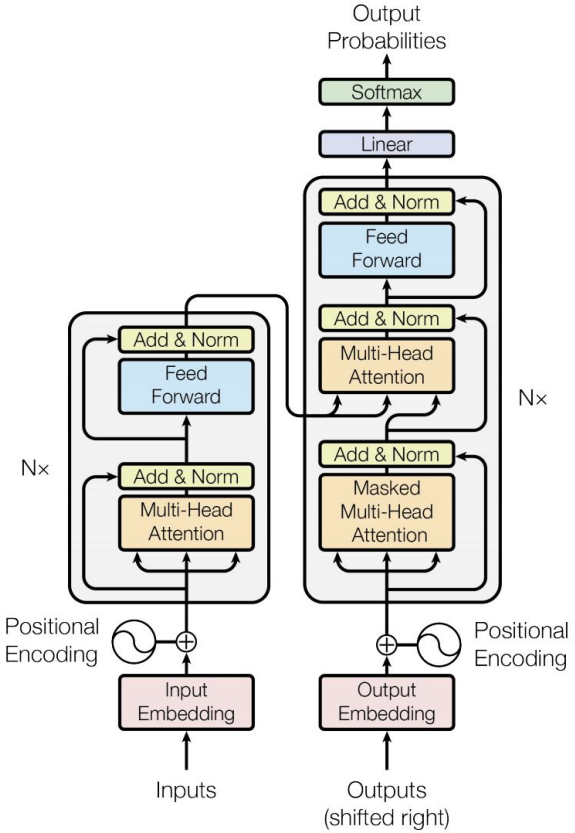


3.1 Encoder and Decoder Stacks

Encoder: The encoder is composed of a stack of $N = 6$ identical layers. Each layer has two sub-layers. The first is a multi-head self-attention mechanism, and the second is a simple, position-wise fully connected feed-forward network. We employ a residual connection [11] around each of the two sub-layers, followed by layer normalization [1]. That is, the output of each sub-layer is $\mathrm{LayerNorm}(x + \mathrm{Sublayer}(x))$, where $\mathrm{Sublayer}(x)$ is the function implemented by the sub-layer itself. To facilitate these residual connections, all sub-layers in the model, as well as the embedding layers, produce outputs of dimension $d_{\text{model}} = 512$.

Decoder: The decoder is also composed of a stack of $N = 6$ identical layers. In addition to the two sub-layers in each encoder layer, the decoder inserts a third sub-layer, which performs multi-head attention over the output of the encoder stack. Similar to the encoder, we employ residual connections around each of the sub-layers, followed by layer normalization. We also modify the self-attention sub-layer in the decoder stack to prevent positions from attending to subsequent positions. This masking, combined with fact that the output embeddings are offset by one position, ensures that the predictions for position i can depend only on the known outputs at positions less than i.

Model Descriptions in the Original Paper

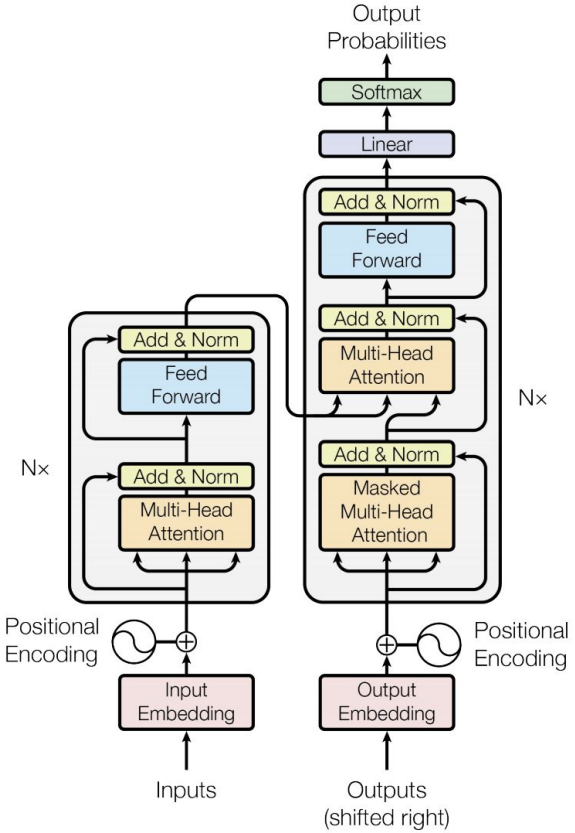


3.1 Encoder and Decoder Stacks

Encoder: The encoder is composed of a stack of $N = 6$ identical layers. Each layer has two sub-layers. The first is a multi-head self-attention mechanism, and the second is a simple, position-wise fully connected feed-forward network. We employ a residual connection [11] around each of the two sub-layers, followed by layer normalization [1]. That is, the output of each sub-layer is $\mathrm{LayerNorm}(x + \mathrm{Sublayer}(x))$, where $\mathrm{Sublayer}(x)$ is the function implemented by the sub-layer itself. To facilitate these residual connections, all sub-layers in the model, as well as the embedding layers, produce outputs of dimension $d_{\text{model}} = 512$.

Decoder: The decoder is also composed of a stack of $N = 6$ identical layers. In addition to the two sub-layers in each encoder layer, the decoder inserts a third sub-layer, which performs multi-head attention over the output of the encoder stack. Similar to the encoder, we employ residual connections around each of the sub-layers, followed by layer normalization. We also modify the self-attention sub-layer in the decoder stack to prevent positions from attending to subsequent positions. This masking, combined with fact that the output embeddings are offset by one position, ensures that the predictions for position i can depend only on the known outputs at positions less than i.

