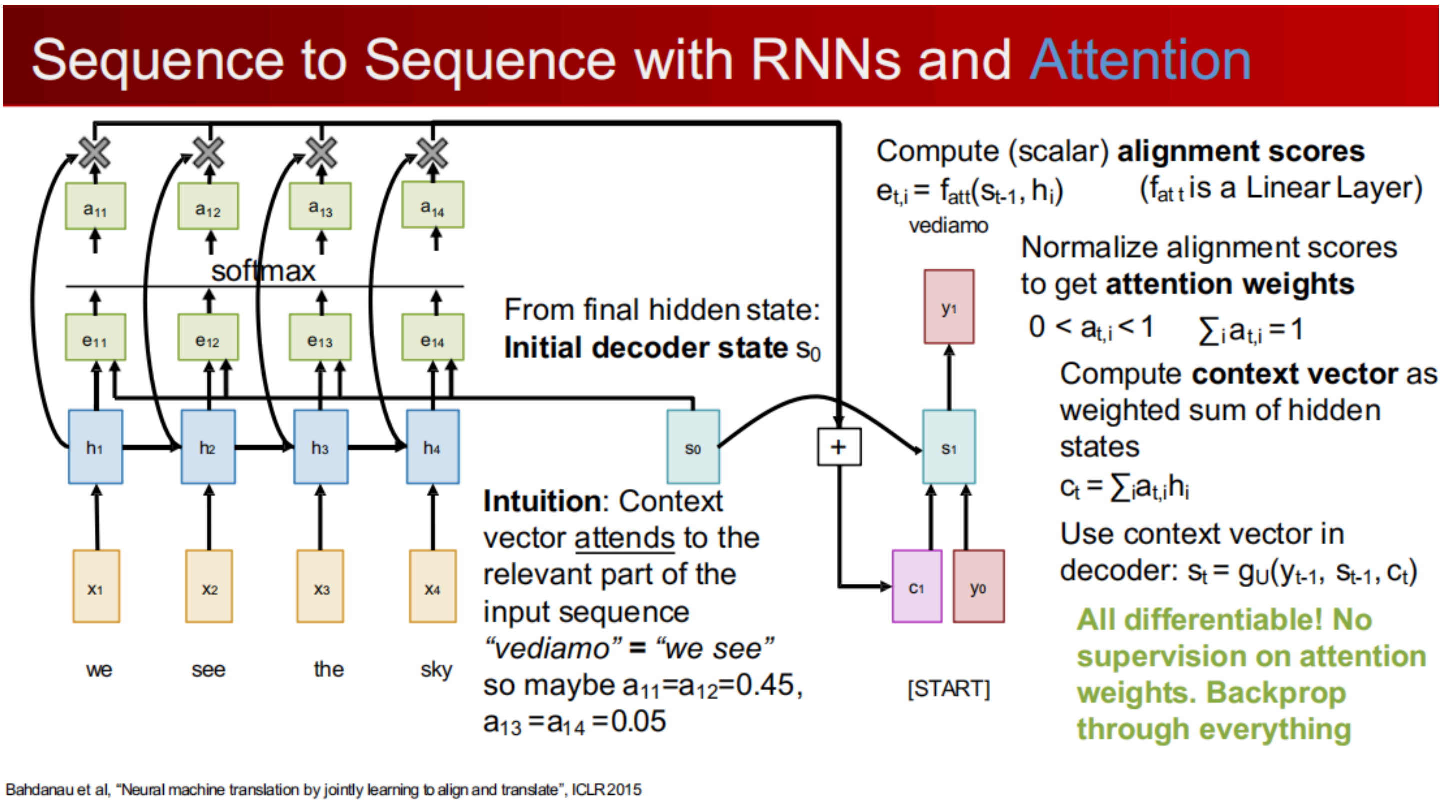
Attention and Transformers

Attention

attention is introduced to solve the limited representation ability of context token in sequence-to-sequence RNNs. It allows the decoder to see all the words in the input sequence at once, and focus on the relevant parts of the input sequence. That saves time(decoder predict all the words in the output sequence at once. But at eval time, it still needs to generate word by word) and improves ability(better context), but it also increase the storage.

algorithm



NOTE: it's the hidden state in the decoder and the hidden state of the encoder that are used to calculate similarity between them.

