Transformers

CS 288 Spring 2026

UC Berkeley

cal-cs288.github.io/sp26

Berkeley EECS

BAIR

Course Progress So Far

Class #	Topic
1	Intro & n-gram LM
2	Word representation
3	Text classification (Neural networks)
4	Sequence models (RNNs)
5	Seq2seq models
6	Seq2seq (cont'ed) & Transformers
7	Transformers (cont'ed)
8	Pre-training, fine-tuning & prompting
9	Prompting (cont'ed), scaling laws, data curation
10	Post-training
11	Inference methods & Evals
12	Experimental design and human annotation
13	Question answering
14	Retrieval and RAG

"Architecture"

Today

Next lecture

"LLM Recipe"

Class #	Topic
15	Mixture-of-Experts
16	Impact & Social implication
17	*No class*
18	Test-time compute & reasoning models
19	LLM agent
20	VLM models
21	Interactive embodied agents
22	Guest lecture
23	Guest lecture
24	Pragmatics
25	Guest lecture
26	Guest lecture
27	*Project poster presentation*
28	*Project poster presentation*

Transformers

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

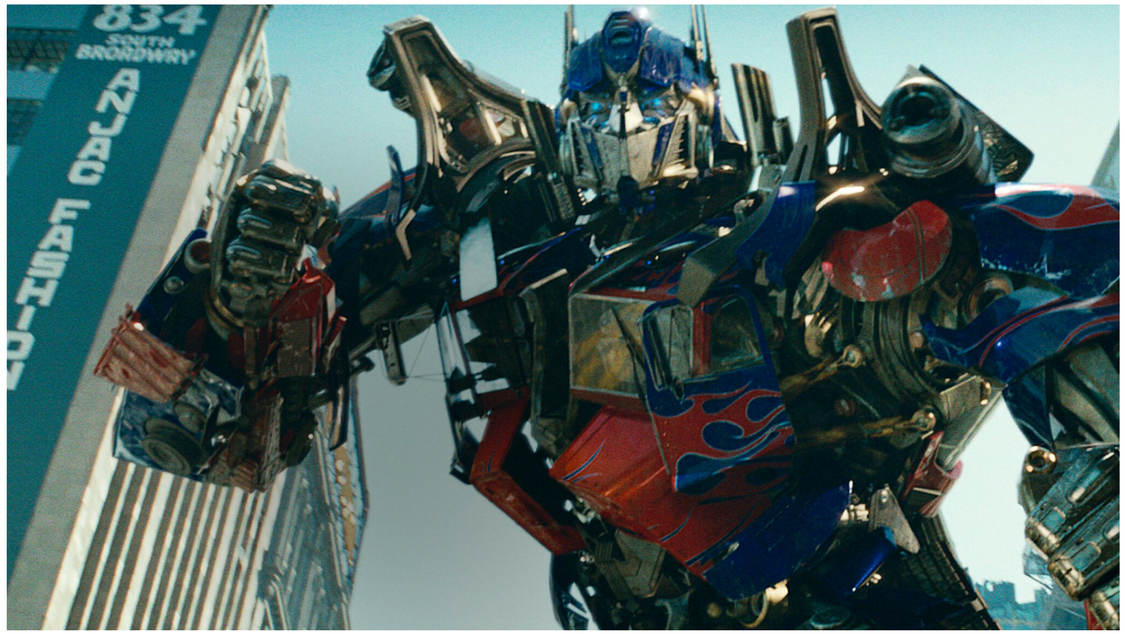
Llion Jones* Google Research llion@google.com

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

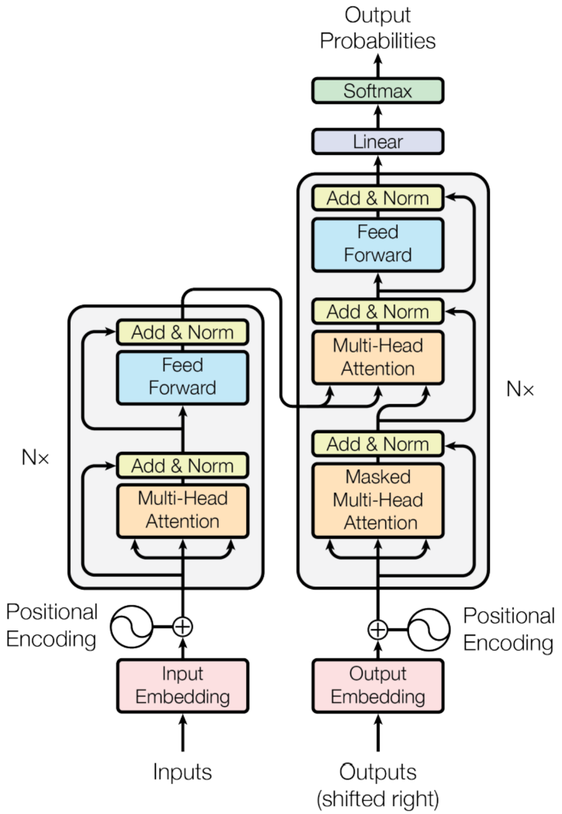
Łukasz Kaiser* Google Brain lukaszkaiser@google.com

Illia Polosukhin* ‡ illia.polosukhin@gmail.com

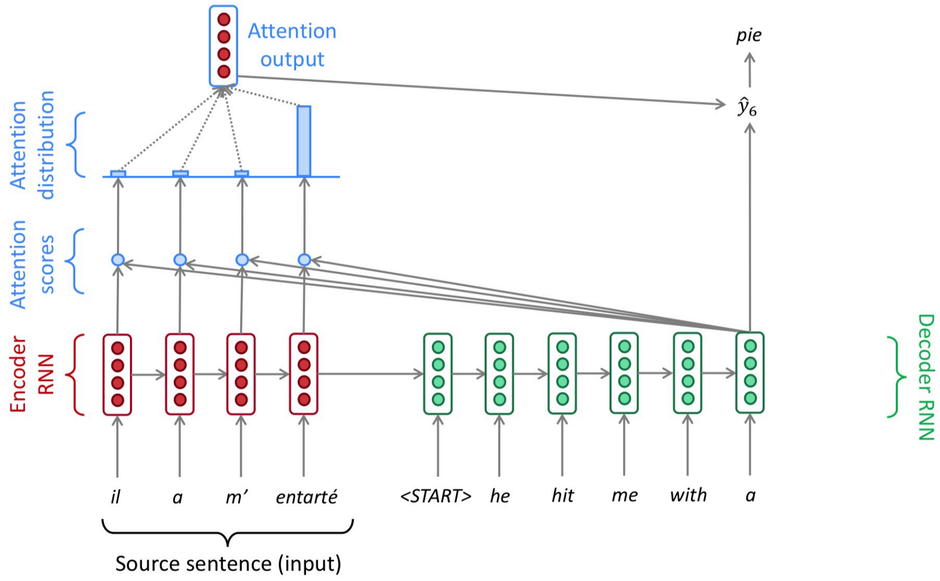
(Vaswani et al., 2017)



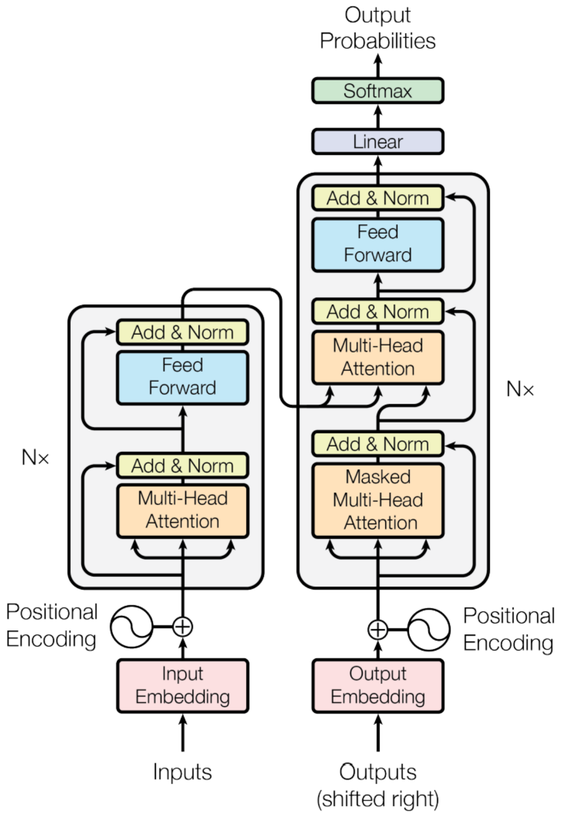
Transformer encoder-decoder



- Transformer encoder + Transformer decoder
- Designed and experimented on NMT
- Can be viewed as a replacement for seq2seq + attention based on RNNs



Transformer encoder-decoder



- Transformer encoder = a stack of **encoder layers**
- Transformer decoder = a stack of **decoder layers**

Transformer encoder: BERT, RoBERTa, ELECTRA

Transformer decoder: GPT-2, GPT-3, Llama

Transformer encoder-decoder: T5, BART

- Key innovation: **multi-head, self-attention**
- Clean & effective architecture design
- Transformers don't have any recurrence structure

$$\mathbf{h}_t = f(\mathbf{h}_{t-1}, \mathbf{x}_t) \in \mathbb{R}^h$$

Issues with recurrent RNs

- Longer sequences can lead to vanishing gradients -> it's hard to capture long-distance information

- RNNs lack parallelizability
  - Forward and backward passes have O(sequence length) unparallelizable operations
  - GPUs can perform a bunch of independent computations at once!
  - Inhibits training on very large datasets

RNNs / LSTMs → seq2seq → seq2seq + attention → attention only = Transformers!

Transformers have become a new building block to replace RNNs

Transformers: Roadmap

1. Attention in Transformers

- From attention to self-attention
- From self-attention to multi-head self-attention
- From multi-head self-attention to masked multi-head self-attention

2. Zooming out: Transformers

- Embeddings and positional encoding
- Feedforward layers
- Residual connections + layer normalization
- Transformer encoder vs Transformer decoder vs. Transformer encoder-decoder

3. Modern variants of Transformers

- Embeddings and positional encoding
- Feedforward layers
- Pre-norm vs. Post-norm
- Layernorm vs. RMSNorm
- Activations
- Positional Encodings
- Attention

Transformers: Roadmap

1. Attention in Transformers

- From attention to self-attention
- From self-attention to multi-head self-attention
- From multi-head self-attention to masked multi-head self-attention

2. Zooming out: Transformers

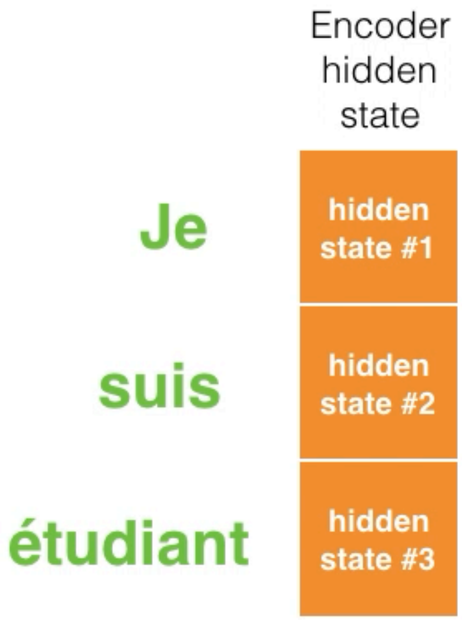
- Embeddings and positional encoding
- Feedforward layers
- Residual connections + layer normalization
- Transformer encoder vs Transformer decoder vs. Transformer encoder-decoder

3. Modern variants of Transformers

- Embeddings and positional encoding
- Feedforward layers
- Pre-norm vs. Post-norm
- Layernorm vs. RMSNorm
- Activations
- Positional Encodings
- Attention

Attention to Self-Attention

Recap: Attention



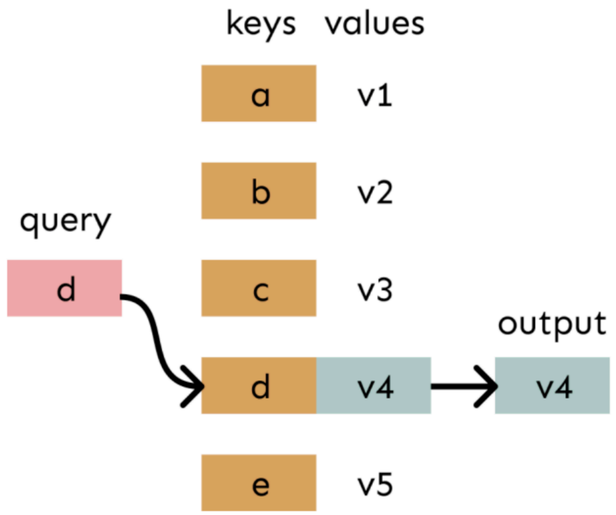
https://jalammar.github.io/visualizing-neural-machine-translation-mechanics-of-seq2seq-models-with-attention/

(credits: Jay Alammar)

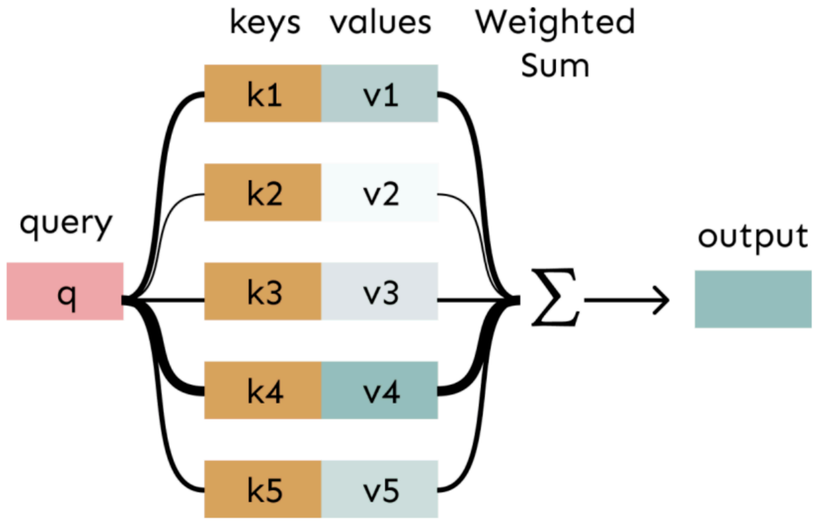
Attention as a soft, averaging lookup table

- We can think of **attention** as performing fuzzy lookup a in **key-value store**

Lookup table: a table of keys that map to values. The query matches one of the keys, returning its value.



