Transformer

Why Transformers?

- In sequence-to-sequence problems such as neural machine translation - initial proposals were based on use of RNNs in an encoder-decoder architecture
- These architectures have a great limitation when working with long sequences
  - Ability to retain information from first elements lost when new elements incorporated into the sequence
  - In encoder - hidden state in every step associated with a certain word in the input sentence, usually one of the most recent
  - If decoder only accesses last hidden state of decoder, it will lose relevant information about first elements of sequence

Why Transformers?

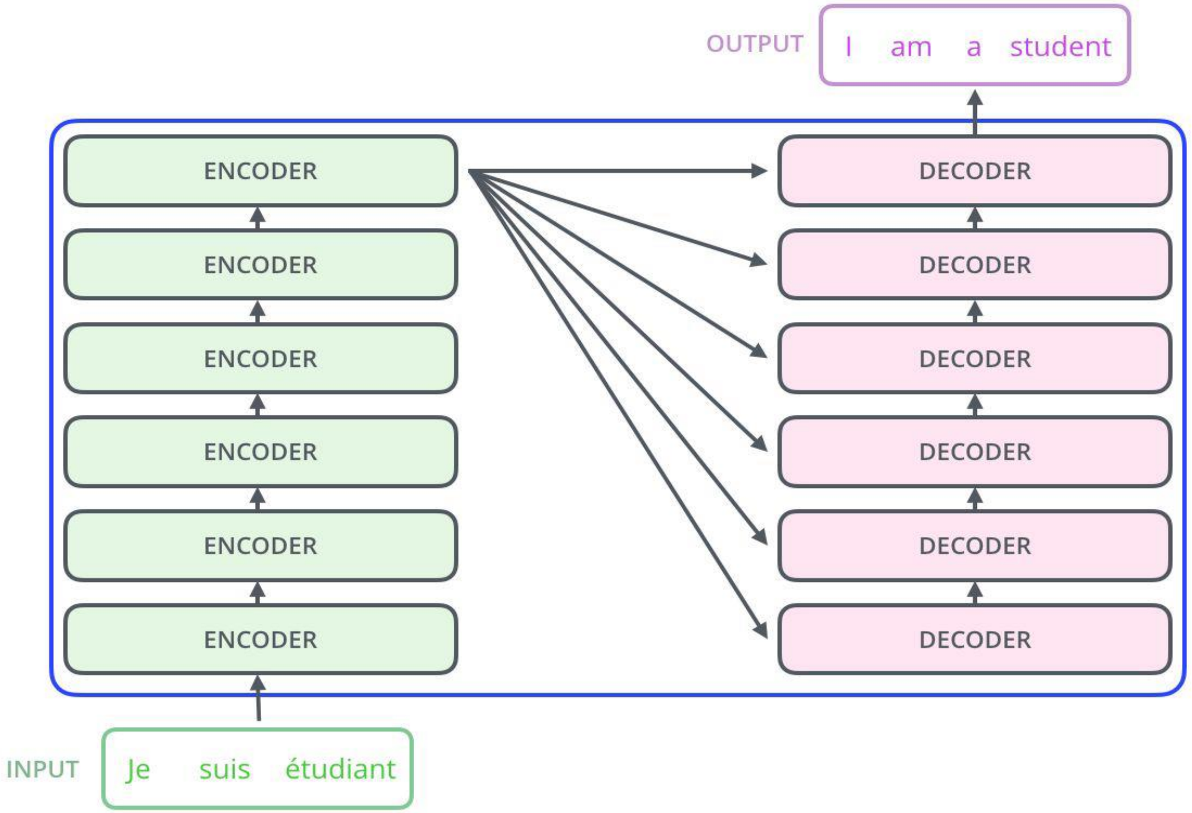
- Instead of paying attention to last state of encoder:
  - In each step of decoder - look at all states of encoder
  - Access information about all elements of input sequence
- This is what **attention** does
  - Extracts information from whole sequence - a **weighted sum of all past encoder states**
  - Allows decoder to assign greater weight or importance to a certain element of input for each element of output
  - Learning to focus in right element of input to predict next output element

Transformers and Attention

- Transformer is a neural network layer that relies on **attention**
  - Attention is a method of gathering relevant contextual information
  - Allows neural network to weigh importance of different elements within a sequence while generating representations for each element
  - Enables models to attend to entire sequence simultaneously
  - Ability to consider global context and capture dependencies between any pair of elements within the sequence
- State-of-the-art models across various domains consist almost entirely of transformer layers

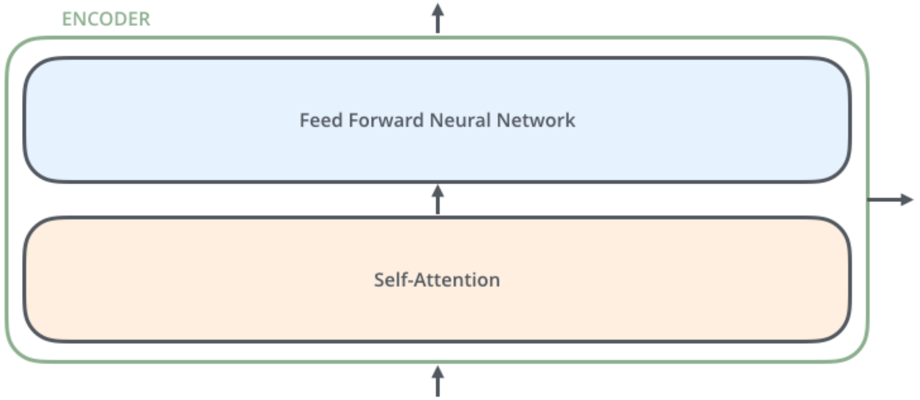
High Level Overview

- Use machine translation as example, this is what Transformers were initially developed for

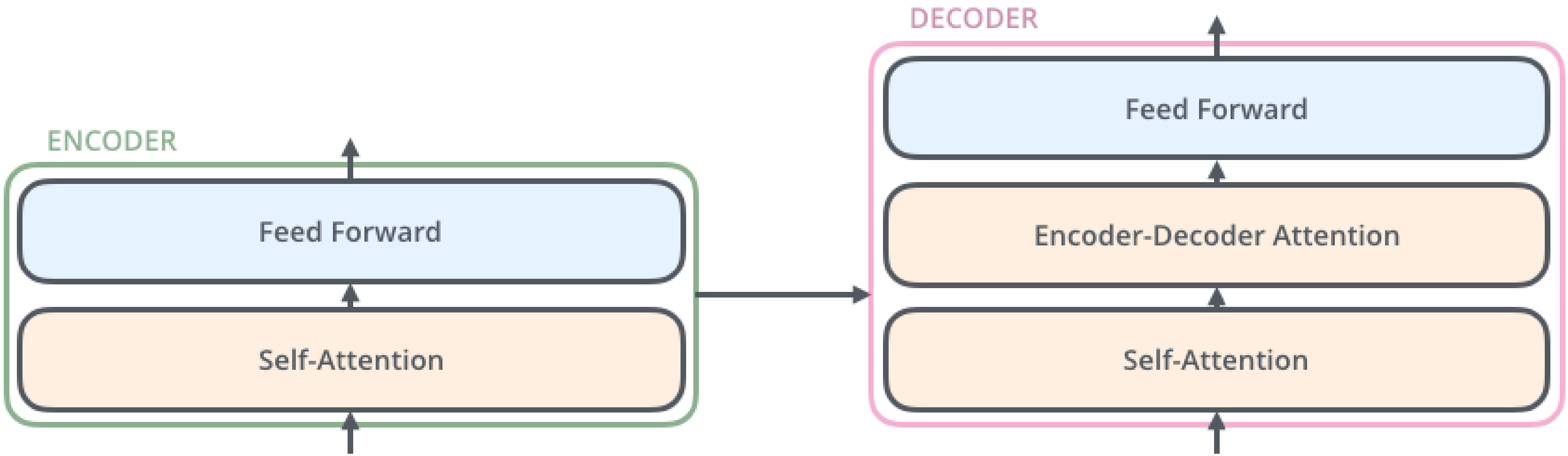


High Level Overview

- Encoding component is a stack of encoders
  - All identical in structure (they do not share weights)
  - Inputs flow through a self-attention layer
  - Helps encoder look at other words in input sentence as it encodes a specific word
  - Outputs of self-attention layer fed to a feed-forward neural network
  - Exact same feed-forward network is independently applied to each position
- Decoding component is a stack of decoders of same number
  - Decoder has both those layers
  - Between them is an attention layer that helps decoder focus on relevant parts of input sentence



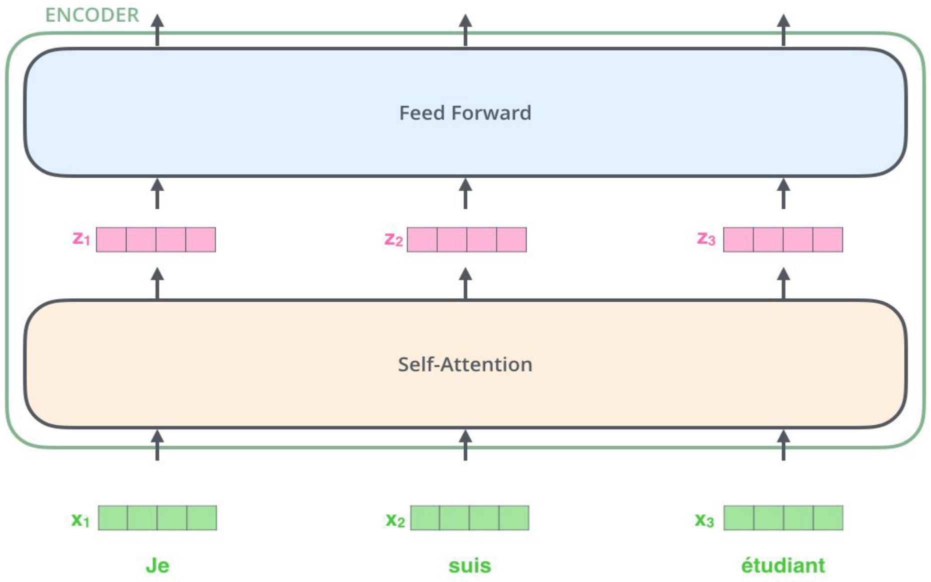
https://jalammar.github.io/illustrated-transformer/


ENCODER
Feed Forward
Self-Attention
DECODER
Feed Forward
Encoder-Decoder Attention
Self-Attention