attention layer

Attention Layer

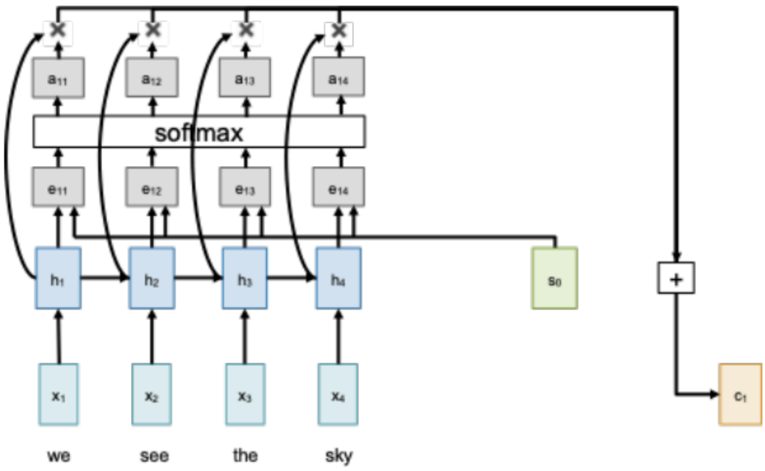
Inputs:

Query vector: **Q**[N_Q x D_Q]

Data vectors: **X**[N_X x D_X]

Key matrix: W_K[D_X x D_Q]

Value matrix: W_V [D_X x D_V]



Computation:

Keys: **K** = XW_K [N_X x D_Q]

Values: **V** = XW_V [N_X x D_V]

Similarities: E = $QK^T / \sqrt{D_Q}$ [N_Q x N_X]

$E_{ij} = Q_i \cdot K_j / \sqrt{D_Q}$

Attention weights: A = softmax(E, dim=1) [N_Q x N_X]

Output vector: **Y** = A**V** [N_Q x D_V]

$Y_i = \sum_j A_{ij} V_j$

Changes

- Use scaled dot product for similarity
- Multiple **query** vectors
- Separate **key** and **value**

NOTE: why do we divide the attention weight by the square root of the hidden state dimension? Because it makes training more stable. If we make assumption that q(Nq x D) and k(Nk x D) are of normal distribution with variance equals to 1, then the production of q and k will have a variance of D. We divide the weight by the square root of D to make the product also has a variance of 1.

self-attention layer

Self-Attention Layer

Inputs:

Input vectors: **X** [N x D_{in}]

Key matrix: W_K[D_{in} x D_{out}]

Value matrix: W_V [D_{in} x D_{out}]

Query matrix: W_Q[D_{in}x D_{out}]

Each **input** produces one **output**, which is a mix of information from all **inputs**

Shapes get a little simpler:

- N input vectors, each D_{in}
- Almost always $D_Q = D_V = D_{out}$

Computation:

Queries: **Q** = XW_Q [N x D_{out}]

Keys: **K** = XW_K [N x D_{out}]

Values: **V** = XW_V [N x D_{out}]

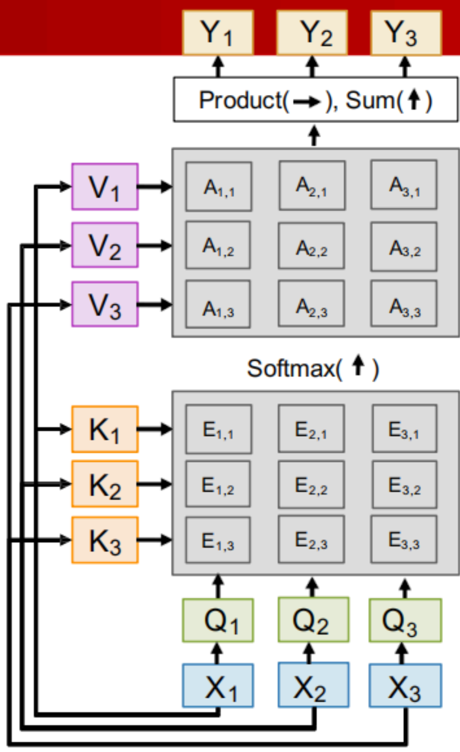
Similarities: E = $QK^T / \sqrt{D_Q}$ [N x N]

$E_{ij} = Q_i \cdot K_j / \sqrt{D_Q}$

Attention weights: A = softmax(E, dim=1) [N x N]

Output vector: **Y** = A**V** [N x D_{out}]

$Y_i = \sum_j A_{ij} V_j$



NOTE: in previous algorithm, the key is actually not the word, but the hidden state, which means it's comparing the query to the sentence before this word. But in self-attention layer, the query is comparing to the word representation itself. This may seems good, but it actually lose some syntax information. In fact, naive self-attention layer is permutation equivariant, which means it doesn't care about the order of the words(or syntax). To solve this we introduce a positional encoding to add the order information.

