SC 4001 CE/CZ 4042 Neural Network and Deep Learning

Last update: 17 Mar 2024

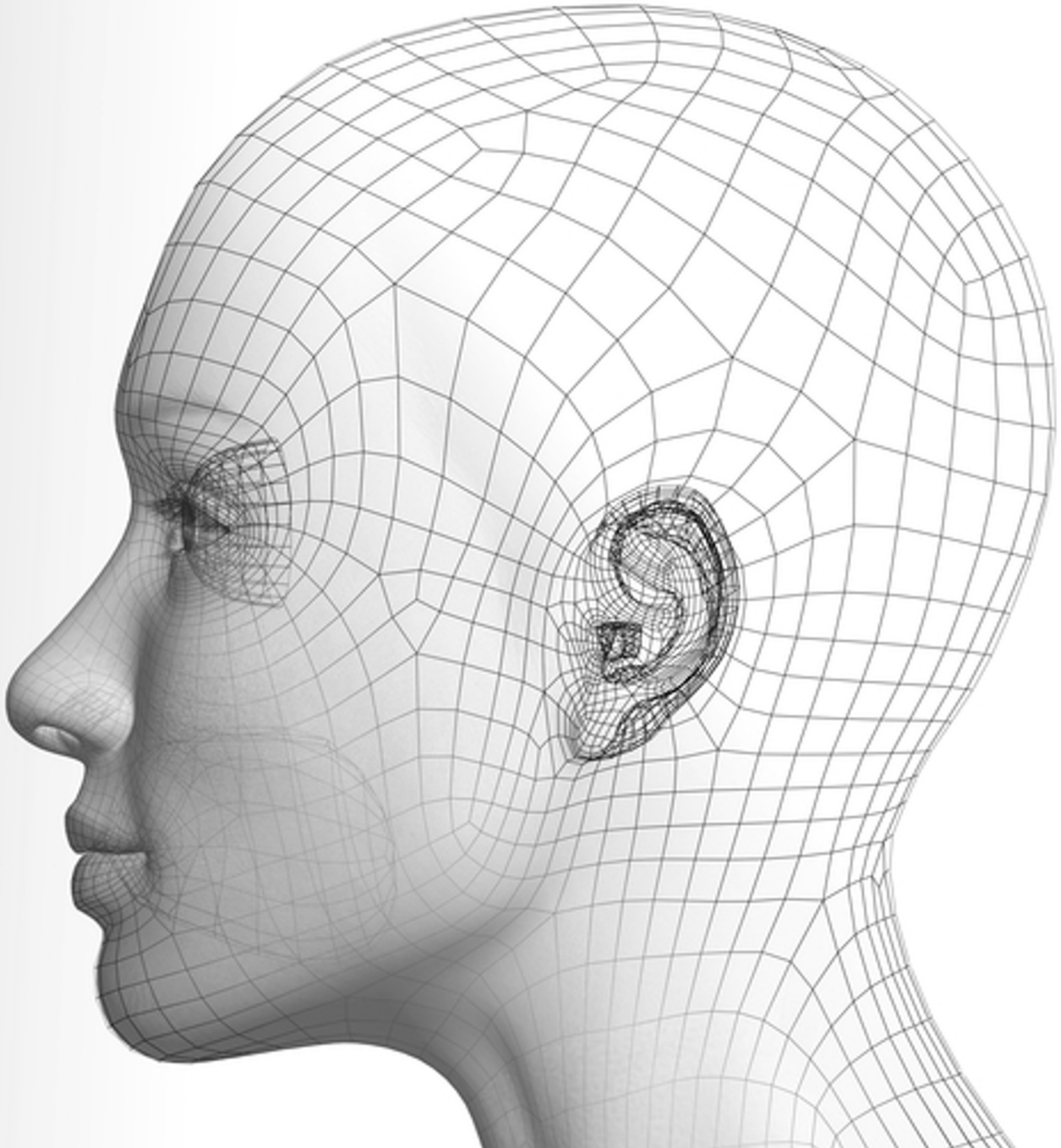
Attention

Xingang Pan

潘新钢

https://xingangpan.github.io/

https://twitter.com/XingangP



Outline

- Attention
- Transformers
- Vision Transformers

Attention


"Where's Waldo?"

Attention

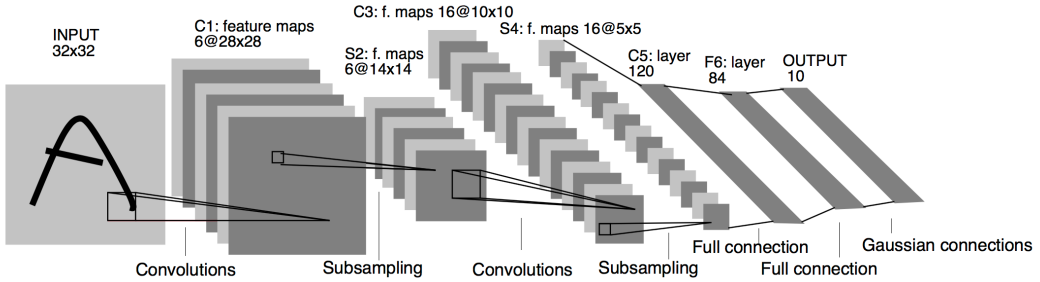
Attention is a mechanism that a model can learn to make predictions by selectively attending to a given set of data.

The amount of attention is quantified by learned weights and thus the output is usually formed as a weighted average.

Transformers

Background

- Convolutional neural networks (CNN) has been dominating
  - Greater scale
  - More extensive connections
  - More sophisticated forms of convolution



- Transformers
  - Competitive alternative to CNN
  - Generally found to perform best in settings with large amounts of training data
  - Enable multi-modality learning

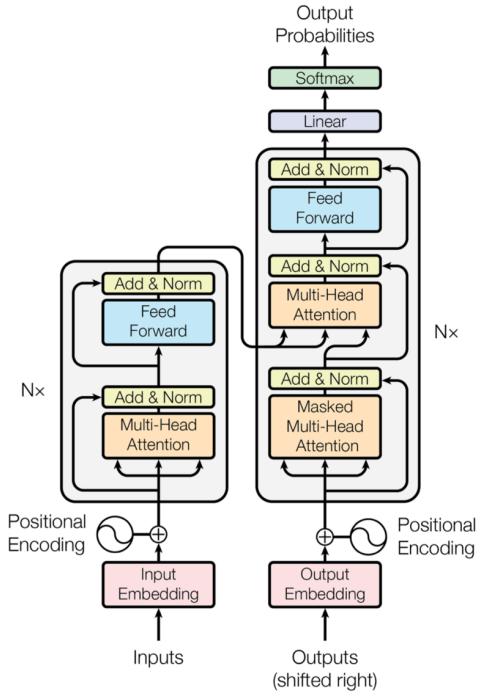


Figure 1: The Transformer - model architecture.

Ashish Vaswani et al., Attention Is All You Need, NIPS 2017 (from Google)

Background

GPT-4

What is the best way to learn knowledge about Transformers (a machine learning model)?

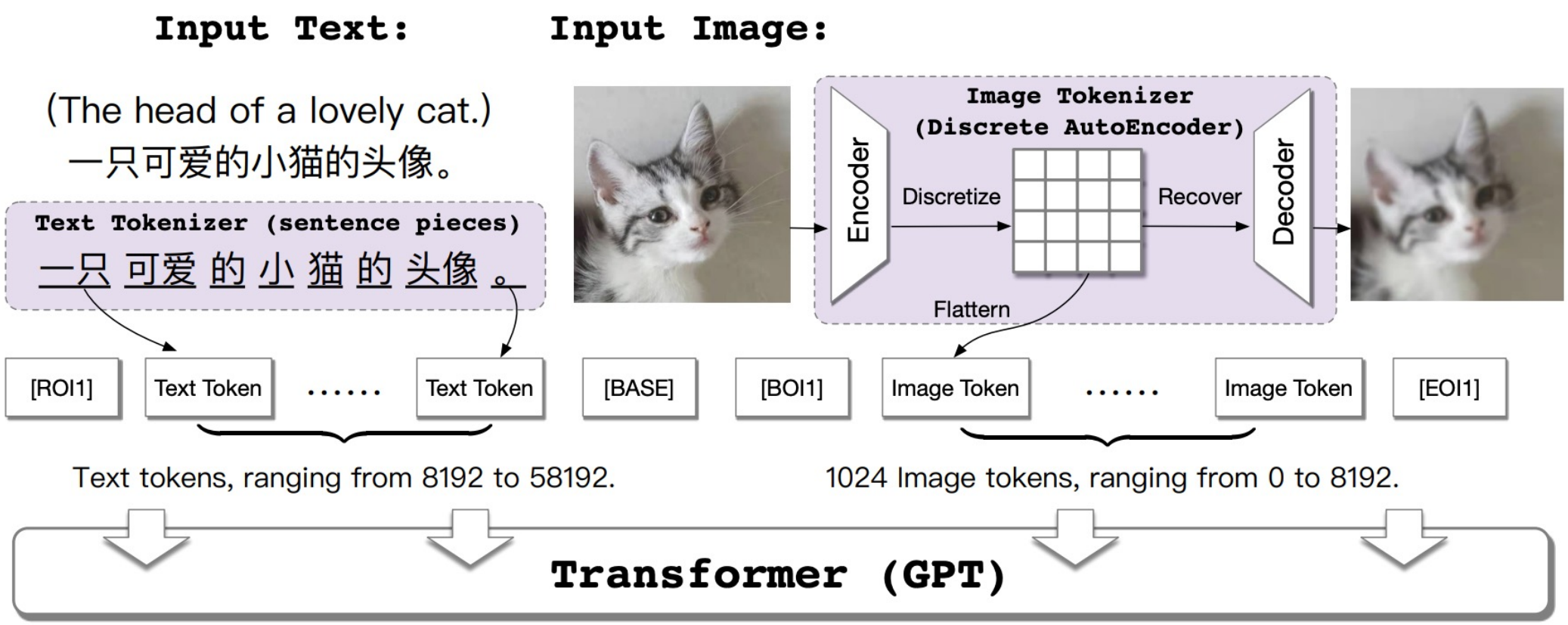
Transformers are an important concept in modern machine learning, especially in the field of natural language processing. They are the key model behind the technology in GPT-3 and GPT-4, BERT, T5, and many other influential architectures.

Here is a suggested path for learning about Transformers:

1. Understand the Basics of Machine Learning and Deep Learning

Before you get into Transformers, it's crucial to have a solid understanding of Machine Learning and Deep Learning. Knowing the fundamental concepts of Deep Learning such as neural networks, activation functions, loss functions, backpropagation, convolutional neural networks (CNNs), recurrent neural networks (RNNs), and Long Short Term Memory (LSTM) units is very helpful.

Background



Ding et al., CogView: Mastering Text-to-Image Generation via Transformers, NeurIPS 2021

Background

Ding et al., [CogView: Mastering Text-to-Image Generation via Transformers](), NeurIPS 2021

Transformers

Notable for its use of attention to model long-range dependencies in data

A sequence-to-sequence model

Model of choice in natural language processing (NLP)

Step-by-step guide to self-attention with illustrations and code https://jalammar.github.io/illustrated-transformer/

Ashish Vaswani et al., Attention Is All You Need, NIPS 2017 (from Google)

Transformers

Like LSTM, Transformer is an architecture for transforming one sequence into another one with the help of two parts (Encoder and Decoder)

But it differs from the existing sequence-to-sequence models because it does not imply any Recurrent Networks (GRU, LSTM, etc.).

- During training, layer outputs can be calculated in parallel, instead of a series like an RNN
- Attention-based models allow modeling of dependencies without regard to their distance in the input or output sequences

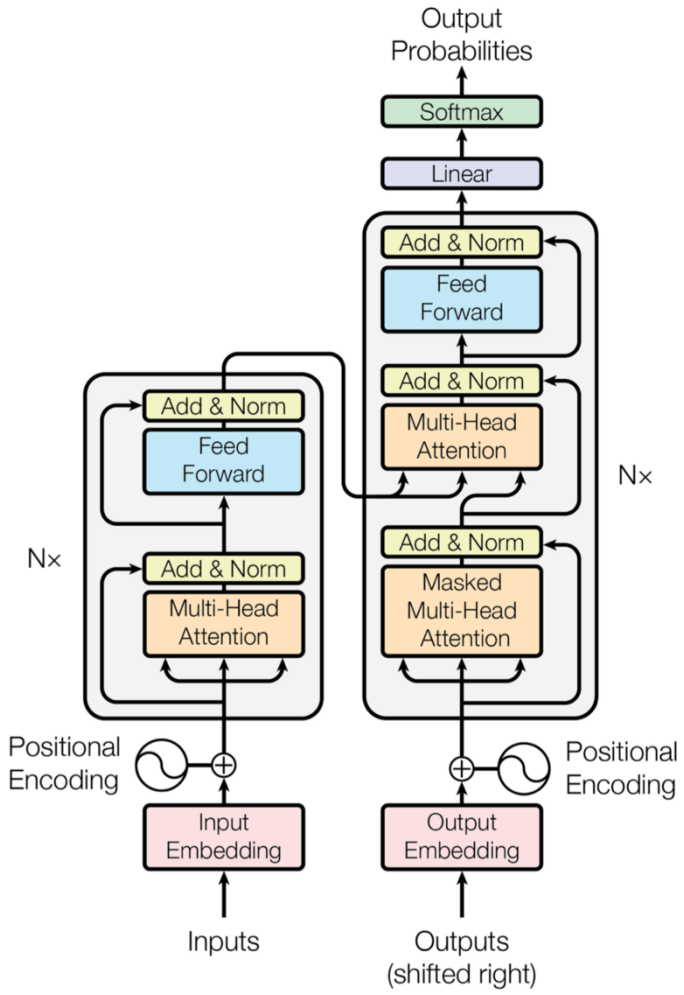


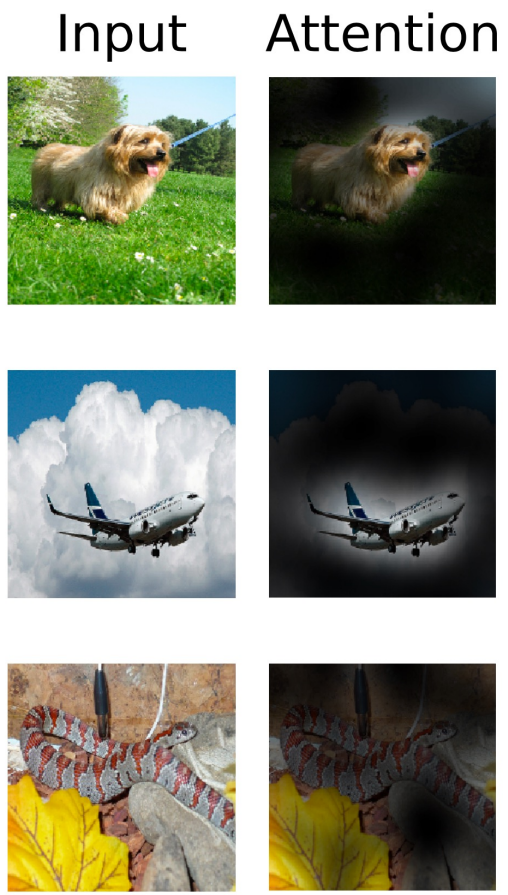
Figure 1: The Transformer - model architecture.

Transformers

Attention is a mechanism that a model can learn to make predictions by selectively attending to a given set of data.

The amount of attention is quantified by learned weights and thus the output is usually formed as a weighted average.

$$\text{Attention} = \mathbf{W}\mathbf{V}$$

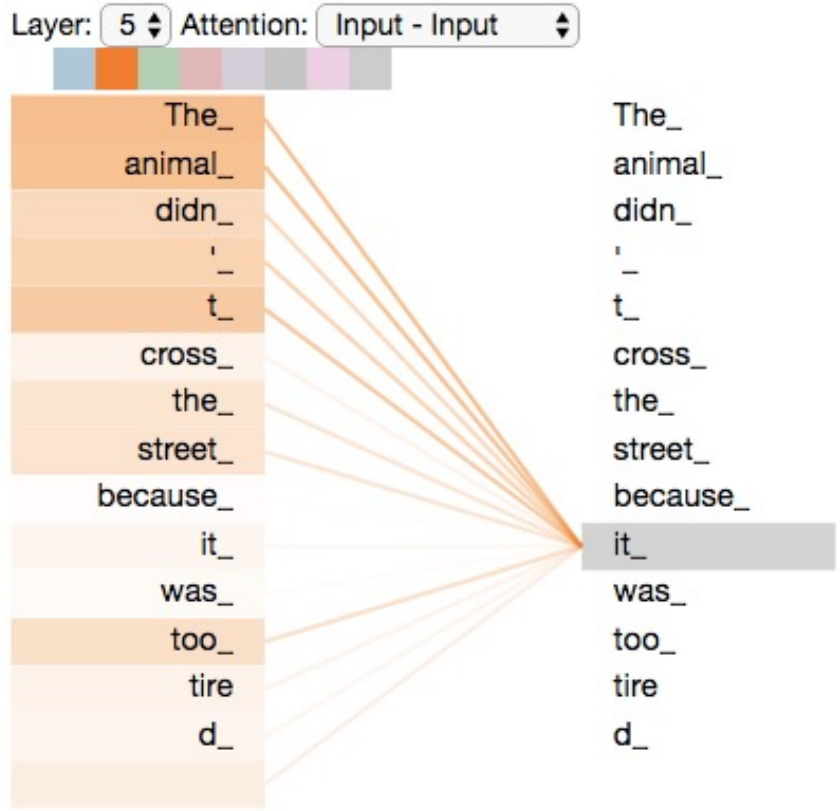


The model attends to image regions that are semantically relevant for classification

Transformers

It is entirely built on the self-attention mechanisms without using sequence-aligned recurrent architecture

Self-attention is a type of attention mechanism where the model makes prediction for one part of a data sample using other parts of the observation about the same sample.