Soft look-up table: The query matches to all keys softly to a weight between 0 and 1. The keys' values are multiplied by the weights and summed.



Recap: Attention in a general form

- Assume that we have a set of values $\mathbf{v}_1, \cdots, \mathbf{v}_n \in \mathbb{R}^{d_v}$ and a query vector $\mathbf{q} \in \mathbb{R}^{d_q}$
- Attention always involves the following steps:
  - Computing the **attention scores** $\mathbf{e} = g(\mathbf{q}, \mathbf{v}_i) \in \mathbb{R}^n$
  - Taking softmax to get **attention distribution** α

$$\alpha = \text{softmax}(\mathbf{e}) \in \mathbb{R}^n$$

  - Using attention distribution to take **weighted sum** of values:

$$\mathbf{a} = \sum_{i=1}^{n} \alpha_i \mathbf{v}_i \in \mathbb{R}^{d_v}$$

Attention in a (more) general form

- A model general form: use a set of keys and values $(\mathbf{k}_1, \mathbf{v}_1), \cdots, (\mathbf{k}_n, \mathbf{v}_n), \mathbf{k}_i \in \mathbb{R}^{d_k}, \mathbf{v}_i \in \mathbb{R}^{d_v}$ where keys are used to compute the attention scores and values are used to compute the output vector
- Attention always involves the following steps:
  - Computing the **attention scores** $\mathbf{e} = g(\mathbf{q}, \mathbf{k}_i) \in \mathbb{R}^n$
  - Taking softmax to get **attention distribution** α

$$\alpha = \text{softmax}(\mathbf{e}) \in \mathbb{R}^n$$

  - Using attention distribution to take **weighted sum** of values:

$$\mathbf{a} = \sum_{i=1}^{n} \alpha_i \mathbf{v}_i \in \mathbb{R}^{d_v}$$

Attention in a (more) general form

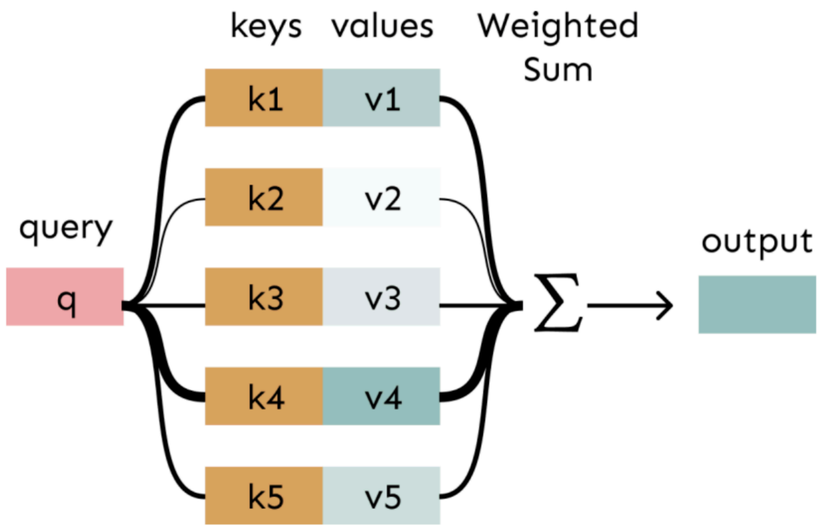
- A model general form: use a set of keys and values $(\mathbf{k}_1, \mathbf{v}_1), \cdots, (\mathbf{k}_n, \mathbf{v}_n), \mathbf{k}_i \in \mathbb{R}^{d_k}, \mathbf{v}_i \in \mathbb{R}^{d_v}$ where keys are used to compute the attention scores and values are used to compute the output vector
- Attention always involves the following steps:
  - Computing the **attention scores** $\mathbf{e} = g(\mathbf{q}, \mathbf{k}_i) \in \mathbb{R}^n$
  - Taking softmax to get **attention distribution** α

$$\alpha = \text{softmax}(\mathbf{e}) \in \mathbb{R}^n$$

  - Using attention distribution to take **weighted sum** of values:

$$\mathbf{a} = \sum_{i=1}^{n} \alpha_i \mathbf{v}_i \in \mathbb{R}^{d_v}$$

Attention in a (more) general form



Query is the subject of attention:
"Here is what I'm looking for"

Key and **value** come from the elements we're attending to — the object of attention

Key: "Here is what I have"
Value: "If you decide I'm relevant, here is the info I'll give you"

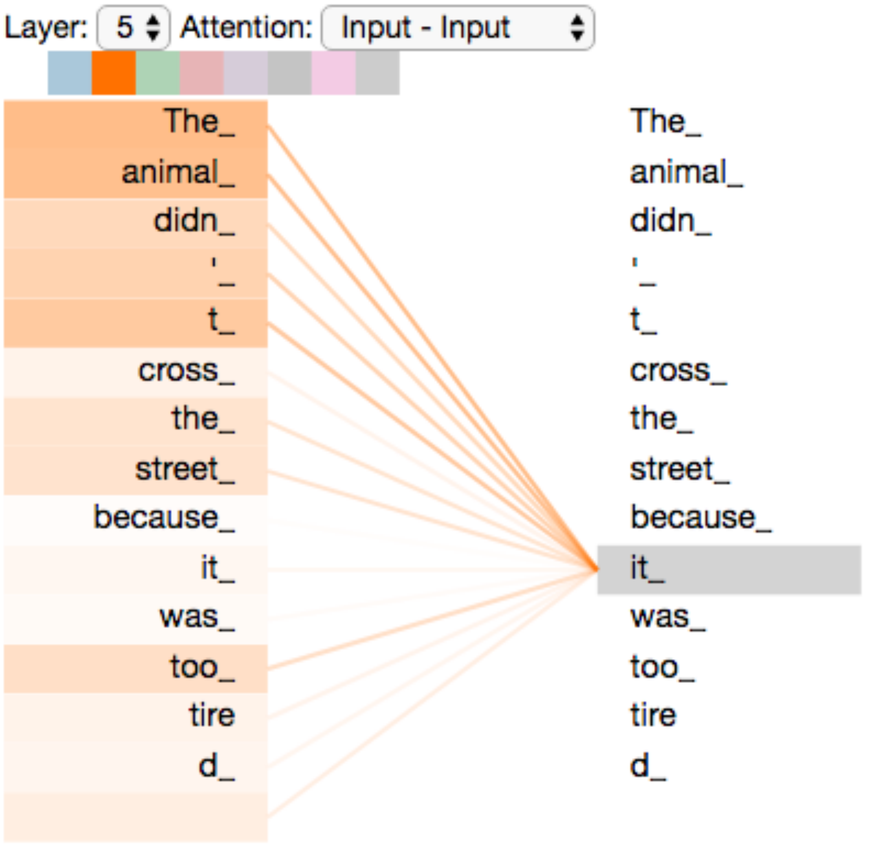
Query and **key** are used to compute an attention distribution. Afterward, the weighted sum is computed using attention distribution & **values**.

In RNN seq2seq:

- Query = current decoder hidden state
- Keys=Values=encoder hidden states

Self-attention

- Self-attention: Attention from the sequence to itself
- Self-attention: let's use each word in a sequence as the query, and all other words in the sequence as keys and values



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

Self-attention

- A self attention layer maps a sequence of input vectors $\mathbf{x}_1, \cdots, \mathbf{x}_n \in \mathbb{R}^{d_1}$ to a sequence of n vectors $\mathbf{z}_1, \cdots, \mathbf{z}_n \in \mathbb{R}^{d_2}$
- The same abstraction as RNNs - use as a drop-in replacement for an RNN layer $\mathbf{z}_t = g(\mathbf{W}\mathbf{z}_{t-1} + \mathbf{U}\mathbf{x}_t + \mathbf{b}) \in \mathbb{R}^{h_2}$

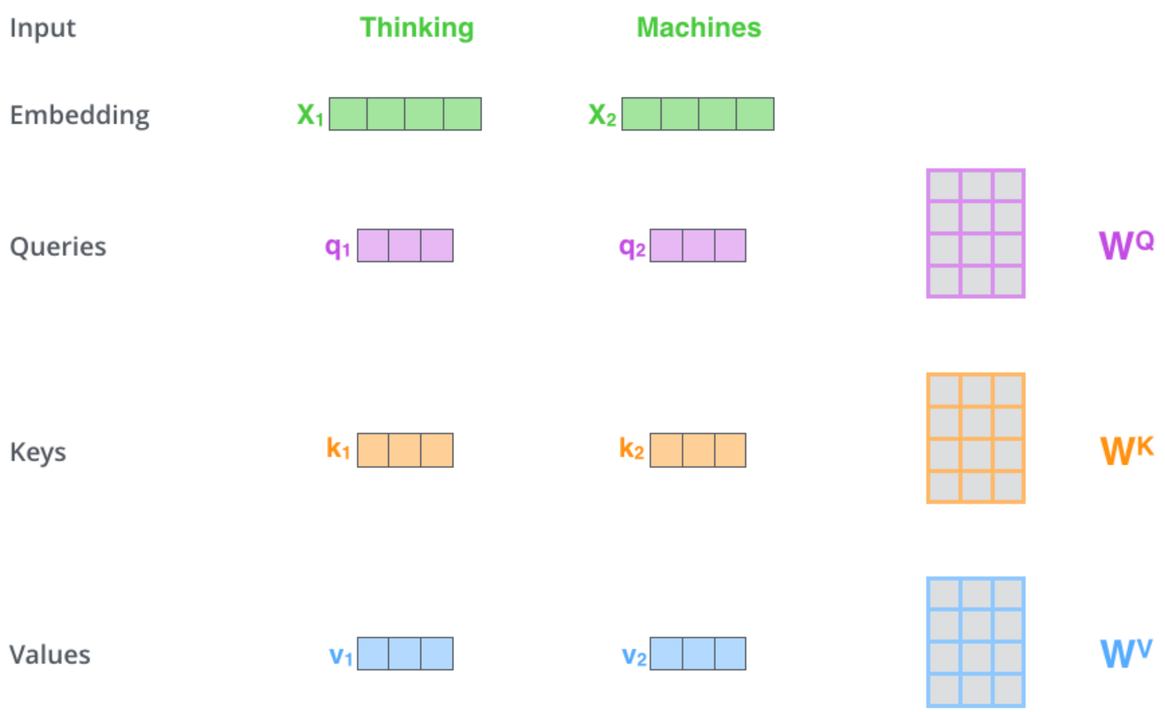
Self-attention

- Step 1: Transform each input vector into three vectors: query, key, and value vectors

$$\mathbf{q}_i = \mathbf{x}_i \mathbf{W}^Q \in \mathbb{R}^{d_q}, \quad \mathbf{W}^Q \in \mathbb{R}^{d_1 \times d_q}$$

$$\mathbf{k}_i = \mathbf{x}_i \mathbf{W}^K \in \mathbb{R}^{d_k}, \quad \mathbf{W}^K \in \mathbb{R}^{d_1 \times d_k}$$

$$\mathbf{v}_i = \mathbf{x}_i \mathbf{W}^V \in \mathbb{R}^{d_v}, \quad \mathbf{W}^V \in \mathbb{R}^{d_1 \times d_v}$$



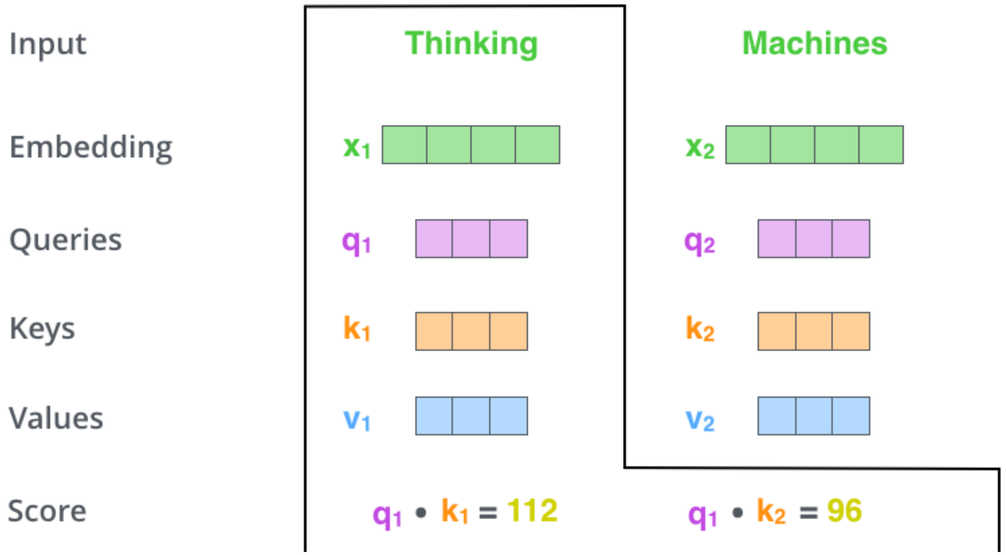
https://jalammar.github.io/illustrated-transformer/

Self-attention

- Step 2: For each query-key pair, calculate the weight based on a doc product.
  - Normalize to add to one using softmax.