NOTE: when training self-attention layer, we use a 'mask' (set the similarity to -inf so that after softmax, the weight will be zero) to prevent the model from looking at the future words. That's because if we provide the whole sentence to the model, then the answer is just the next word, we are not sure if the model has understood the sentence or learn the trivial way to generate the next word to perform well at training time. This manner occures also because we want to process all the inputs at once.

NOTE: what's the relationship between position encoding and masking? Though masking is not introduced to solve permutation equivariance problem, it still works in that way, so will it take the work of position encoding, and we don't need position encoding if we use masking? The answer is no. Masking is used in decoder to run predictions in parallel, but it's still poor at solving permutation equivariance problem. So example, in sentence 'abcde', if we are trying to decode 'e', we will calculate it's similarity with 'abcd', but it's the same as doing so with 'bdac', so masking doesn't solve the problem completely.

NOTE: personal opinion: position encoding is used in encoder and mask is not used in encoder to better learn the word representation; position encoding is used in decoder to provide syntax information of output sequence, and maybe self-attention is used on already-generated outputs to better grasp the context before making cross-attentio with the input sequence. Mask is used in decoder to run predictions in parallel.

multiheaded attention

Multiheaded Self-Attention Layer

Run H copies of Self-Attention in parallel

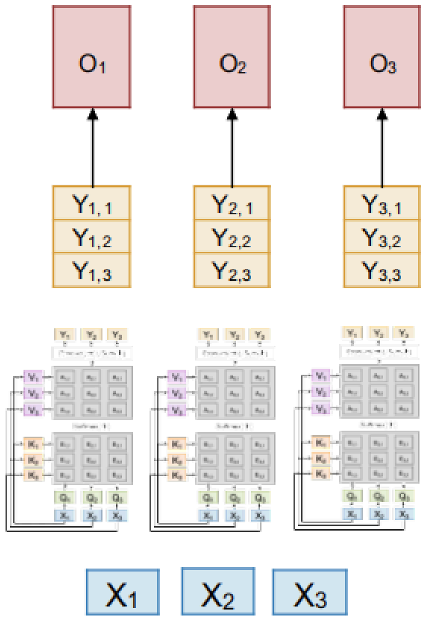
Inputs:
Input vectors: X [N x D_{in}]
Key matrix: W_K [D_{in} x D_{out}]
Value matrix: W_V [D_{in} x D_{out}]
Query matrix: W_Q [D_{in} x D_{out}]

Computation:
Queries: $Q = XW_Q$ [N x D_{out}]
Keys: $K = XW_K$ [N x D_{out}]
Values: $V = XW_V$ [N x D_{out}]
Similarities: $E = QK^T / \sqrt{D_Q}$ [N x N]
$E_{ij} = Q_i \cdot K_j / \sqrt{D_Q}$
Attention weights: A = softmax(E, dim=1) [N x N]
Output vector: $Y = AX$ [N x D_{out}]
$Y_i = \sum_j A_{ij} V_j$

Output projection fuses data from each head

Stack up the H independent outputs for each input X

H = 3 independent self-attention layers (called heads), each with their own weights



Multiheaded Self-Attention Layer

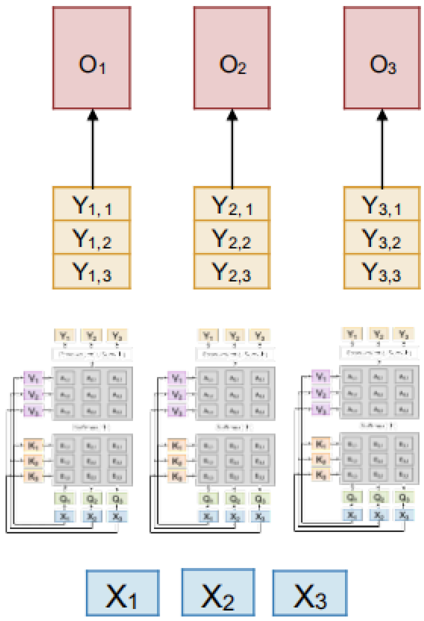
Run H copies of Self-Attention in parallel