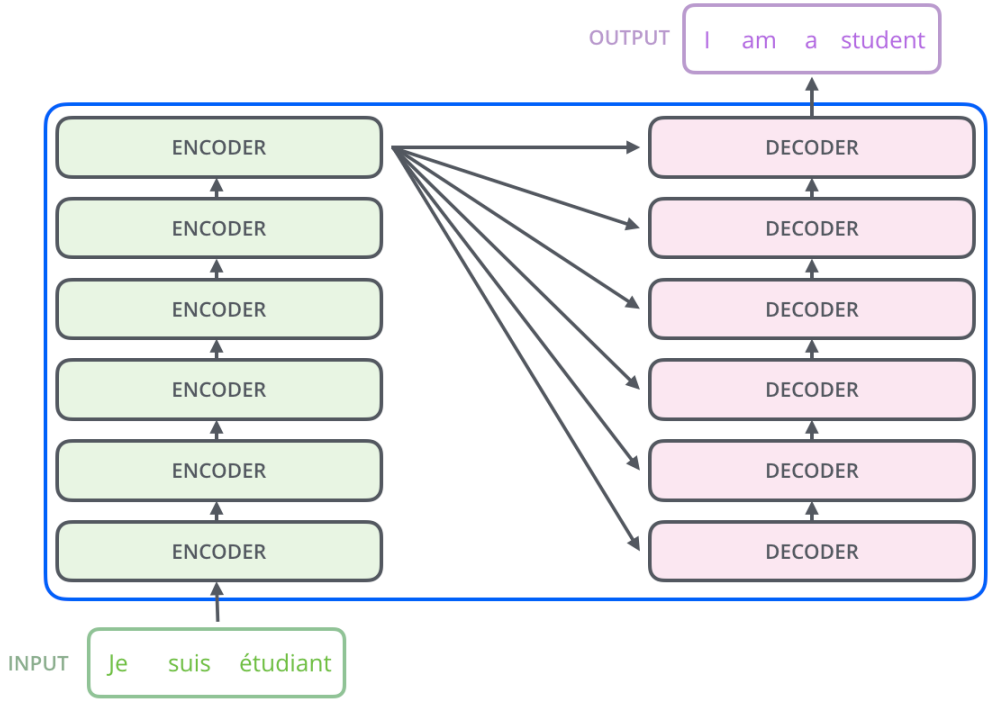
What does “it” in this sentence refer to? Is it referring to the street or to the animal?

When the model is processing the word “it”, self-attention allows it to associate “it” with “animal”.

Transformers

The encoding component is a stack of encoders

The decoding component is a stack of decoders of the same number

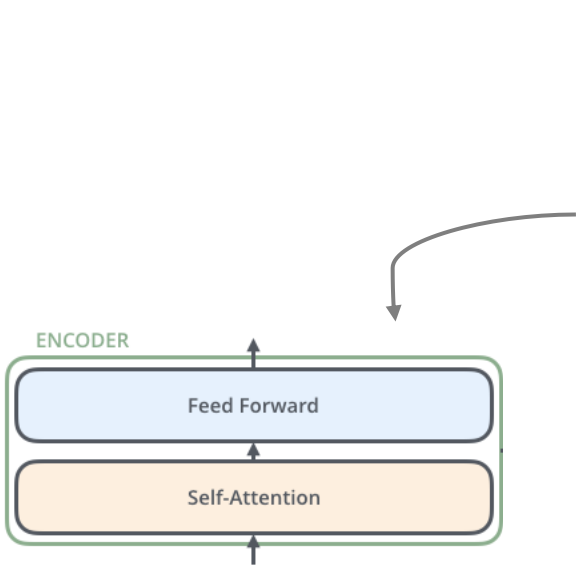


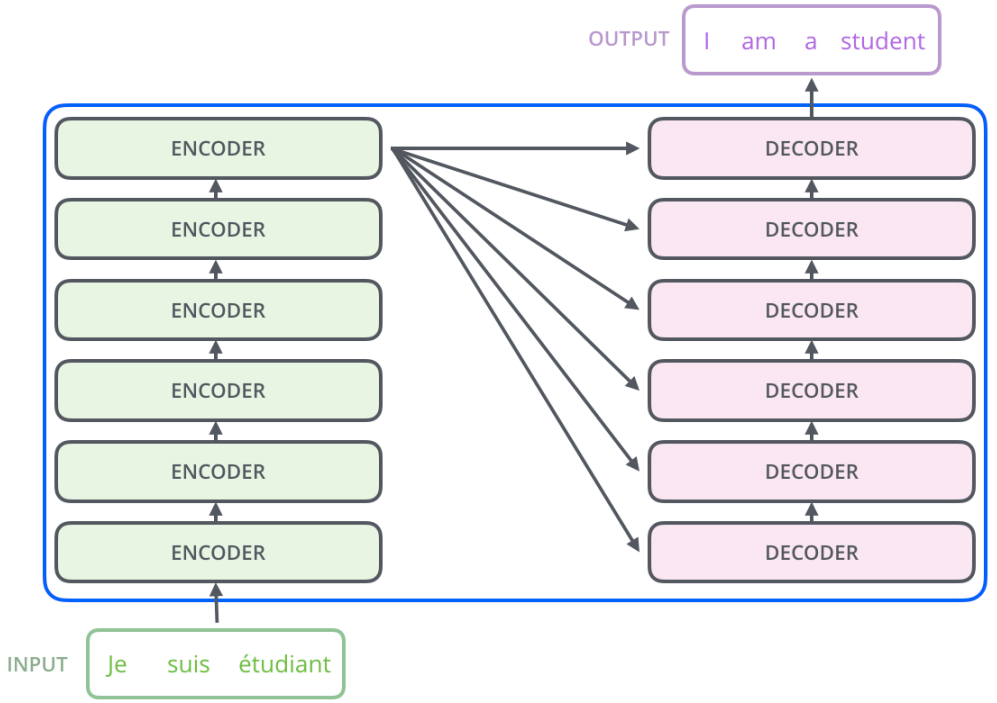
Transformers

The encoder's inputs first flow through a self-attention layer – a layer that helps the encoder look at other words in the input sentence as it encodes a specific word

The outputs of the self-attention layer are fed to a feed-forward neural network.

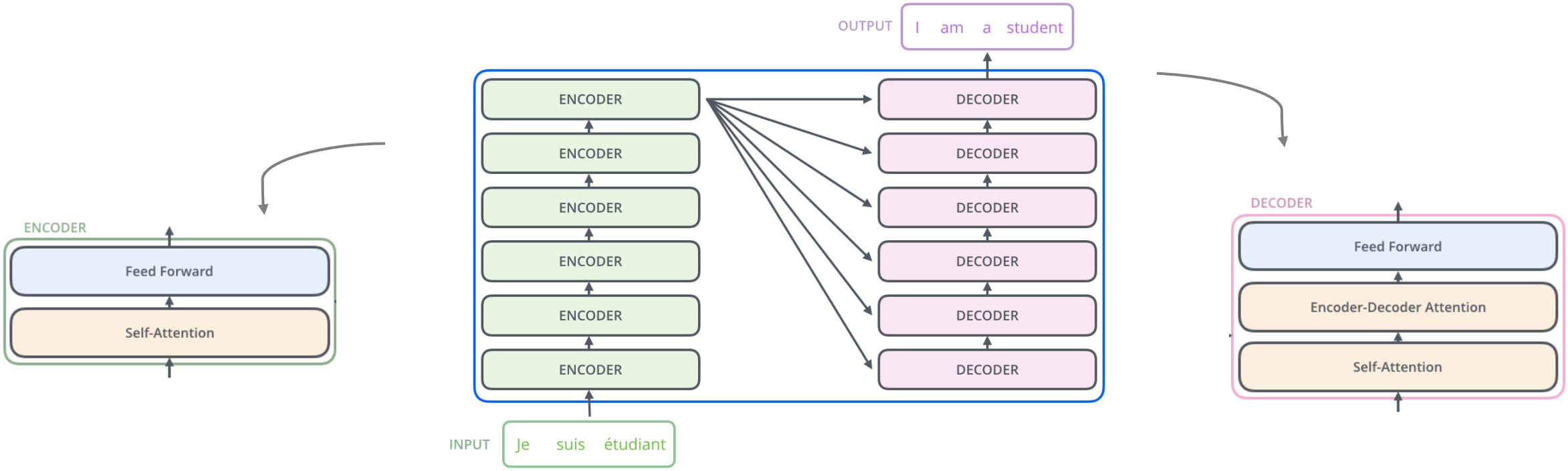
The exact same feed-forward network is independently applied to each position (each word/token).



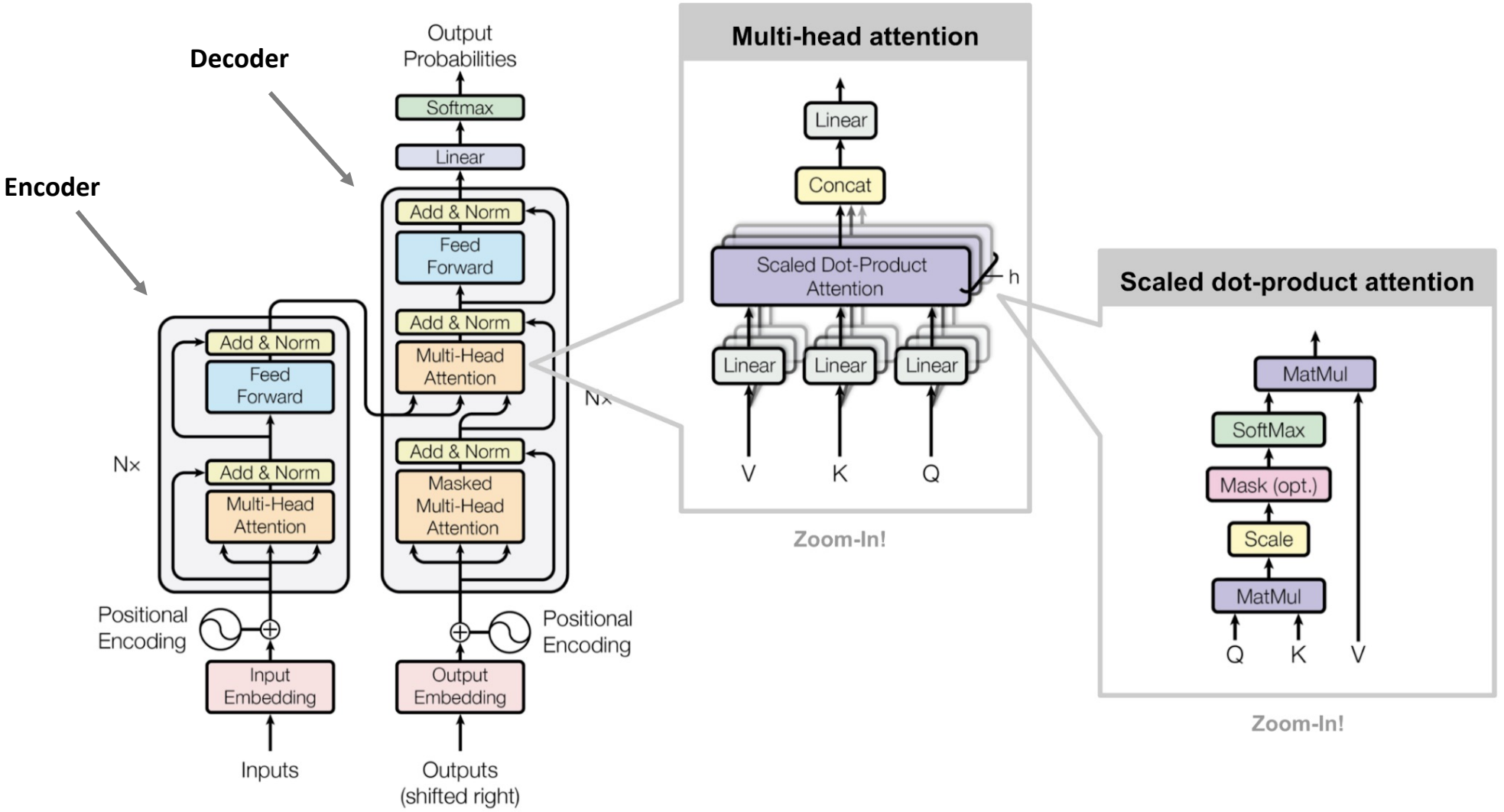


Transformers

The decoder has both those layers, but between them is an attention layer that helps the decoder focus on relevant parts of the input sentence



Transformers

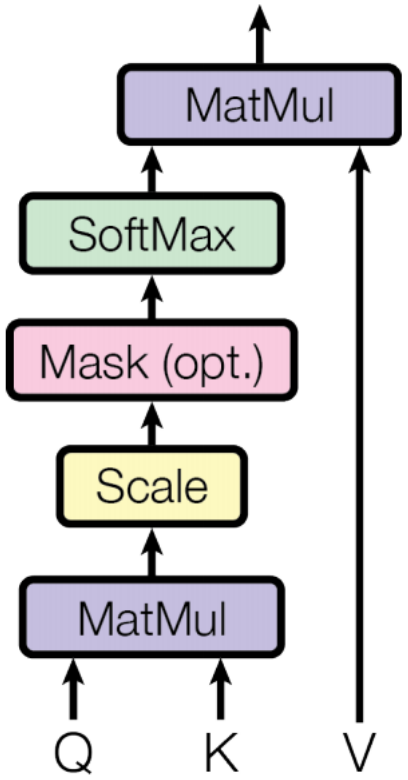


Transformers

Self-attention = Scaled dot-product attention

The output is a weighted sum of the values, where the weight assigned to each value is determined by the dot-product of the query with all the keys

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



Scaled Dot-Product Attention

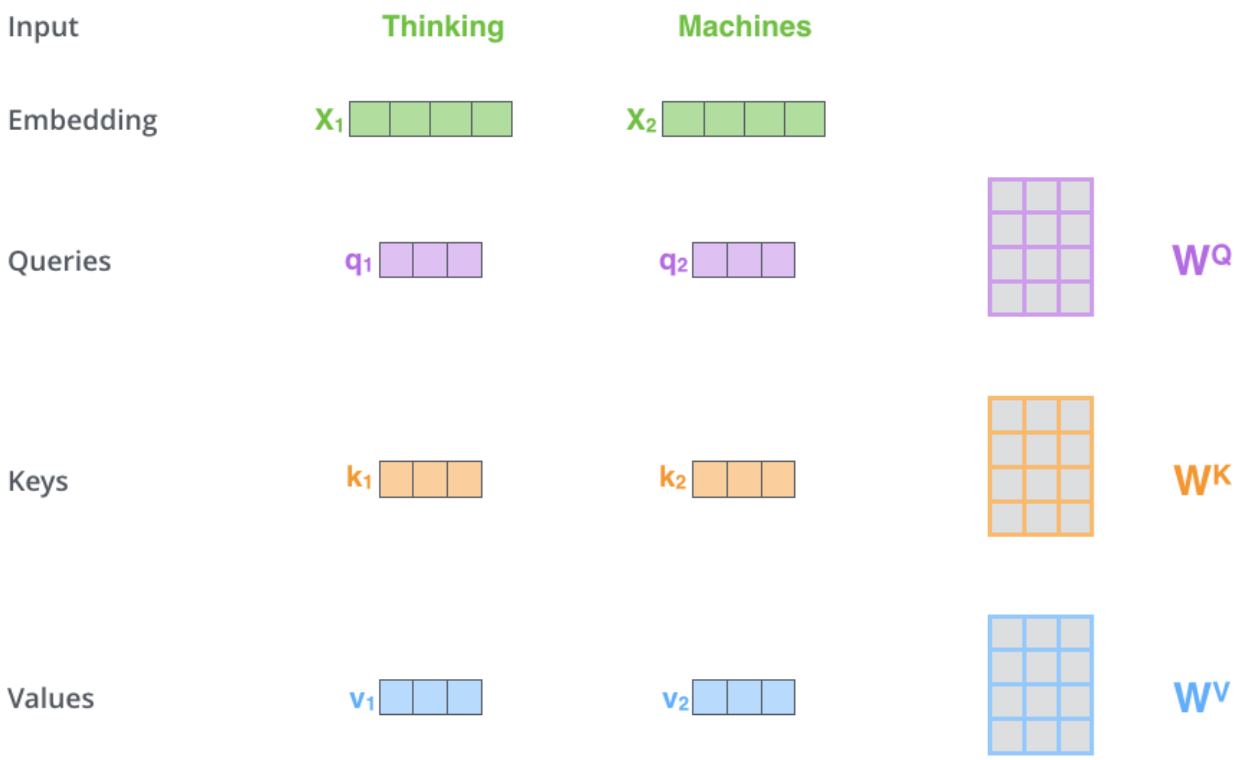
Self-Attention in Detail

First Step

Create three vectors from each of the encoder's input vectors (in this case, the **embedding** of each word).