$$\alpha_{i,j} = \text{softmax}\left(\mathbf{q}_i \cdot \mathbf{k}_j\right)$$

  - Problem: scale of doc product increases as dimensions get larger

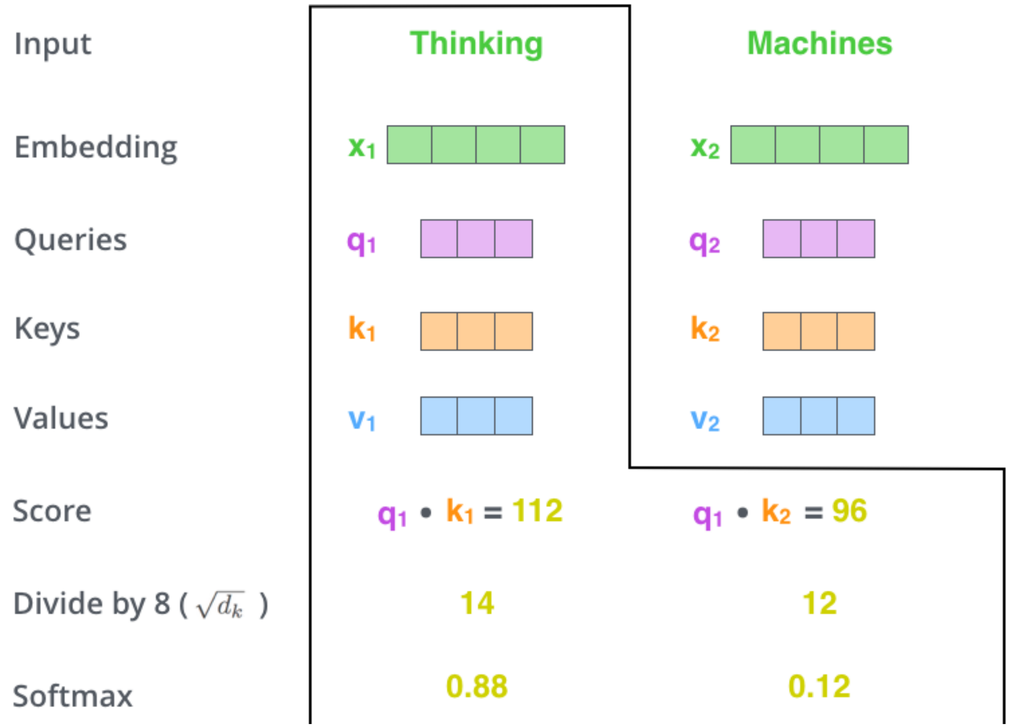


https://jalammar.github.io/illustrated-transformer/

Self-attention

- Step 2: For each query-key pair, calculate the weight based on a doc product.
  - Normalize to add to one using softmax.
  - Problem: scale of doc product increases as dimensions get larger
  - Fix: Scale by size of the vector

$$\alpha_{i,j} = \text{softmax}\left(\frac{\mathbf{q}_i \cdot \mathbf{k}_j}{\sqrt{d_k}}\right)$$

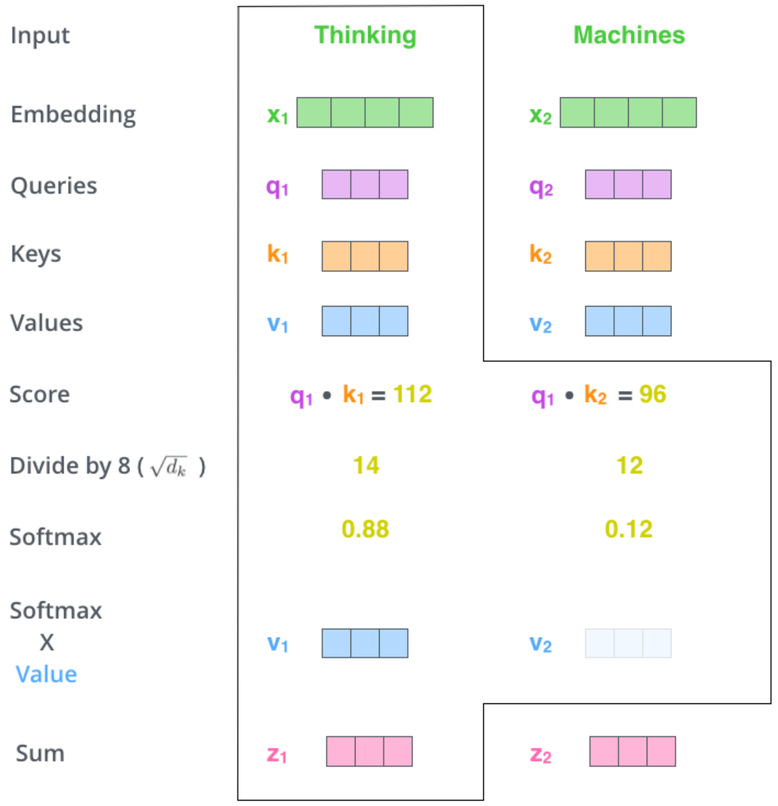


https://jalammar.github.io/illustrated-transformer/

Self-attention

- Step 3: Compute output for each input as weighted some of values

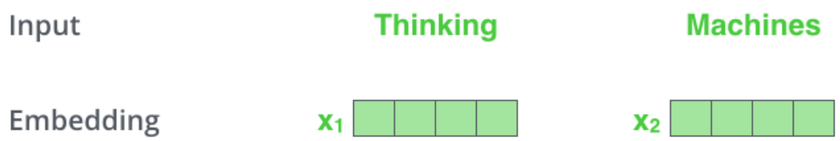
$$\mathbf{z}_i = \sum_{i=1}^{n} \alpha_{i,j} \mathbf{v}_j \in \mathbb{R}^{d_v}$$



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

Self-attention: Summary

Goal: map a sequence of input vectors $\mathbf{x}_1, \cdots, \mathbf{x}_n \in \mathbb{R}^{d_1}$ to a sequence of n vectors $\mathbf{z}_1, \cdots, \mathbf{z}_n \in \mathbb{R}^{d_2}$



Step 1: Compute query, key, and value

$$\mathbf{q}_i = \mathbf{x}_i \mathbf{W}^Q \in \mathbb{R}^{d_q} \quad \mathbf{k}_i = \mathbf{x}_i \mathbf{W}^K \in \mathbb{R}^{d_k} \quad \mathbf{v}_i = \mathbf{x}_i \mathbf{W}^V \in \mathbb{R}^{d_v}$$

Self-attention: Summary

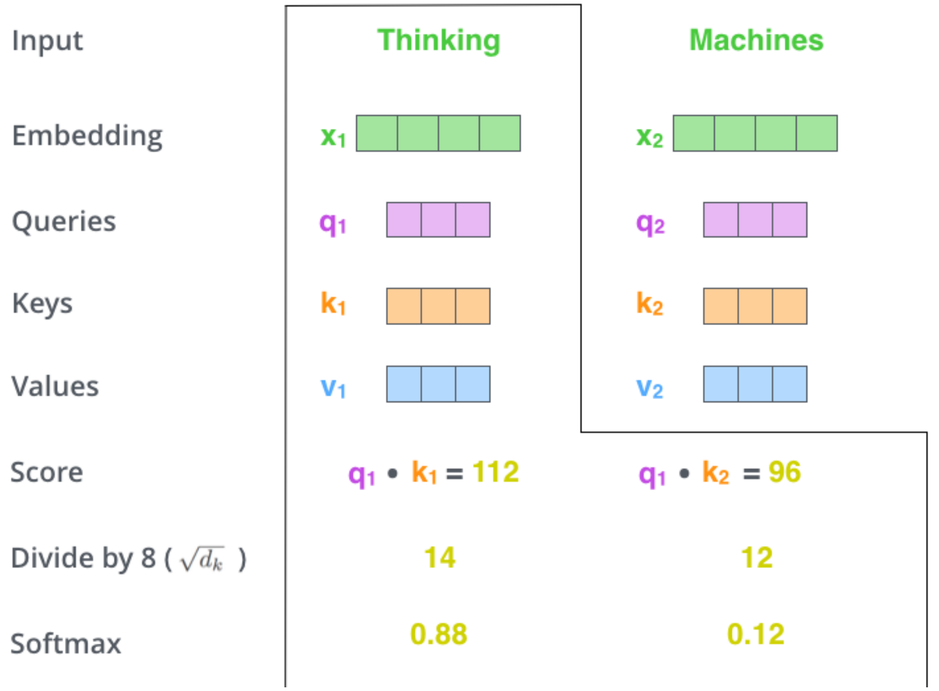
Goal: map a sequence of input vectors $\mathbf{x}_1, \cdots, \mathbf{x}_n \in \mathbb{R}^{d_1}$ to a sequence of n vectors $\mathbf{z}_1, \cdots, \mathbf{z}_n \in \mathbb{R}^{d_2}$

Step 1: Compute query, key, and value

$$\mathbf{q}_i = \mathbf{x}_i \mathbf{W}^Q \in \mathbb{R}^{d_q} \quad \mathbf{k}_i = \mathbf{x}_i \mathbf{W}^K \in \mathbb{R}^{d_k} \quad \mathbf{v}_i = \mathbf{x}_i \mathbf{W}^V \in \mathbb{R}^{d_v}$$

Step 2: Compute attention distribution

$$\alpha_{i,j} = \text{softmax}\left(\frac{\mathbf{q}_i \cdot \mathbf{k}_j}{\sqrt{d_k}}\right)$$



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

Self-attention: Summary

Goal: map a sequence of input vectors $\mathbf{x}_1, \cdots, \mathbf{x}_n \in \mathbb{R}^{d_1}$ to a sequence of n vectors $\mathbf{z}_1, \cdots, \mathbf{z}_n \in \mathbb{R}^{d_2}$

Step 1: Compute query, key, and value

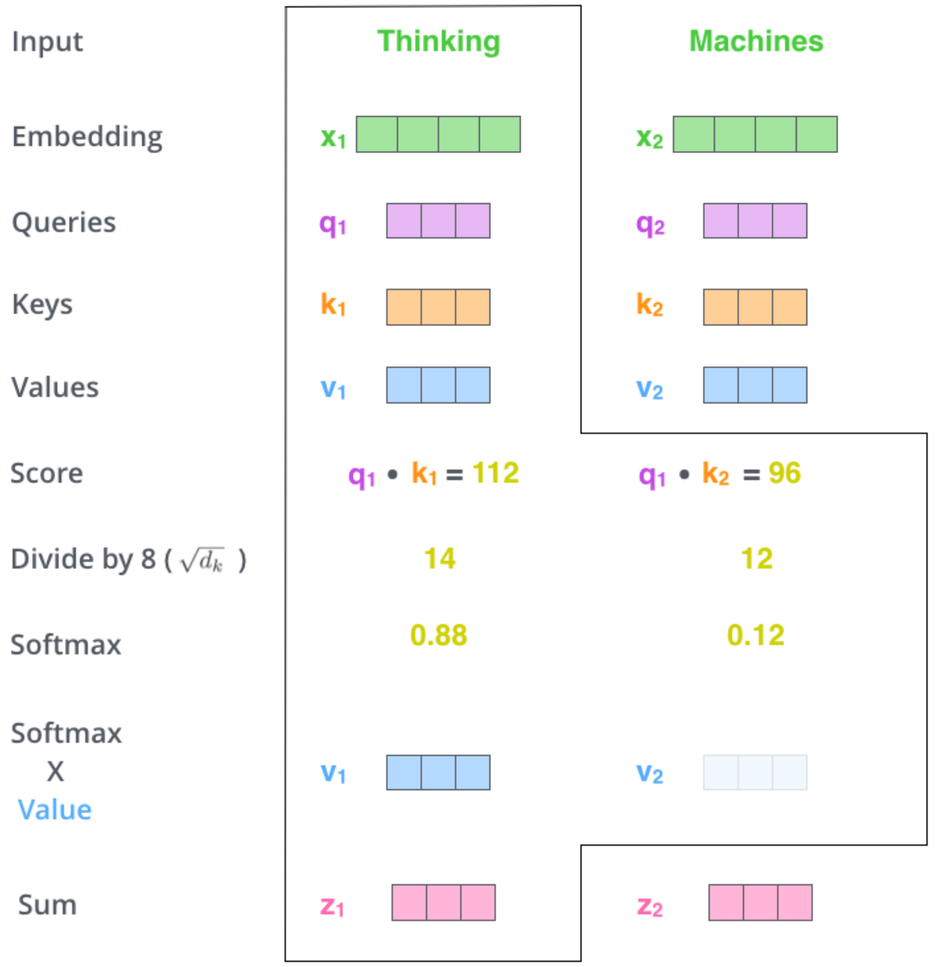
$$\mathbf{q}_i = \mathbf{x}_i \mathbf{W}^Q \in \mathbb{R}^{d_q} \quad \mathbf{k}_i = \mathbf{x}_i \mathbf{W}^K \in \mathbb{R}^{d_k} \quad \mathbf{v}_i = \mathbf{x}_i \mathbf{W}^V \in \mathbb{R}^{d_v}$$

Step 2: Compute attention distribution

$$\alpha_{i,j} = \text{softmax}\left(\frac{\mathbf{q}_i \cdot \mathbf{k}_j}{\sqrt{d_k}}\right)$$

Step 3: Compute the final output

$$\mathbf{z}_i = \sum_{i=1}^{n} \alpha_{i,j} \mathbf{v}_j \in \mathbb{R}^{d_v}$$



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

Feb 10 lecture starts from here

CS 288 Advanced Natural Language Processing

Course website: cal-cs288.github.io/sp26

Ed: edstem.org/us/join/XvztdK

- Class starts at 15:40!
- Today: A1 due, team registration for A3 & project due, at **5:59pm PST**
- A2 is out! (Today's lecture is super relevant!)
- Lecture plan: Complete Transformers

Recap: Transformers

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

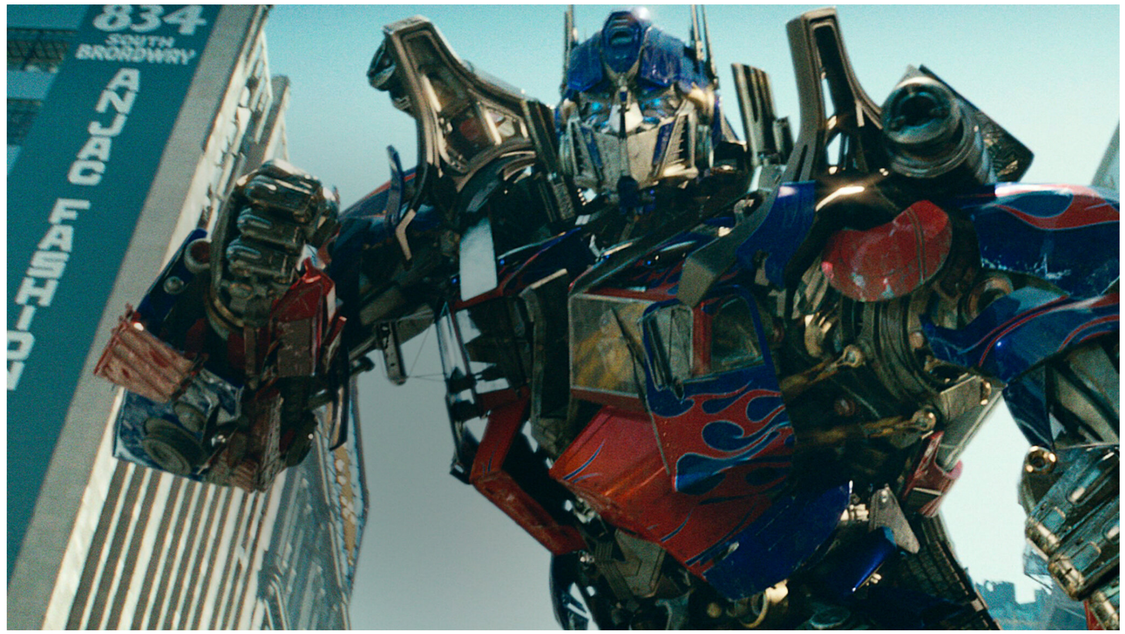
Llion Jones* Google Research llion@google.com