High Level Overview

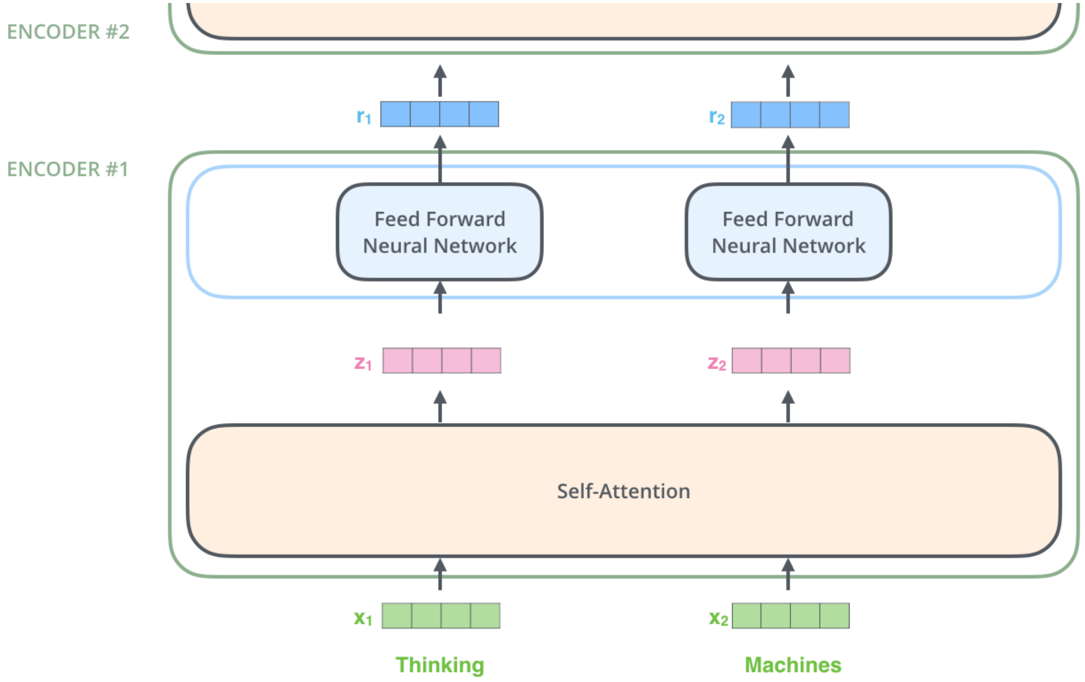
- Embed input into tokens (fixed dimensional vector)

- Process with encoder layer



High Level Overview

- Encoder receives a list of vectors as input
  - Processes by passing vectors into a ‘self-attention’ layer
  - Then into a feed-forward neural network
  - Then sends out output upwards to next encoder



Self-Attention

- Ex., in the sentence:

  She poured water from the pitcher to the cup until it was full.

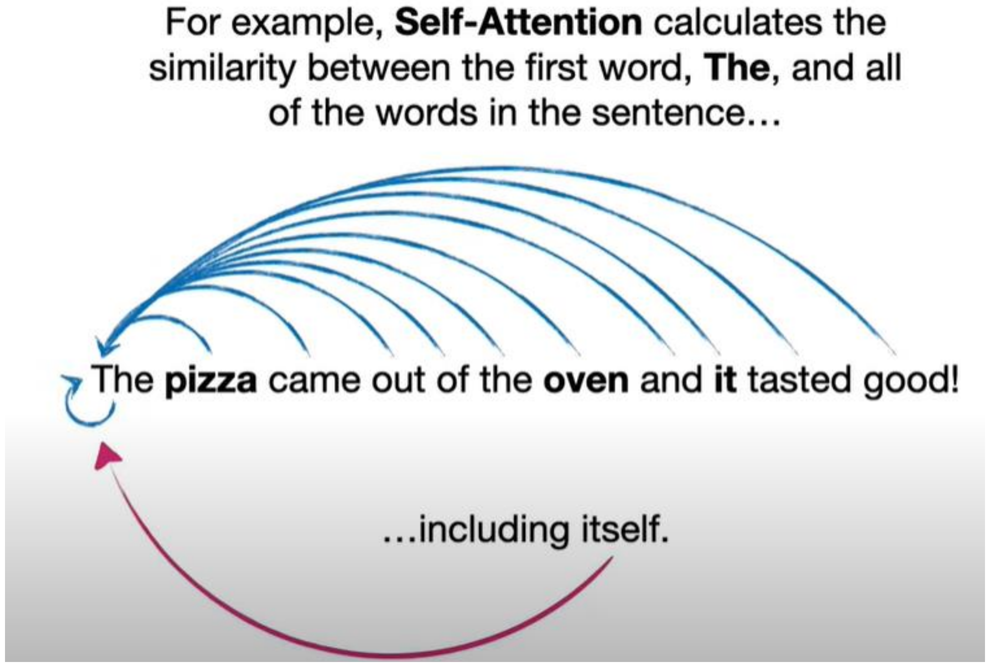
- "***it***" refers to the cup
- While in the sentence:

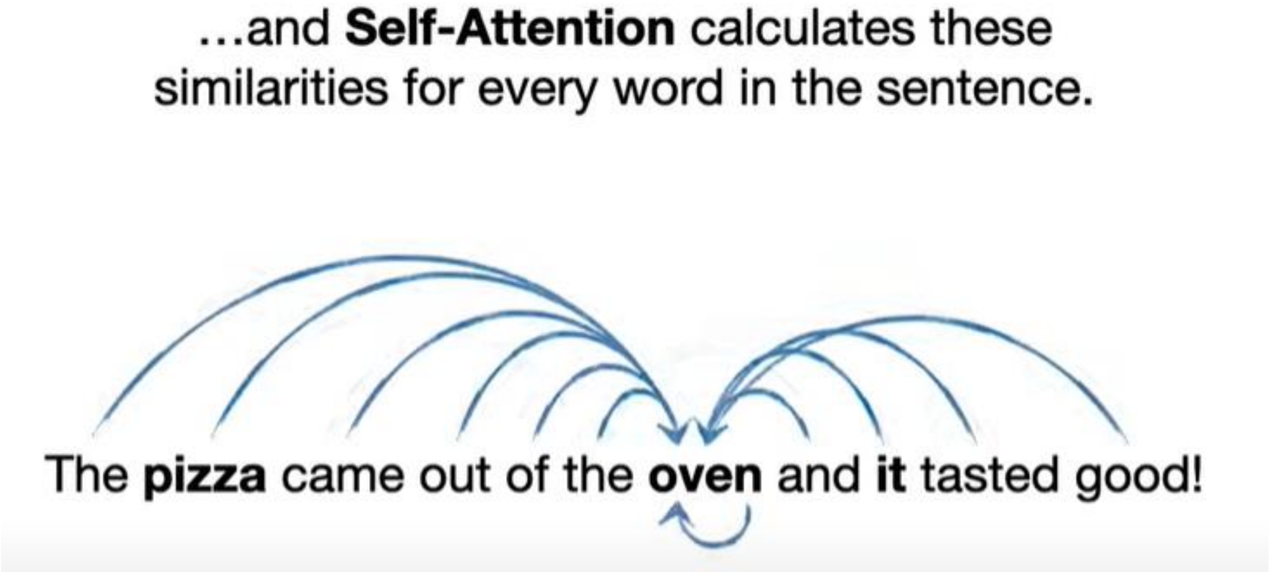
  She poured water from the pitcher to the cup until it was empty.

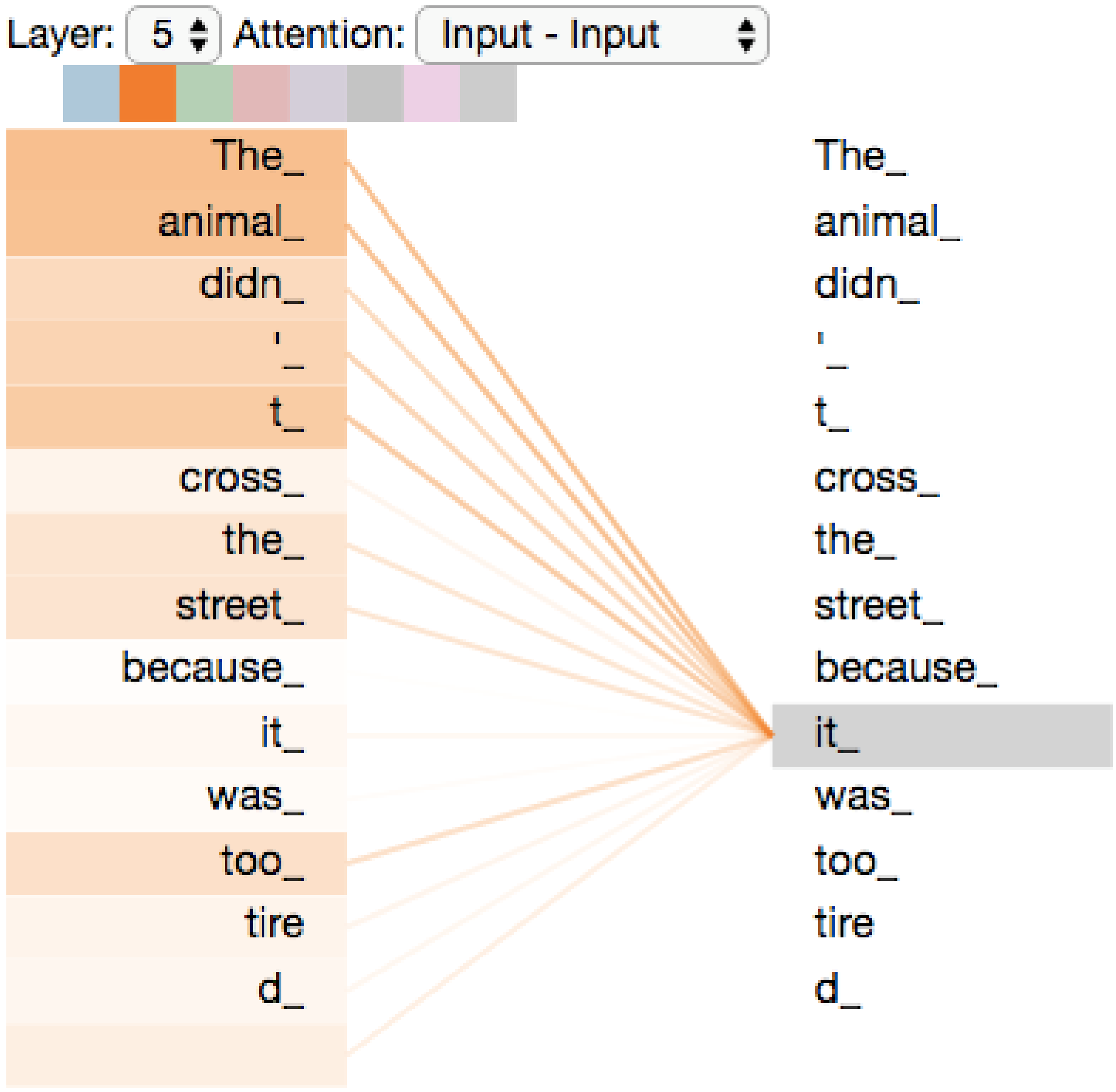
- "***it***" refers to the pitcher
- *"Meaning is a result of relationships between things, and self-attention is a general way of learning relationships,"* - Ashish Vaswani

Self-Attention

- Works by seeing how similar each word is to all of the words in the sentence, including itself
  - Allows model to calculate attention weights, which determine importance of each element in a sequence to all other elements