So for each word, we create a **Query vector**, a **Key vector**, and a **Value vector**.

These vectors are created by multiplying the embedding by three matrices that we trained during the training process.



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

Self-Attention in Detail

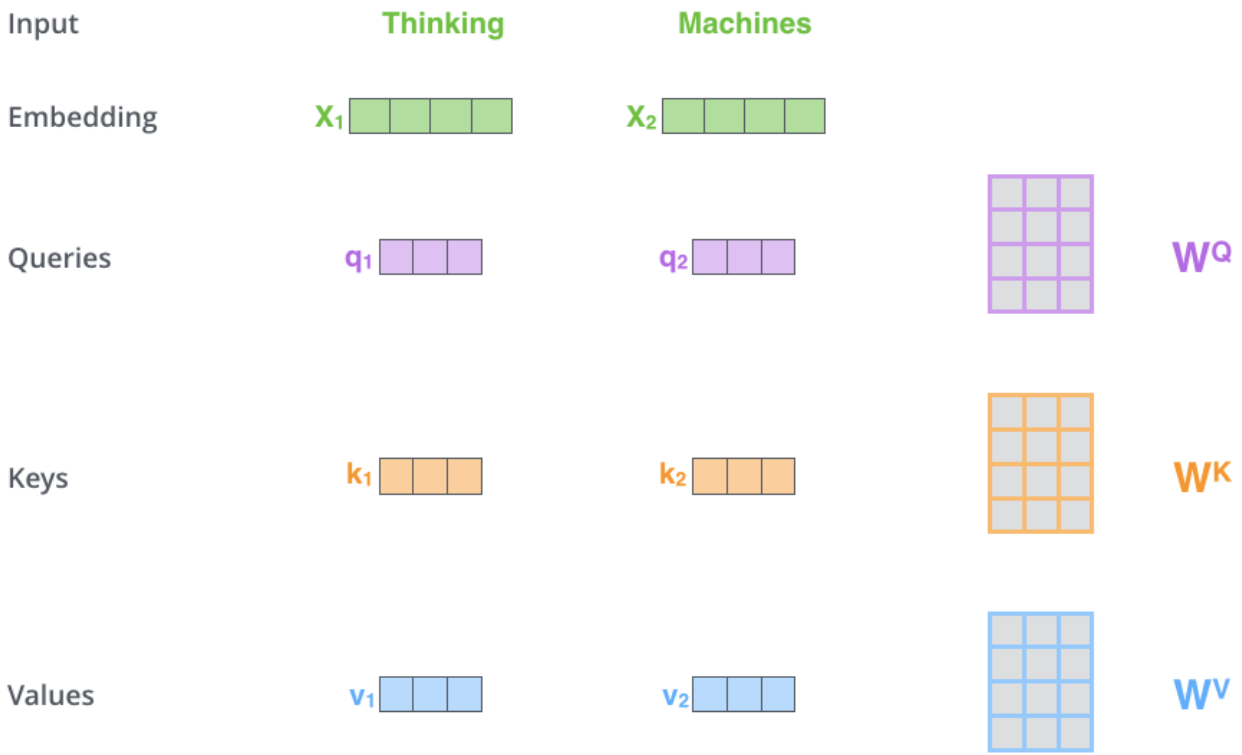
First Step

What are the "query", "key", and "value" vectors?

The names *"query", "key"* are inherited from the field of information retrieval

The dot product operation returns a measure of similarity between its inputs, so the weights $\frac{\mathbf{Q}\mathbf{K}^{\mathrm{T}}}{\sqrt{d_k}}$ depend on the relative similarities between the n-th query and all of the keys

The softmax function means that the key vectors "compete" with one another to contribute to the final result.



$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{SoftMax}\left(\frac{\mathbf{Q}\mathbf{K}^{\mathrm{T}}}{\sqrt{d_k}}\right)\mathbf{V}$$

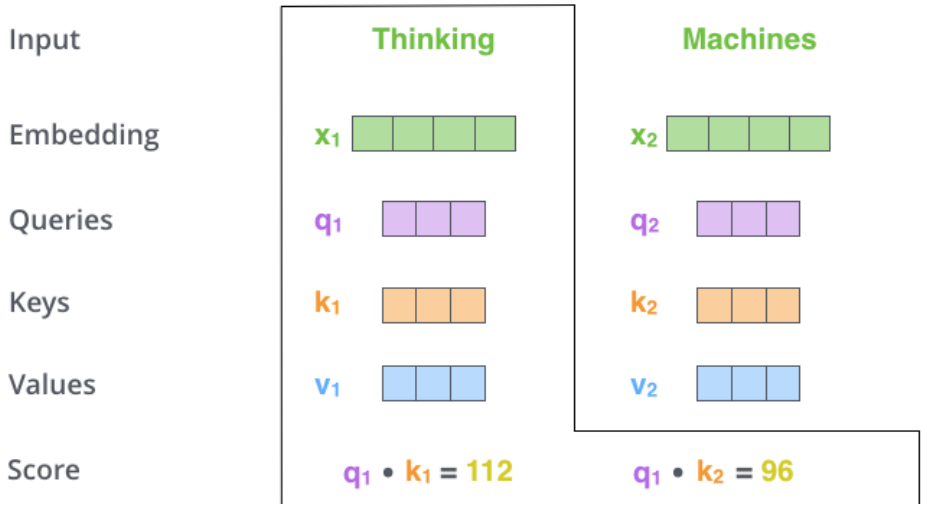
Self-Attention in Detail

Second Step

Calculate a score for each word of the input sentence against a word.

The score determines how much focus to place on other parts of the input sentence as we encode a word at a certain position.

The score is calculated by taking the dot product of the query vector with the key vector of the respective word we're scoring.



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

Self-Attention in Detail

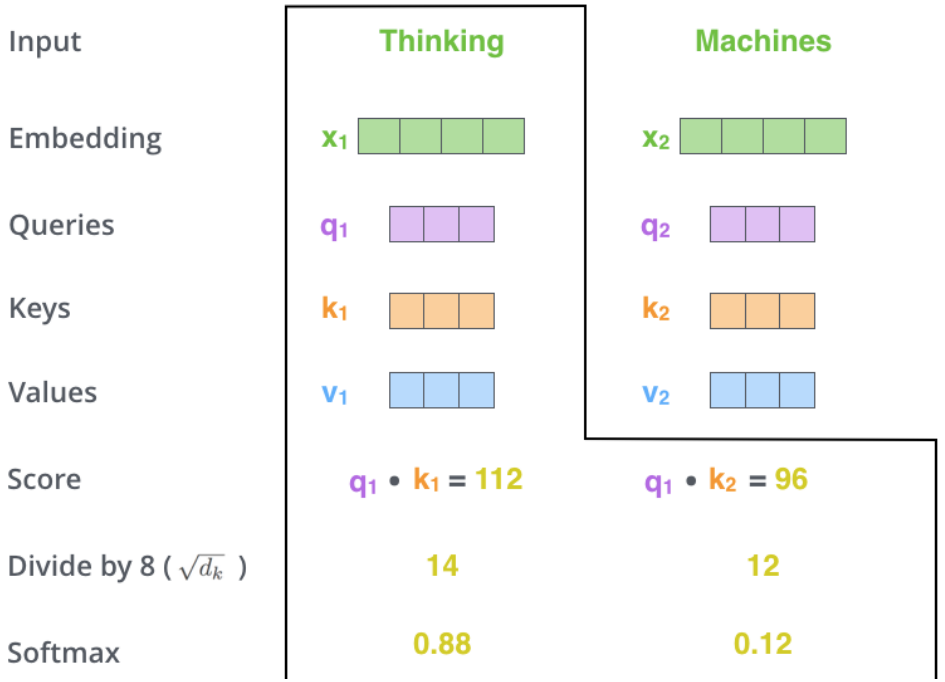
Third Step

Divide the scores by $\sqrt{d_k}$, the square root of the dimension of the key vectors

This leads to having more stable gradients (large similarities will cause softmax to saturate and give vanishing gradients)

Fourth Step

Softmax for normalization



$$z_1 = q_1 . k_1 / \sqrt{d_k} = 112/\sqrt{64} = 112/8 = 14, z_2 = q_1 . k_2 / \sqrt{d_k} = 96/\sqrt{64} = 96/8 = 12$$

$$softmax(z_1) = exp(z_1) / \sum_{i=1}^{2}(exp(z_i)) = 0.88, softmax(z_2) = exp(z_2) / \sum_{i=1}^{2}(exp(z_i)) = 0.12$$

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

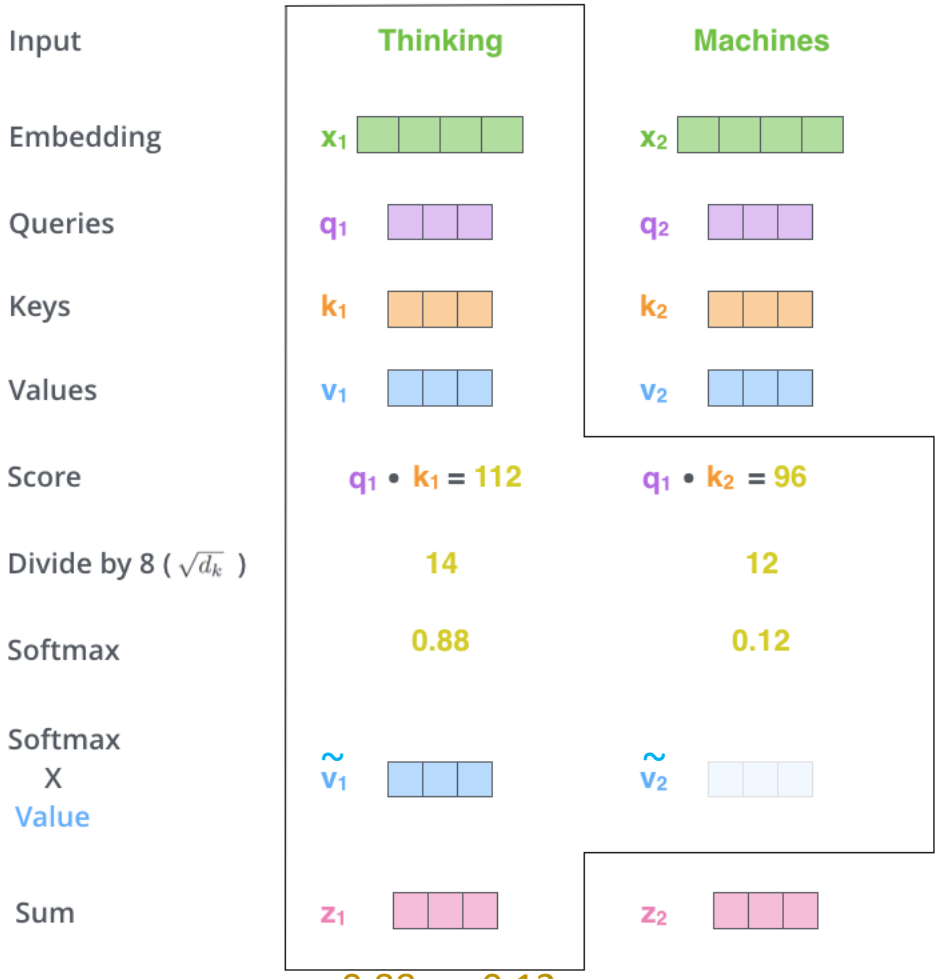
Self-Attention in Detail

Fifth Step

Multiply each value vector by the softmax score

Sixth Step

Sum up the weighted value vectors to get the output of the self-attention layer at this position (for the first word)



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

Matrix Calculation of Self-Attention

First Step

Calculate the Query, Key, and Value matrices.

Pack our embeddings into a matrix X, and multiplying it by the weight matrices we've trained (W^Q, W^K, W^V)

Every row in the X matrix corresponds to a word in the input sentence.

Second Step

Calculate the outputs of the self-attention layer.
SoftMax is row-wise

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

Example

Assuming we have two sequences:
$(1, 2, 3)$
$(4, 5, 6)$

And the given $\mathbf{W}^Q$, $\mathbf{W}^K$, $\mathbf{W}^V$ matrices are respectively given as

$$\begin{pmatrix} 0.01 & 0.03 \\ 0.02 & 0.02 \\ 0.03 & 0.01 \end{pmatrix}, \begin{pmatrix} 0.05 & 0.05 \\ 0.06 & 0.05 \\ 0.07 & 0.05 \end{pmatrix}, \begin{pmatrix} 0.02 & 0.02 \\ 0.01 & 0.02 \\ 0.01 & 0.01 \end{pmatrix}$$

Compute the output of the scaled-dot product attention, Attention($\mathbf{Q}$, $\mathbf{K}$, $\mathbf{V}$).

eg9.1.ipynb

Example

Step 1: Find $\mathbf{Q}, \mathbf{K}, \mathbf{V}$

$$\mathbf{Q} = \mathbf{X}\mathbf{W}^{Q} = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix} \begin{pmatrix} 0.01 & 0.03 \\ 0.02 & 0.02 \\ 0.03 & 0.01 \end{pmatrix} = \begin{pmatrix} 0.14 & 0.10 \\ 0.32 & 0.28 \end{pmatrix}$$

$$\mathbf{K} = \mathbf{X}\mathbf{W}^{K} = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix} \begin{pmatrix} 0.5 & 0.5 \\ 0.6 & 0.5 \\ 0.7 & 0.5 \end{pmatrix} = \begin{pmatrix} 0.38 & 0.30 \\ 0.92 & 0.75 \end{pmatrix}$$

$$\mathbf{V} = \mathbf{X}\mathbf{W}^{V} = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix} \begin{pmatrix} 0.02 & 0.02 \\ 0.01 & 0.02 \\ 0.01 & 0.01 \end{pmatrix} = \begin{pmatrix} 0.07 & 0.09 \\ 0.19 & 0.24 \end{pmatrix}$$

eg9.1.ipynb

Example

Step 2: Find Attention$(\mathbf{Q}, \mathbf{K}, \mathbf{V})$

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{Softmax}\left(\frac{\mathbf{Q}\mathbf{K}^{\mathrm{T}}}{\sqrt{d_k}}\right)\mathbf{V}$$

$$\frac{e^{z_j}}{\sum_j e^{z_j}} = \frac{e^{0.0588}}{e^{0.0588} + e^{0.1441}} = 0.4787$$

Perform row-wise SoftMax

$$\text{Softmax}\left(\frac{\mathbf{Q}\mathbf{K}^{\mathrm{T}}}{\sqrt{d_k}}\right) = \text{Softmax}\left(\frac{\begin{pmatrix}0.14 & 0.10\\ 0.32 & 0.28\end{pmatrix}\begin{pmatrix}0.38 & 0.92\\ 0.30 & 0.75\end{pmatrix}}{\sqrt{2}}\right) = \text{Softmax}\begin{pmatrix}0.0588 & 0.1441\\ 0.1454 & 0.3566\end{pmatrix} = \begin{pmatrix}0.4787 & 0.5213\\ 0.4474 & 0.5526\end{pmatrix}$$

$$\text{Softmax}\left(\frac{\mathbf{Q}\mathbf{K}^{\mathrm{T}}}{\sqrt{d_k}}\right)\mathbf{V} = \begin{pmatrix}0.4787 & 0.5213\\ 0.4474 & 0.5526\end{pmatrix}\begin{pmatrix}0.07 & 0.09\\ 0.19 & 0.24\end{pmatrix} = \begin{pmatrix}0.1326 & 0.1682\\ 0.1363 & 0.1729\end{pmatrix}$$