Model Descriptions in the Original Paper

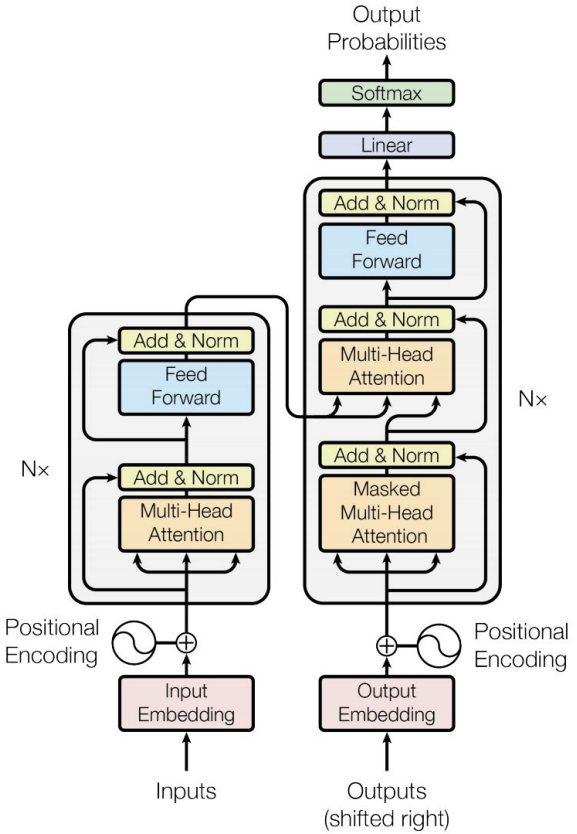


3.1 Encoder and Decoder Stacks

Encoder: The encoder is composed of a stack of $N = 6$ identical layers. Each layer has two sub-layers. The first is a multi-head self-attention mechanism, and the second is a simple, position-wise fully connected feed-forward network. We employ a residual connection [11] around each of the two sub-layers, followed by layer normalization [1]. That is, the output of each sub-layer is $\mathrm{LayerNorm}(x + \mathrm{Sublayer}(x))$, where $\mathrm{Sublayer}(x)$ is the function implemented by the sub-layer itself. To facilitate these residual connections, all sub-layers in the model, as well as the embedding layers, produce outputs of dimension $d_{\text{model}} = 512$.

Decoder: The decoder is also composed of a stack of $N = 6$ identical layers. In addition to the two sub-layers in each encoder layer, the decoder inserts a third sub-layer, which performs multi-head attention over the output of the encoder stack. Similar to the encoder, we employ residual connections around each of the sub-layers, followed by layer normalization. We also modify the self-attention sub-layer in the decoder stack to prevent positions from attending to subsequent positions. This masking, combined with fact that the output embeddings are offset by one position, ensures that the predictions for position i can depend only on the known outputs at positions less than i.

Model Descriptions in the Original Paper

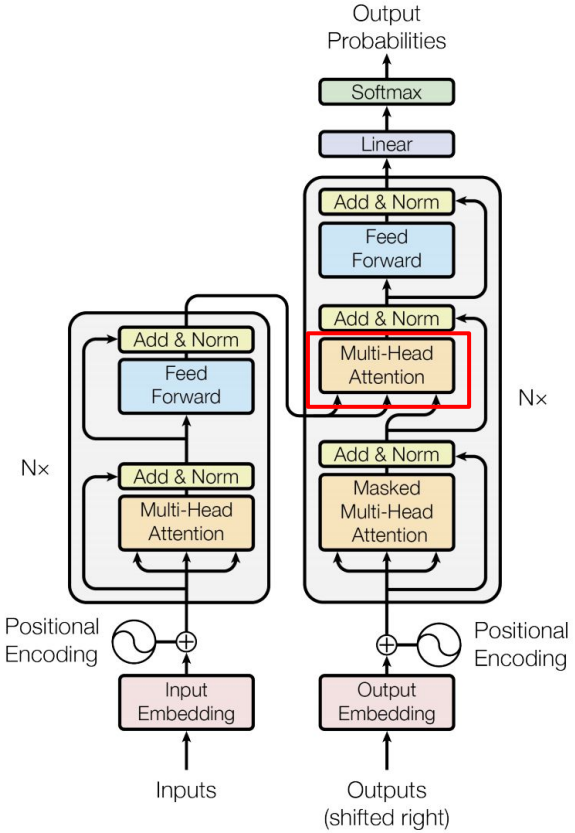


3.1 Encoder and Decoder Stacks

Encoder: The encoder is composed of a stack of $N = 6$ identical layers. Each layer has two sub-layers. The first is a multi-head self-attention mechanism, and the second is a simple, position-wise fully connected feed-forward network. We employ a residual connection [11] around each of the two sub-layers, followed by layer normalization [1]. That is, the output of each sub-layer is $\mathrm{LayerNorm}(x + \mathrm{Sublayer}(x))$, where $\mathrm{Sublayer}(x)$ is the function implemented by the sub-layer itself. To facilitate these residual connections, all sub-layers in the model, as well as the embedding layers, produce outputs of dimension $d_{\text{model}} = 512$.

Decoder: The decoder is also composed of a stack of $N = 6$ identical layers. In addition to the two sub-layers in each encoder layer, the decoder inserts a third sub-layer, which performs multi-head attention over the output of the encoder stack. Similar to the encoder, we employ residual connections around each of the sub-layers, followed by layer normalization. We also modify the self-attention sub-layer in the decoder stack to prevent positions from attending to subsequent positions. This masking, combined with fact that the output embeddings are offset by one position, ensures that the predictions for position i can depend only on the known outputs at positions less than i.