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

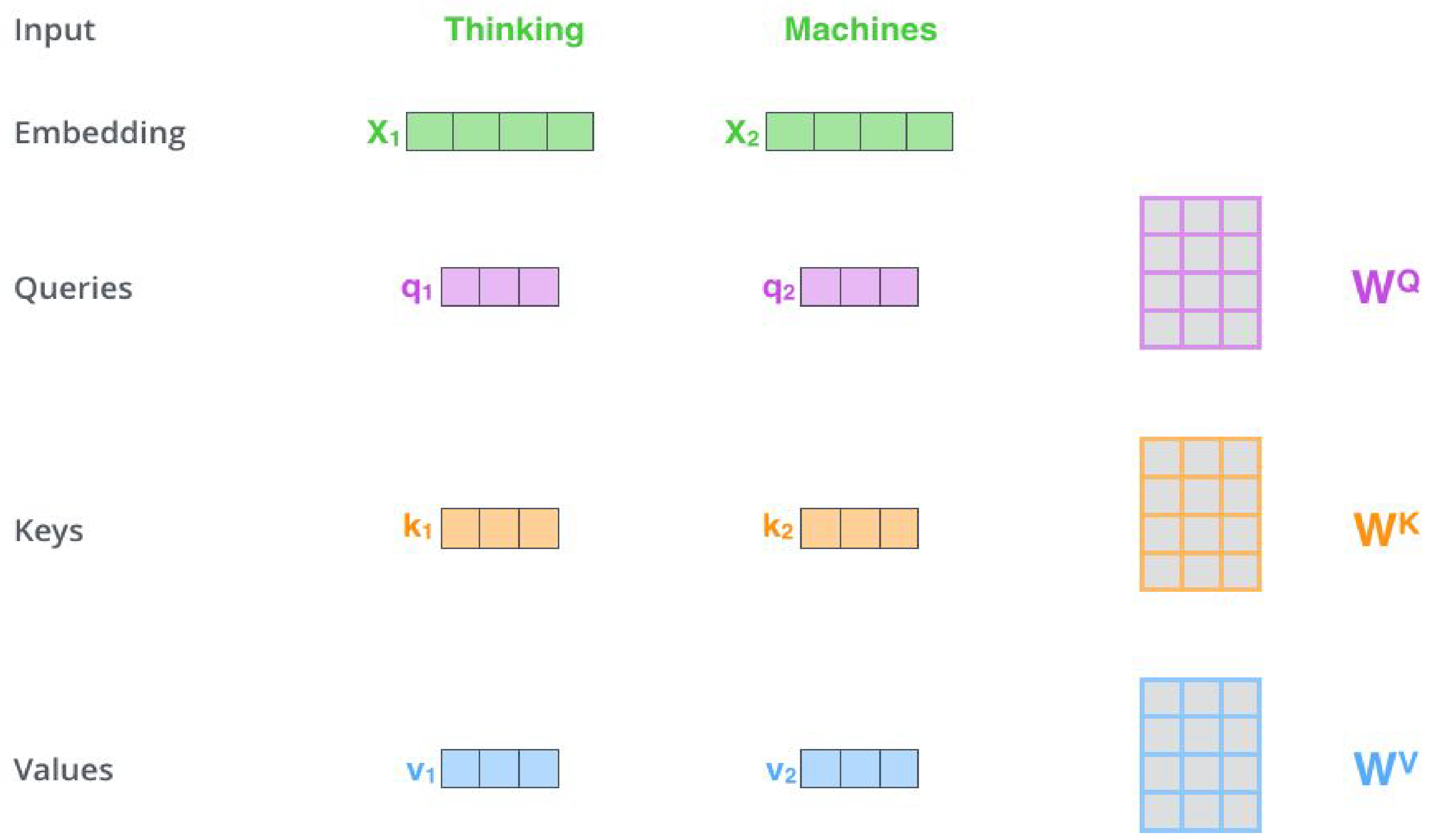
Self-Attention

- After similarities are computed, they are used to determine how Transformer encodes each word
  - Ex., if many sentences about **pizza** show that the word **it** is more commonly associated with **pizza** then **oven**, then similarity score for **pizza** will have a larger impact on how **it** is encoded

Self-Attention

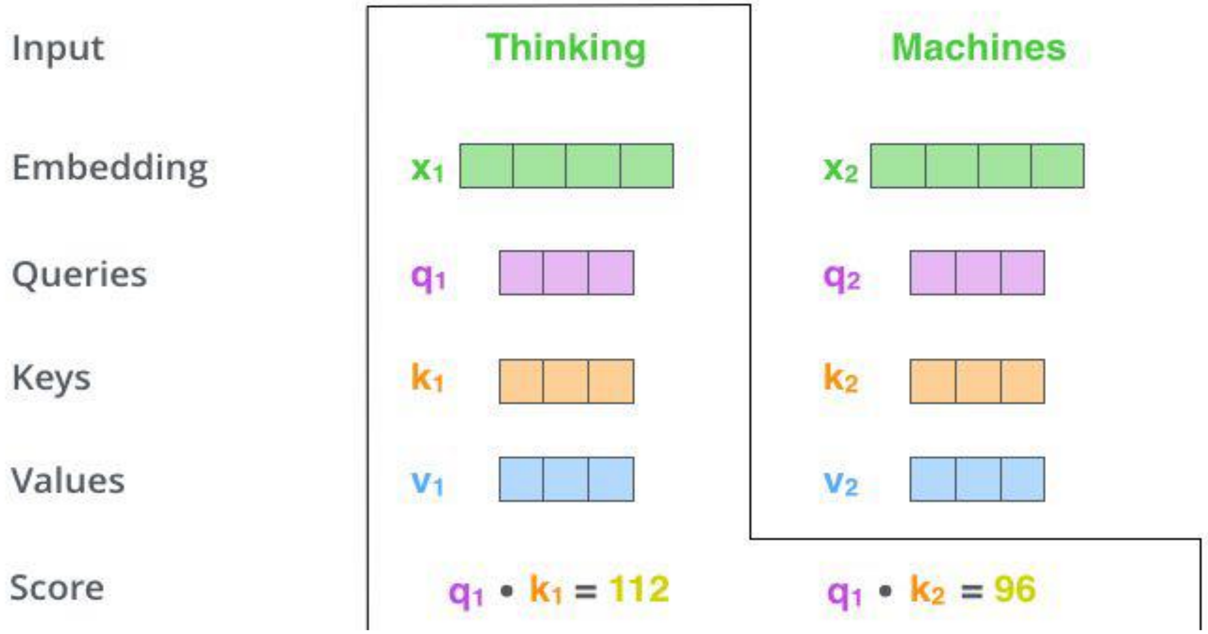
- Given an input sequence, **X**
- Project to Queries, Keys and Values using linear transforms.
- $\mathbf{Q} = \mathbf{W}^{Q}\mathbf{X}$, $\mathbf{K} = \mathbf{W}^{K}\mathbf{X}$, $\mathbf{V} = \mathbf{W}^{V}\mathbf{X}$
  - **Query** - captures element for which attention weights will be calculated
  - **Key, Value** - provide contextual information
- By employing separate linear transformations, attention mechanism allows model to learn different projections for queries, keys, and values
  - Enables model to capture distinct aspects of input elements and facilitate meaningful comparisons during calculation of attention weights

Self-Attention



Self-Attention

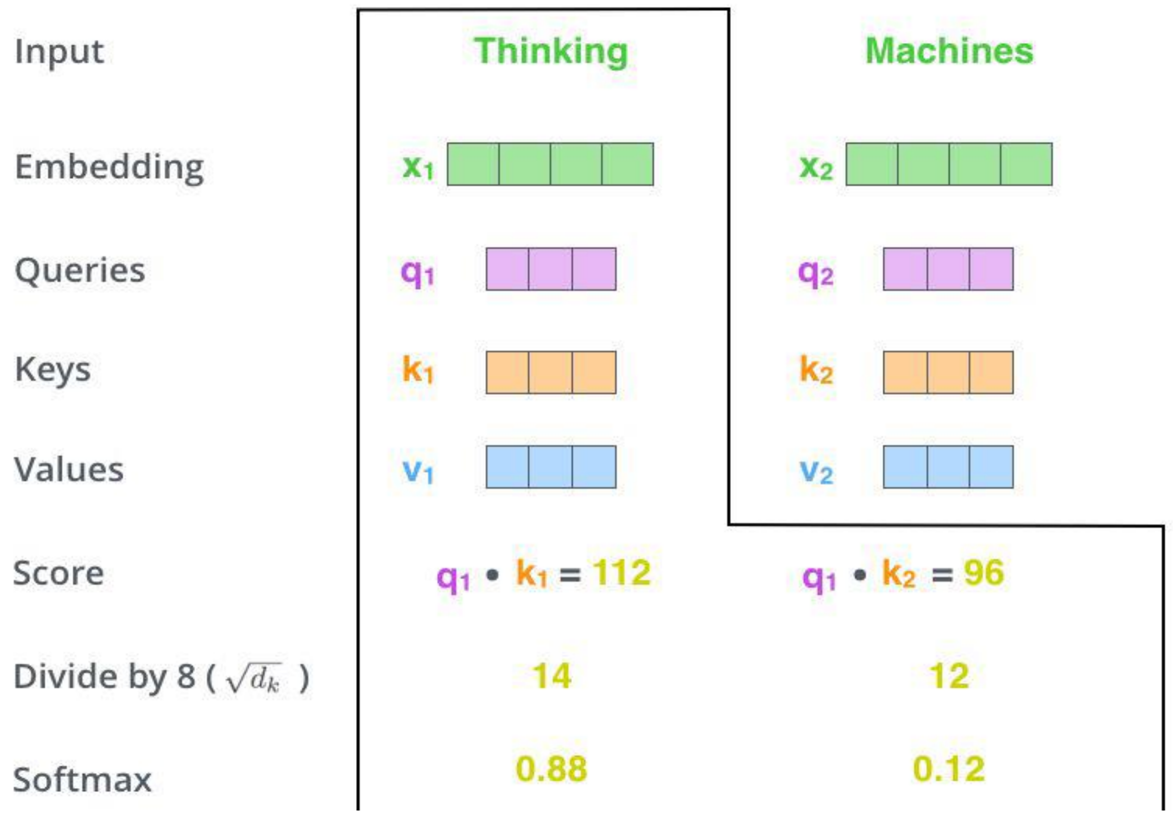
- Calculate a score (**QK**^{T})
  - For each query, how relevant are all the other words?