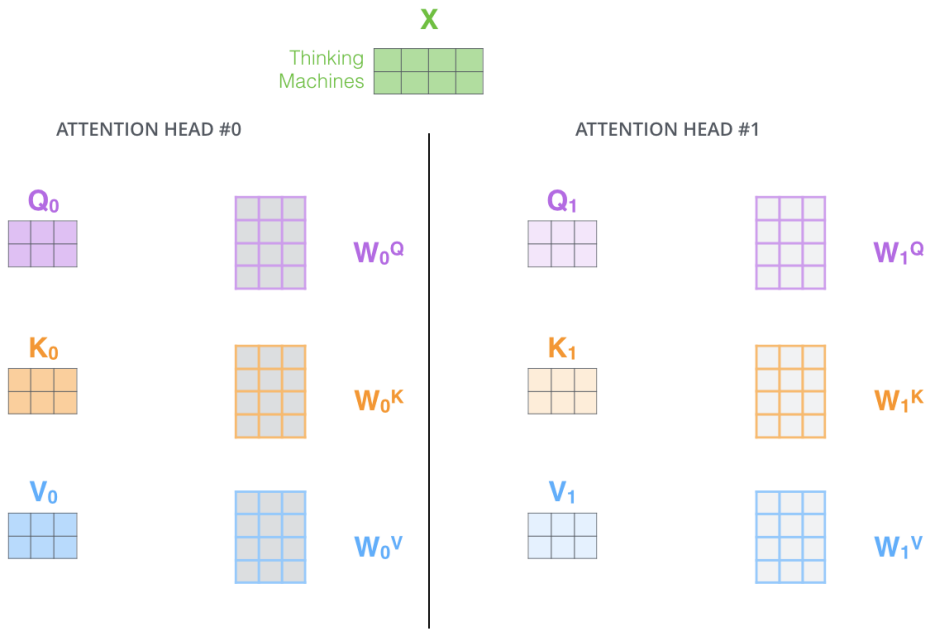
eg9.1.ipynb

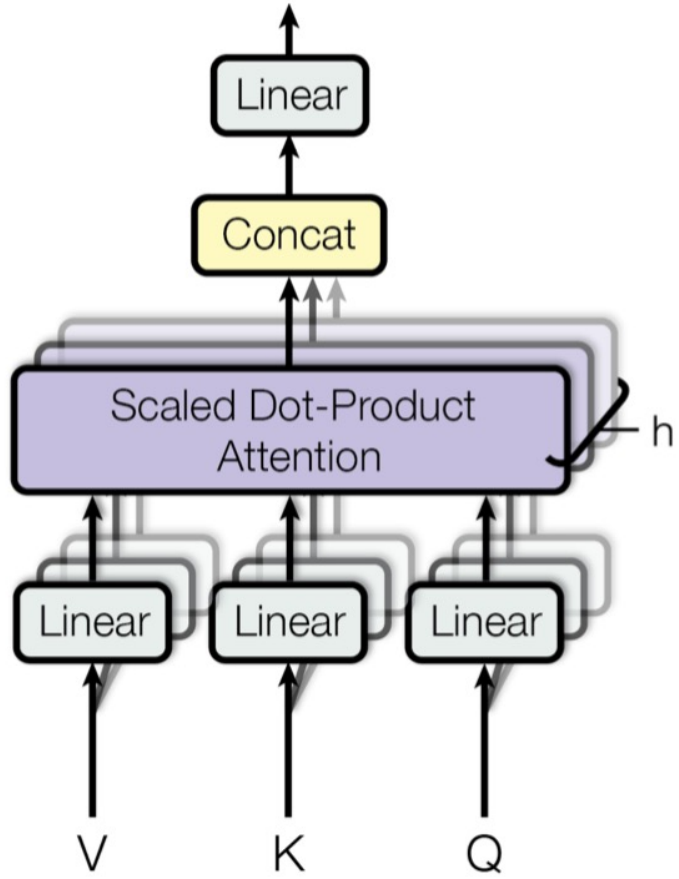
Multi-Head Self-Attention

Multi-Head Self-Attention

Rather than only computing the attention once, the multi-head mechanism runs through the scaled dot-product attention multiple times in parallel.

Due to different linear mappings, each head is presented with different versions of keys, queries, and values



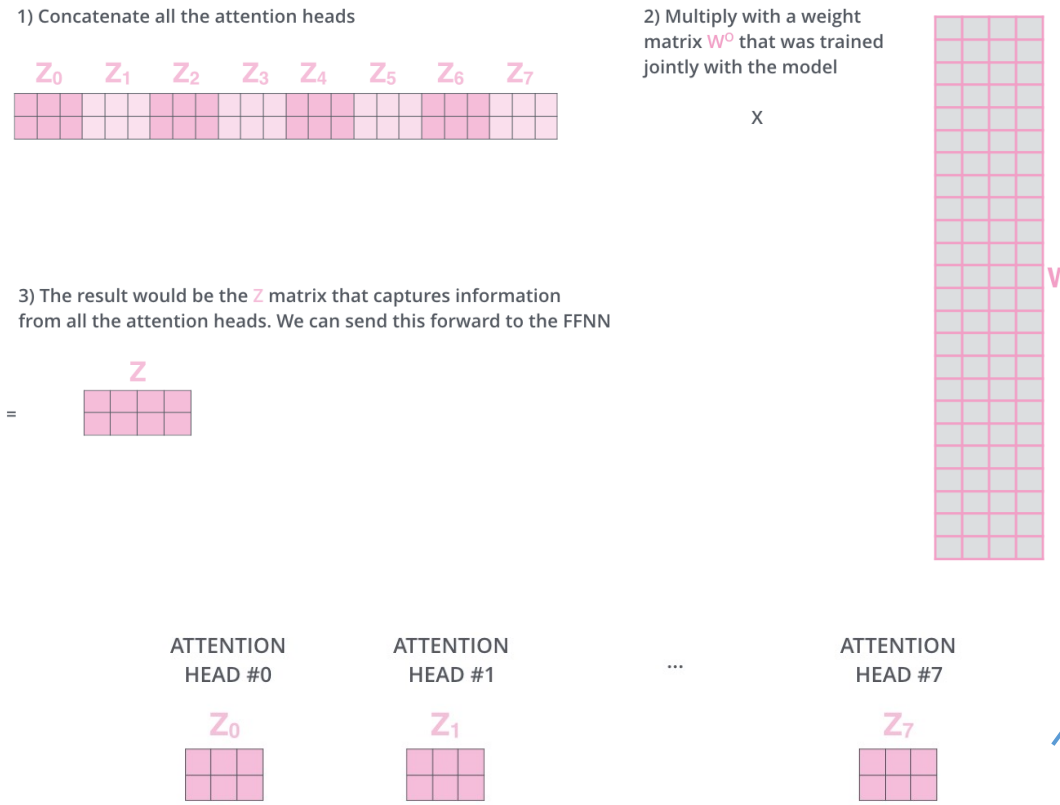


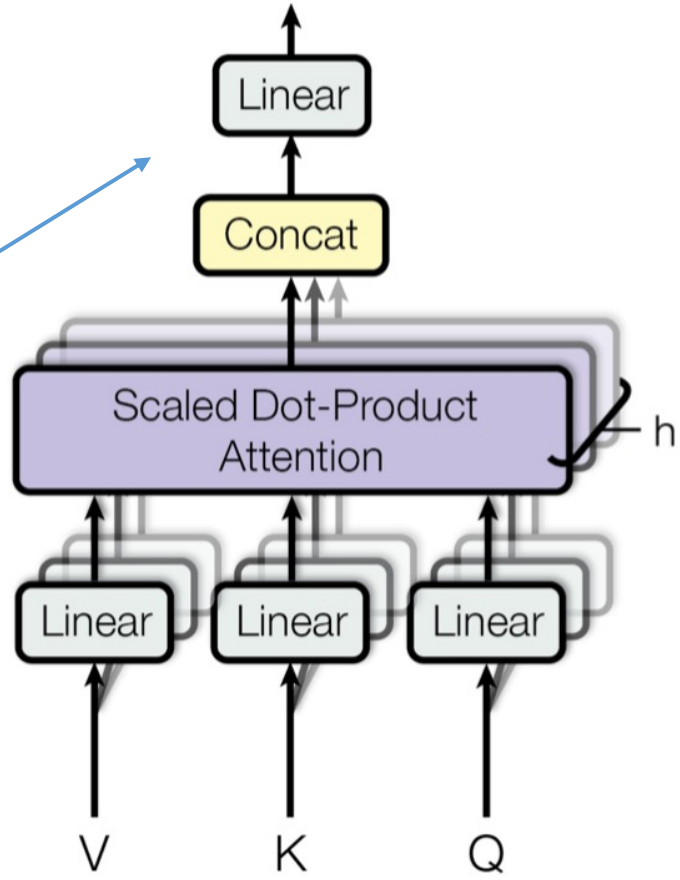
Multi-head Attention

Multi-Head Self-Attention

Multi-Head Self-Attention

The independent attention outputs are simply concatenated and linearly transformed into the expected dimensions.



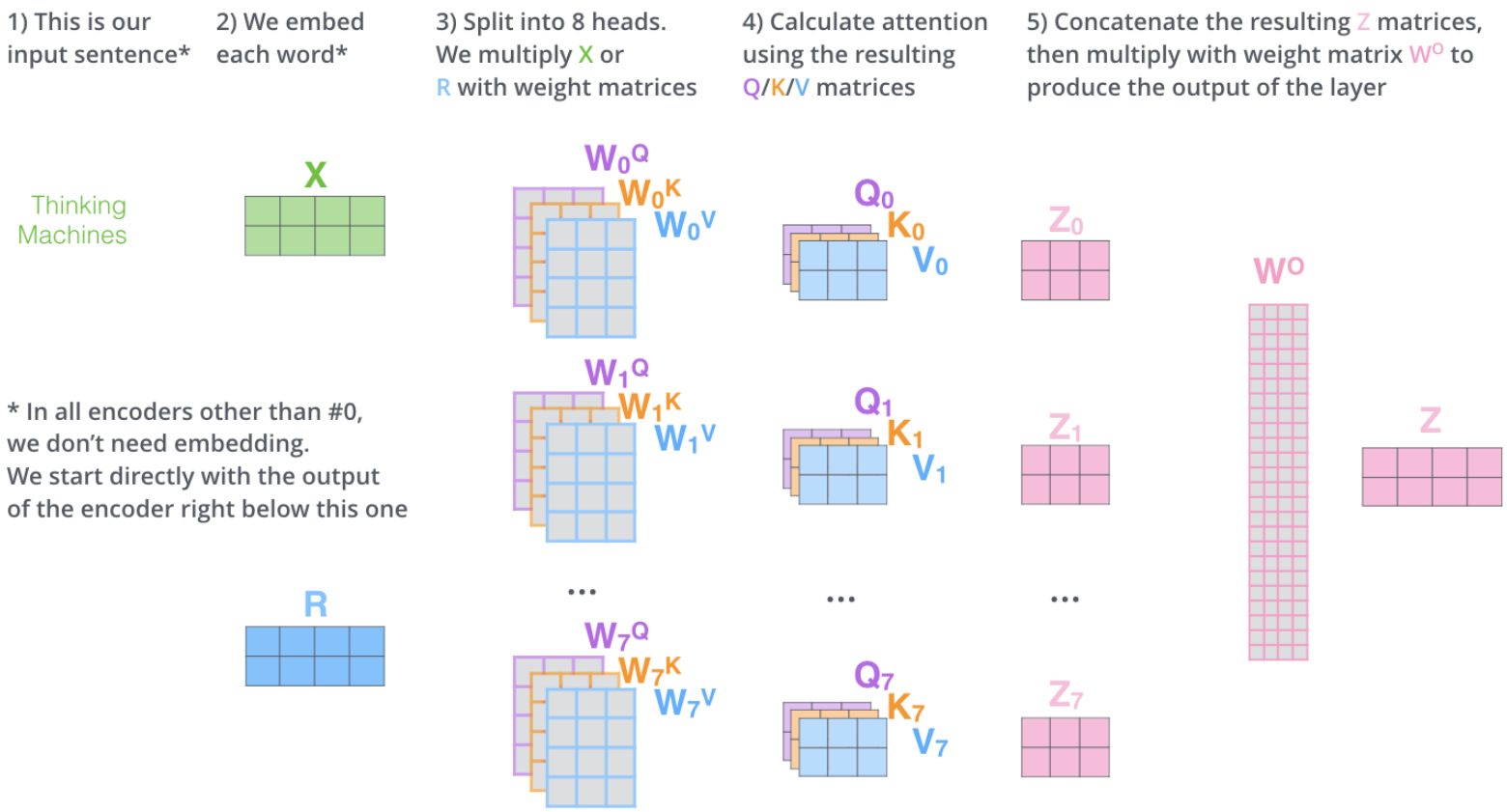


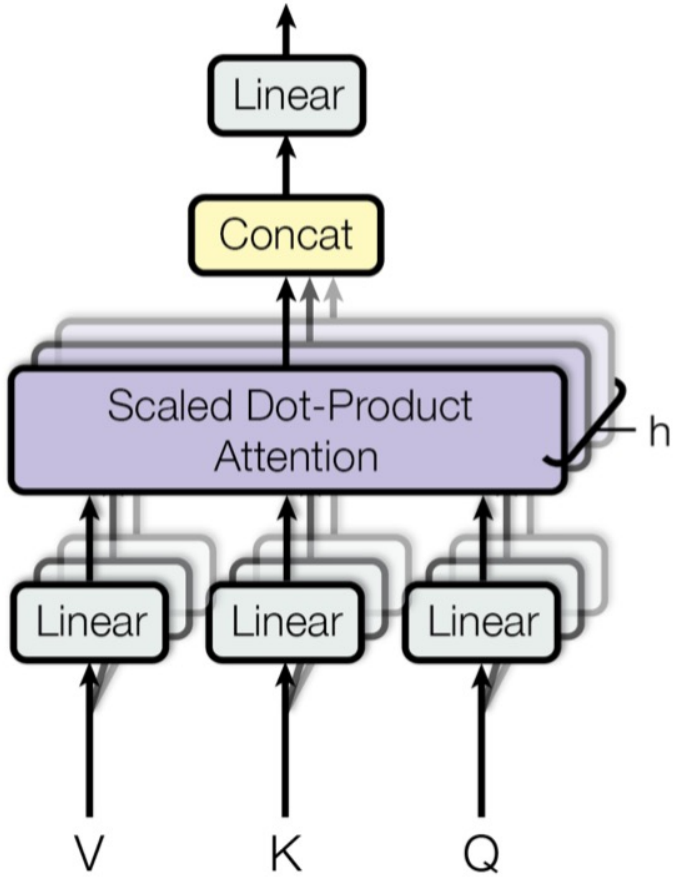
Multi-head Attention

Multi-Head Self-Attention

Multi-Head Self-Attention

The big picture. Note that after the split each head can have a reduced dimensionality, so the total computation cost is the same as a single head attention with full dimensionality.





Multi-head Attention

Multi-Head Self-Attention

Why?