Inputs:
Input vectors: X [N x D]
Key matrix: W_K [D x HD_H]
Value matrix: W_V [D x HD_H]
Query matrix: W_Q [D x HD_H]
Output matrix: W_O [HD_H x D]

Computation:
Queries: $Q = XW_Q$ [H x N x D_H]
Keys: $K = XW_K$ [H x N x D_H]
Values: $V = XW_V$ [H x N x D_H]
Similarities: $E = QK^T / \sqrt{D_Q}$ [H x N x N]
Attention weights: A = softmax(E, dim=1) [H x N x N]
Head outputs: $Y = AV$ [H x N x D_H] => [N x HD_H]
Outputs: $O = YW_O$ [N x D]

Each of the H parallel layers use a qkv dim of D_H = "head dim"

Usually $D_H = D / H$, so inputs and outputs have the same dimension



NOTE: multiheaded attention doesn't increase the calculation cost, but it greatly increase the expressiveness for the model. It allows the model to focus on different parts of the input sequence at different times.

complexity

Self-Attention is Four Matrix Multiplies!

Inputs:
Input vectors: X [N x D]
Key matrix: W_K [D x HD_H]
Value matrix: W_V [D x HD_H]
Query matrix: W_Q [D x HD_H]
Output matrix: W_O [HD_H x D]

Computation:
Queries: $Q = XW_Q$ [H x N x D_H]
Keys: $K = XW_K$ [H x N x D_H]
Values: $V = XW_V$ [H x N x D_H]
Similarities: $E = QK^T / \sqrt{D_Q}$ [H x N x N]
Attention weights: A = softmax(E, dim=1) [H x N x N]
Head outputs: $Y = AV$ [H x N x D_H] => [N x HD_H]
Outputs: $O = YW_O$ [N x D]

1. QKV Projection
   [N x D] [D x $3HD_H$] => [N x $3HD_H$]
   Split and reshape to get Q, K, V each of shape [H x N x D_H]
2. QK Similarity
   [H x N x D_H] [H x D_H x N] => [H x N x N]
3. V-Weighting
   [H x N x N] [H x N x D_H] => [H x N x D_H]
   Reshape to [N x HD_H]
4. Output Projection
   [N x HD_H] [HD_H x D] => [N x D]

Q1: how much compute does it take as the number of tokens(N) increases?

A1: O(N^2), as a result of the second and third matrix multiplications.

Q2: how much memory does it take as the number of tokens(N) increases?