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

Łukasz Kaiser* Google Brain lukaszkaiser@google.com

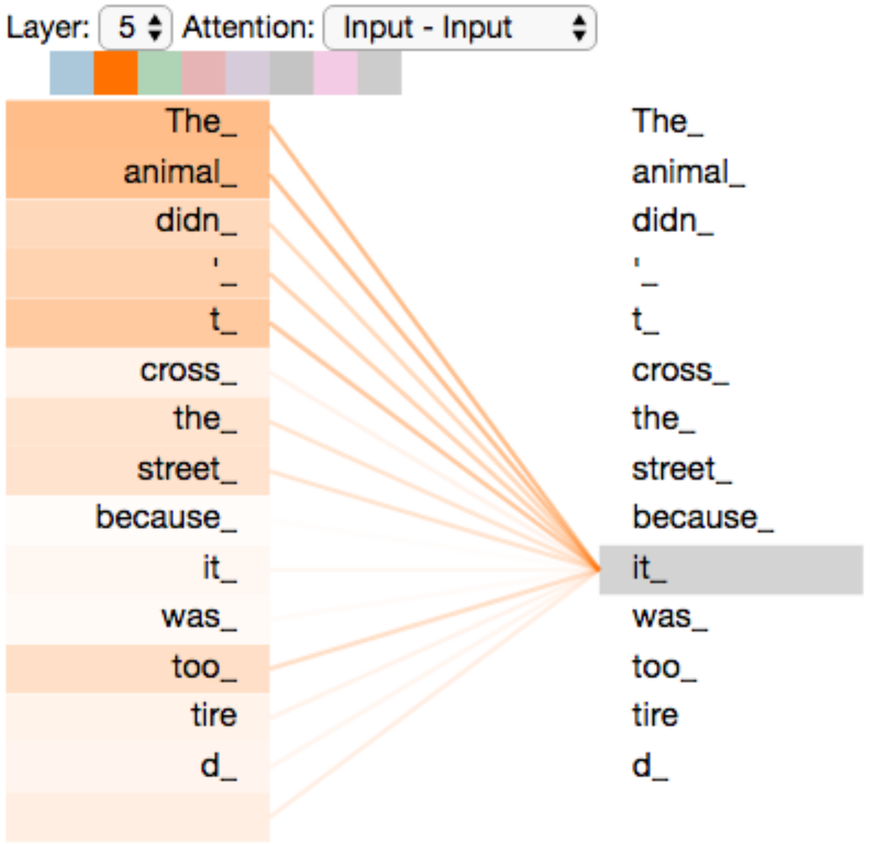
Illia Polosukhin* ‡ illia.polosukhin@gmail.com

(Vaswani et al., 2017)



Recap: Self-attention

- Self-attention: Attention from the sequence to itself
- Self-attention: let's use each word in a sequence as the query, and all other words in the sequence as keys and values

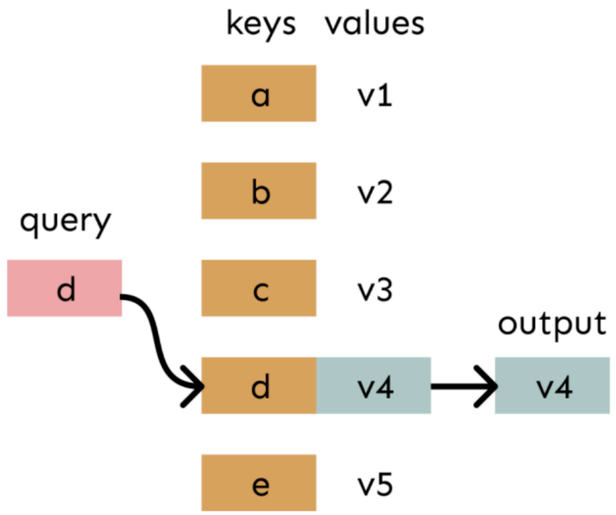


https://jalammar.github.io/illustrated-transformer/

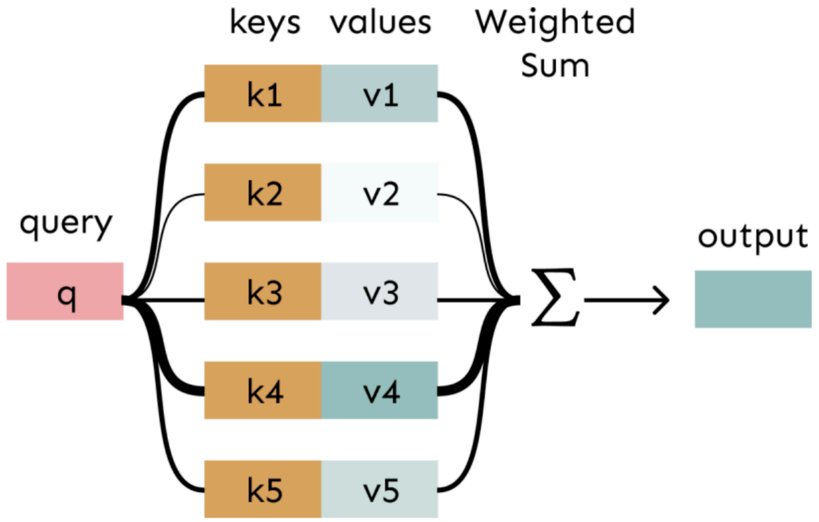
Recap: Attention as a soft, averaging lookup table

- We can think of **attention** as performing fuzzy lookup a in **key-value store**

Lookup table: a table of keys that map to values. The query matches one of the keys, returning its value.

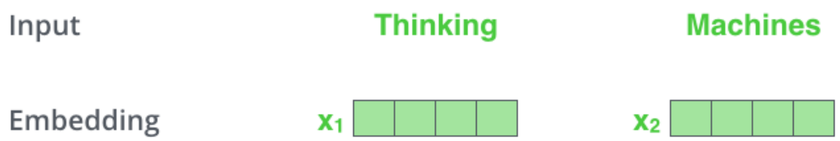


Soft look-up table: The query matches to all keys softly to a weight between 0 and 1. The keys' values are multiplied by the weights and summed.



Self-attention: Summary

Goal: map a sequence of input vectors $\mathbf{x}_1, \cdots, \mathbf{x}_n \in \mathbb{R}^{d_1}$ to a sequence of n vectors $\mathbf{z}_1, \cdots, \mathbf{z}_n \in \mathbb{R}^{d_2}$



Step 1: Compute query, key, and value

$$\mathbf{q}_i = \mathbf{x}_i \mathbf{W}^Q \in \mathbb{R}^{d_q} \quad \mathbf{k}_i = \mathbf{x}_i \mathbf{W}^K \in \mathbb{R}^{d_k} \quad \mathbf{v}_i = \mathbf{x}_i \mathbf{W}^V \in \mathbb{R}^{d_v}$$

Self-attention: Summary

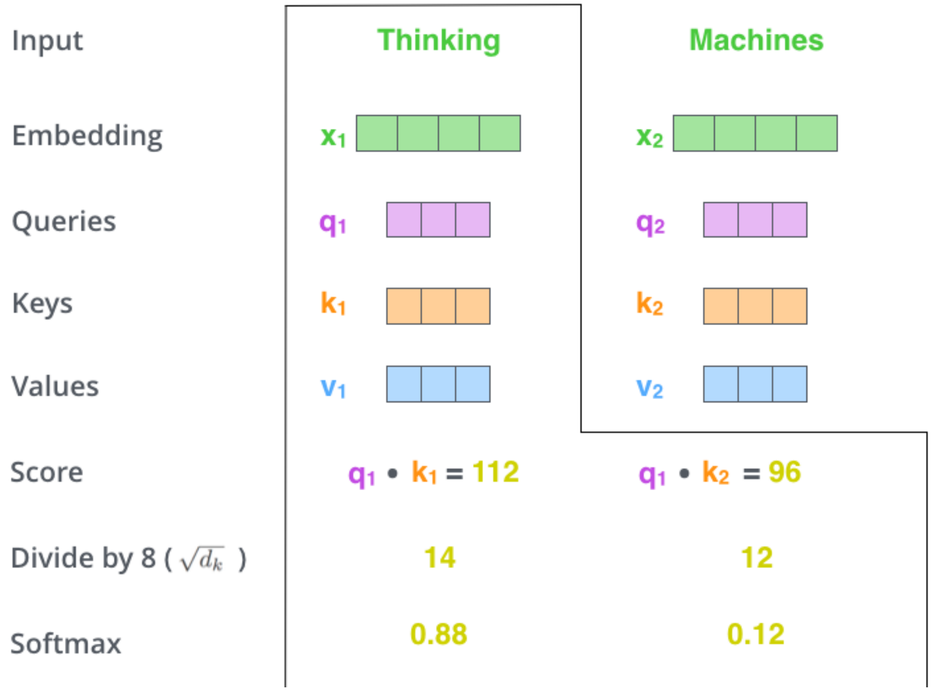
Goal: map a sequence of input vectors $\mathbf{x}_1, \cdots, \mathbf{x}_n \in \mathbb{R}^{d_1}$ to a sequence of n vectors $\mathbf{z}_1, \cdots, \mathbf{z}_n \in \mathbb{R}^{d_2}$

Step 1: Compute query, key, and value

$$\mathbf{q}_i = \mathbf{x}_i \mathbf{W}^Q \in \mathbb{R}^{d_q} \quad \mathbf{k}_i = \mathbf{x}_i \mathbf{W}^K \in \mathbb{R}^{d_k} \quad \mathbf{v}_i = \mathbf{x}_i \mathbf{W}^V \in \mathbb{R}^{d_v}$$

Step 2: Compute attention distribution

$$\alpha_{i,j} = \text{softmax}\left(\frac{\mathbf{q}_i \cdot \mathbf{k}_j}{\sqrt{d_k}}\right)$$



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

Self-attention: Summary

Goal: map a sequence of input vectors $\mathbf{x}_1, \cdots, \mathbf{x}_n \in \mathbb{R}^{d_1}$ to a sequence of n vectors $\mathbf{z}_1, \cdots, \mathbf{z}_n \in \mathbb{R}^{d_2}$

Step 1: Compute query, key, and value

$$\mathbf{q}_i = \mathbf{x}_i \mathbf{W}^Q \in \mathbb{R}^{d_q} \quad \mathbf{k}_i = \mathbf{x}_i \mathbf{W}^K \in \mathbb{R}^{d_k} \quad \mathbf{v}_i = \mathbf{x}_i \mathbf{W}^V \in \mathbb{R}^{d_v}$$

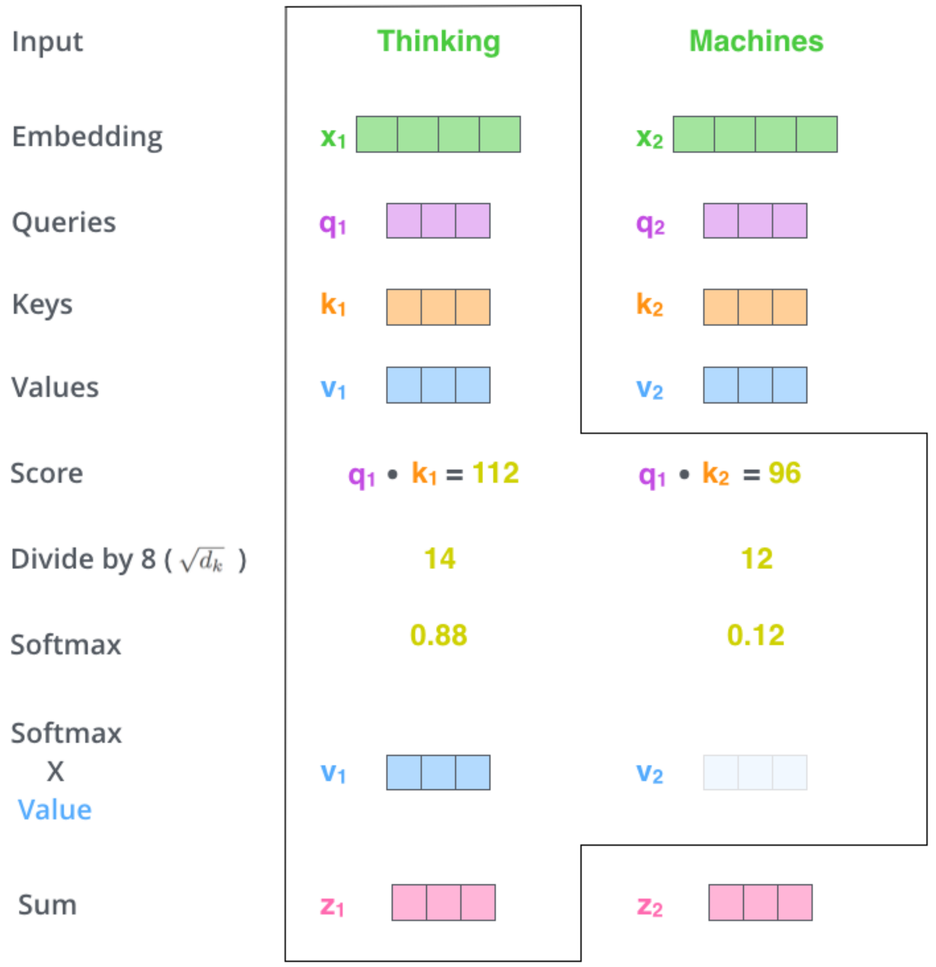
Step 2: Compute attention distribution

$$\alpha_{i,j} = \text{softmax}\left(\frac{\mathbf{q}_i \cdot \mathbf{k}_j}{\sqrt{d_k}}\right)$$