Model Descriptions in the Original Paper

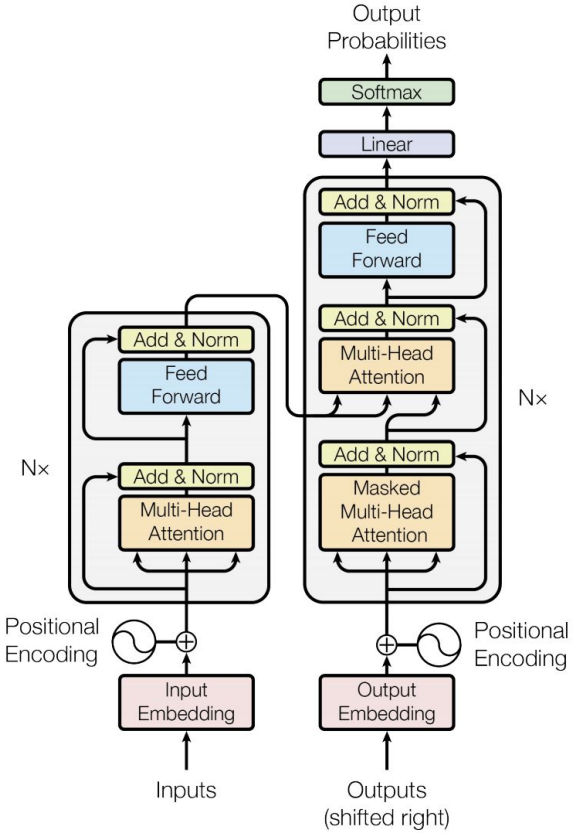


3.1 Encoder and Decoder Stacks

Encoder: The encoder is composed of a stack of $N = 6$ identical layers. Each layer has two sub-layers. The first is a multi-head self-attention mechanism, and the second is a simple, position-wise fully connected feed-forward network. We employ a residual connection [11] around each of the two sub-layers, followed by layer normalization [1]. That is, the output of each sub-layer is $\mathrm{LayerNorm}(x + \mathrm{Sublayer}(x))$, where $\mathrm{Sublayer}(x)$ is the function implemented by the sub-layer itself. To facilitate these residual connections, all sub-layers in the model, as well as the embedding layers, produce outputs of dimension $d_{\text{model}} = 512$.

Decoder: The decoder is also composed of a stack of $N = 6$ identical layers. In addition to the two sub-layers in each encoder layer, the decoder inserts a third sub-layer, which performs multi-head attention over the output of the encoder stack. Similar to the encoder, we employ residual connections around each of the sub-layers, followed by layer normalization. We also modify the self-attention sub-layer in the decoder stack to prevent positions from attending to subsequent positions. This masking, combined with fact that the output embeddings are offset by one position, ensures that the predictions for position i can depend only on the known outputs at positions less than i.

Model Descriptions in the Original Paper

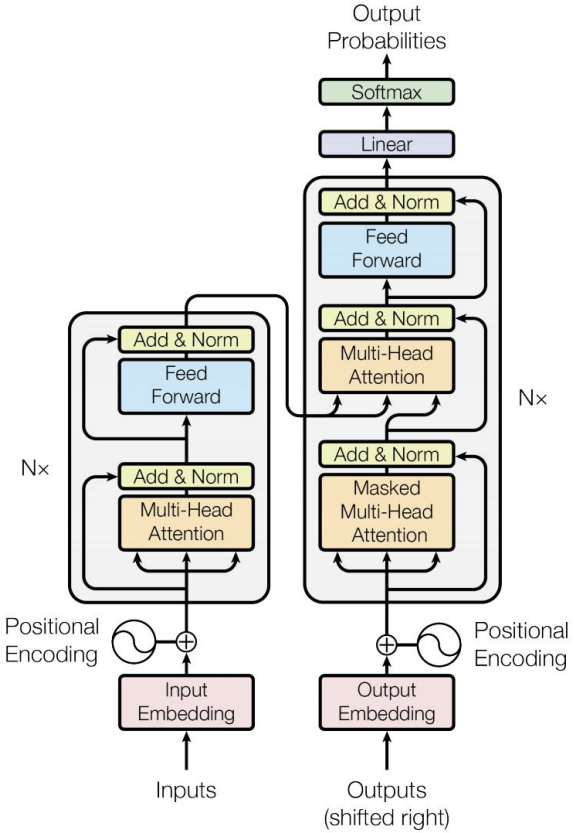


3.1 Encoder and Decoder Stacks

Encoder: The encoder is composed of a stack of $N = 6$ identical layers. Each layer has two sub-layers. The first is a multi-head self-attention mechanism, and the second is a simple, position-wise fully connected feed-forward network. We employ a residual connection [11] around each of the two sub-layers, followed by layer normalization [1]. That is, the output of each sub-layer is $\mathrm{LayerNorm}(x + \mathrm{Sublayer}(x))$, where $\mathrm{Sublayer}(x)$ is the function implemented by the sub-layer itself. To facilitate these residual connections, all sub-layers in the model, as well as the embedding layers, produce outputs of dimension $d_{\text{model}} = 512$.

Decoder: The decoder is also composed of a stack of $N = 6$ identical layers. In addition to the two sub-layers in each encoder layer, the decoder inserts a third sub-layer, which performs multi-head attention over the output of the encoder stack. Similar to the encoder, we employ residual connections around each of the sub-layers, followed by layer normalization. We also modify the self-attention sub-layer in the decoder stack to prevent positions from attending to subsequent positions. This masking, combined with fact that the output embeddings are offset by one position, ensures that the predictions for position i can depend only on the known outputs at positions less than i.