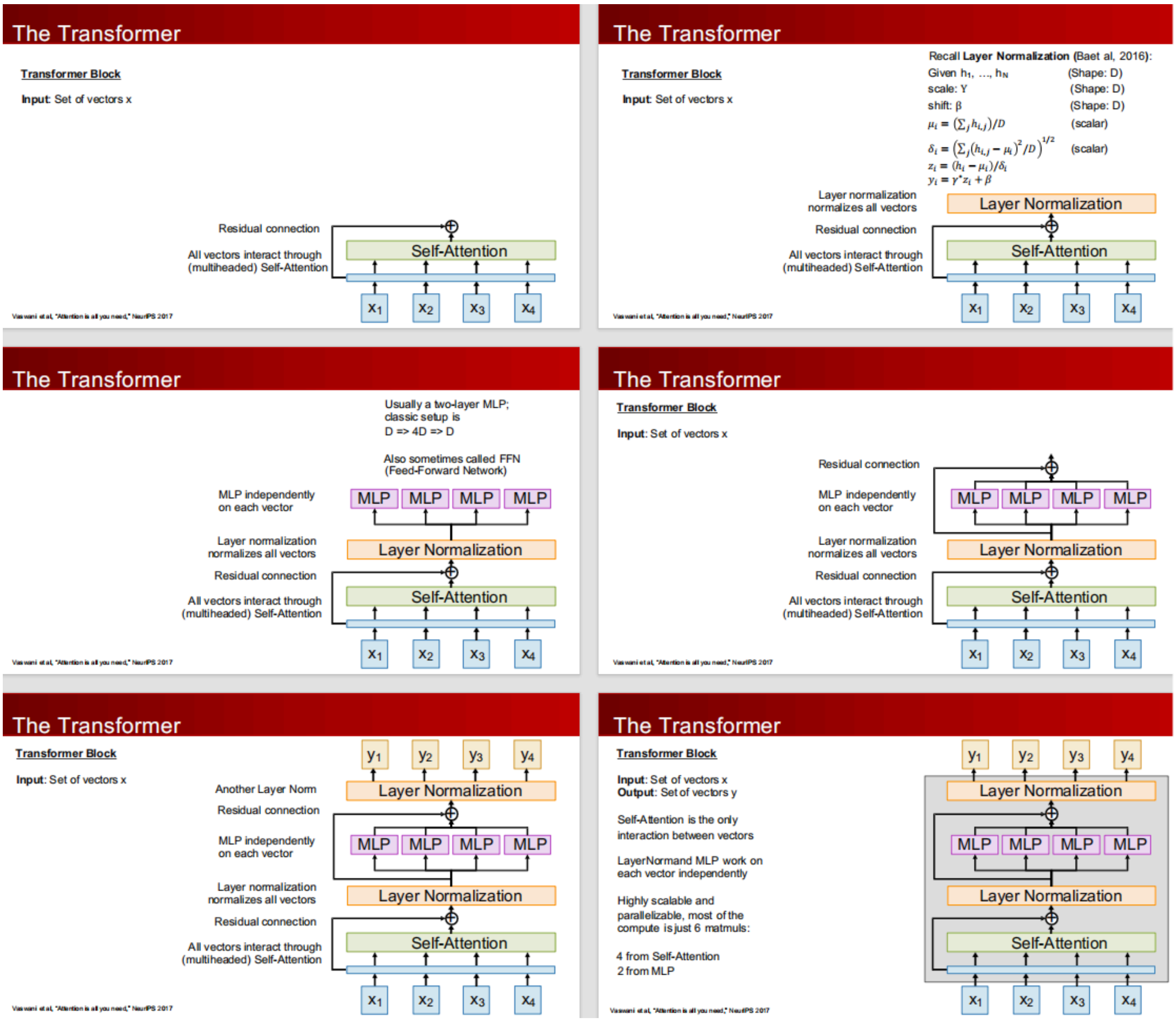
A2: O(N^2), as a result of the attention weights.

S2: use Flash Attention can calculate the second and third step together without storing attention weights. This helps to reduce memory cost to O(N)

Three Ways of Processing Sequences

1. RNN
   (+) Theoretically good at long sequences: O(N) compute and memory for a sequence of length N
   (-) Not parallelizable. Need to compute hidden states sequentially
2. CNN
   (+) Theoretically good at long sequences: O(N) compute and memory for a sequence of length N
   (-) Not parallelizable. Need to compute hidden states sequentially
3. Self-Attention
   (+) Great for long sequences; each output depends directly on all inputs
   (+) Highly parallel, it’s just 4 matmuls
   (-) Expensive: O(N2) compute, O(N) memory for sequence of length N

Transformer



NOTE:

1. layer normalization is applied to every word representation.
2. self-attention layer also has a FC layer at the end

The Transformer

Transformer Block

Input: Set of vectors x
Output: Set of vectors y

Self-Attention is the only interaction between vectors

LayerNormand MLP work on each vector independently

Highly scalable and parallelizable, most of the compute is just 6 matmuls:

4 from Self-Attention
2 from MLP

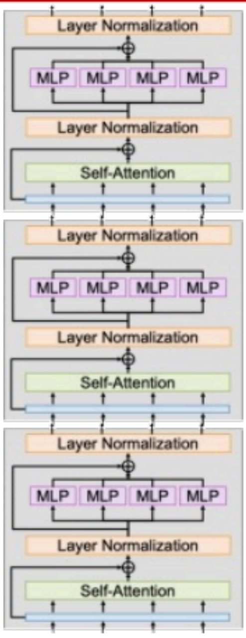
A **Transformer** is just a stack of identical Transformer blocks!