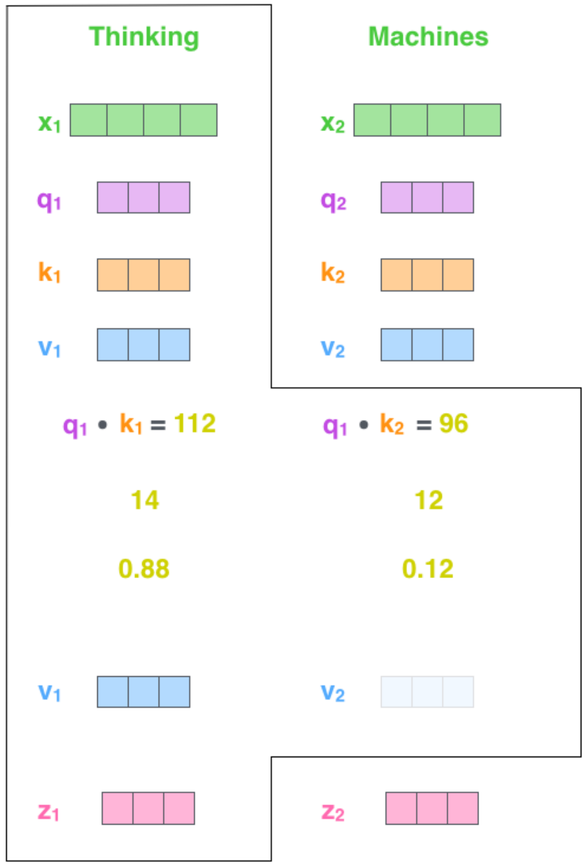
Step 3: Compute the final output

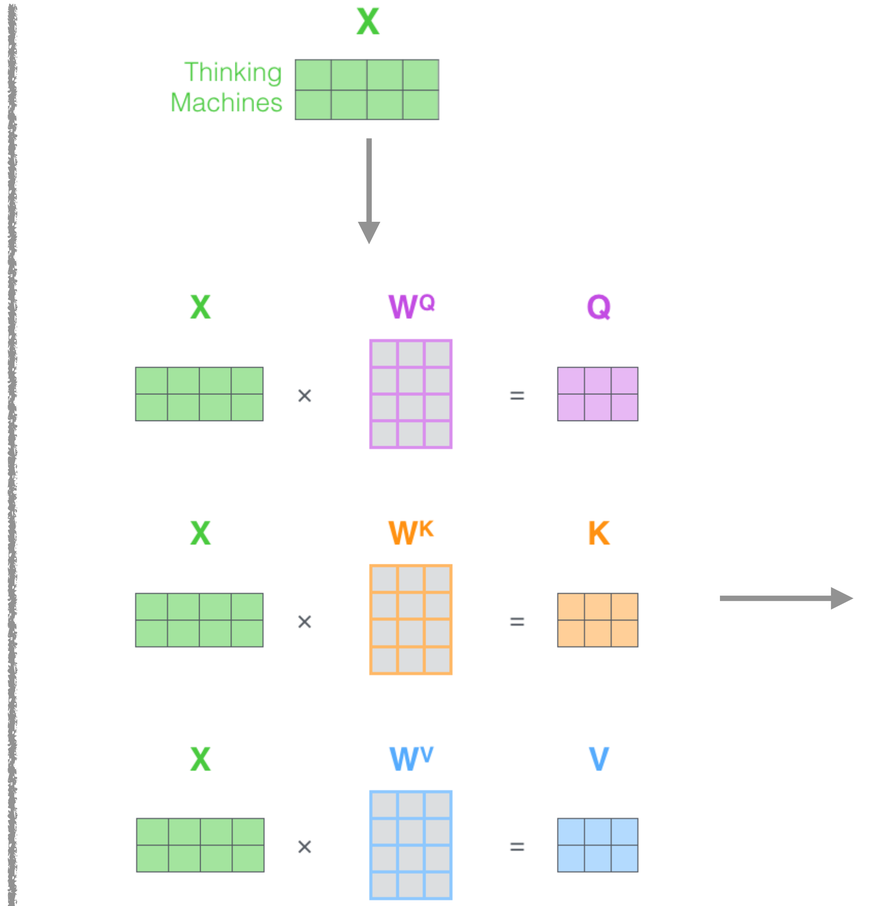
$$\mathbf{z}_i = \sum_{i=1}^{n} \alpha_{i,j} \mathbf{v}_j \in \mathbb{R}^{d_v}$$

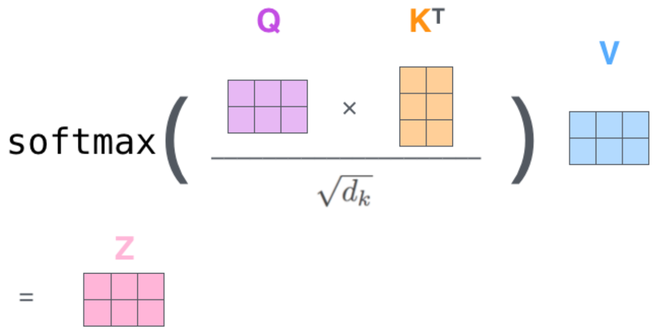
Here,
- $d_q = d_k$
- $d_v = d_2$ (although we'll cover other cases very soon)



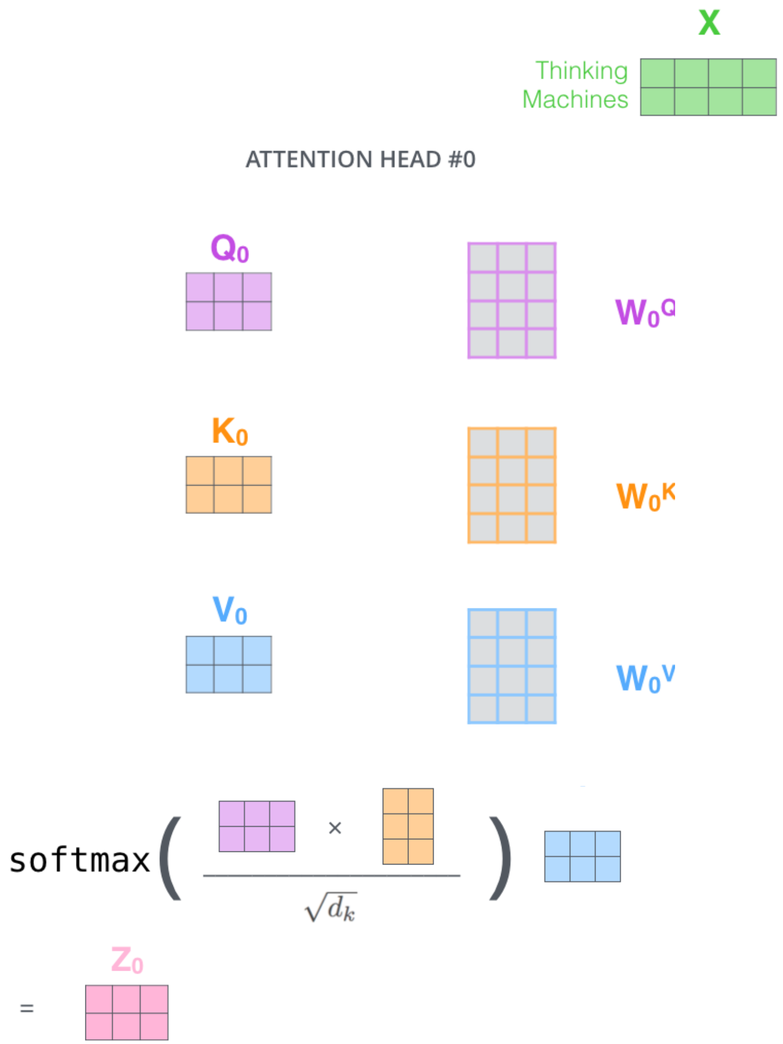
New: Self-attention in matrix notation



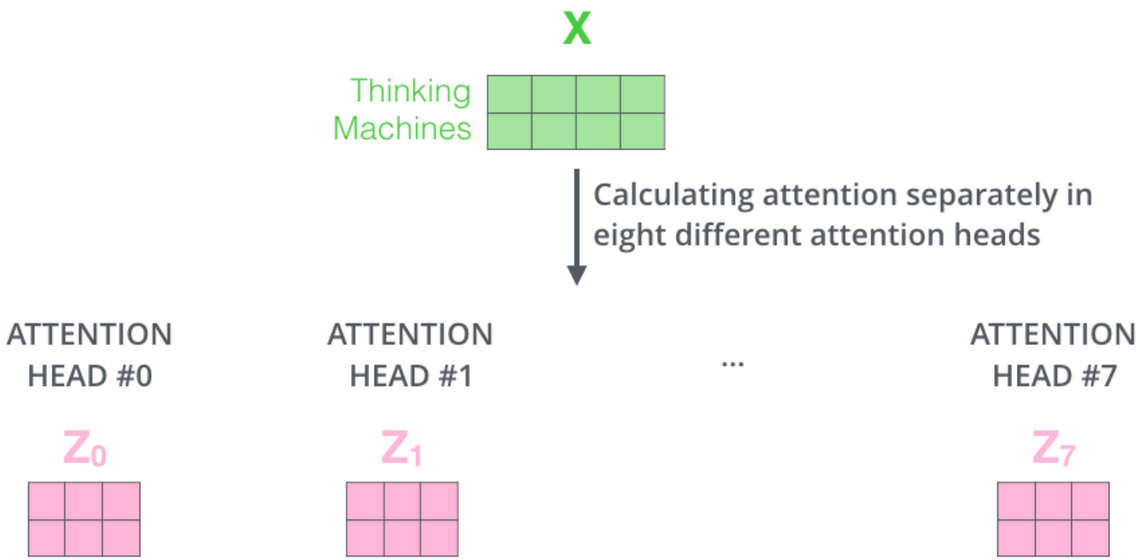




Multi-head self-attention (MHA)



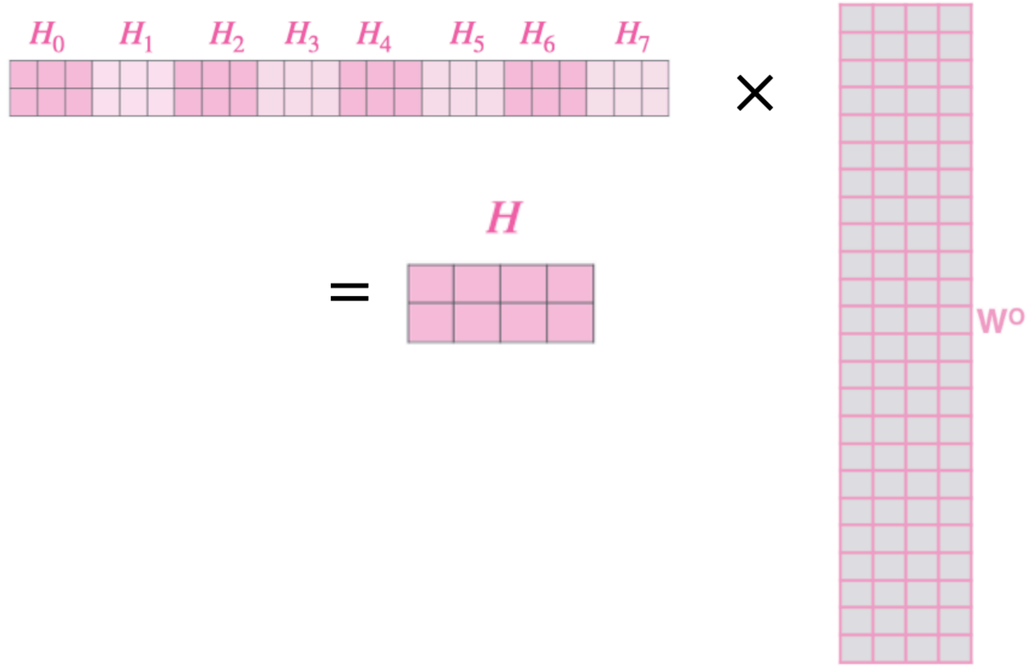
Multi-head self-attention (MHA)



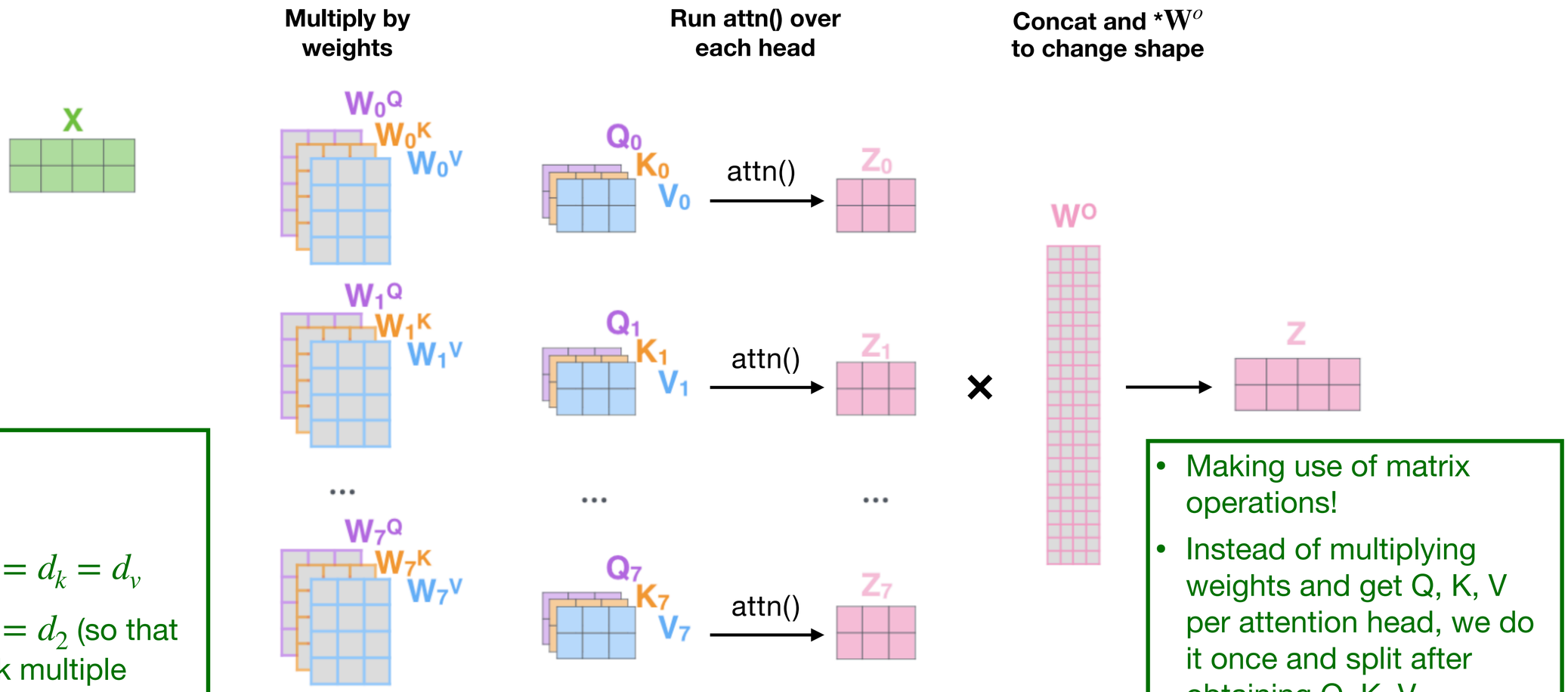
- Intuition: It's better to use multiple attention functions instead of one!
- Each attention function ("head") focuses on different key positions / content.