Model Descriptions in the Original Paper



3.1 Encoder and Decoder Stacks

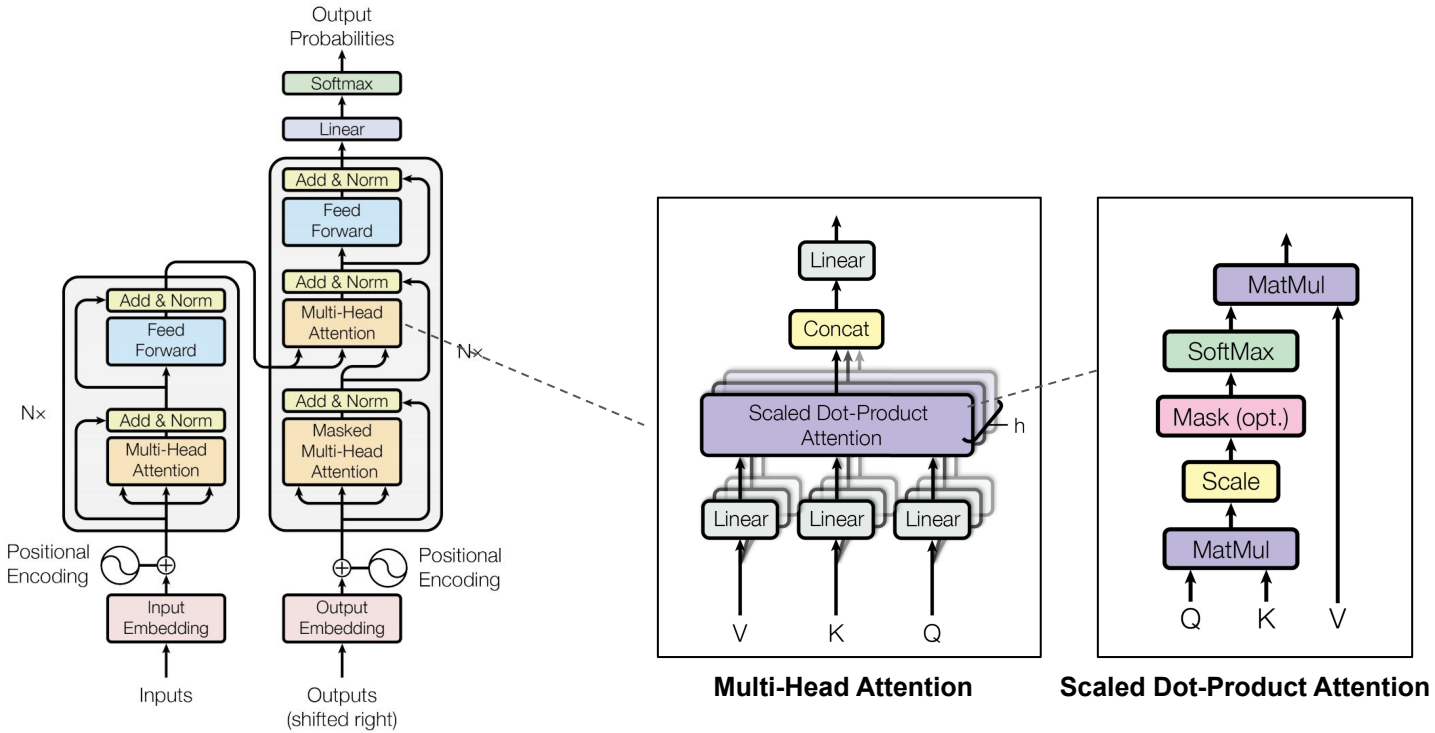
Encoder: The encoder is composed of a stack of $N = 6$ identical layers. Each layer has two sub-layers. The first is a multi-head self-attention mechanism, and the second is a simple, position-wise fully connected feed-forward network. We employ a residual connection [11] around each of the two sub-layers, followed by layer normalization [1]. That is, the output of each sub-layer is $\mathrm{LayerNorm}(x + \mathrm{Sublayer}(x))$, where $\mathrm{Sublayer}(x)$ is the function implemented by the sub-layer itself. To facilitate these residual connections, all sub-layers in the model, as well as the embedding layers, produce outputs of dimension $d_{\text{model}} = 512$.

Decoder: The decoder is also composed of a stack of $N = 6$ identical layers. In addition to the two sub-layers in each encoder layer, the decoder inserts a third sub-layer, which performs multi-head attention over the output of the encoder stack. Similar to the encoder, we employ residual connections around each of the sub-layers, followed by layer normalization. We also modify the self-attention sub-layer in the decoder stack to prevent positions from attending to subsequent positions. This masking, combined with fact that the output embeddings are offset by one position, ensures that the predictions for position i can depend only on the known outputs at positions less than i.

PyTorch Code Reading

- Transformer
- TransformerEncoder
- TransformerEncoderLayer
- MultiheadAttention
- TransformerDecoder
- TransformerDecoderLayer

This Week's Goal:
Understanding the Transformer Architecture 😊



Model Architecture

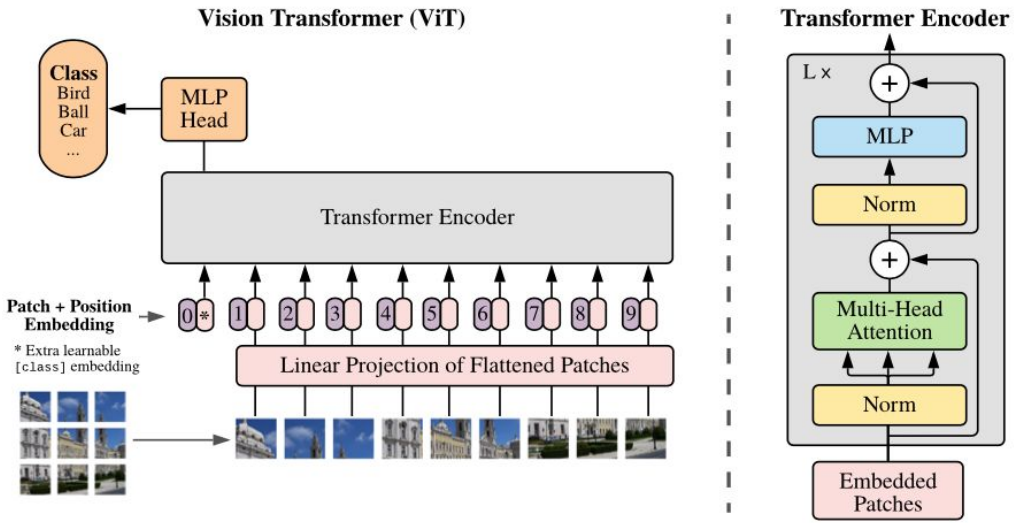
Multi-Head Attention

Scaled Dot-Product Attention

Transformers for Computer Vision

Vision Transformer (ViT) [Dosovitskiy et al. 2021]