They have not changed much since 2017… but have gotten a lot bigger

Original: [Vaswani et al, 2017]
12 blocks, D=1024, H=16, N=512
213M params

GPT-2: [Radford et al, 2019]
48 blocks, D=1600, H=25, N=1024
1.5B params

GPT-3: [Brown etal, 2020]
96 blocks, D=12288, H=96, N=2048
175B params



Vaswani et al, "Attention is all you need," NeurIPS 2017

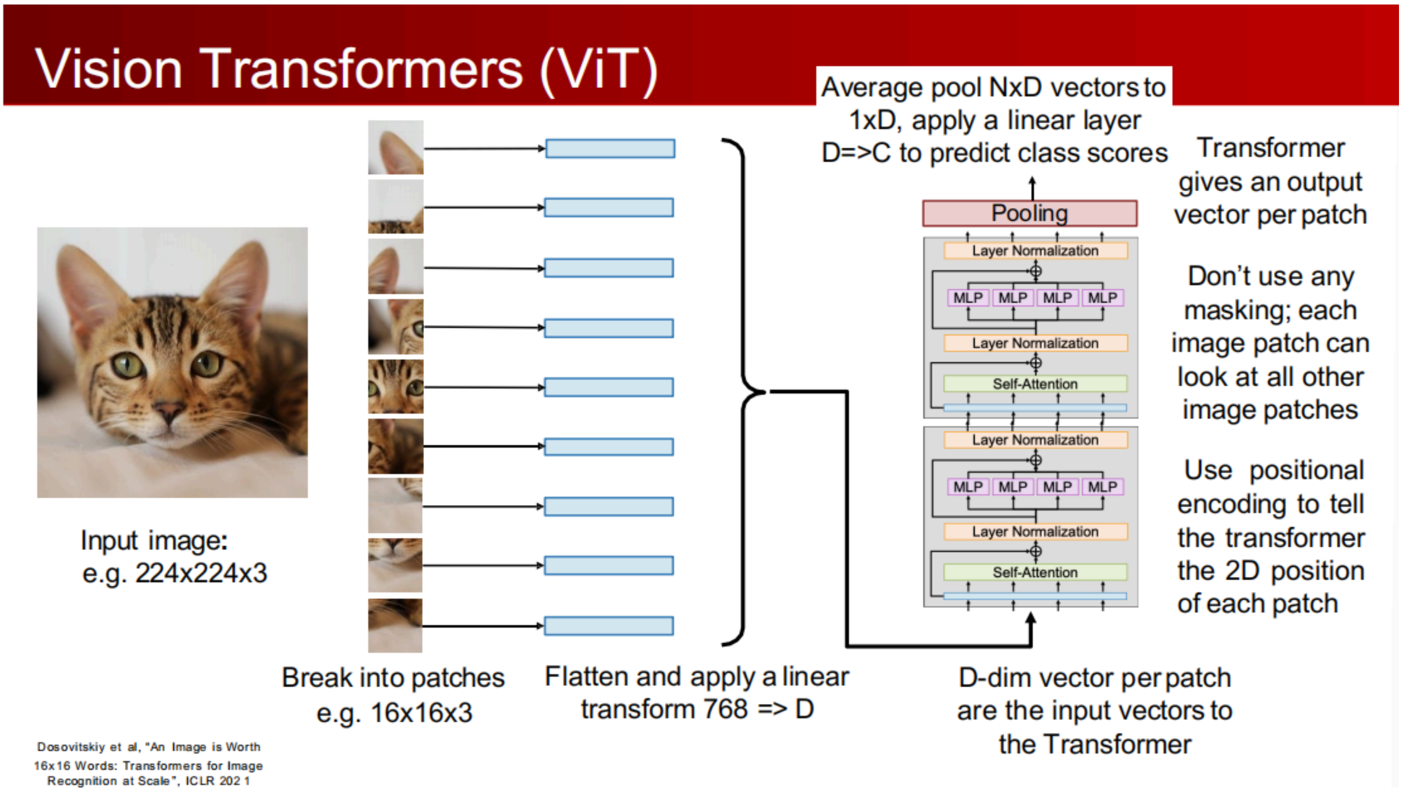
NOTE:

1. parameters:
    - self-attention: Q(DxD), K(DxD), V(DxD), output(DxD)
    - feed-forward: FC1(Dx4D), FC2(4DxD)