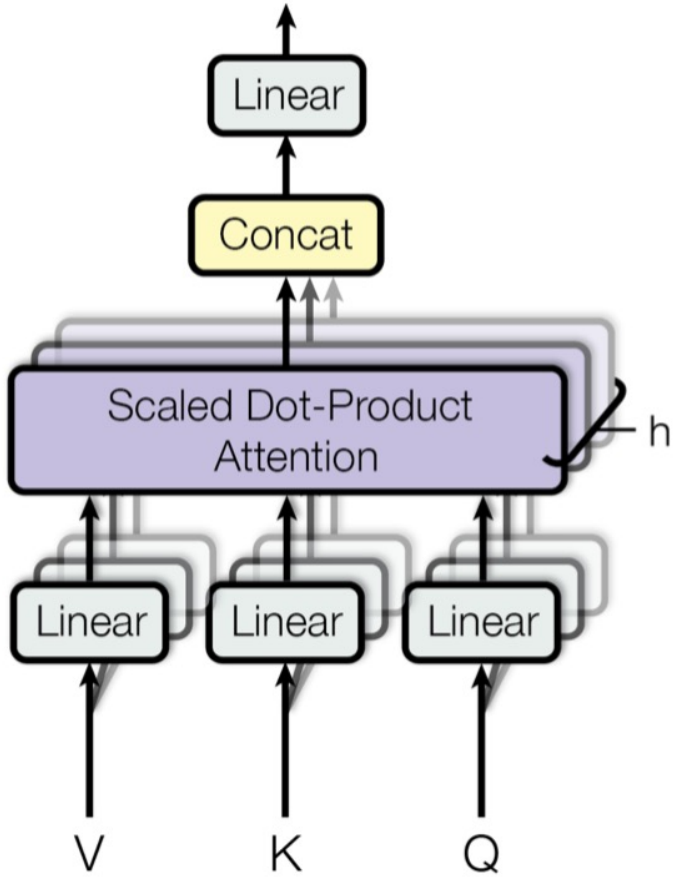
"Multi-head attention allows the model to jointly attend to information from different representation subspaces at different positions. "

An intuitive example

Given a sentence:

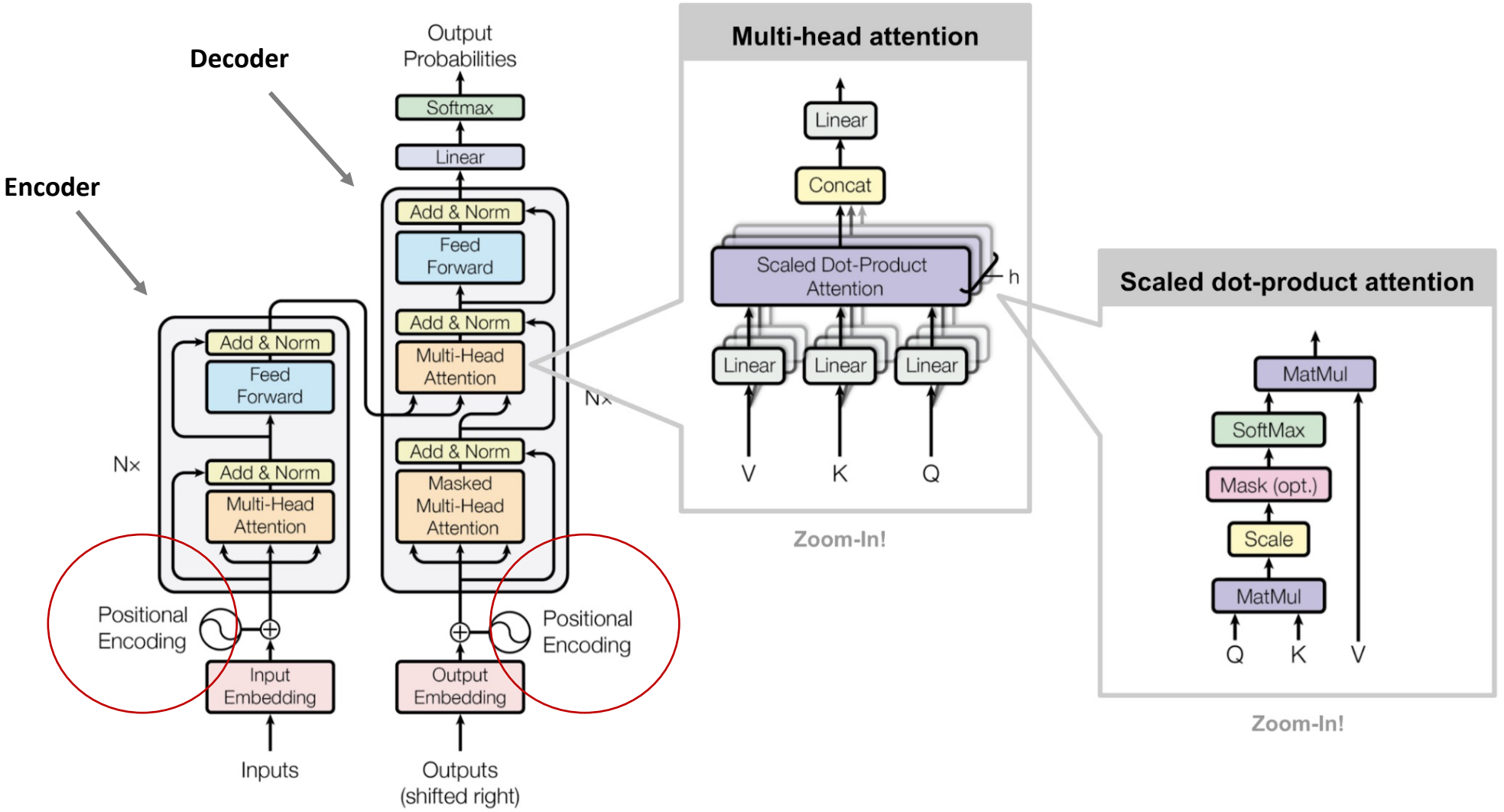
"Deep learning (also known as deep structured learning) is part of a broader family of machine learning methods based on artificial neural networks with representation learning."

Given "representation learning", the first head attends to "Deep learning" while the second head attends to the more general term "machine learning methods".



Multi-head Attention

Positional Encoding



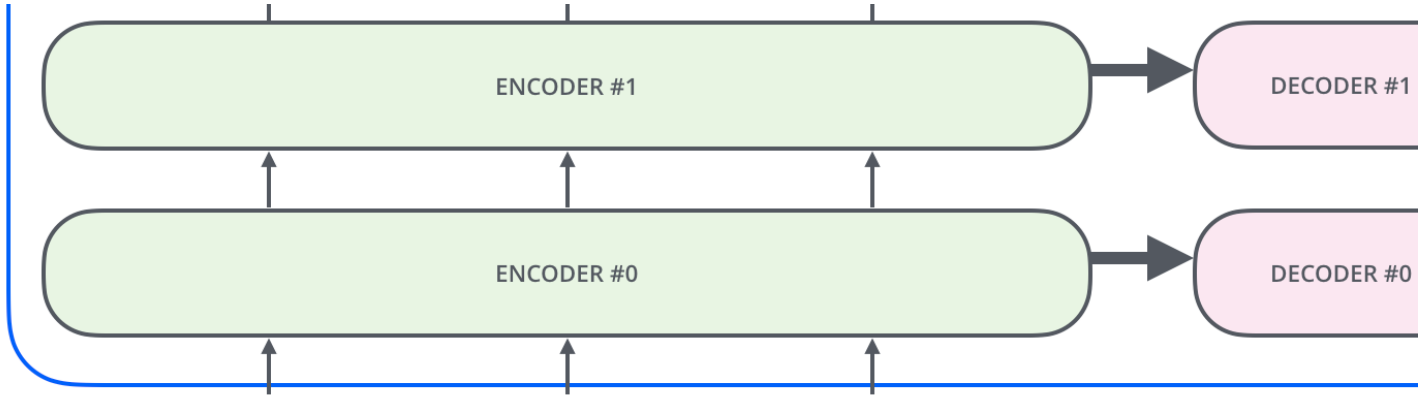
Positional Encoding

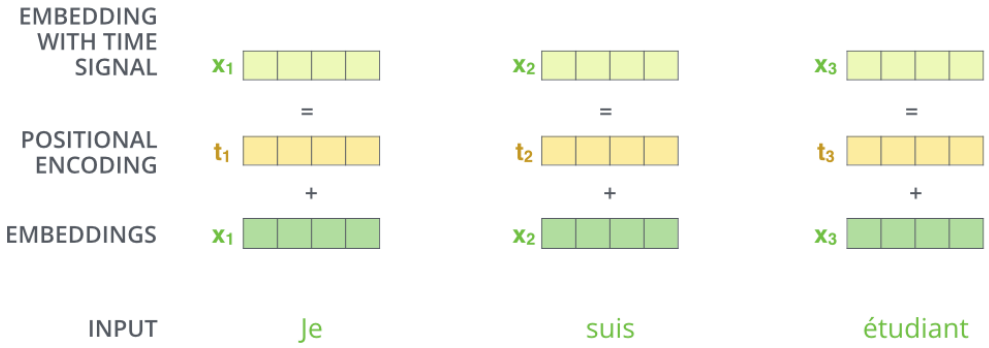
Self-attention layer works on sets of vectors and it doesn't know the order of the vectors it is processing

The positional encoding has the same dimension as the input embedding

Adds a vector to each input embedding to give information about the relative or absolute position of the tokens in the sequence

These vectors follow a specific pattern





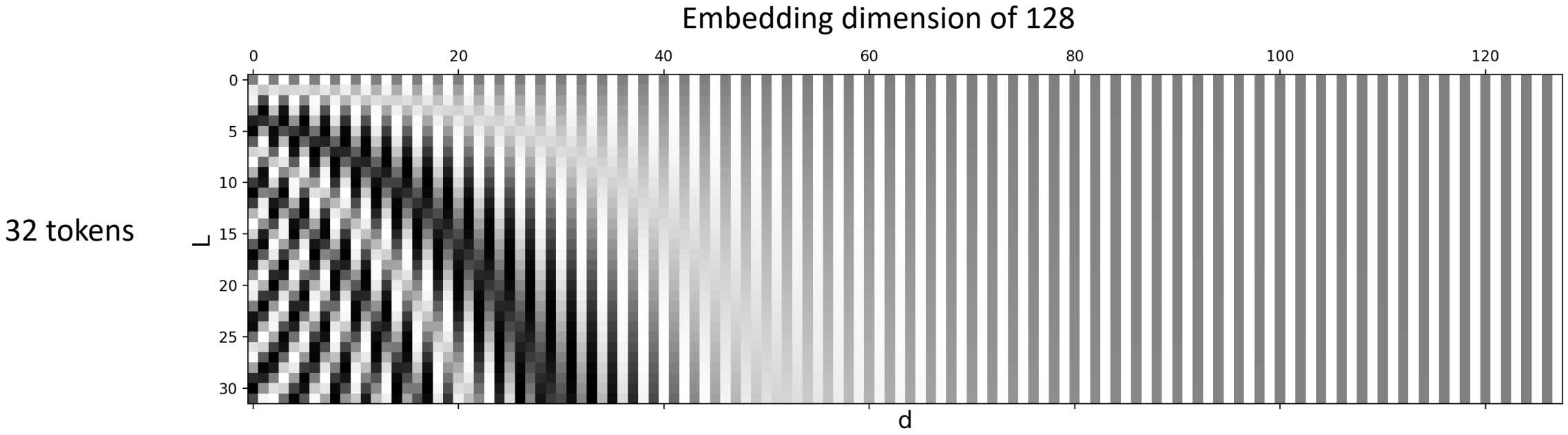
Positional Encoding

What might this pattern look like?

Each row corresponds the a positional encoding of a vector.

The first row would be the vector we'd add to the embedding of the first word in an input sequence.

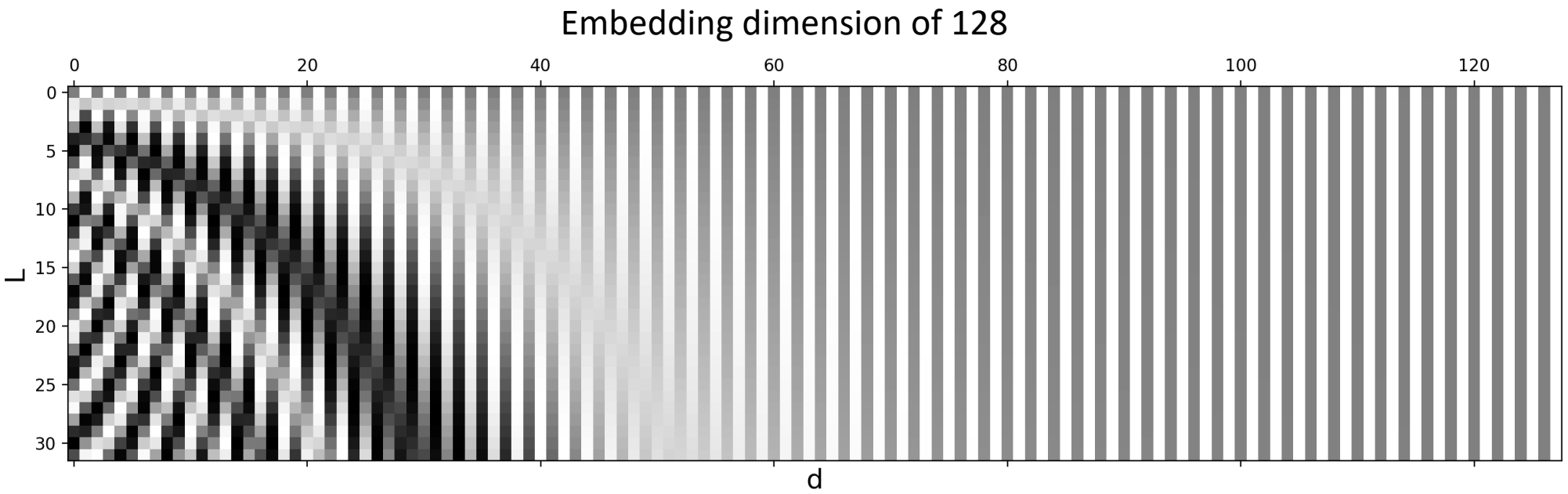
Each position is uniquely encoded and the encoding can deal with sequences longer than any sequence seen in the training time.



Sinusoidal positional encoding with 32 tokens and embedding dimension of 128. The value is between -1 (black) and 1 (white) and the value 0 is in gray.

Positional Encoding

32 tokens



*Sinusoidal positional encoding - interweaves the two signals (*sine for even indices and cosine for odd indice*)*

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

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

where pos is the position and i is the dimension, $[0, \ldots, d_{model}/2\ )$

Example: Positional Encoding

Example:

Given the following *Sinusoidal positional encoding*, calculate the $PE(pos = 1)$ for the first five dimensions [0, 1, 2, 3, 4]. Assume $d_{model} = 512$

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

Solution:

Given $pos = 1$ and $d_{model} = 512$

At dimension 0, $2i = 0$ thus $i = 0$, therefore $PE_{(pos,2i)} = PE_{(1,0)} = \sin(1/10000^{0/512})$

At dimension 1, $2i + 1 = 1$ thus $i = 0$, therefore $PE_{(pos,2i+1)} = PE_{(1,1)} = \cos(1/10000^{0/512})$

At dimension 2, $2i = 2$ thus $i = 1$, therefore $PE_{(pos,2i)} = PE_{(1,2)} = \sin(1/10000^{2/512})$

At dimension 3, $2i + 1 = 3$ thus $i = 1$, therefore $PE_{(pos,2i+1)} = PE_{(1,3)} = \cos(1/10000^{2/512})$

At dimension 4, $2i = 4$ thus $i = 2$, therefore $PE_{(pos,2i)} = PE_{(1,4)} = \sin(1/10000^{4/512})$

Implementation

```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:
        """
        Arguments:
            x: Tensor, shape ``[seq_len, batch_size, embedding_dim]``
        """
        x = x + self.pe[:x.size(0)]
        return self.dropout(x)
```

The constructor initializes the module and sets up the dropout layer with the given dropout probability.

d_model is the dimension of the embeddings (or the depth of the model).

max_len is the maximum expected length of the sequences.

Implementation

```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:
        """
        Arguments:
            x: Tensor, shape ``[seq_len, batch_size, embedding_dim]``
        """
        x = x + self.pe[:x.size(0)]
        return self.dropout(x)
```

position is a tensor containing integers from 0 to max_len-1, representing each position in the sequence.

Implementation

```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:
        """
        Arguments:
            x: Tensor, shape ``[seq_len, batch_size, embedding_dim]``
        """
        x = x + self.pe[:x.size(0)]
        return self.dropout(x)
```

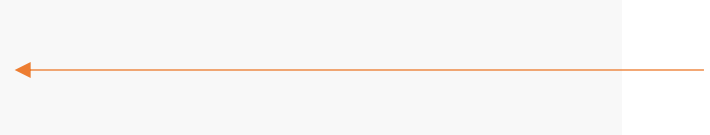
div_term is a scaling term used to adjust the rate of the sinusoidal functions.

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

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

Implementation

```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:
        """
        Arguments:
            x: Tensor, shape ``[seq_len, batch_size, embedding_dim]``
        """
        x = x + self.pe[:x.size(0)]
        return self.dropout(x)
```



The sinusoidal functions (sine for even indices and cosine for odd indices) are applied to the positions and the results are stored in pe. This creates a unique positional encoding for each position.