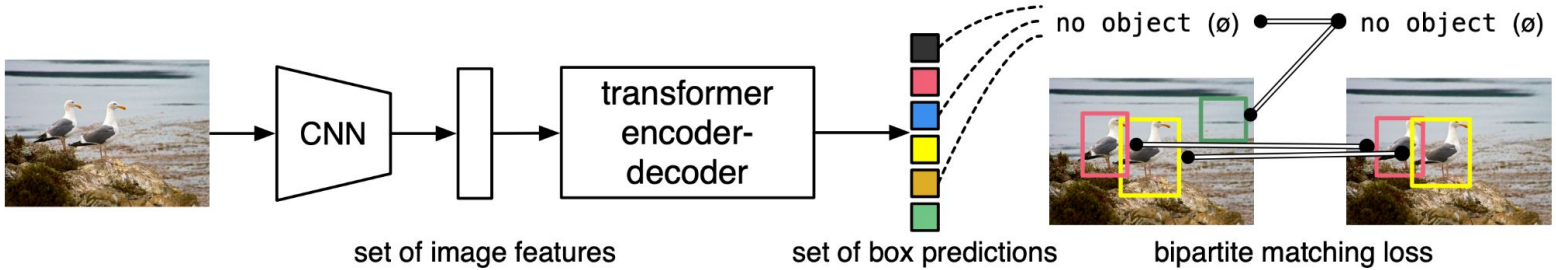
- RGB Image → N Patches → Flattening patches
  - Each patch = (3 * P^2)-dimensional vector



A. Dosovitskiy et al. “An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale”, ICLR ‘21

Detection Transformer (DETR) [Carion et al. 2020]

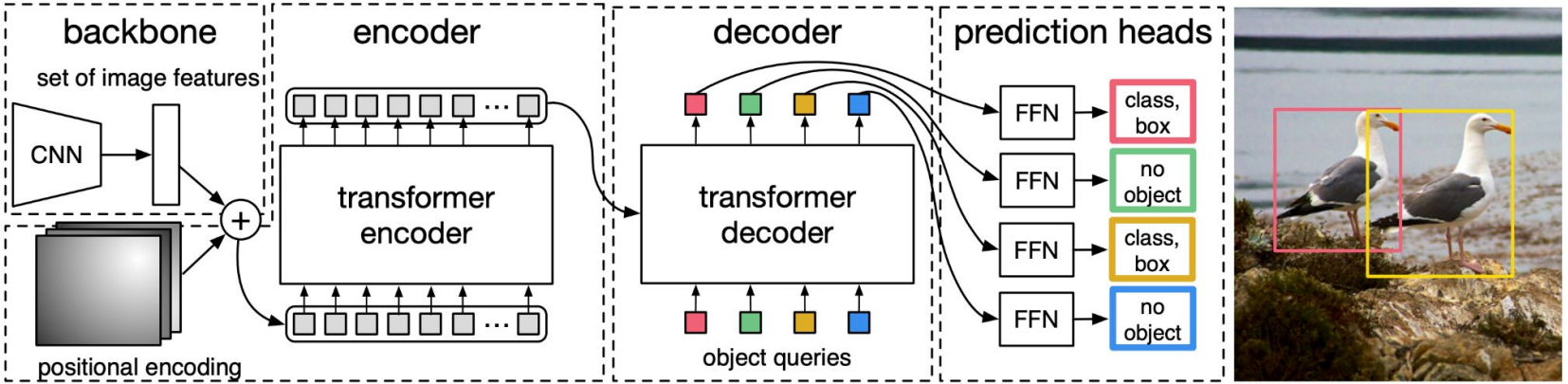
- CNN + Transformer for object detection



N. Carion et al. “End-to-End Object Detection with Transformers”, arXiv 2020.

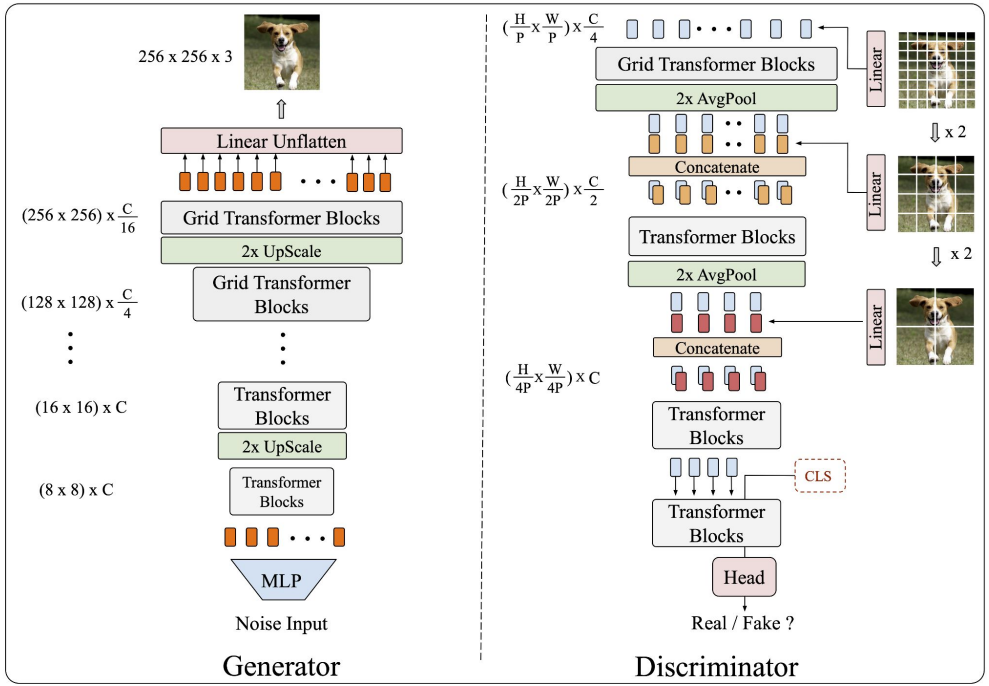
Detection Transformer (DETR) [Carion et al. 2020]