Transformers for Language Modeling

- Learn an embedding matrix at the start of the model to convert words into vectors.
- Given vocab size V and model dimension D, it's a lookup table of shape [V x D].
- Use masked attention inside each transformer block so each token can only see the ones before it
- At the end, learn a projection matrix of shape [D x V] to project each D-dim vector to a V-dim vector of scores for each element of the vocabulary.
- Train to predict next token using softmax + cross-entropy loss
- good for translation

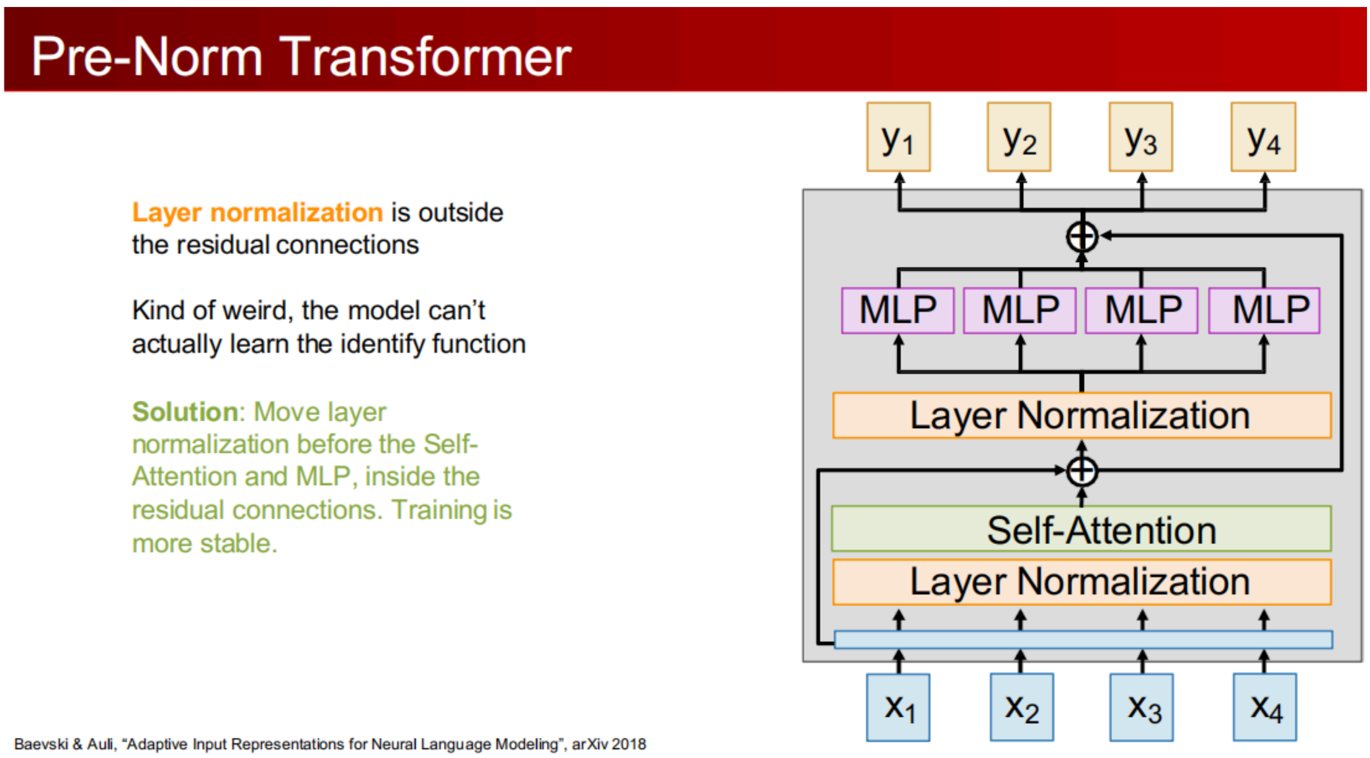
Vision Transformers (ViT)



NOTE:

1. another way to process image patch instead of flatten it is to apply conv layers with output channels equals to D.