Multi-head self-attention (MHA)

- Problem: Our final output should be one vector per word, not eight vectors per word. So we need a way to condense these eight down into a single matrix.
- Solution: We concat the matrices then multiply them by an additional weights matrix $\mathbf{W}^O \in \mathbb{R}^{(md_v)\times d_2}$ (m = # of heads)



Multi-head self-attention (MHA): Summary

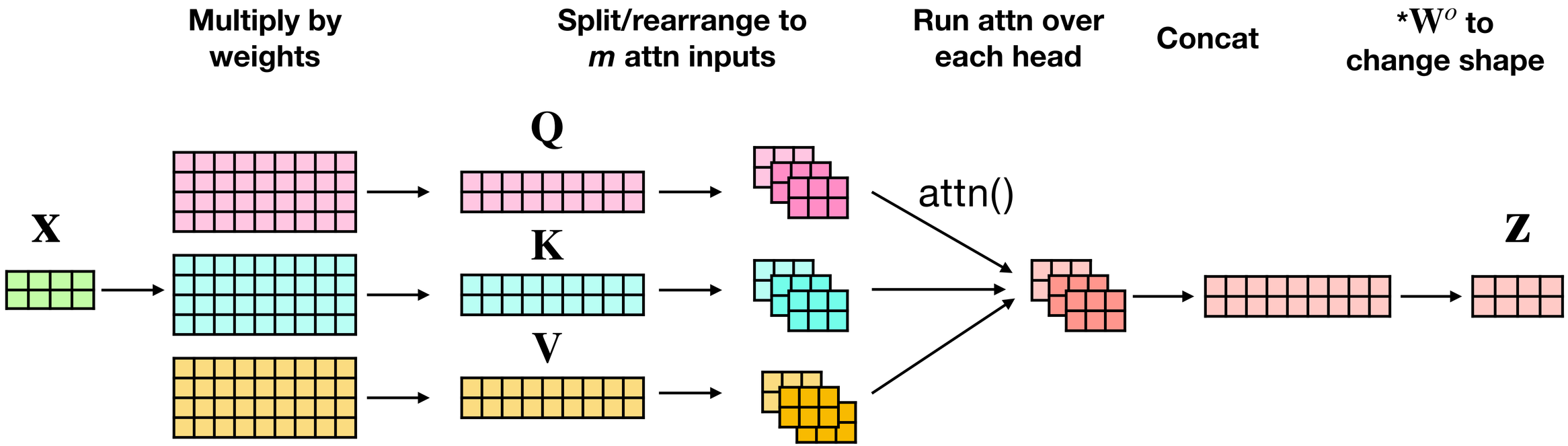


Here,

- $d_q = d_k$
- Typically, $d_q = d_k = d_v$
- Typically, $d_1 = d_2$ (so that we can stack multiple layers of MHA)

- Making use of matrix operations!
- Instead of multiplying weights and get Q, K, V per attention head, we do it once and split after obtaining Q, K, V

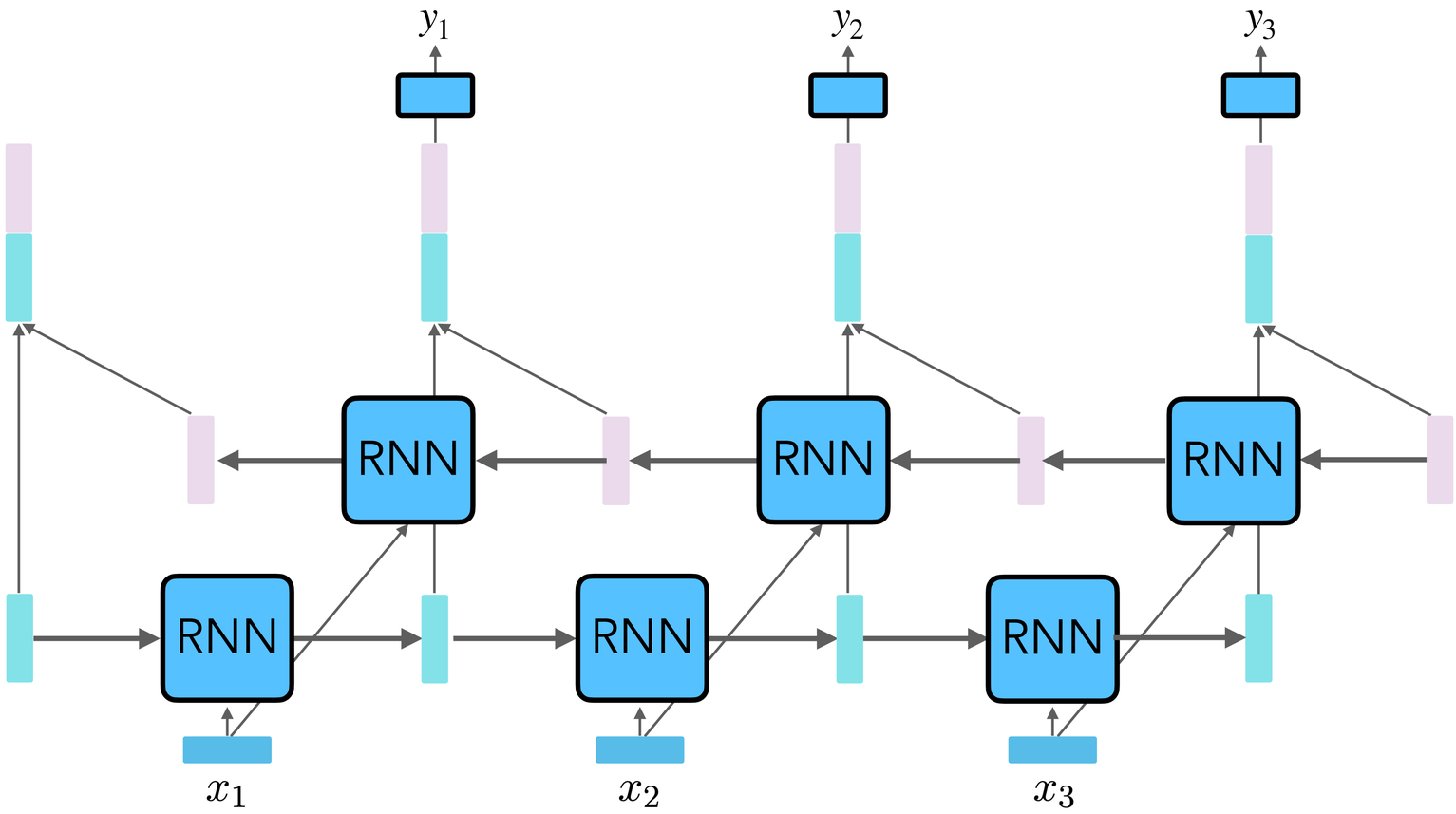
Another visualization: Multi-head self-attention



Masked Multi-head Self-Attention

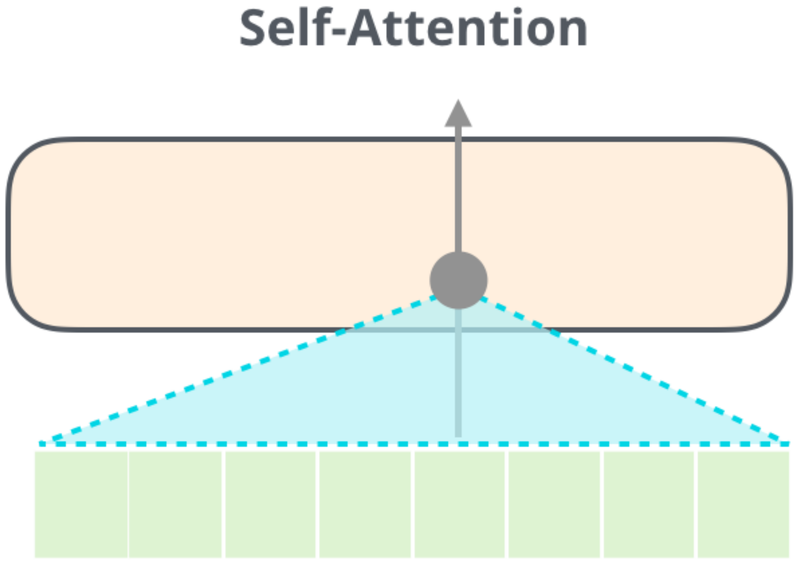
Recap from RNNs

- Bi-directionality is important in language representations → Bi-directional RNNs!
- For text generation, you can't use bidirectionally.



Self-attention

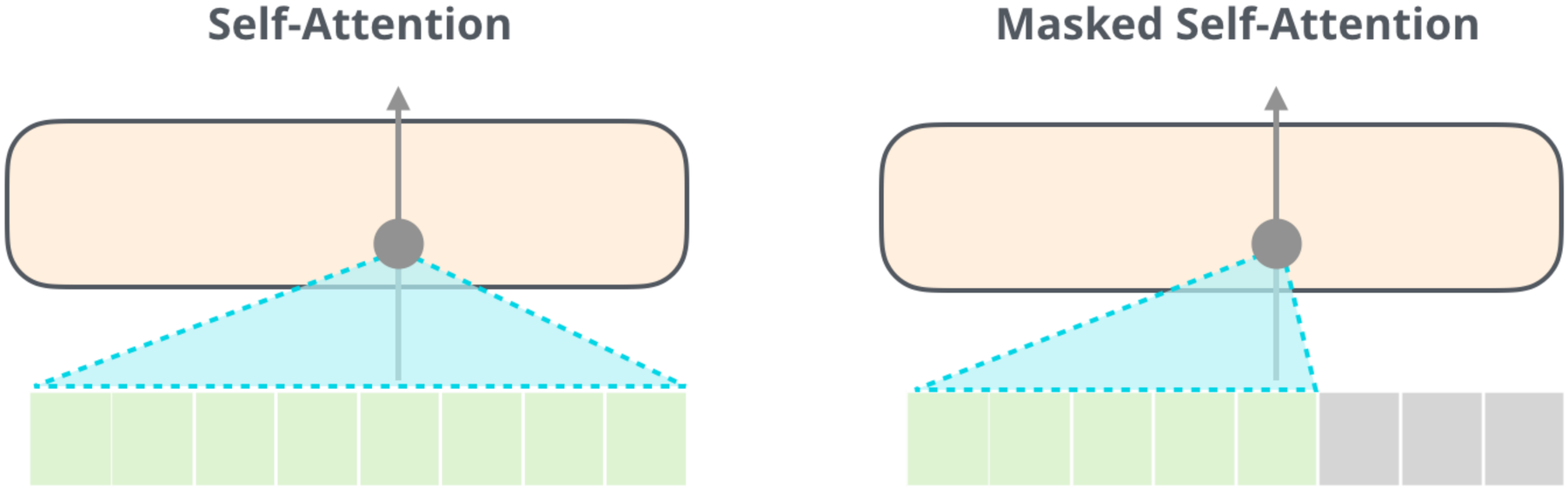
- Default self-attention is bidirectional
- But you can’t see the future text for generation!



https://jalammar.github.io/illustrated-gpt2/

Masked (casual) self-attention

- Default self-attention is bidirectional
- But you can't see the future text for generation!

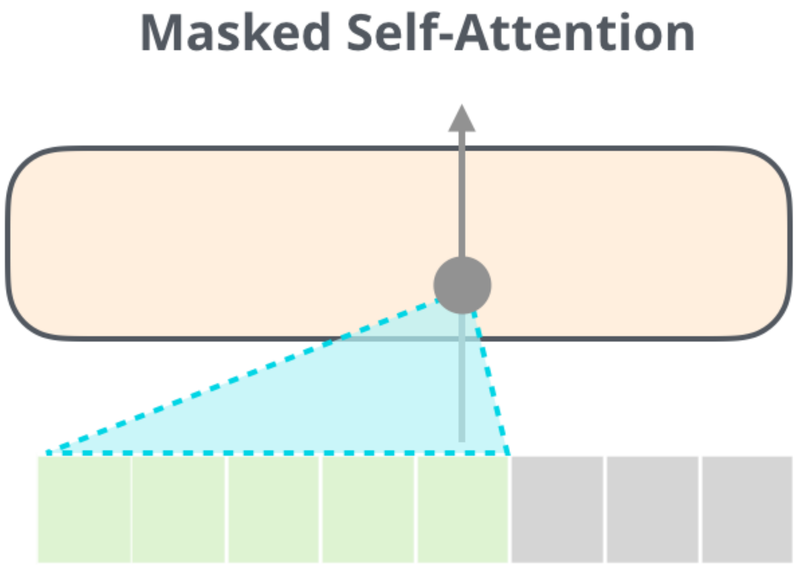


- We should ensure every query $\mathbf{q}_i$ only attends to $\{(\mathbf{k}_j, \mathbf{v}_j)\}, j \leq i$

https://jalammar.github.io/illustrated-gpt2/

Masked (casual) self-attention

- We should ensure every query $\mathbf{q}_i$ only attends to $\{(\mathbf{k}_j, \mathbf{v}_j)\}, j \leq i$



- At inference: Generation is autoregressive, so this is satisfied naturally
- Training
  - The model is given the whole sequence, and we apply loss to each individual token
  - How to ensure attention to previous tokens only ?