- CNN + Transformer for object detection



N. Carion et al. “End-to-End Object Detection with Transformers”, arXiv 2020.

TransGAN [Jian et al. 2021]

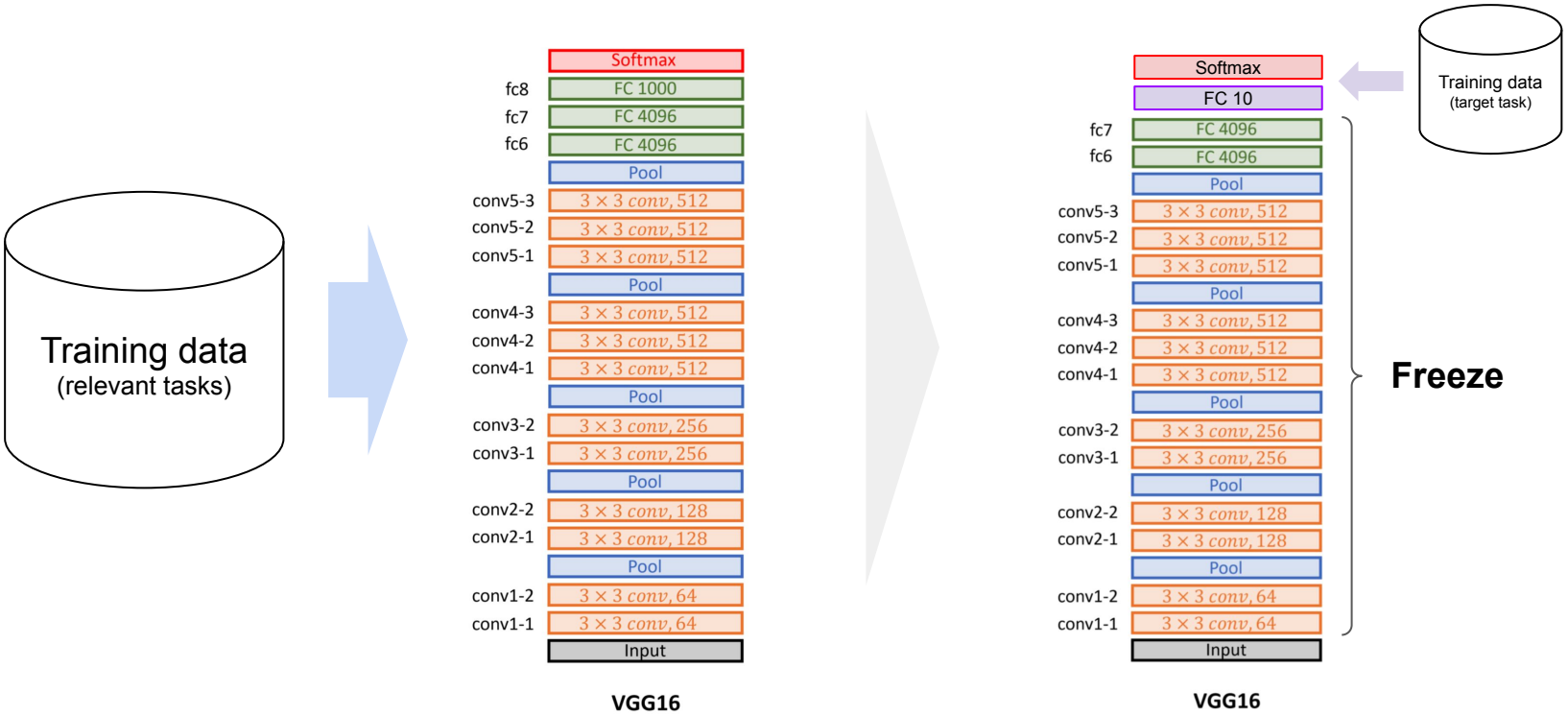


Yifan Jiang, Shiyu Chang, Zhangyang Wang, “TransGAN: Two Pure Transformers Can Make One Strong GAN, and That Can Scale Up”, arXiv 2021.

Pre-trained Language Models

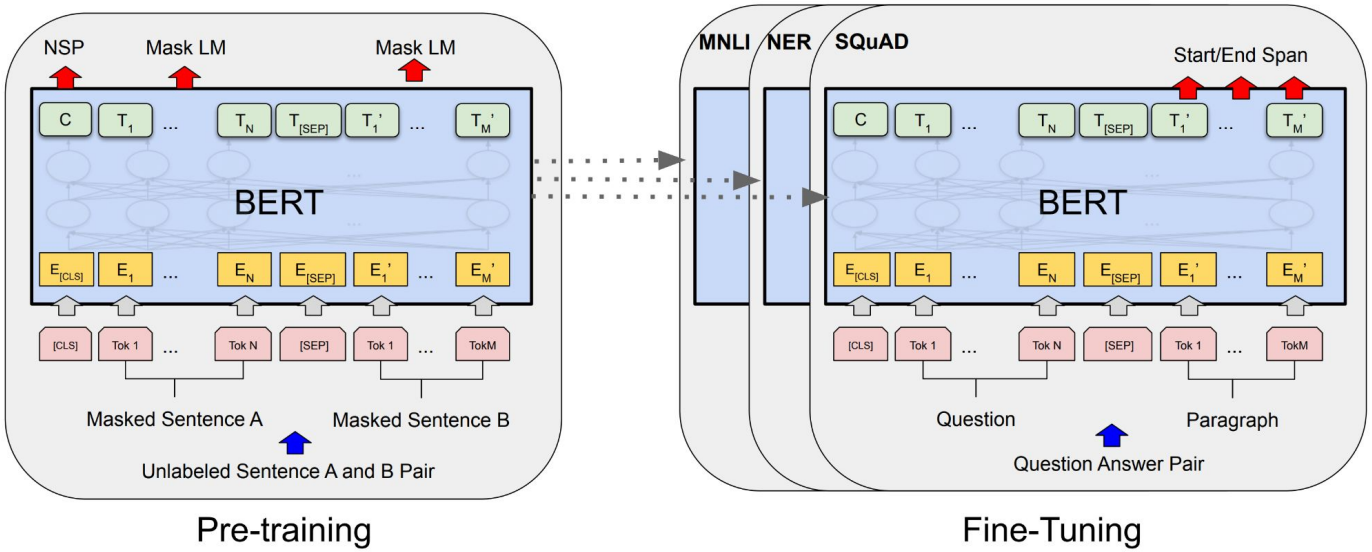
What are Pre-training and Fine-tuning?