Self-Attention

- Normalized using a softmax function to ensure that weights sum up to one - Softmax($\mathbf{QK}^T$)
  - Normalized attention weights represent importance or relevance of each element in sequence with respect to query element

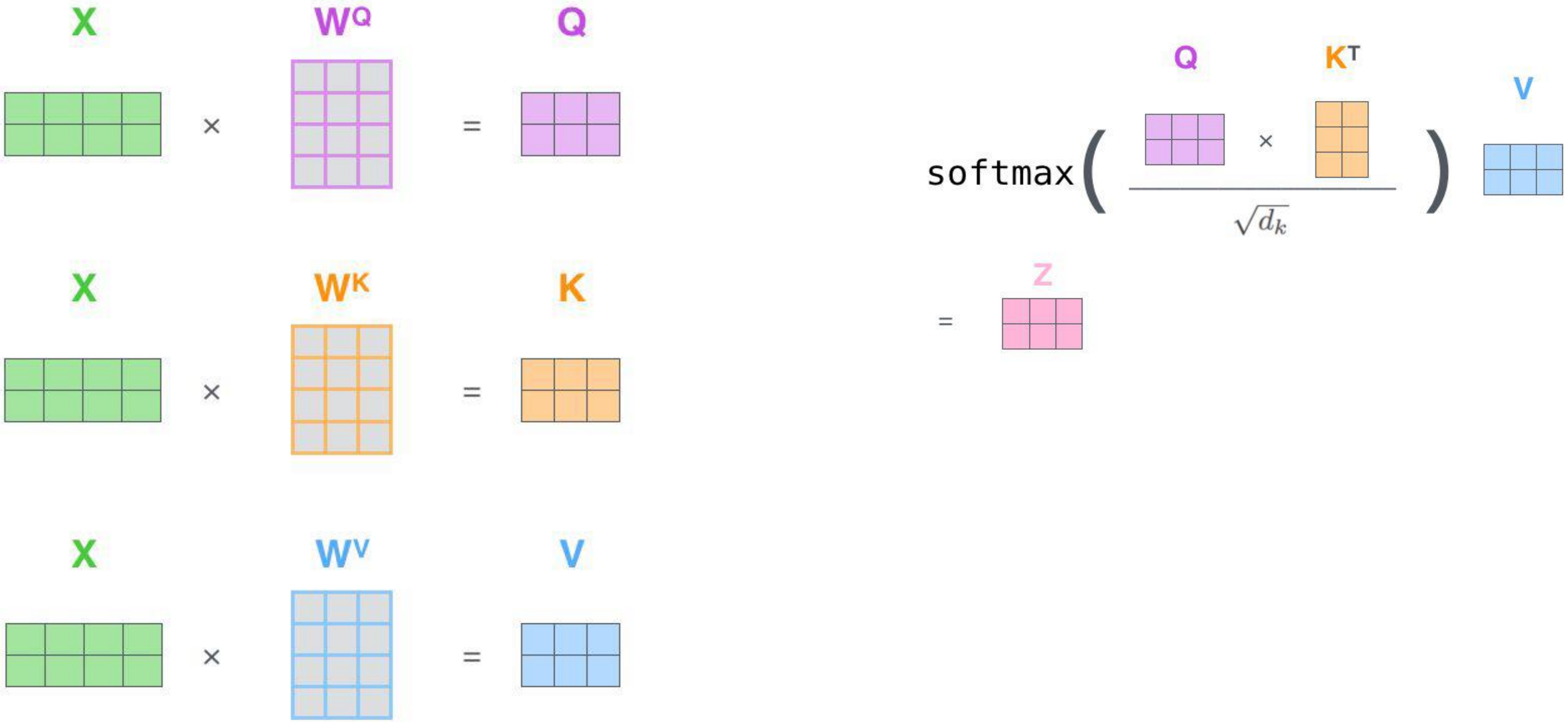
Self-Attention



Self-Attention

- Multiply attention weights with corresponding value representations - aggregated to produce final output representation
  - Combines weighted values
  - Incorporates information from all elements in sequence according to their relevance
- $\mathbf{Z} = \text{Softmax}(\mathbf{Q}\mathbf{K}^T)\mathbf{V}$

Self Attention: As a Matrix



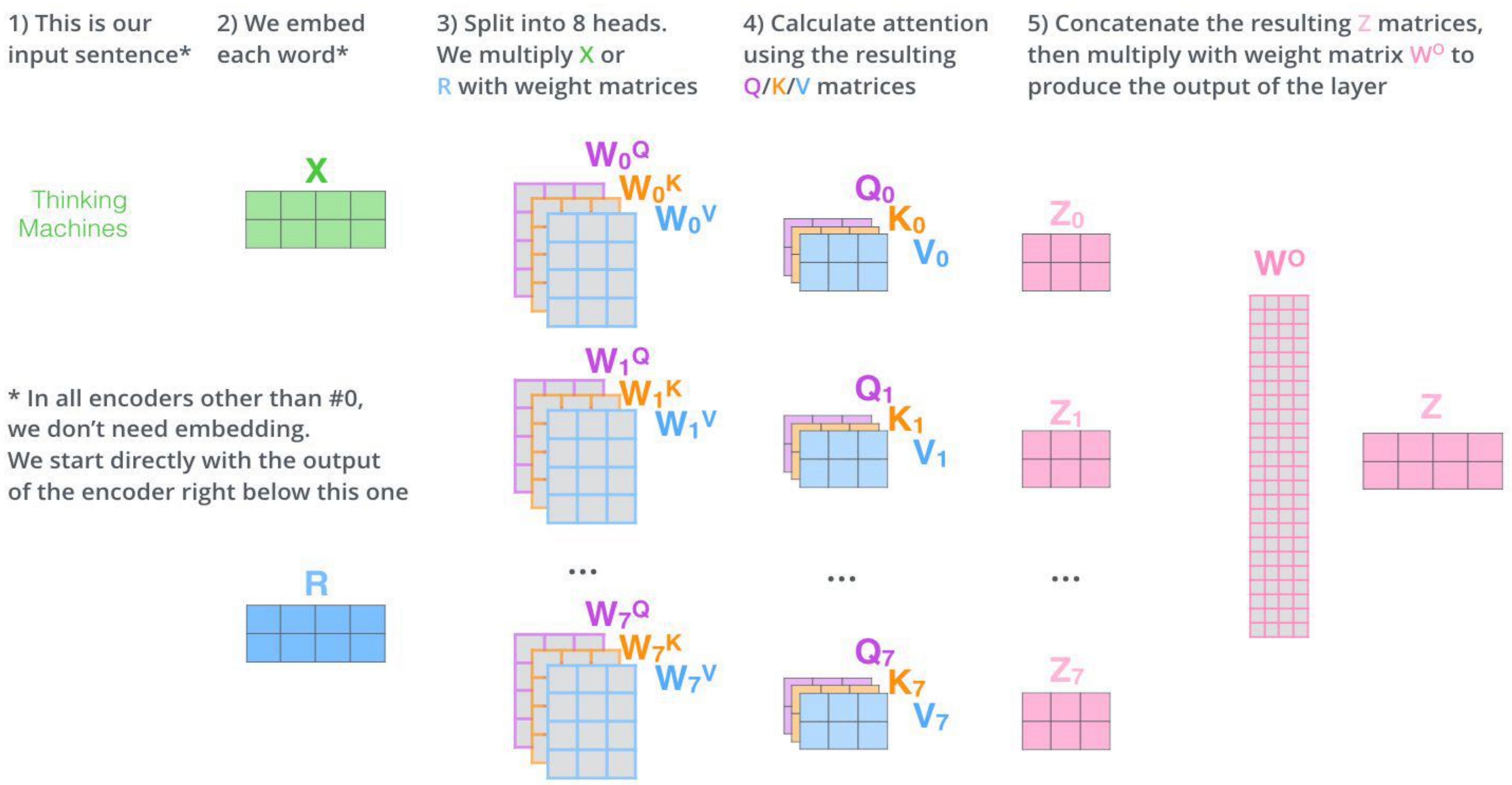
Multiple Attention Heads (MHA)

- Involves learning multiple sets of query, key, and value transformations - known as attention heads
  - Each attention head operates independently
  - Attends to different aspects of input sequence
  - Provide model with multiple perspectives or interpretations of input
  - Captures various patterns and relationships
  - Model can leverage different representations for different parts of sequence
  - Enables model to extract more nuanced and comprehensive features enhancing its ability to understand and process input sequence

Combining Outputs from Attention Heads

- Outputs from each attention head in multi-head attention combined to create a unified representation
  - Combination can be achieved through concatenation
- Combined output from multi-head attention retains representations learned by each attention head
  - Provides model with a richer understanding of input sequence
- Aggregated representation serves as input for subsequent layers or tasks in Transformer architecture

Self-Attention: Multiple Heads



Benefits of Multi-Head Attention (MHA)

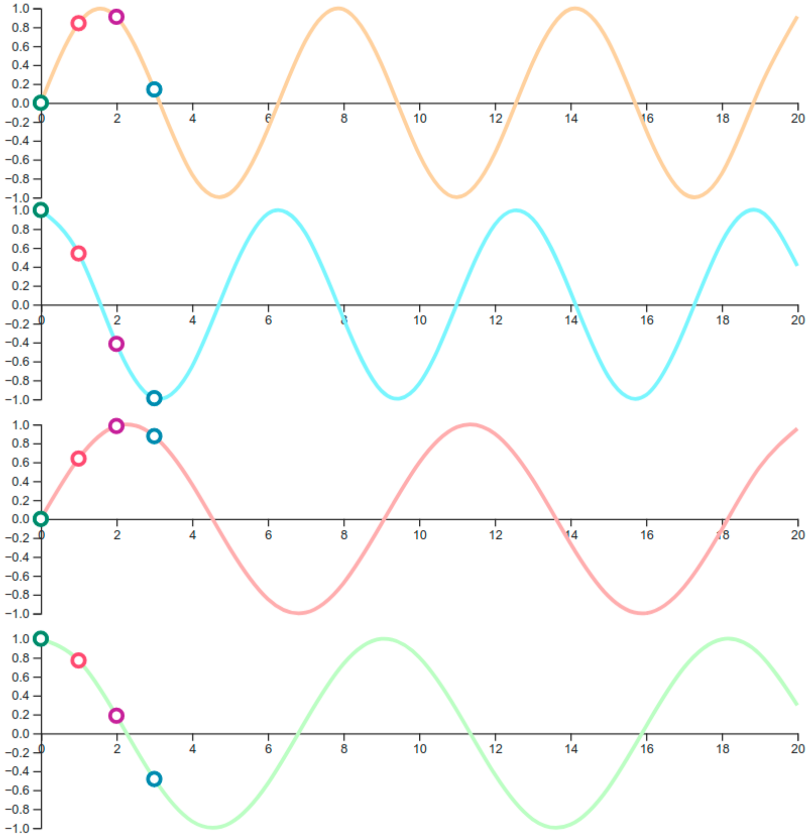
- Capturing Different Relationships: Attending to different subspaces of input sequence
  - Allows model to capture a variety of relationships and dependencies
  - Enabling model to learn a diverse range of features
- Increased Modeling Capacity: Model has more parameters and capacity to learn complex relationships and representations
  - Helps improve expressiveness of model and its ability to handle intricate dependencies in input sequence
- Parallelization: Highly amenable to parallel computation
  - Each attention head operates independently - computations for different attention heads can be performed simultaneously
  - Leading to more efficient training and inference

Positional Embeddings

- What if the ordering of input vectors conveys information as well?
- Position of a word in a sentence matters!
  - “The man ate a fish” != “The fish ate a man”
- Self-attention is permutation invariant!
  - Learned positional embedding
  - At input - add a learned vector to each token
  - Representation of token changes depending on its input position

Positional Embeddings

- Sinusoidal positional embedding



p0	p1	p2	p3	
0.000	0.841	0.909	0.141	i=0
1.000	0.540	-0.416	-0.990	i=1
0.000	0.638	0.983	0.875	i=2
1.000	0.770	0.186	-0.484	i=3