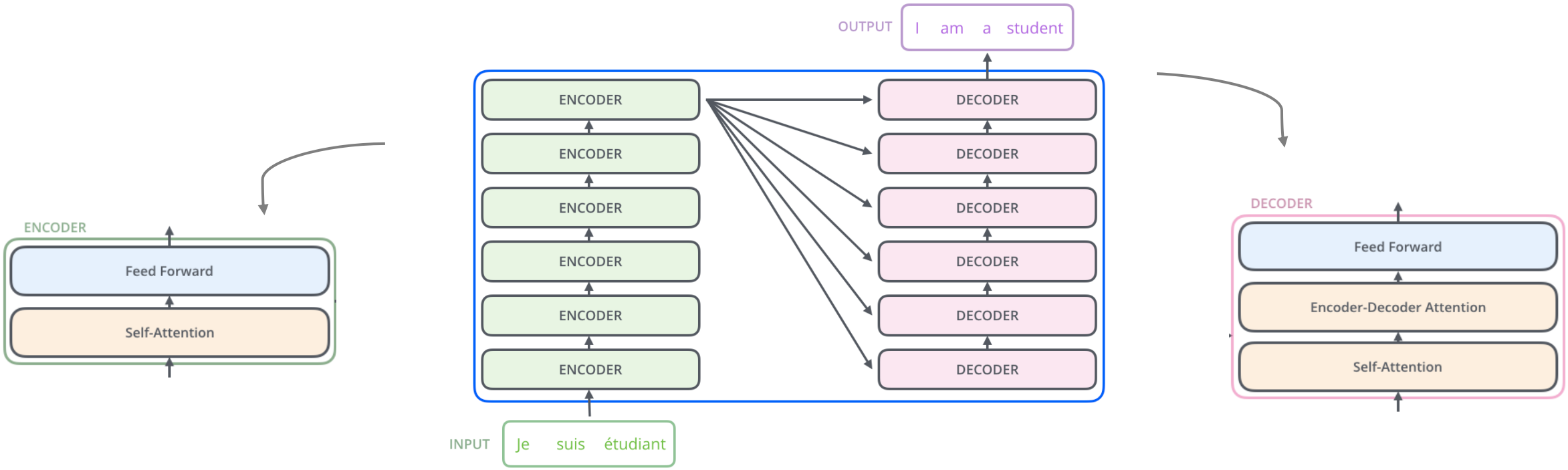
Implementation

```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:
        """
        Arguments:
            x: Tensor, shape ``[seq_len, batch_size, embedding_dim]``
        """
        x = x + self.pe[:x.size(0)]
        return self.dropout(x)
```

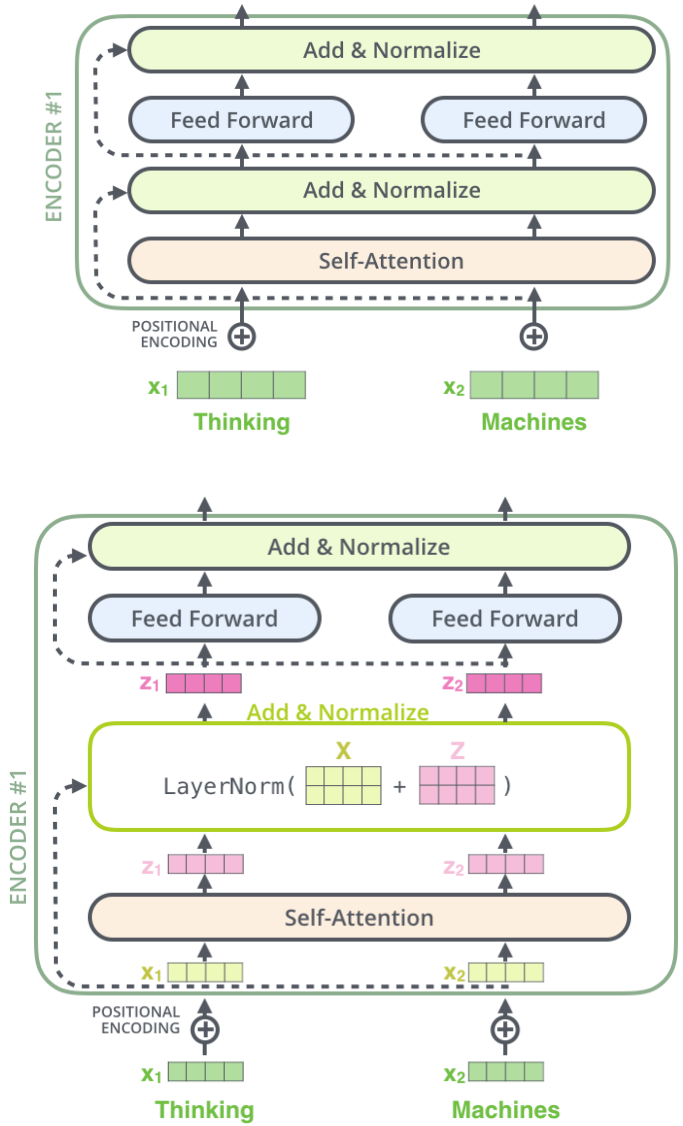
The positional encoding (self.pe) corresponding to the length of the input sequence is added to the input tensor x.

This addition operation effectively combines the positional information with the embeddings of the tokens.

Transformers



Transformer Encoder



If we're to visualize the vectors and the layer-norm operation associated with self attention, it would look like this:

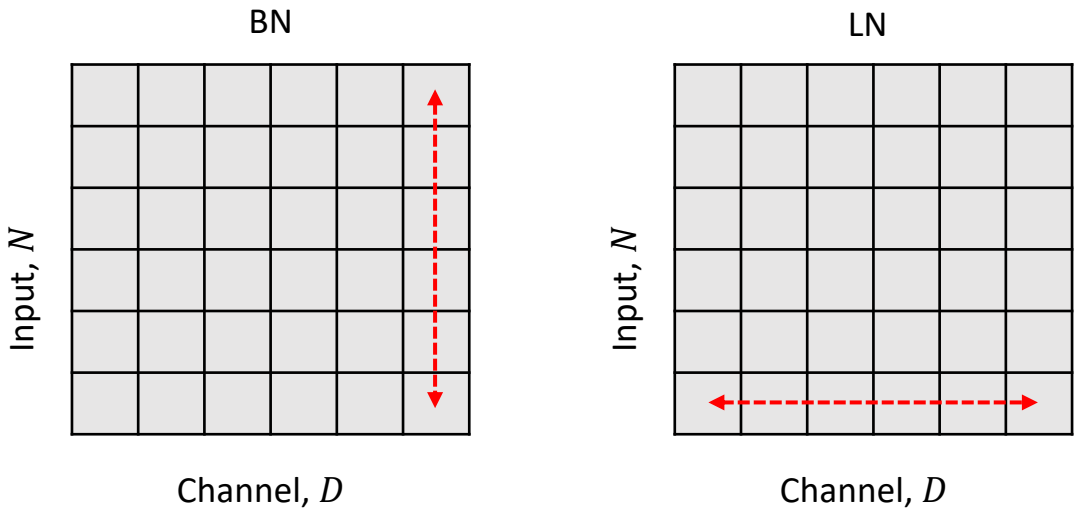
Encoder

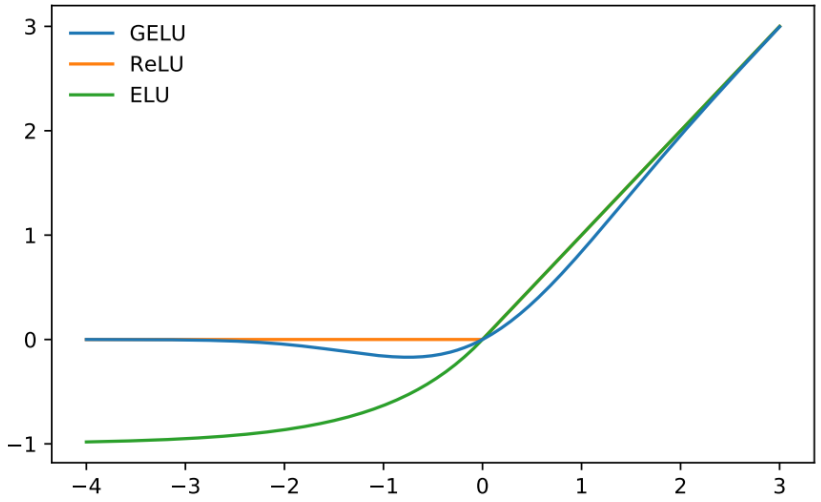
- A stack of $N = 6$ identical layers.
- Each layer has a multi-head self-attention layer and a simple position-wise fully connected feed-forward network.

$$\mathrm{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2$$

- The linear transformations are the same across different positions, they use different parameters from layer to layer.
- Each sub-layer adopts a residual connection and a layer normalization.

Transformer Encoder





Layer Normalization (LN) [1]

- The pixels along the red arrow are normalized by the same mean and variance, computed by aggregating the values of these pixels.
- BN is found unstable in Transformers [2]
- Works well with RNN and now being used in Transformers

Gaussian Error Linear Units (GELU) [3]

- Can be thought of as a smoother ReLU
- Used in GPT-3, BERT, and most other Transformers

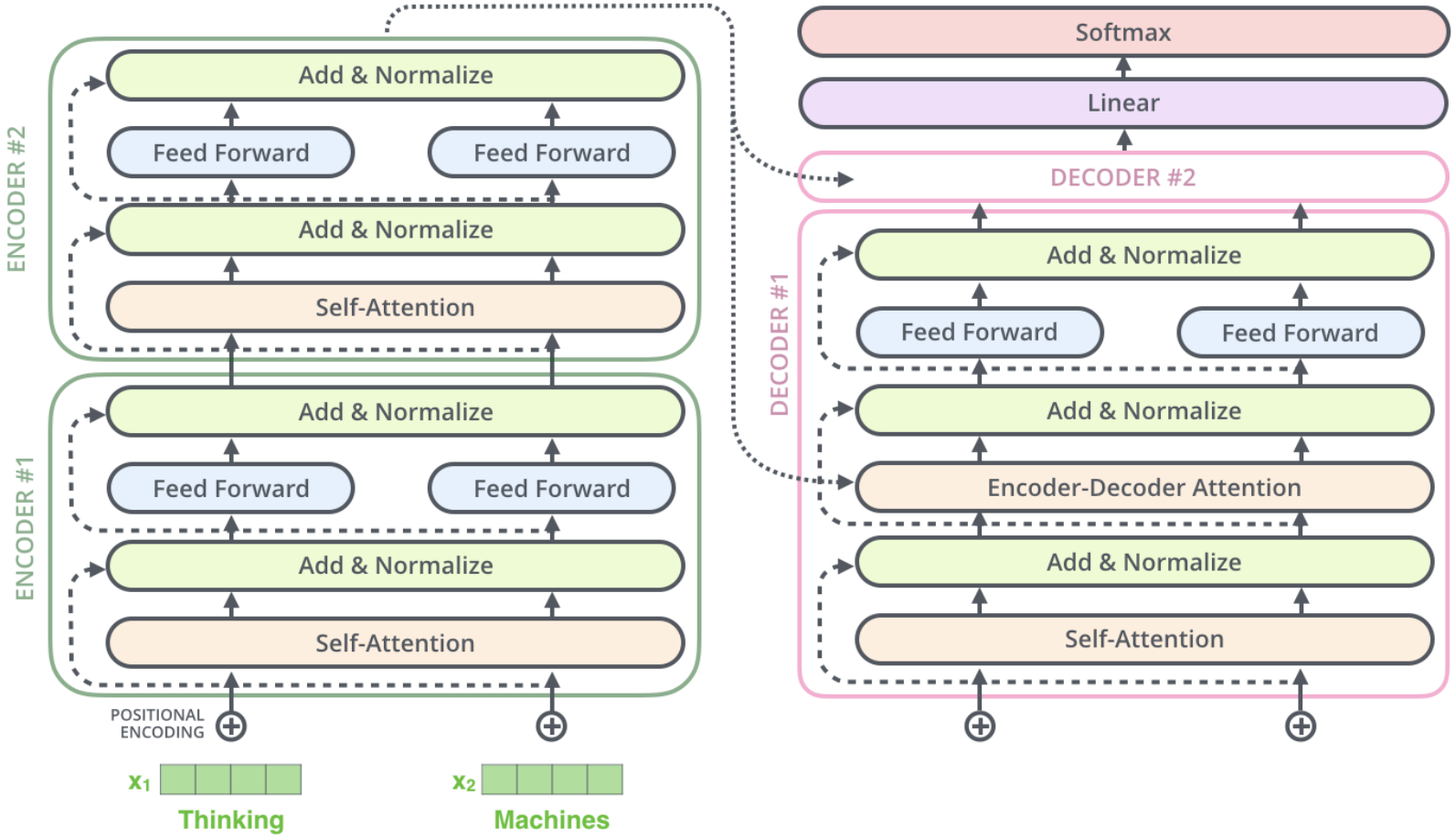
[1] Jimmy Lei Ba, Jamie Ryan Kiros, Geoffrey E. Hinton, Layer Normalization, arXiv:1607.06450

[2] Sheng Shen, Zhewei Yao, Amir Gholami, Michael Mahoney, Kurt Keutzer, Rethinking Batch Normalization in Transformers, ICML 2020

[3] Dan Hendrycks, Kevin Gimpel, Gaussian Error Linear Units (GELUs), arXiv:1606.08415

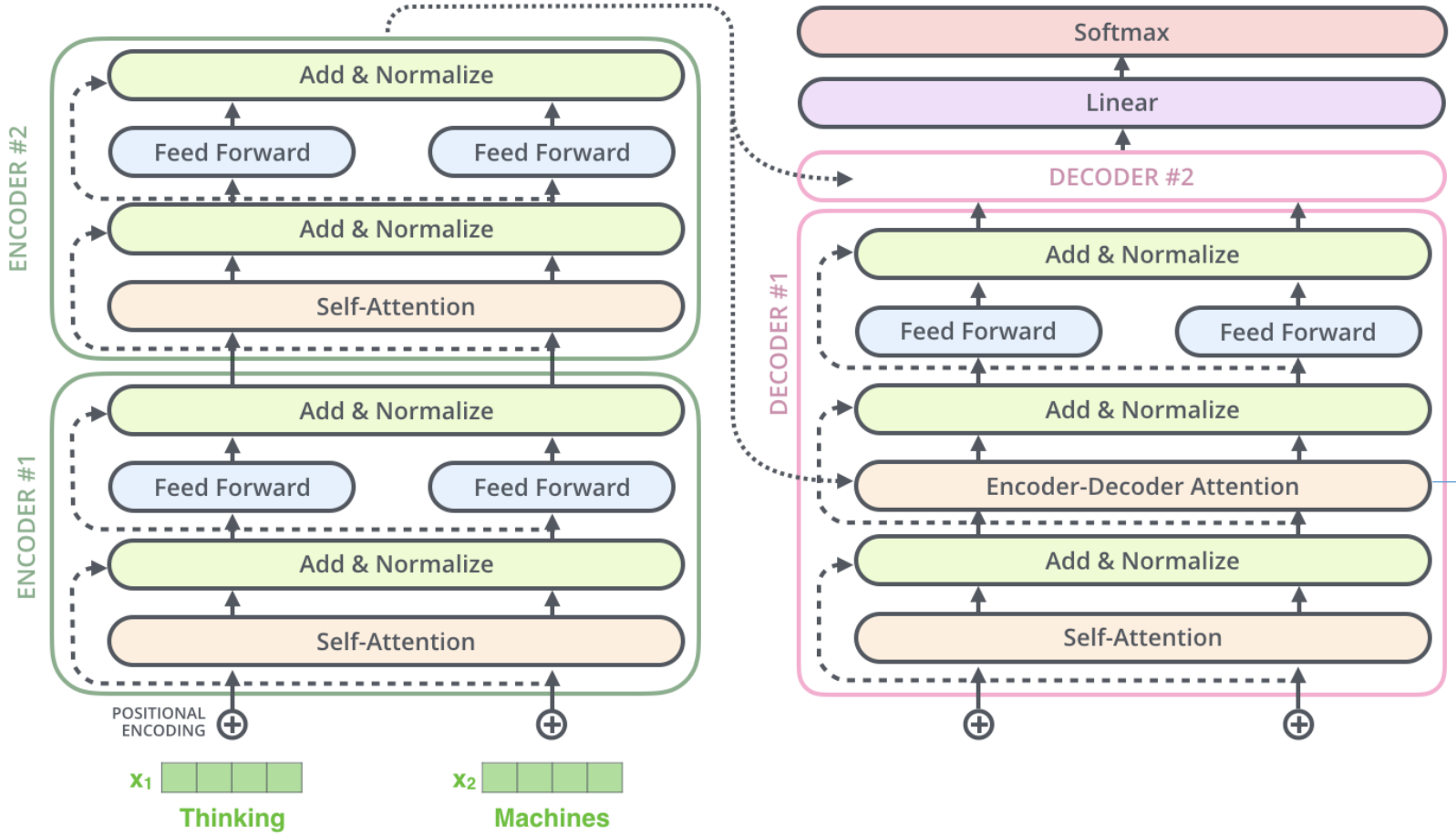
Transformer Encoder

The outputs of encoder go to the sub-layers of the decoder as well. If we're to think of a Transformer of two stacked encoders and decoders, it would look something like this:



Transformer Encoder

The outputs of encoder go to the sub-layers of the decoder as well. If we're to think of a Transformer of two stacked encoders and decoders, it would look something like this:

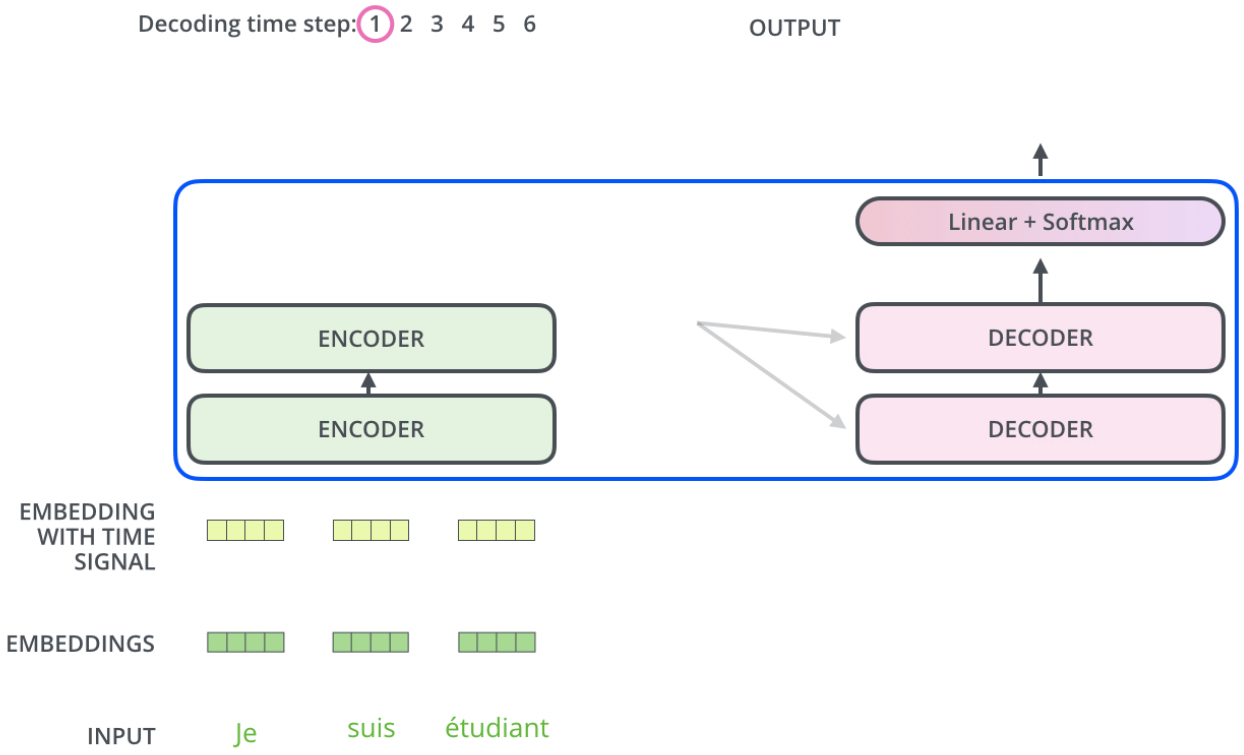


The "Encoder-Decoder Attention" layer works just like multiheaded self-attention, except it creates its Queries matrix from the layer below it, and takes the Keys and Values matrix from the output of the encoder stack.

Transformer Decoder

How encoder and decoder work together

- The encoder start by processing the input sequence
- The output of the top encoder is then transformed into a set of attention vectors **K** and **V**.
- These are to be used by each decoder in its "encoder-decoder attention" layer which helps the decoder focus on appropriate places in the input sequence

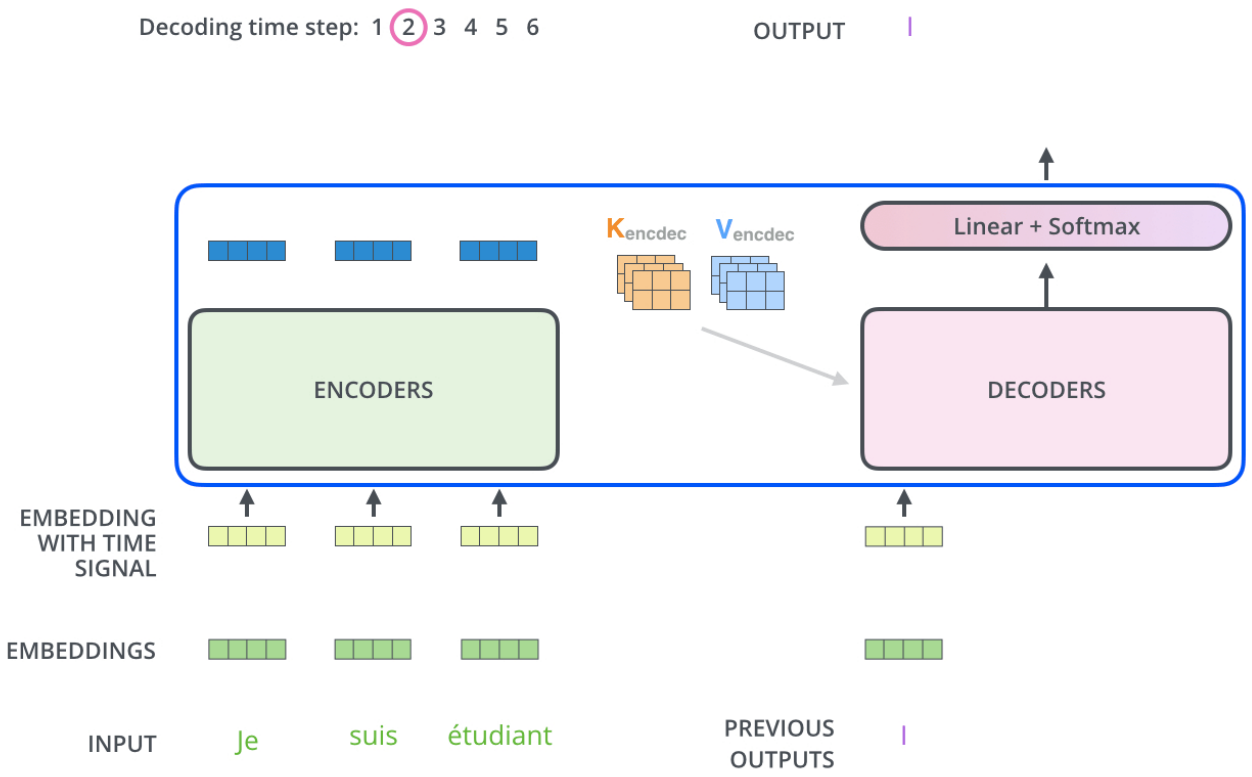


After finishing the encoding phase, we begin the decoding phase. Each step in the decoding phase outputs an element from the output sequence (the English translation sentence in this case).

Transformer Decoder

How encoder and decoder work together

- The output of each step is fed to the bottom decoder in the next time step
- Embed and add positional encoding to those decoder inputs. Process the inputs
- Repeat the process until a special symbol is reached indicating the transformer decoder has completed its output.

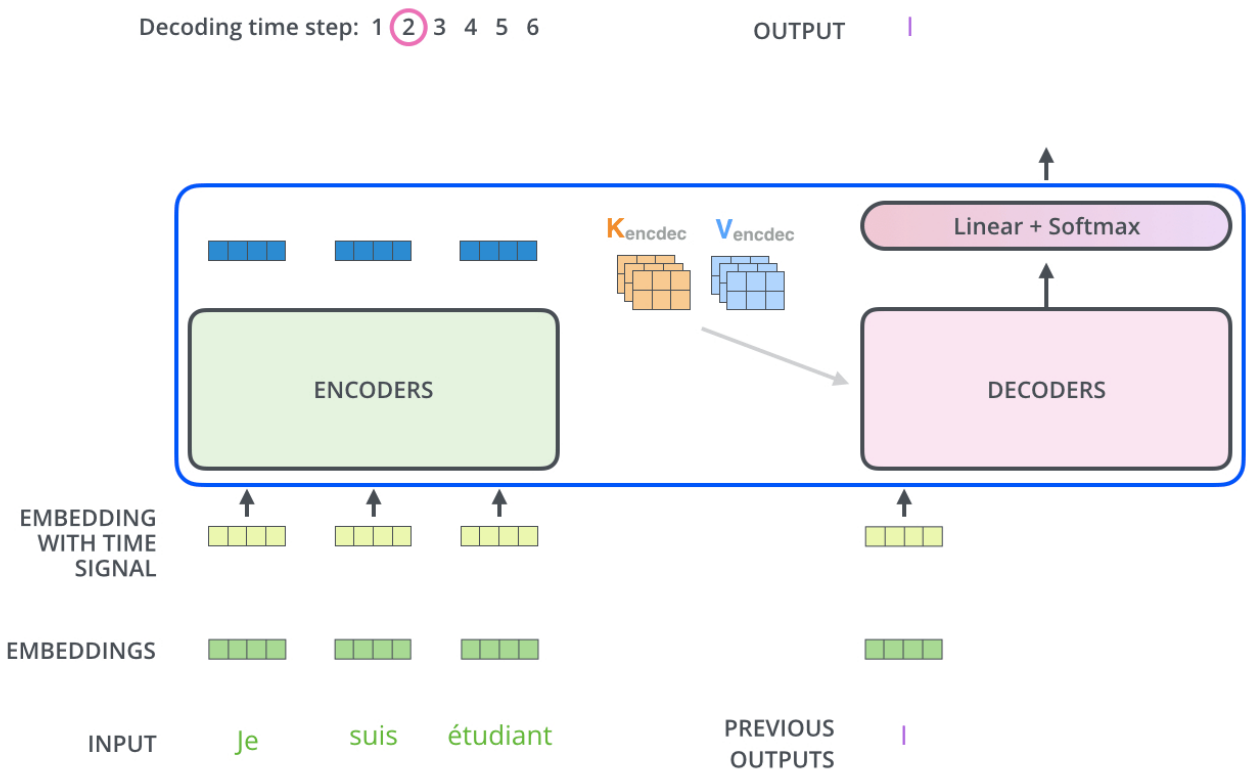


Transformer Decoder

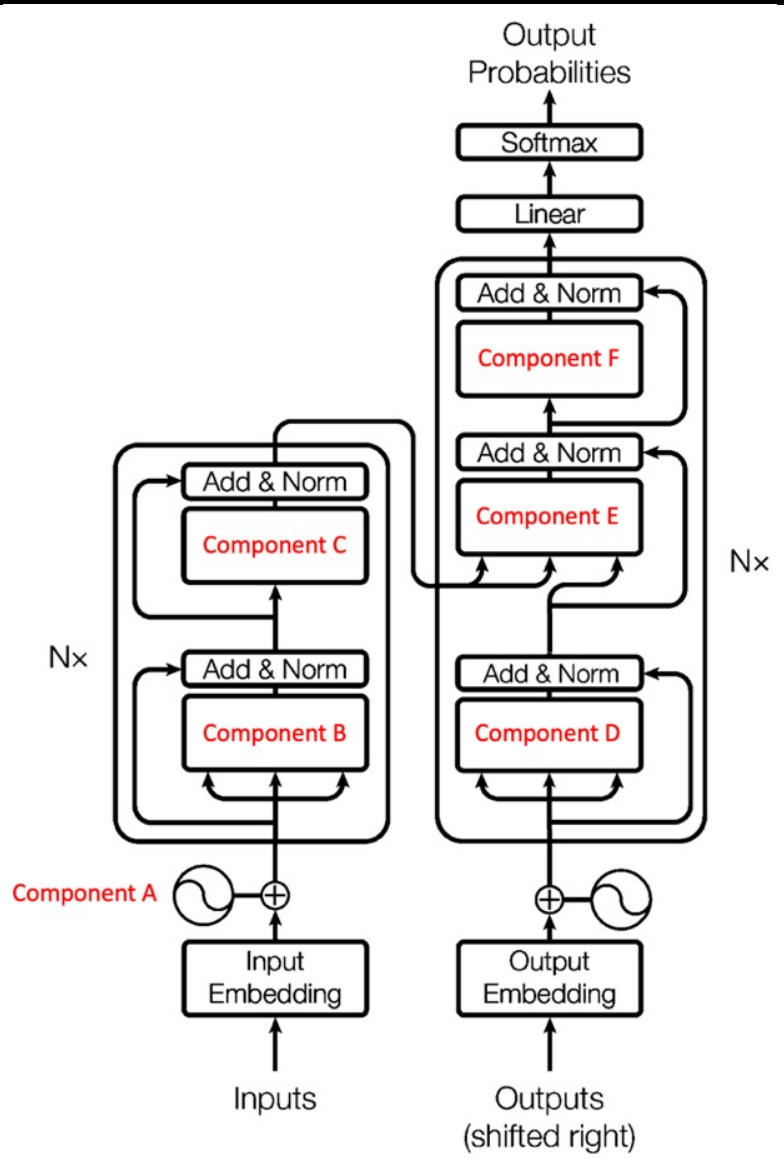
How encoder and decoder work together

- The output of each step is fed to the bottom decoder in the next time step
- Embed and add positional encoding to those decoder inputs. Process the inputs
- Repeat the process until a special symbol is reached indicating the transformer decoder has completed its output.

Note:
In the decoder, *the self-attention layer is only allowed to attend to earlier positions in the output sequence*. This is done by masking future positions (setting them to -inf) before the softmax step in the self-attention calculation.



Transformer

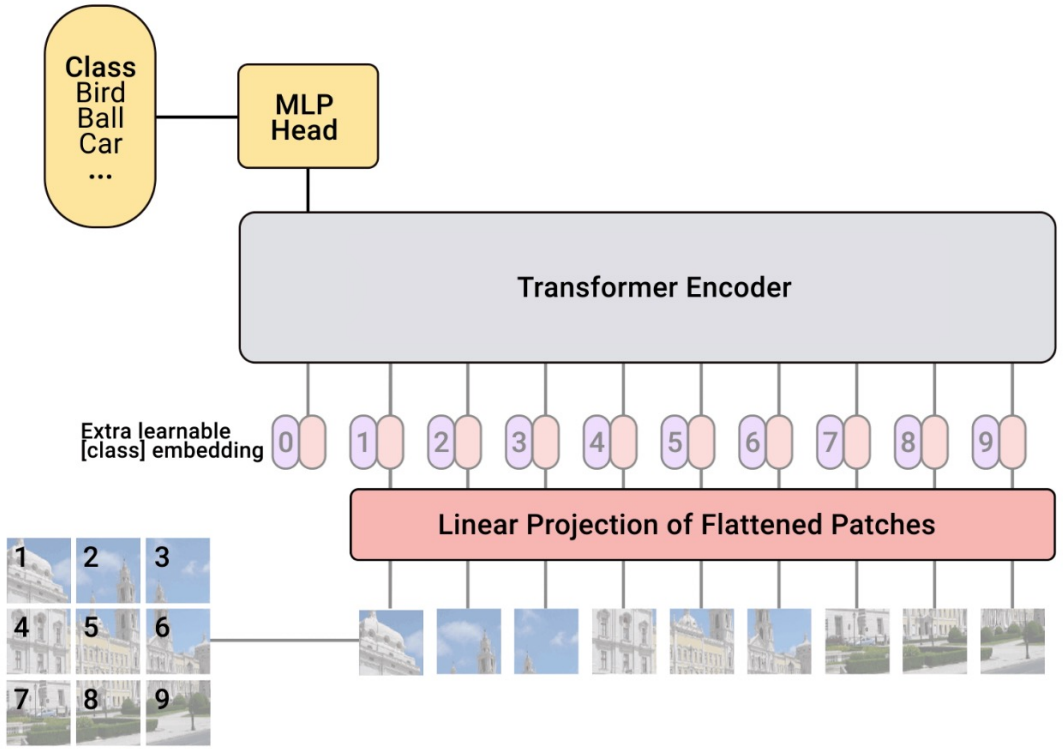


Vision Transformer

Vision Transformer (ViT)

- Do not have decoder

- Reshape the image $\mathbf{x} \in \mathbb{R}^{H \times W \times C}$ into a sequence of flattened 2D patches $\mathbf{x} \in \mathbb{R}^{N \times (P^2 \cdot C)}$
  - (H, W) is the resolution of the original image
  - C is the number of channels
  - (P, P) is the resolution of each image patch
  - $N = HW/P^2$ is the resulting number of patches



Alexey Dosovitskiy et al., An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale, ICLR 2021