Solution: Masking!

https://jalammar.github.io/illustrated-gpt2/

Masked (casual) self-attention

Queries

robot	must	obey	orders

X

Keys

robot	must	obey	orders
robot	must	obey	orders
robot	must	obey	orders
robot	must	obey	orders

=

Scores (before softmax)

0.11	0.00	0.81	0.79
0.19	0.50	0.30	0.48
0.53	0.98	0.95	0.14
0.81	0.86	0.38	0.90

Apply Attention Mask →

Masked Scores (before softmax)

0.11	-inf	-inf	-inf
0.19	0.50	-inf	-inf
0.53	0.98	0.95	-inf
0.81	0.86	0.38	0.90

Softmax (along rows) →

Scores

1	0	0	0
0.48	0.52	0	0
0.31	0.35	0.34	0
0.25	0.26	0.23	0.26

Transformers: Roadmap

1. Attention in Transformers

- From attention to self-attention
- From self-attention to multi-head self-attention
- From multi-head self-attention to masked multi-head self-attention

2. Zooming out: Transformers

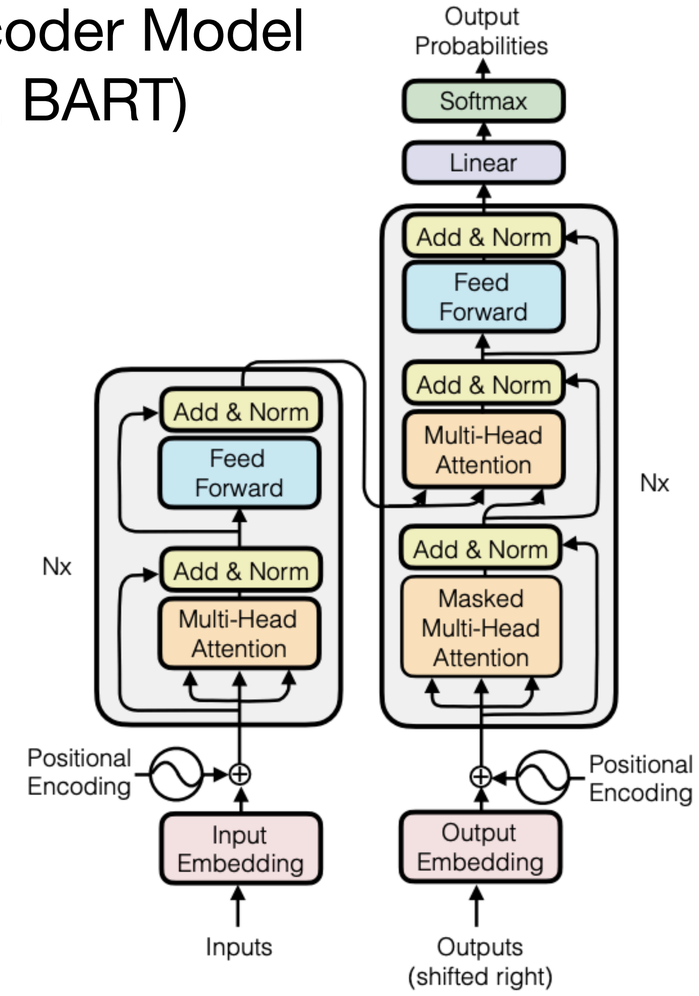
- Embeddings and positional encoding
- Feedforward layers
- Residual connections + layer normalization
- Transformer encoder vs Transformer decoder vs. Transformer encoder-decoder

3. Modern variants of Transformers

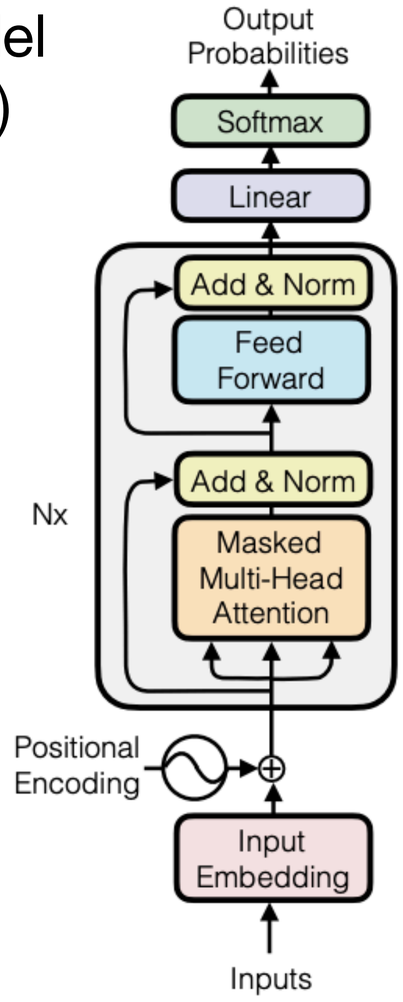
- Embeddings and positional encoding
- Feedforward layers
- Pre-norm vs. Post-norm
- Layernorm vs. RMSNorm
- Activations
- Positional Encodings
- Attention

Recap: Transformer encoder-decoder

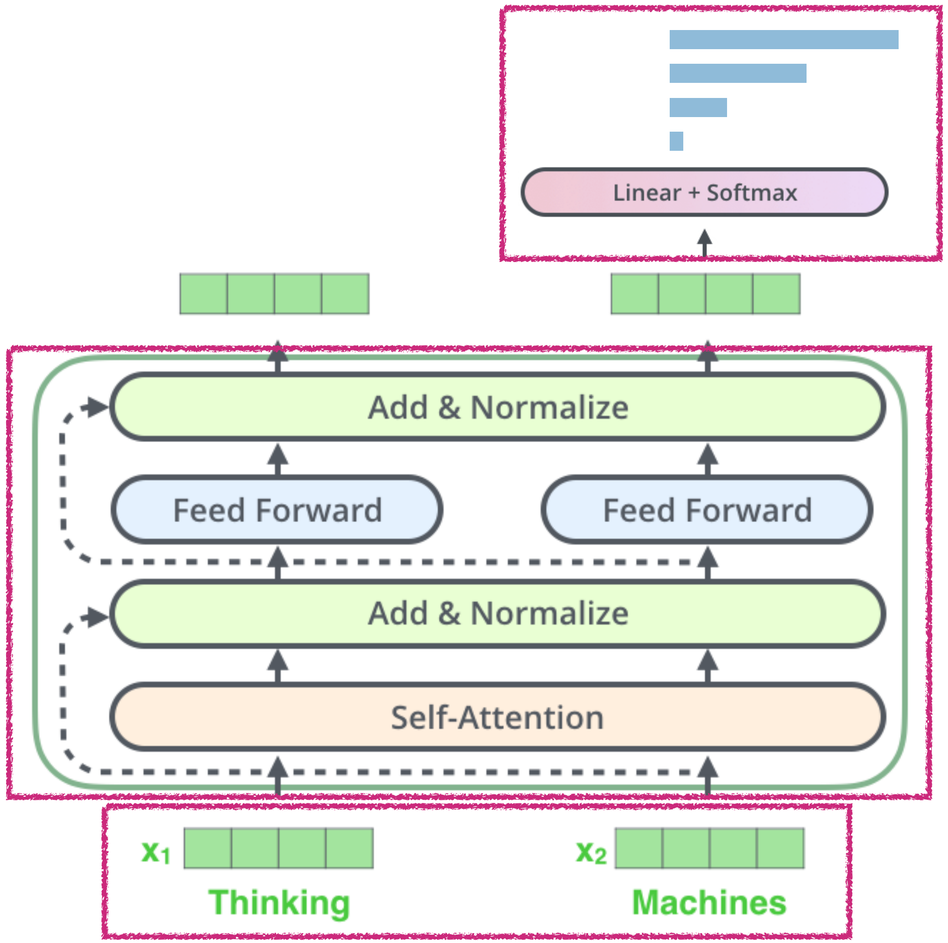
Encoder-Decoder Model
(e.g., T5, BART)



Decoder-only Model
(e.g., GPT, Llama)



Transformers: Basic idea

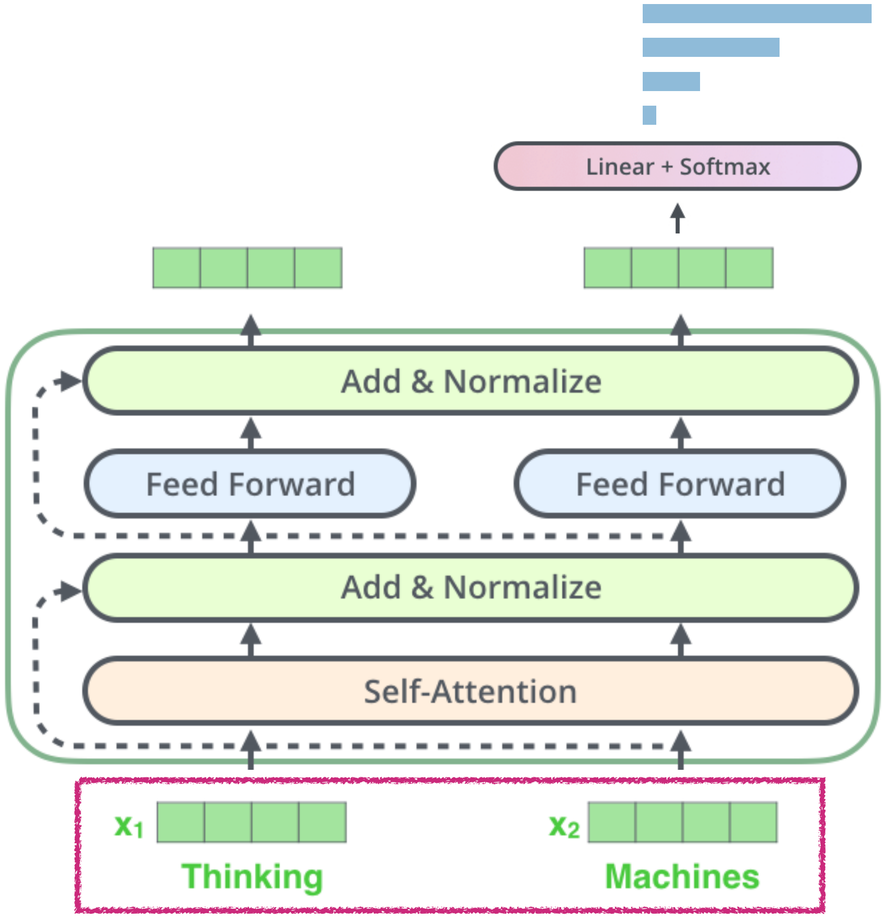


3. Softmax

2. Stacked Transformer Layers

1. Input & Embeddings

Transformers: Basic idea



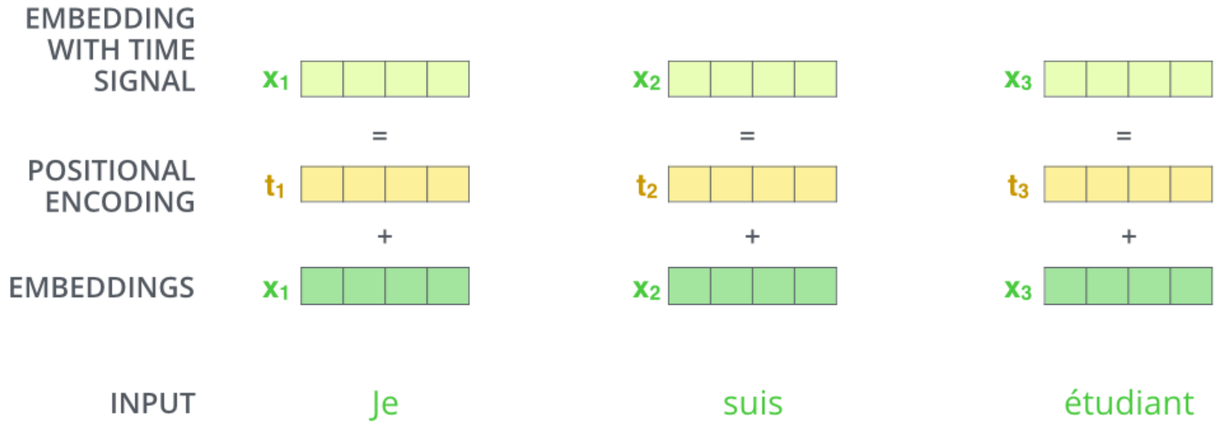
1. Input & Embeddings

Input & Embeddings

Goal: Given a sequence of words (tokens) $x_1, \cdots, x_n$, produce n number of d-dimensional vectors

For a token x at a position p in the sequence,
the input layer outputs

$$\mathbf{E}_x + \mathrm{PE}(p) \in \mathbb{R}^d,$$



$\mathbf{E}_x$: x's word embedding

- $\mathbf{E}_x$ is x's word embedding
- Involving a look-up table $\mathbf{E} \in \mathbb{R}^{V \times d}$
  (V=vocab size)

$\mathrm{PE}(p)$: a positional encoding for the position p

Why positional encoding?