Positional Encoding

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

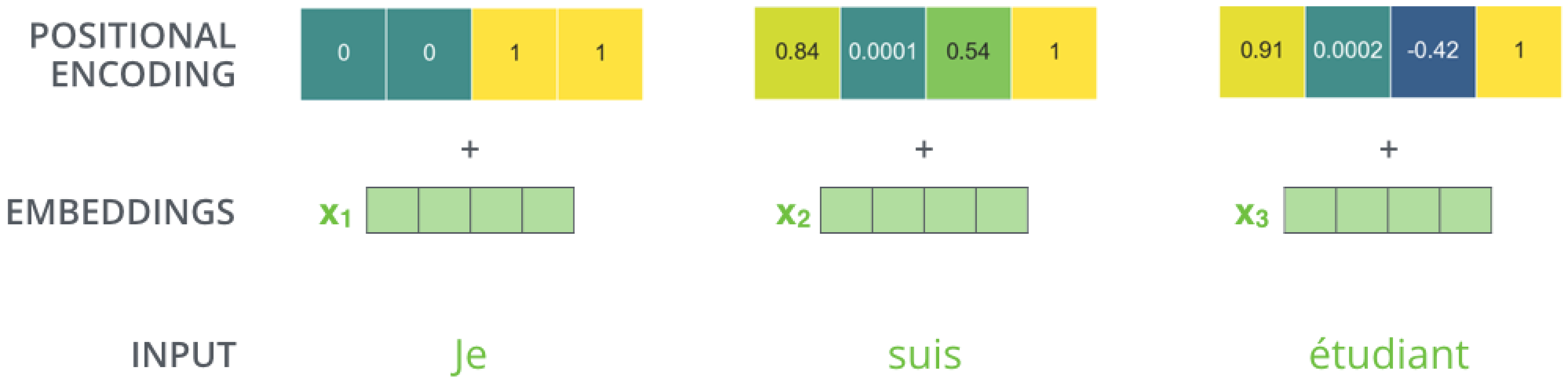
$$PE_{(pos,2i+1)} = cos(\frac{pos}{10000^{2i/d_{\text{model}}}})$$

Settings: d = 50
The value of each positional encoding depends on the *position* (*pos*) and *dimension* (*d*). We calculate result for every *index* (*i*) to get the whole vector.

pos0 pos1 pos2 pos3

Positional Embeddings

- Each positional encoding contains values between 1 and -1



Transformer of [Vaswani et al. Attention is all You Need]

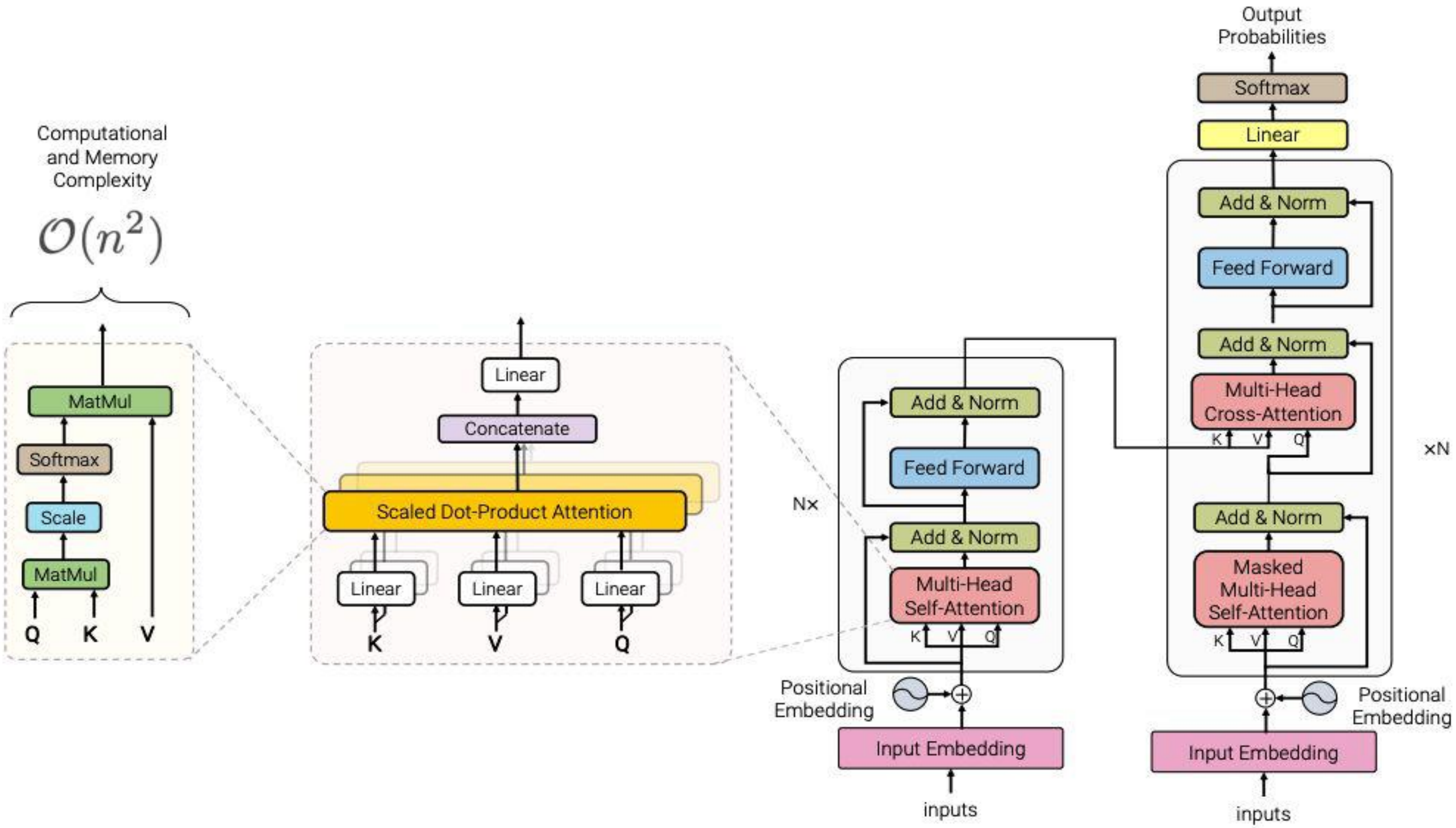


Figure 1: Architecture of the standard Transformer (Vaswani et al., 2017)

Pros/Cons

- Pros:
  - A generic architecture:
  - Operates on any inputs that can be tokenized!
  - Parallelizable
  - Empirically shown to perform excellently at scale

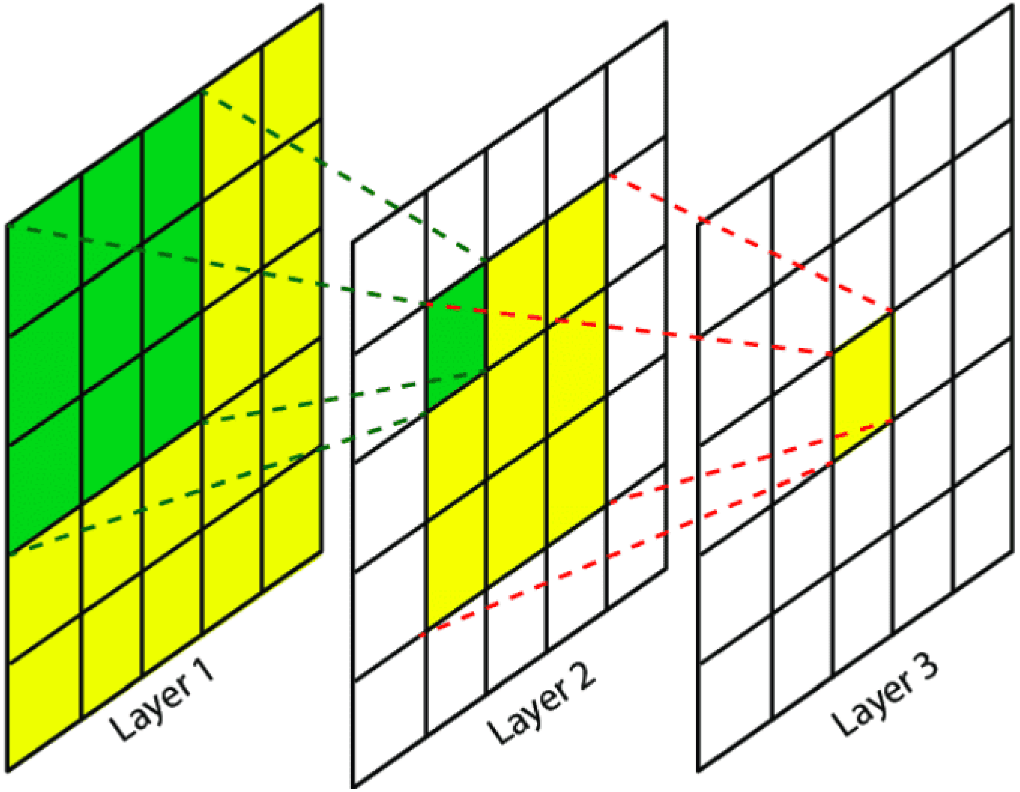
Cons:

- Quadratic complexity
  - Each token attends to every other token
  - N tokens → N^2 operations
  - Prohibitive as the number of tokens increases!
- Most powerful language models are extremely expensive
- Can overfit easily on smaller datasets

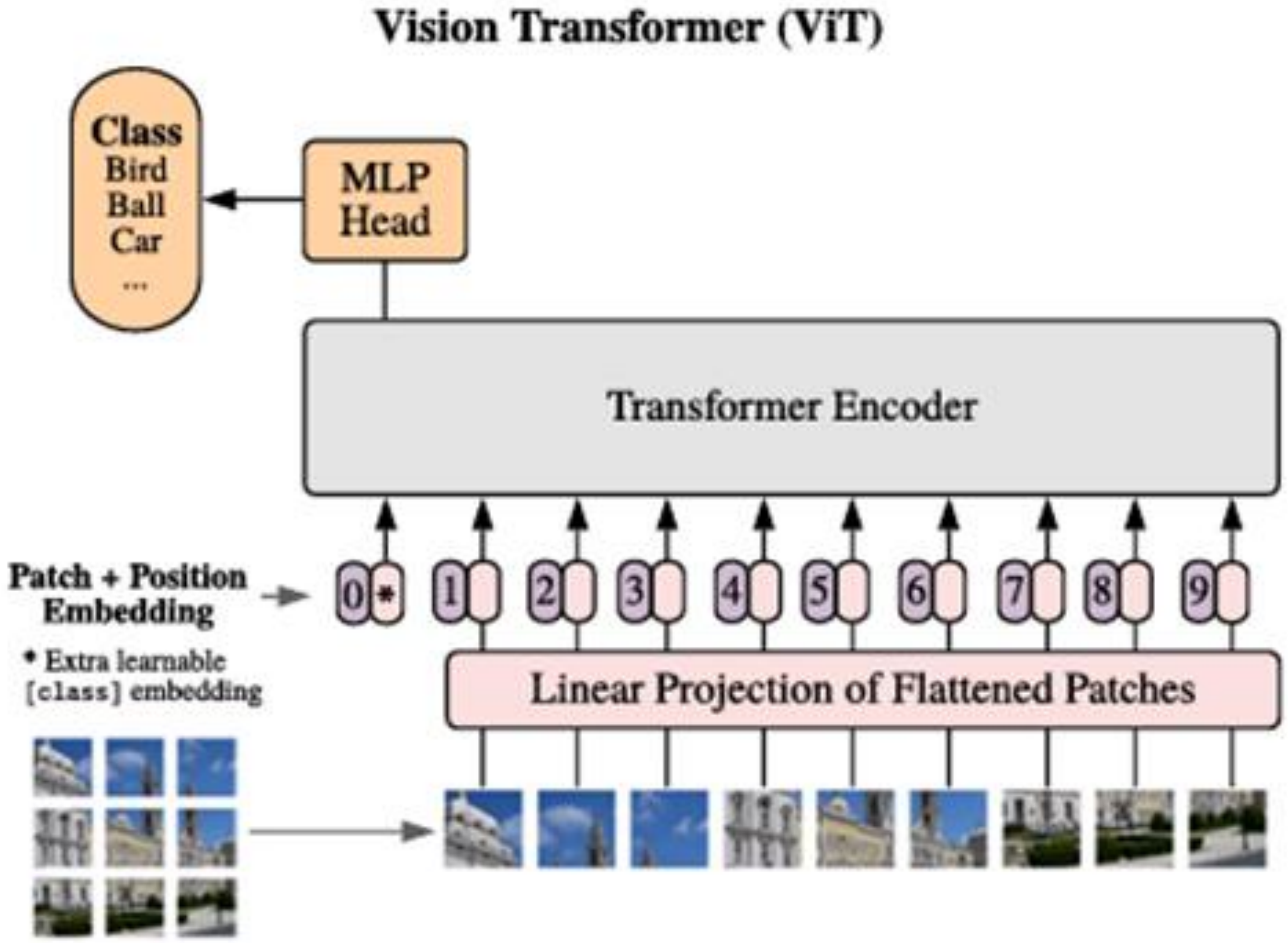
Vision Transformer

CNNs

- CNN uses kernel to aggregate local information in each layer
  - Passed to next layer which aggregates local information again but with a larger field of view
  - It looks at information already aggregated by previous layer
  - Receptive field becomes more global in each layer
- Vision Transformer succeeds at having a large receptive field