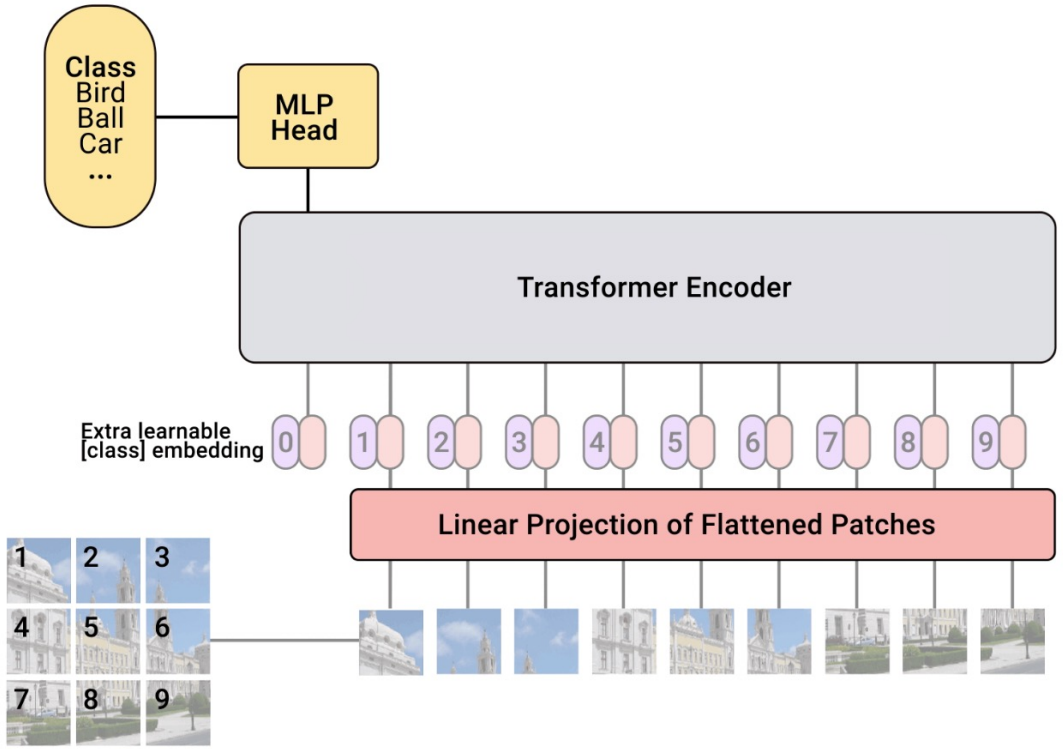
Vision Transformer (ViT)

Prepend a learnable embedding ($\mathbf{z}_0^0 = \mathbf{x}_{\text{class}}$) to the sequence of embedded patches

$$\mathbf{z}_0 = [\mathbf{x}_{\text{class}};\ \mathbf{x}_p^1\mathbf{E};\ \mathbf{x}_p^2\mathbf{E};\ \cdots\ ;\ \mathbf{x}_p^N\mathbf{E}] + \mathbf{E}_{pos}$$

Patch embedding - Linearly embed each of them to D dimension with a trainable linear projection $\mathbf{E} \in \mathbb{R}^{(P^2 \cdot C) \times D}$

Add learnable position embeddings $\mathbf{E}_{pos} \in \mathbb{R}^{(N+1) \times D}$ to retain positional information



Alexey Dosovitskiy et al., An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale, ICLR 2021

Vision Transformer (ViT)

Feed the resulting sequence of vectors $\mathbf{z}_0$ to a standard Transformer encoder

$$\mathbf{z}_0 = [\mathbf{x}_{\text{class}};\ \mathbf{x}_p^1\mathbf{E};\ \mathbf{x}_p^2\mathbf{E};\ \cdots\ ;\ \mathbf{x}_p^N\mathbf{E}] + \mathbf{E}_{pos}, \qquad \mathbf{E} \in \mathbb{R}^{(P^2 \cdot C) \times D},\ \mathbf{E}_{pos} \in \mathbb{R}^{(N+1) \times D}$$

Transformer Encoder

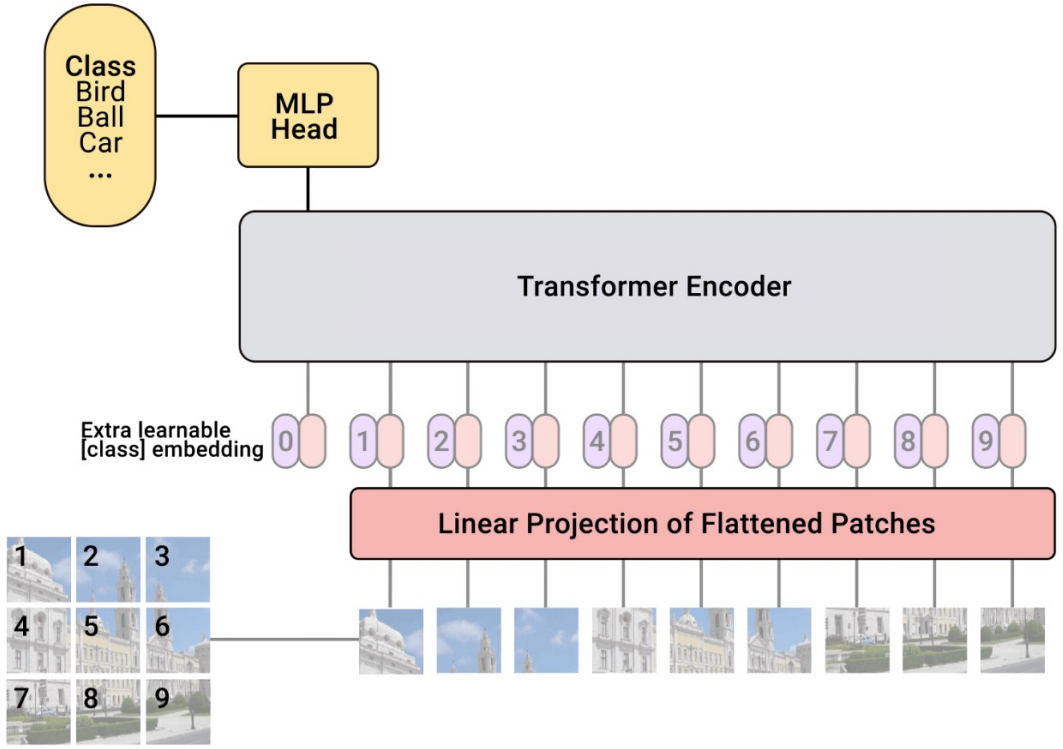
$$\mathbf{z}'_\ell = \text{MSA}(\text{LN}(\mathbf{z}_{\ell-1})) + \mathbf{z}_{\ell-1}, \qquad \ell = 1 \ldots L$$

$$\mathbf{z}_\ell = \text{MLP}(\text{LN}(\mathbf{z}'_\ell)) + \mathbf{z}'_\ell, \qquad \ell = 1 \ldots L$$

Classification Head

$$\mathbf{y} = \text{LN}(\mathbf{z}_L^0)$$

The output of the additional [class] token is transformed into a class prediction via a small multi-layer perceptron (MLP) with tanh as non-linearity in the single hidden layer.

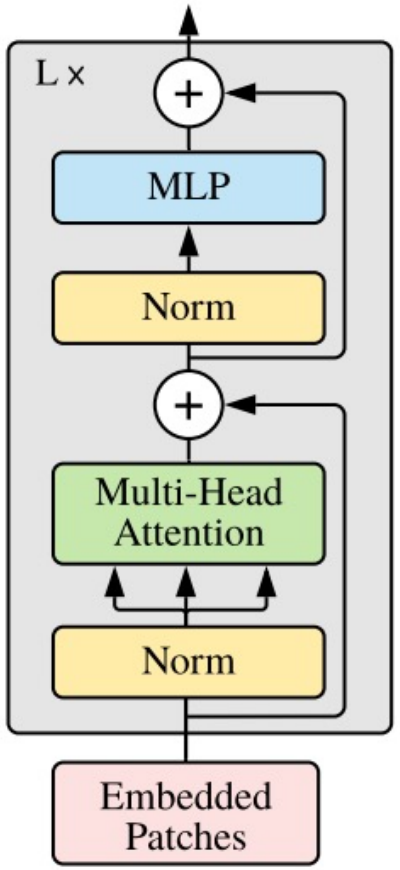


Vision Transformer (ViT)

Transformer Encoder

- Consists of a multi-head self-attention module (MSA), followed by a 2-layer MLP (with GELU)
- LayerNorm (LN) is applied before MSA module and MLP, and a residual connection is applied after each module.

$$\mathbf{z}'_\ell = \mathrm{MSA}(\mathrm{LN}(\mathbf{z}_{\ell-1})) + \mathbf{z}_{\ell-1},$$
$$\mathbf{z}_\ell = \mathrm{MLP}(\mathrm{LN}(\mathbf{z}'_\ell)) + \mathbf{z}'_\ell,$$



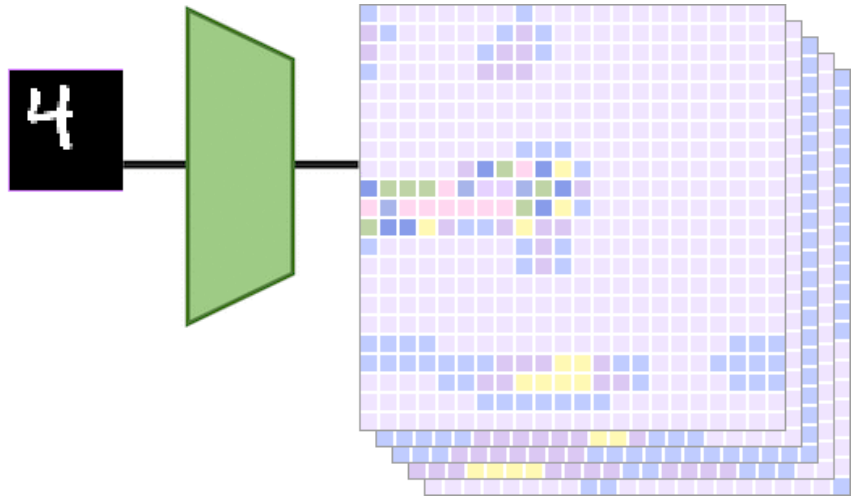
Transformer Encoder

Inductive Bias

- ViT has much less image-specific inductive bias than CNNs

- Inductive bias in CNN
  - Locality
  - Two-dimensional neighborhood structure
  - Translation equivariance

- ViT
  - Only MLP layers are local and translationally equivariant. Self-attention layer is global
  - Two dimensional neighborhood is used sparingly – i) only at the beginning where we cut image into patches, ii) learnable position embedding (spatial relations have to be learned from scratch)



An equivariant mapping is a mapping which preserves the algebraic structure of a transformation.

A translation equivariant mapping is a mapping which, when the input is translated, leads to a translated mapping

Vision Transformer (ViT)

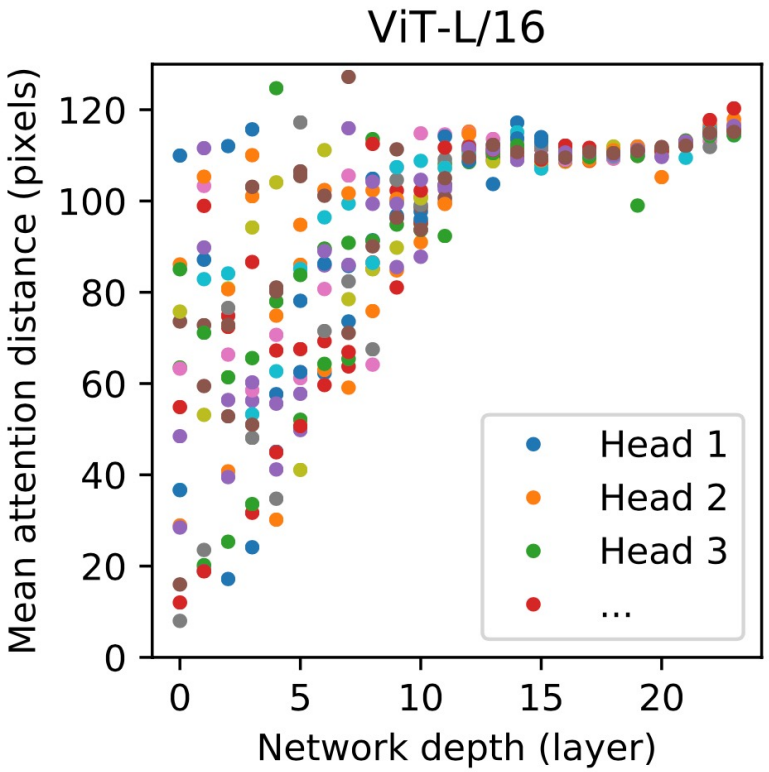
Examine the attention distance, analogous to receptive field in CNN

Compute the average distance in image space across which information is integrated, based on the attention weights.

Attention distance was computed for 128 example images by averaging the distance between the query pixel and all other pixels, weighted by the attention weight.

Each dot shows the mean attention distance across images for one of 16 heads at one layer. Image width is 224 pixels.

An example: if a pixel is 20 pixels away and the attention weight is 0.5 the distance is 10.



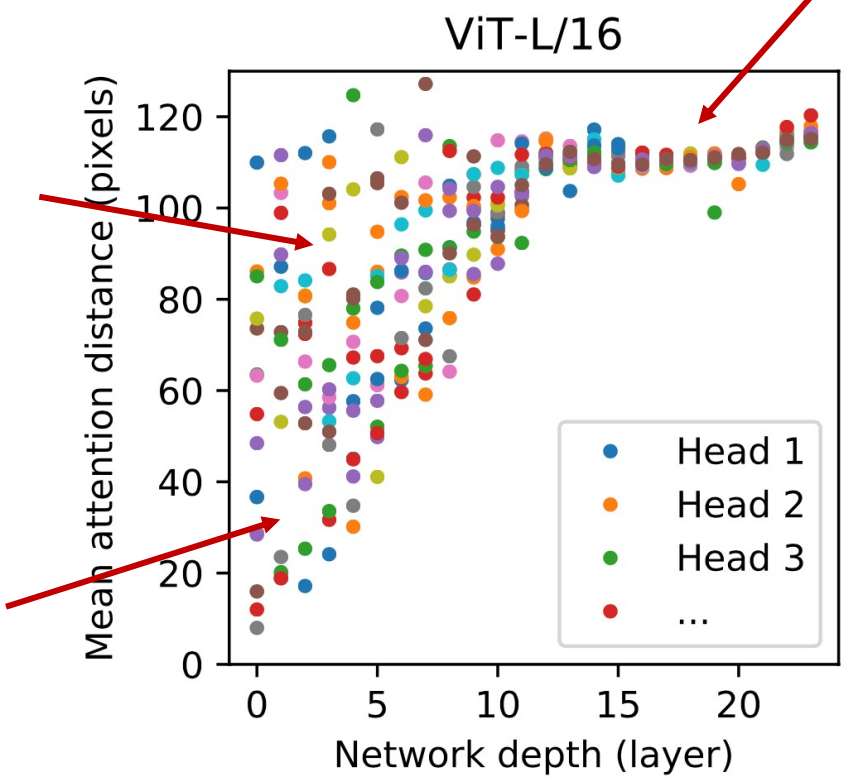
Vision Transformer (ViT)

Examine the attention distance, analogous to receptive field in CNN

The attention distance increases with network depth

Some heads attend to most of the image already in the lowest layers, showing the capability of ViT in integrating information globally

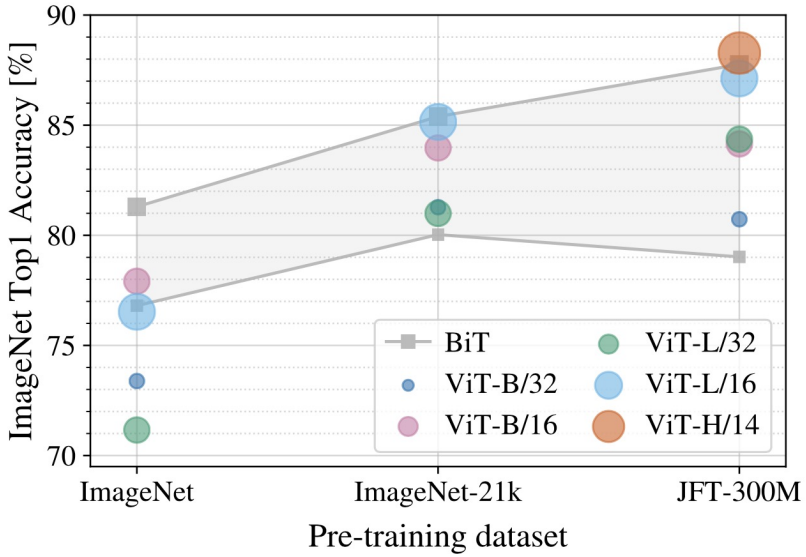
Other attention heads have consistently small attention distances in the low layers



Vision Transformer (ViT)

Performance of ViT

- ViT performs significantly worse than the CNN equivalent (BiT) when trained on ImageNet (1M images).
- However, on ImageNet-21k (14M images) performance is comparable, and on JFT (300M images), ViT outperforms BiT.
- ViT overfits the ImageNet task due to its lack of inbuilt knowledge about images



Model	Layers	Hidden size D	MLP size	Heads	Params
ViT-Base	12	768	3072	12	86M
ViT-Large	24	1024	4096	16	307M
ViT-Huge	32	1280	5120	16	632M

Vision Transformer (ViT)

ViT conducts global self-attention

- Relationships between a token and all other tokens are computed
- Quadratic complexity with respect to the number of tokens
- Not tractable for dense prediction or to represent a high-resolution image