- Example for Computer Vision



What are Pre-training and Fine-tuning?

- Example for NLP



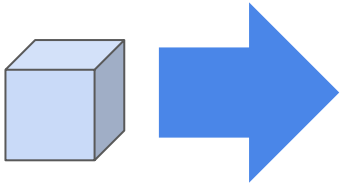
Pre-training & Fine-tuning: Intuition

Pre-training

Pre-training Task 1

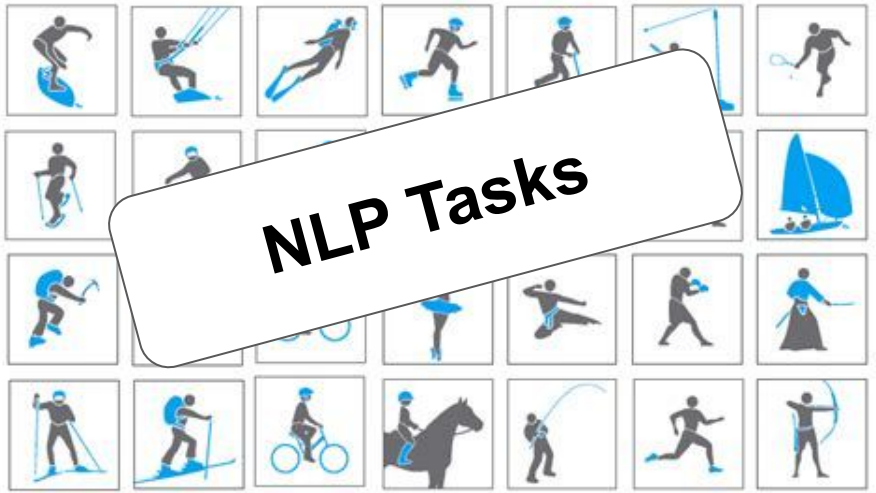
Pre-training Task 2

Fine-tuning



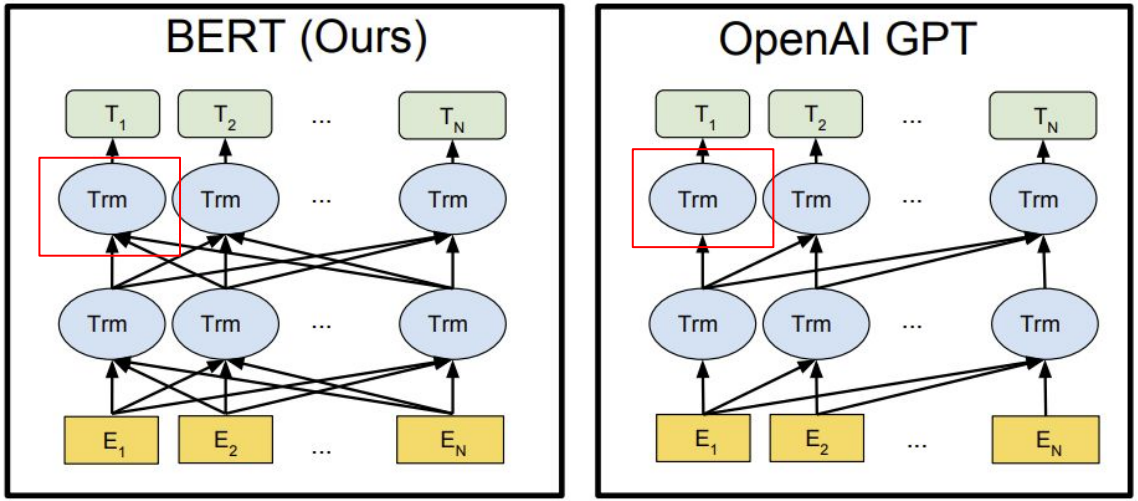
(e.g., BERT)