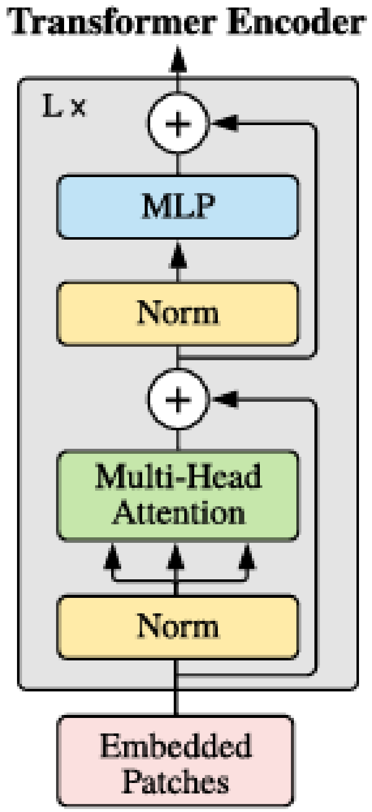
Architecture



https://openreview.net/pdf?id=YicbFdNTTy

Transformer Encoder

- Each encoder layer consists of following:
  - Layer normalization before every block
  - Multi-headed Attention (MHA) block
  - MLP block
  - Residual connections after every block
- Can have multiple encoder layers, *L*

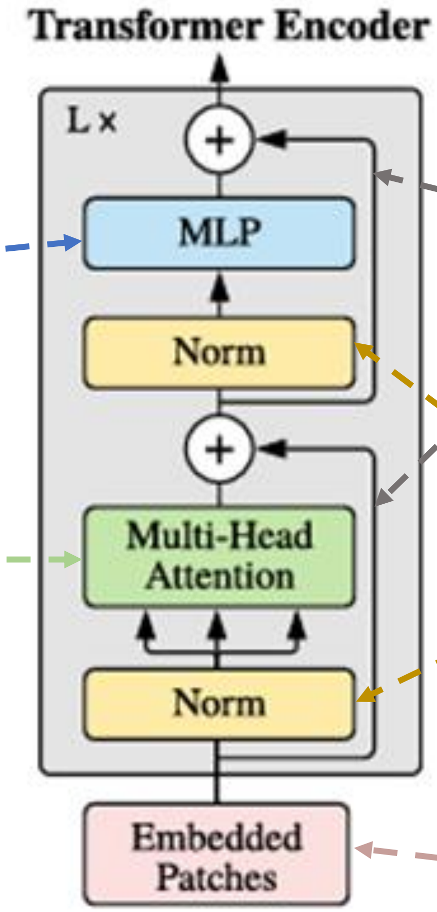


Architecture

Compute heavy blocks

MLP
Allows embeddings to “think” independently

Multi-Head Attention
Allows embeddings to compute



Optimization blocks

Residual Connections
Help gradients flow

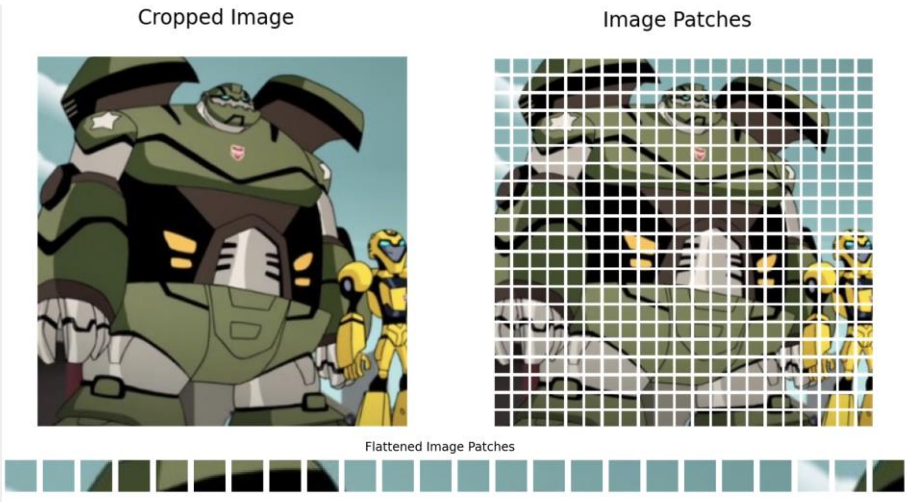
Layer Normalization
Stabilize learning

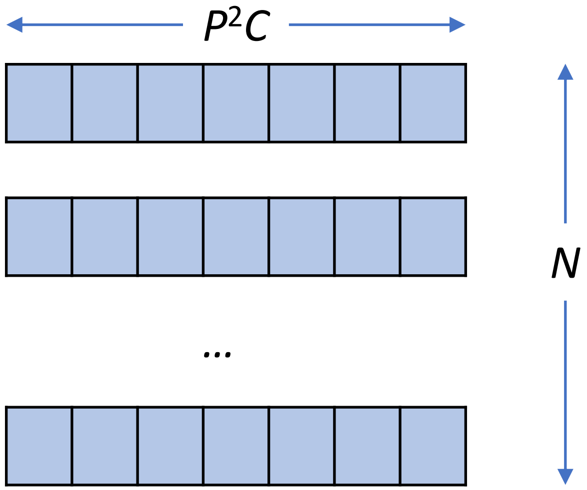
Positional Embeddings
Allow transformer to identify embeddings

https://openreview.net/pdf?id=YicbFdNTTy

Input

- 2D image **X** of size $H * W * C$
- Convert it into patches $\mathbf{x_p}$ of size $P * P * C = P^2C$
- Number of patches $N = \frac{H*W}{P^2}$
- Size of each 1D flattened patch = P^2C
- Dimension of flattened patches = $N * P^2C$

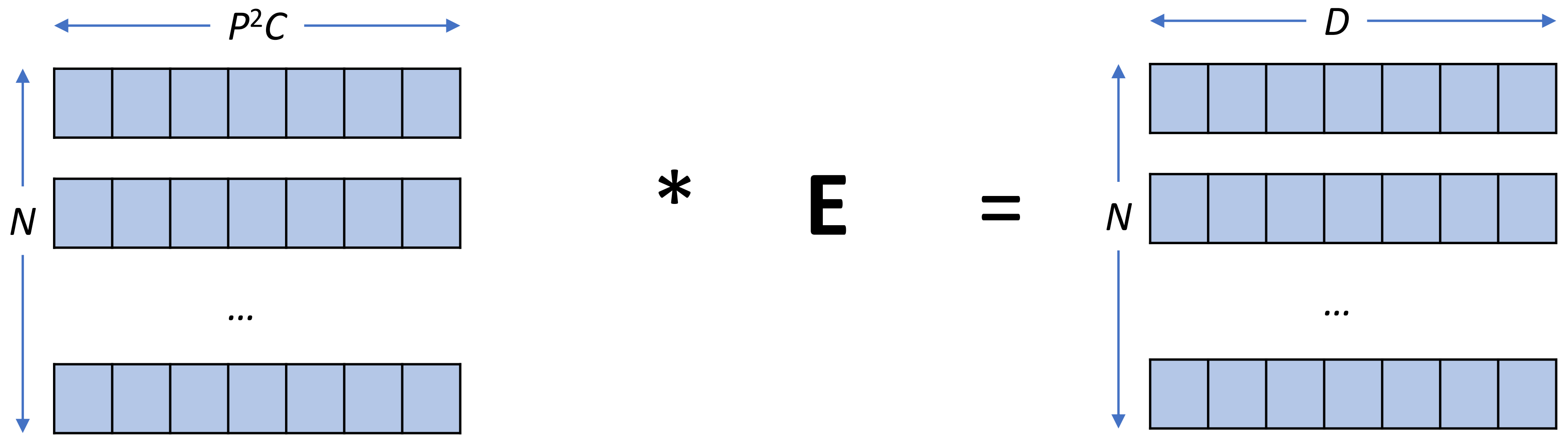




https://gowrishankar.info/blog/transformers-everywhere-patch-encoding-technique-for-vision-transformersvit-explained/

Linear Projection

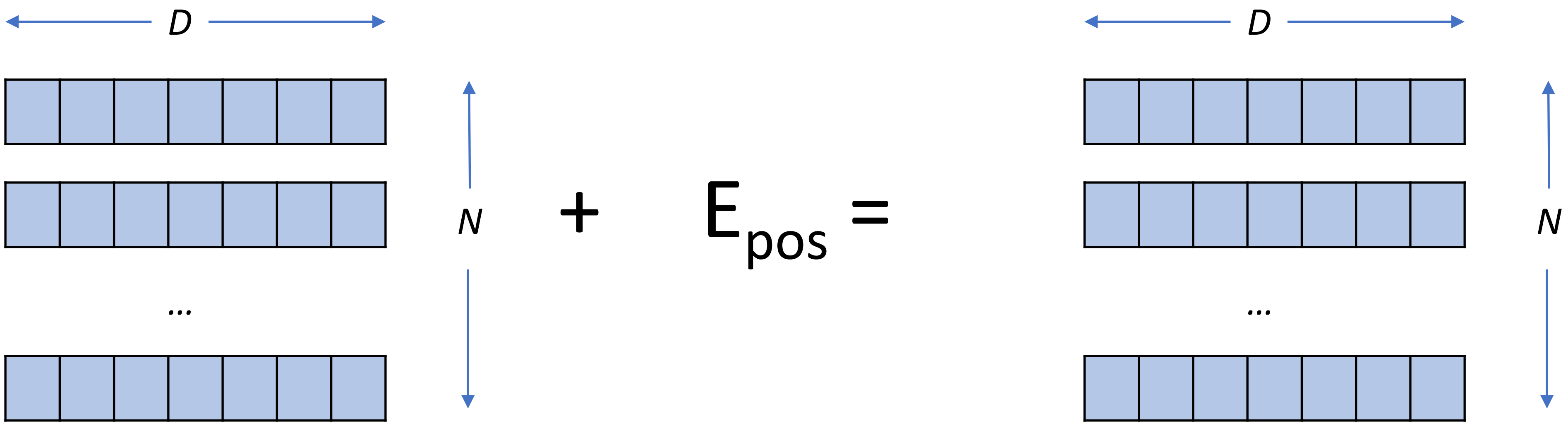
- Map flattened patches to *D* dimensions with a trainable linear projection $E \in R^{(P^2.C)*D}$
- Output of this projection - **patch embeddings**
- $\mathbf{x}_p^1\mathbf{E}; \mathbf{x}_p^2\mathbf{E}; \mathbf{....x}_p^N\mathbf{E};$