Tweaking Transformers



RMSNorm

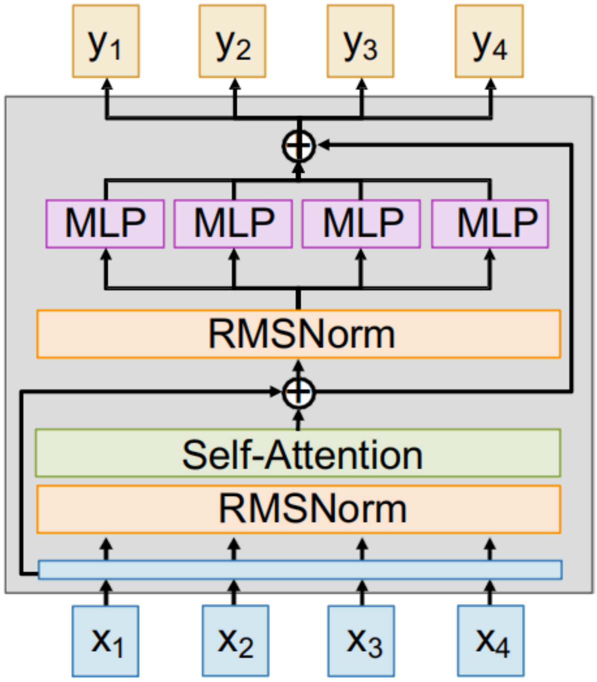
Replace Layer Normalization with Root-Mean-Square Normalization (RMSNorm)

Input: x [shape D]
Output: y [shape D]
Weight: γ [shape D]

$$y_i = \frac{x_i}{RMS(x)} * \gamma_i$$

$$RMS(x) = \sqrt{\varepsilon + \frac{1}{N}\sum_{i=1}^{N} x_i^2}$$

Training is a bit more stable



Zhang and Sennrich, "Root Mean Square Layer Normalization", NeurIPS 2019

NOTE:

1. RMS gives all word representations the same norm.

SwiGLU MLP

<u>**Classic MLP**</u>:

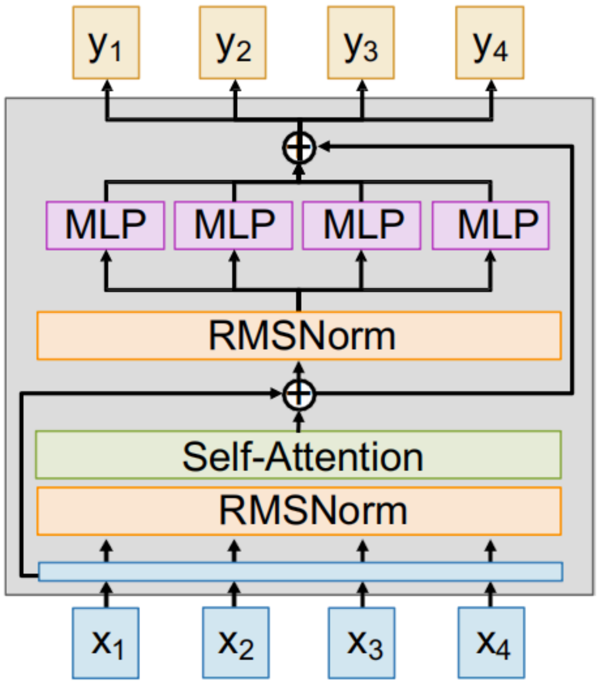
Input: X [N x D]
Weights: W_1 [D x $4D$]
W_2 [$4D$ x D]
Output: $Y = \delta(XW_1)\, W_2$ [N x D]

<u>**SwiGLU MLP**</u>:

Input: X [N x D]
Weights: $W_1, W_2[D \times H]$
$W_3[D \times H]$
Output:

$$Y = (\delta(XW_1) \odot XW_2)W_3$$

Setting H = 8D/3 keeps same total params



Shazeer, "GLU Variants Improve Transformers", 2020

NOTE:

1. SwiGLU is a bit like LSTM. It learns some forget gate and output gate to control the information flow.

Mixture of Experts (MoE)

Learn E separate sets of MLP weights in each block; each MLP is an *expert*

W_1: [D x 4D] => [E x D x 4D]
W_2: [4D x D] => [E x 4D x D]

Each token gets *routed* to A < E of the experts. These are the *active experts*.

Increases params by E,
But only increases compute by A

All of the biggest LLMs today (e.g. GPT4o, GPT4.5, Claude 3.7, Gemini 2.5 Pro, etc) almost certainly use MoE and have > 1T params; but they don't publish details anymore