Downstream tasks



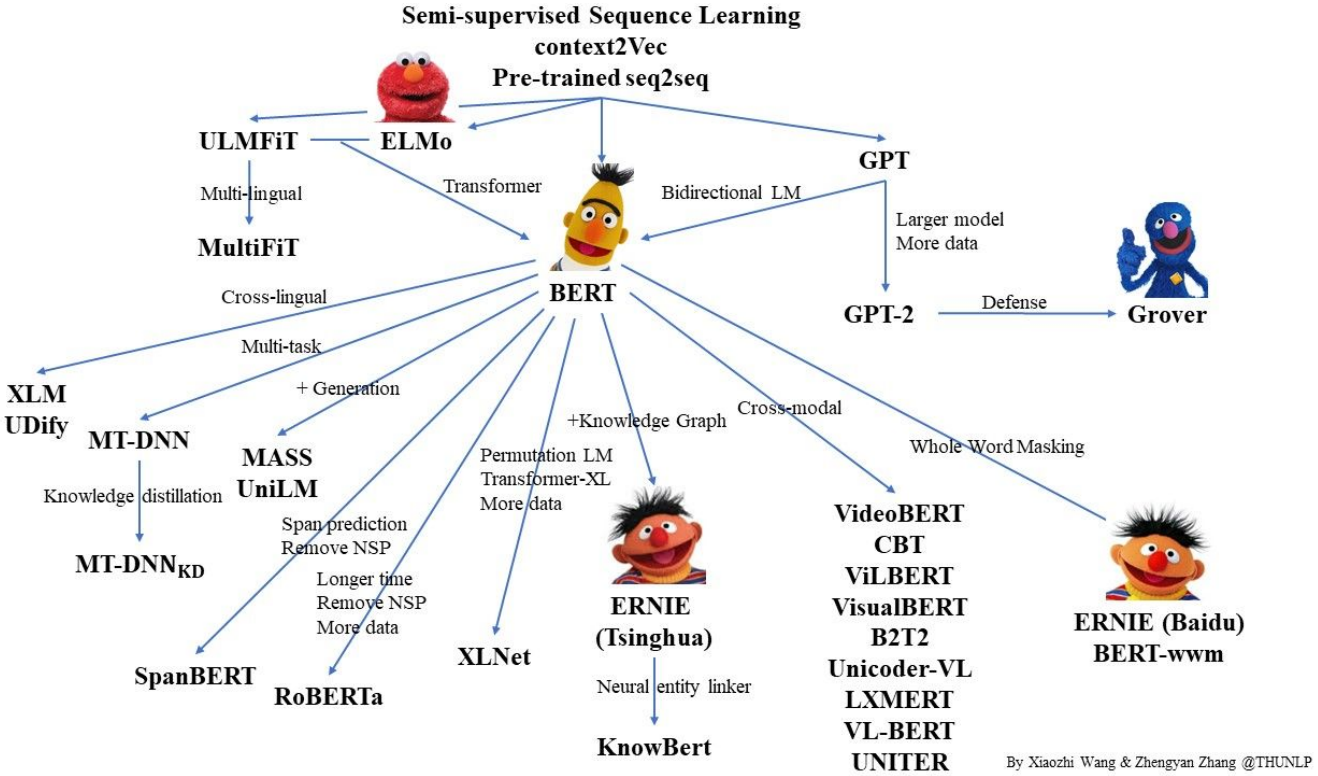
BERT and GPT

- BERT = Bi-directional Transformers (Transformer **Encoder**)
- GPT = left-to-right Transformers (Transformer **Decoder**)



BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding
GPT: Generative Pre-Training

Family Tree of Pre-trained LMs



https://github.com/thunlp/PLMpapers

Pre-trained Model War: Rapid Increase in Model Size

log scale

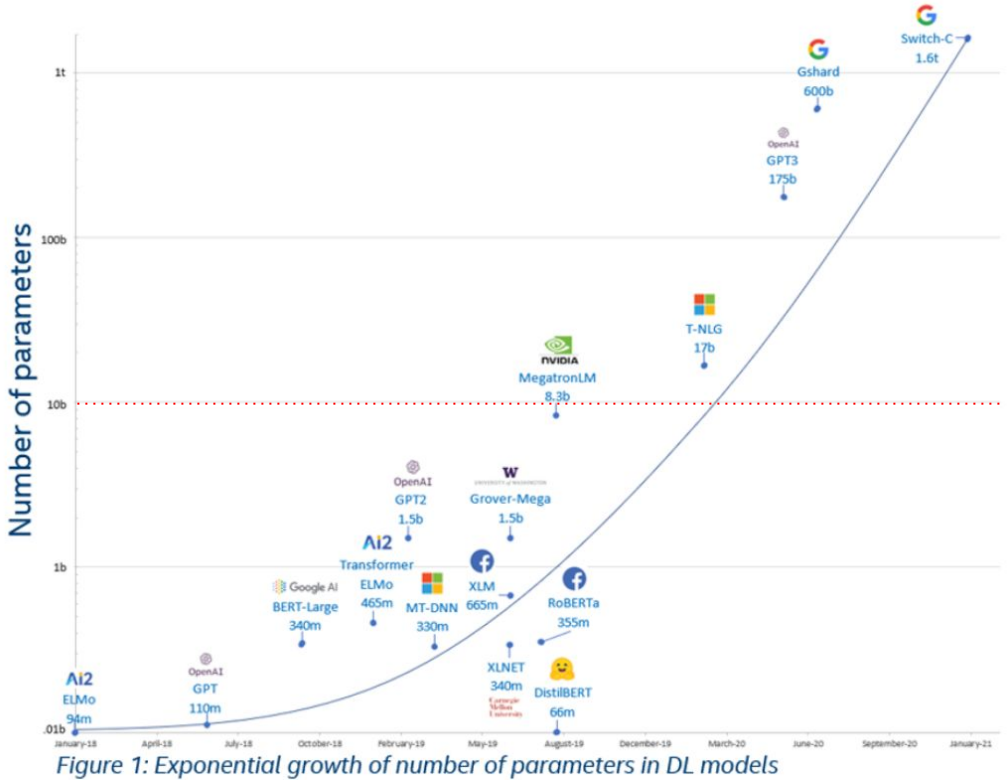


Figure 1: Exponential growth of number of parameters in DL models

2018 2019 2020 2021

Hugging Face's Transformers Library

https://github.com/huggingface/transformers

- You can use a variety of pre-trained language models just with a few lines of Python code

```
>>> from transformers import AutoTokenizer, AutoModel

>>> tokenizer = AutoTokenizer.from_pretrained("bert-base-uncased")
>>> model = AutoModel.from_pretrained("bert-base-uncased")

>>> inputs = tokenizer("Hello world!", return_tensors="pt")
>>> outputs = model(**inputs)
```

Summary

- The Transformer architecture
  - Encoder & Decoder
  - Encoder-decoder attention
- Brief introduction to Transformers for Computer Vision
  - Vision Transformer
  - Detection Transformer
  - TransGAN
- Brief overview of pre-trained Language Models
  - BERT and GPT are Transformer Encoder and Decoder
  - (Almost) all pre-trained models today use the Transformer architecture

Questions?

Further Reading

- The Illustrated Transformer – Jay Alammar – Visualizing machine learning one concept at a time.
- Multi-head attention mechanism: "queries", "keys", and "values," over and over again