Positional Embedding

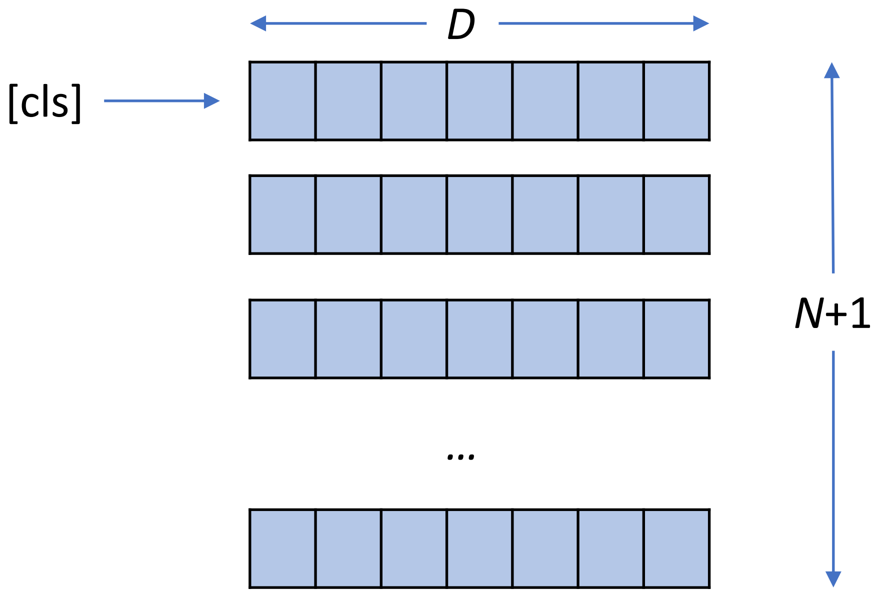
- Add learnable positional embedding $\mathbf{E}_{pos} \in \mathbf{R}^{N*D}$

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



Class Embedding

- Add learnable class embedding [cls] token of shape (1*D)



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

Self Attention

How to allow embeddings to communicate with each other?

- Project each patch embedding three times to compute **Query**, **Key**, **Value**
- These roughly represent:
  - **Query**: "What I am looking for"
  - **Key**: "What I have"
  - **Value:** "What gets communicated"
- Done through learning of three transformation matrices:
  - **Query**: $\mathbf{W}^Q \in \mathbb{R}^{D*d_k}$
  - **Key**: $\mathbf{W}^K \in \mathbb{R}^{D*d_k}$
  - **Value:** $\mathbf{W}^V \in \mathbb{R}^{D*d_v}$
  - d_k is dimension of queries and keys, d_v is dimension of values

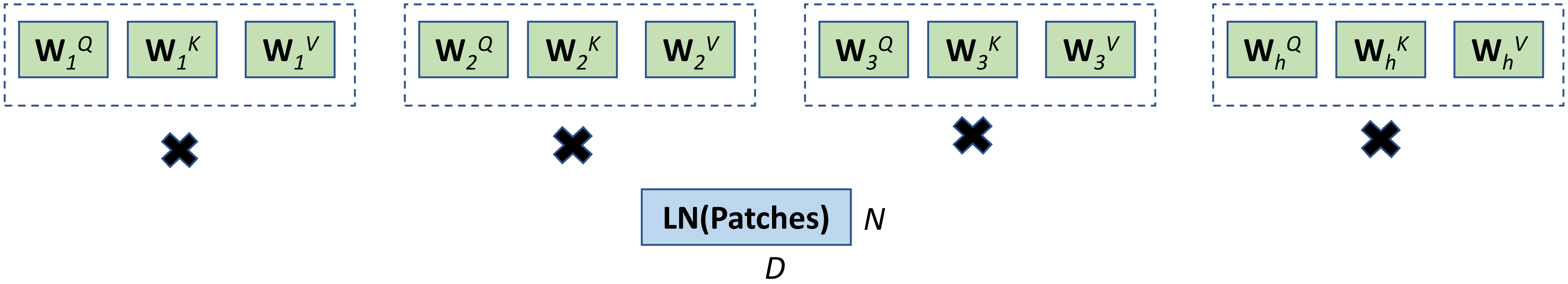
Multi-head Attention (MHA)

Compute **Query**, **Key**, **Value**

- **Query** $\mathbf{Q} = \mathbf{X}\mathbf{W}^{Q} \in \mathbb{R}^{N*d_k}$
- **Key** $\mathbf{K} = \mathbf{X}\mathbf{W}^{K} \in \mathbb{R}^{N*d_k}$
- **Value** $\mathbf{V} = \mathbf{X}\mathbf{W}^{V} \in \mathbb{R}^{N*d_v}$

- Compute $\mathbf{Q}_i$, $\mathbf{K}_i$, $\mathbf{V}_i$ for each head (i = 1..h) with different transformation matrices $\mathbf{W}_i^{Q}$, $\mathbf{W}_i^{K}$, $\mathbf{W}_i^{V}$
- Usually, to make the head dimensions smaller, $d_k = d_v = D/h$

Multi-head Attention (MHA)

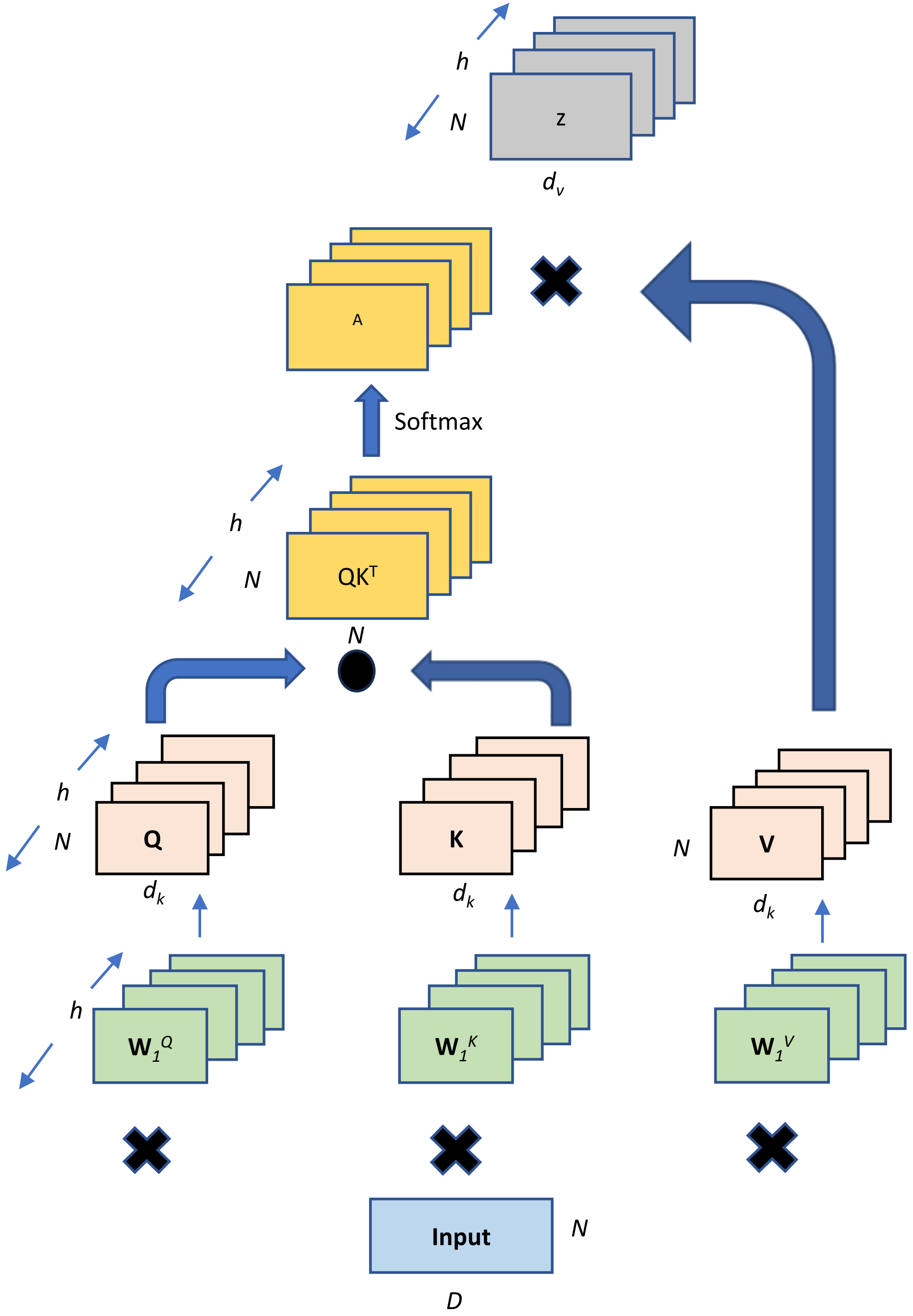


Multi-head Attention (MHA)

- **Attention:** $\mathbf{A}_i$ = softmax($\mathbf{Q}_i\mathbf{K}_i^T/\sqrt{d_k}$) $\in \mathbb{R}^{N*N}$
  - Elements in row 1 of $\mathbf{A}_i$ represent attention of Query 1 against all keys from 1..N
  - Row 2 is attention of Query 2 against all keys and so on..
  - Each row in $\mathbf{A}_i$ sums to 1 (output of softmax are probabilities)
  - Normalized by $\sqrt{d_k}$
    - $q.k$ has variance d_k if q and k are random variables with mean 0 and variance 1

- ***Self Attention*** $\mathbf{z}_i = \mathbf{A}_i\mathbf{V}_i$ $\in \mathbb{R}^{N*d_v}$ //matrix multiplication
  - Row of $\mathbf{z}_i$ is weighted sum of all rows of $\mathbf{V}_i$ with elements from row 1 of $\mathbf{A}_i$ as weights
  - And so on..
  - **Executed in parallel for each head**

https://towardsdatascience.com/into-the-transformer-5ad892e0cee



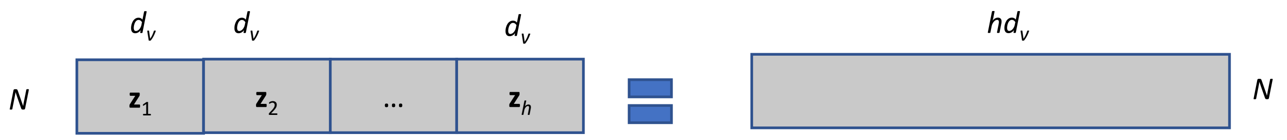
h
N
Z
d_v
A
Softmax
h
N
QK^T
N
h
N
Q
d_k
K
d_k
N
V
d_k
h
W_1^Q
W_1^K
W_1^V
Input
N
D

Multi-head Attention (MHA)

- Concatenate $\mathbf{z}_i$s from all heads: $[\mathbf{z}_1\mathbf{z}_2\ldots\mathbf{z}_h\,]$ $\in \mathbb{R}^{N*hd_v}$
- Multiply with $\mathbf{W^o}$ to get final output of MHA

$$\textbf{Multi-head}(\mathbf{X}) = \mathbf{z} = [\mathbf{z}_1\mathbf{z}_2\ldots\mathbf{z}_h\,]\,\mathbf{W^o}$$

$\mathbf{W^o} \in \mathbb{R}^{hd_v*D}$
$\mathrm{MHA}(\mathbf{X}) \in \mathbb{R}^{N*D}$

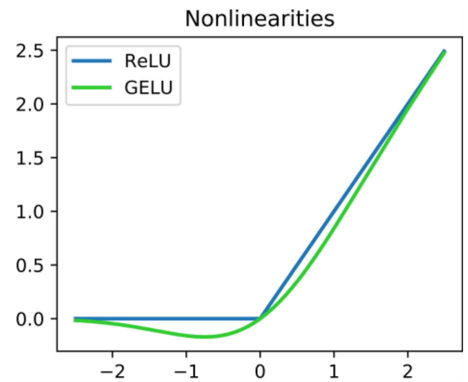


- Complexity of MHA (ignoring the projections) is $O(N^2.D)$
  - Quadratic in sequence length!

Multilayer Perceptron (MLP)

- Apply a 2-layer MLP on each embedding
- Input: $N*D$
- $\text{MLP}(\mathbf{z}) = \mathbf{W}_2\,\sigma(\mathbf{z}*\mathbf{W}_1 + b_1) + b_2$ $\quad W_1: D*d_{mlp}\,;\ W_2: d_{mlp}*D$
- $\sigma(.)$ is a non-linearity. Can be ReLU/GeLU
- Output: $N*D$

ReLU: Rectified Linear Unit
GeLU: Gaussian Error Linear Unit

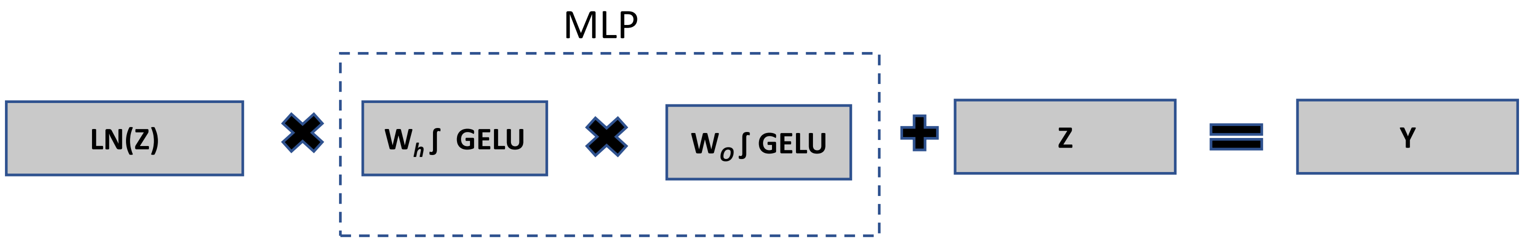


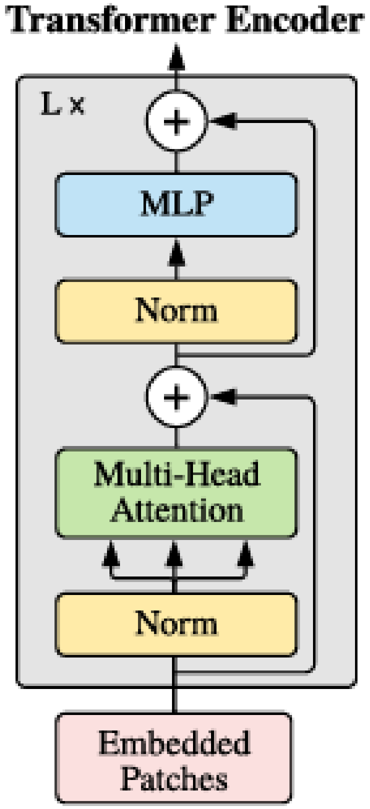
https://ogre51.medium.com/deep-learning-gelu-gaussian-error-linear-unit-activation-function-56168dd5997

MLP

- MLP contains two layers (hidden and output) with a non-linearity

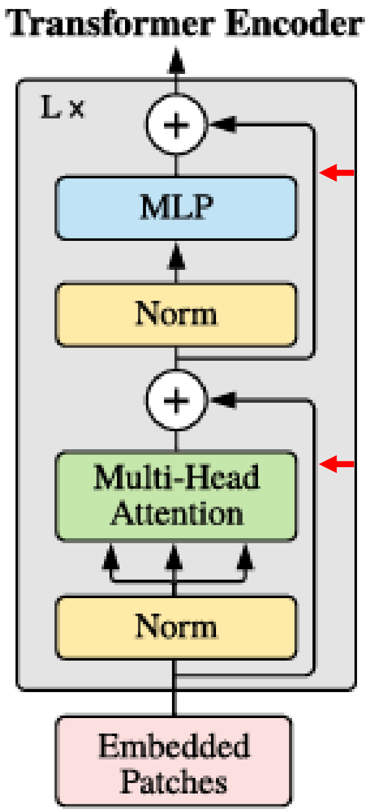
$$\mathbf{Y} = \text{MLP}(\text{LN}(\mathbf{z})) + \mathbf{z}$$





Residual Connection

- Add residual connection to MHA
  - Offer gradients alternative paths, to solve problem of vanishing gradients in very deep architectures
  - Help in optimization

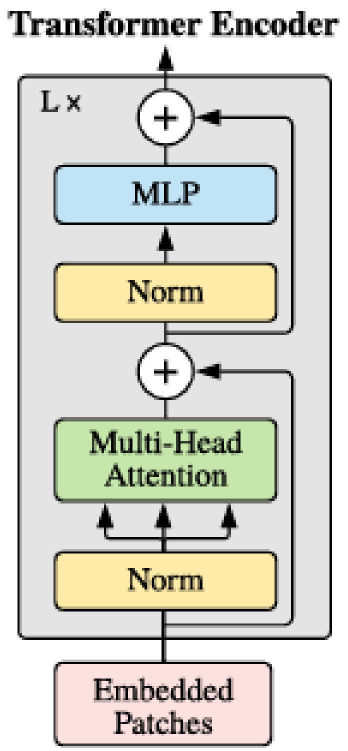


$\mathbf{Y}$ = MLP(norm($\mathbf{z}$)) + $\mathbf{z}$

$\mathbf{z}$ = MHA(norm($\mathbf{X}$)) + $\mathbf{X}$

Layer Normalization

- Helps to stabilize hidden state dynamics and to reduce training time
- Done by scaling with mean and standard deviation for each training example
  - As opposed to batch norm where this is done per feature
- Resulting features multiplied with a scaling factor
  - Learnable during training
- Then added to a shifting factor
  - Learnable during training



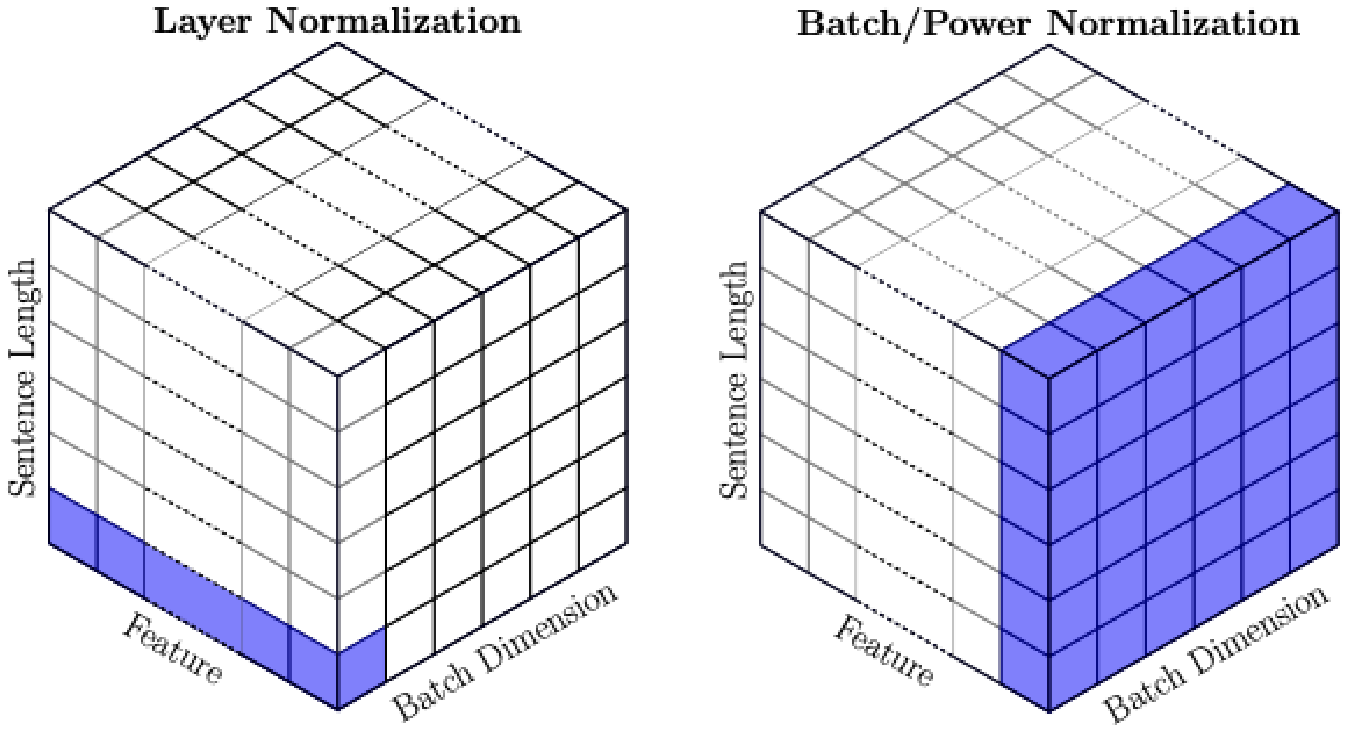
Layer Norm

- Similar to BatchNorm

$$y = \frac{x - \mathrm{E}[x]}{\sqrt{Var[x] + \epsilon}} . \gamma + \beta$$

- Difference with BatchNorm
  - How we estimate E[*x*] and Var[*x*]
  - Input: *N***D* matrices
  - In practice, we process *B***N***D*, where *B* is minibatch size
  - LayerNorm has no dependence on batch dimension, unlike BatchNorm

BatchNorm vs. LayerNorm



https://stats.stackexchange.com/questions/474440/why-do-transformers-use-layer-norm-instead-of-batch-norm

Total parameters

- MHA: $(D*d_k + D*d_k + D*d_k)h + h*d_v*D$
- MLP: $D*d_{ff} + 1*d_{ff} + d_{ff}*D + 1*D$