Self-attention processes all tokens in a sequence simultaneously rather than one by one (unlike RNNs)

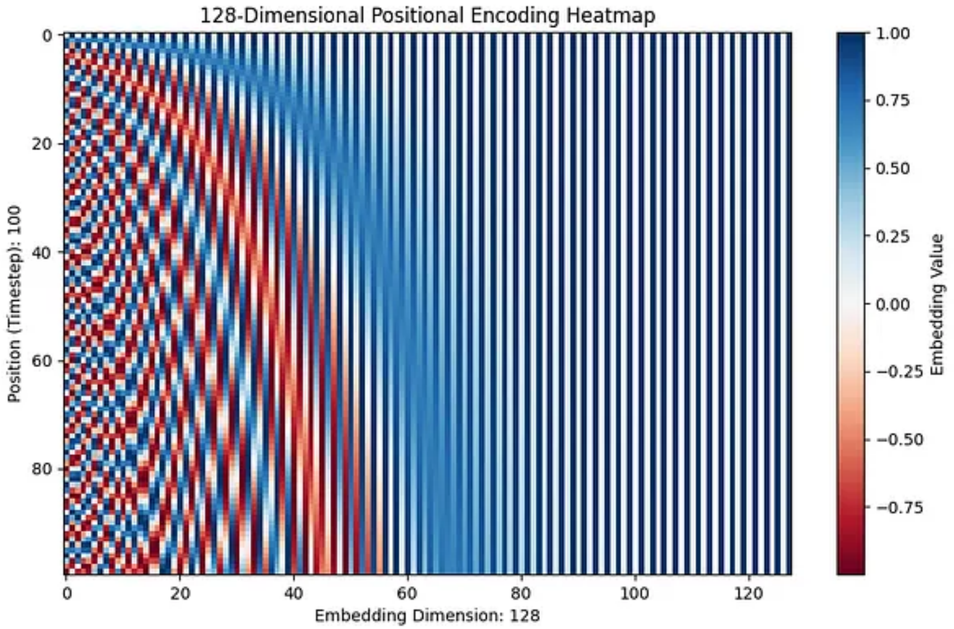
- Extremely fast
- But **“order-blind”**, as mathematically it is **permutation invariant**
  - A: “The dog bit the man.”
  - B: “The man bit the dog.”
  - RNNs know whether “dog” came after “man”
  - Self-Attention doesn’t distinguish them

Sinusoidal Encoding

- Also called sine embeddings
- Add sines and cosines that enable localization
- Properties
  - Every position has a unique signature
  - Normalized range
  - Relative distance is preserved, e.g., the angle between position 1 and 2 is the same as the angle between position 5 and 6

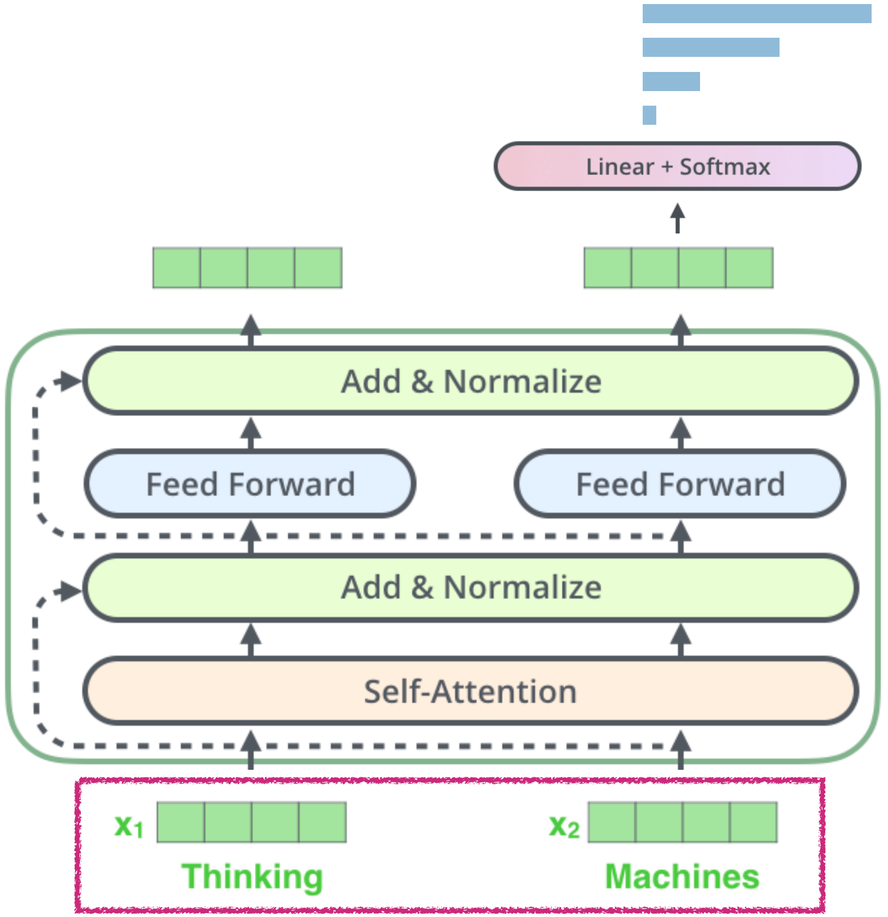
$$\mathrm{PE}(p)_{2i} = \sin\left(\frac{p}{10000^{\frac{2i}{d}}}\right)$$

$$\mathrm{PE}(p)_{2i+1} = \cos\left(\frac{p}{10000^{\frac{2i}{d}}}\right)$$



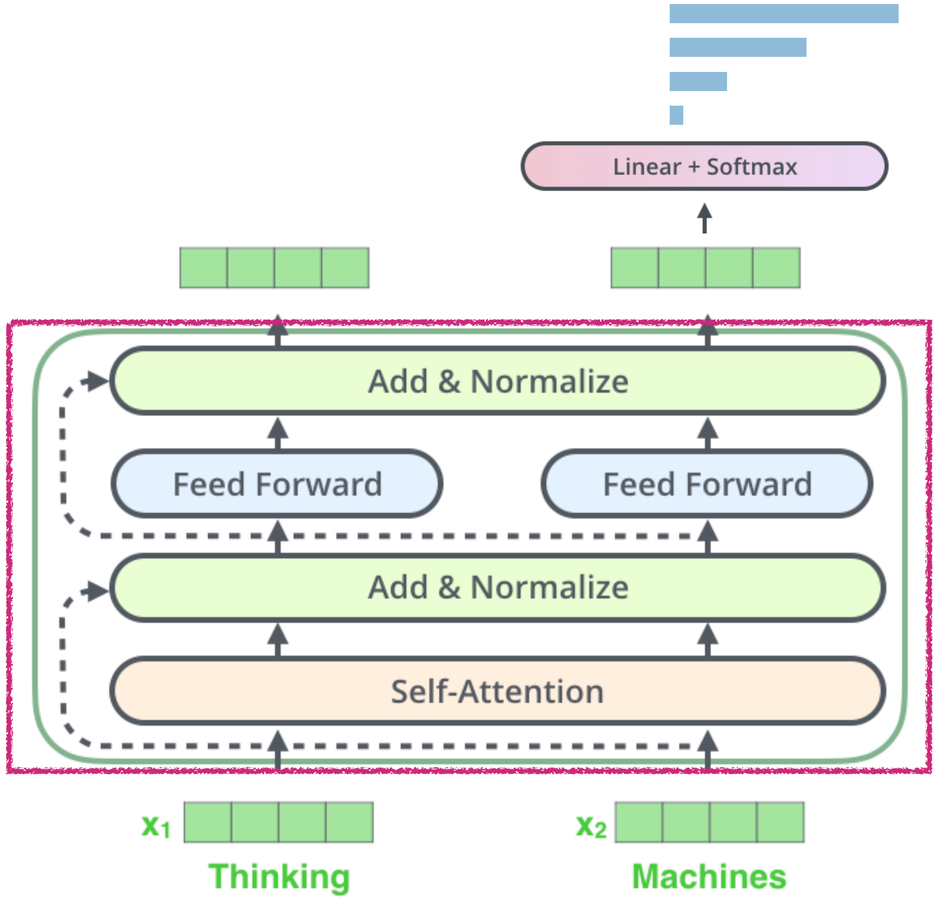
(Later people came up with other positional encodings - more later today)

Transformers: Basic idea



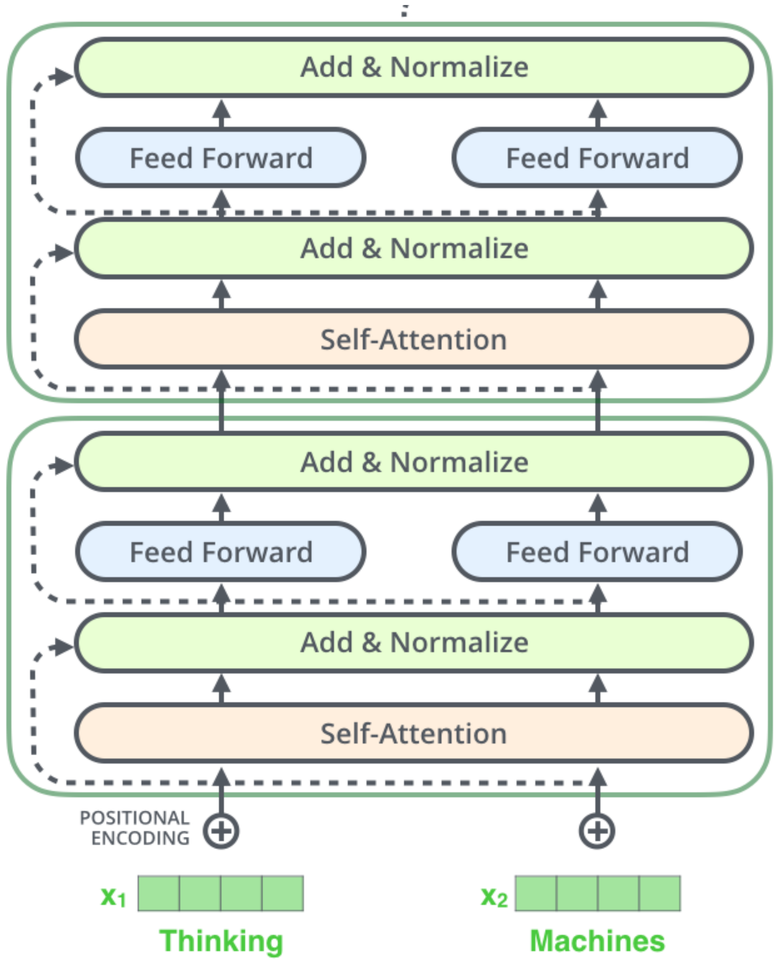
1. Input & Embeddings

Transformers: Basic idea



2. Stacked Transformer Layers

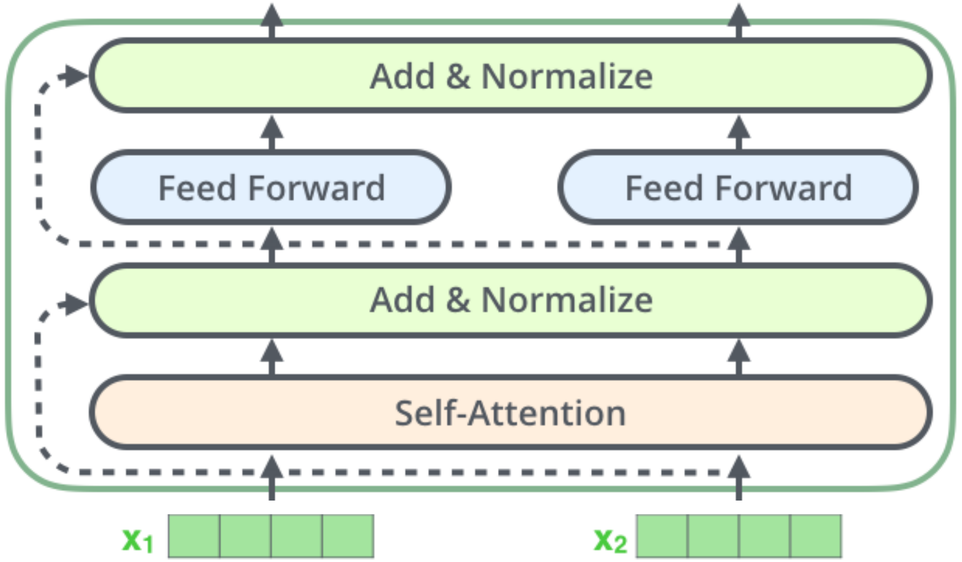
Stacked Transformer layers



Each Transformer maps input vectors $\mathbf{x}_1, \cdots, \mathbf{x}_n \in \mathbb{R}^d$ to output vectors $\mathbf{z}_1, \cdots, \mathbf{z}_n \in \mathbb{R}^d$

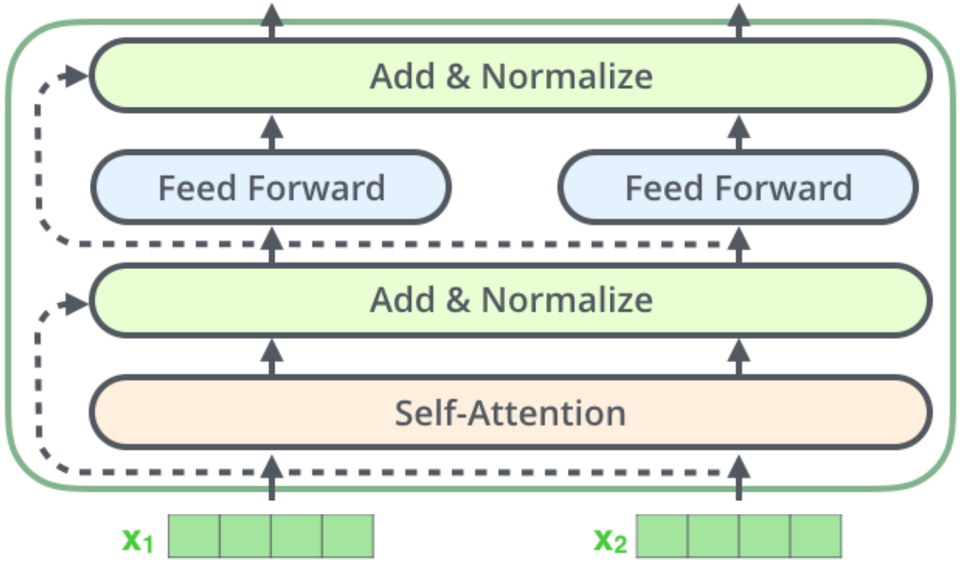
- The first layer takes outputs from the input layer (embeddings + positional encodings) as the inputs
- The second layer takes outputs from the first layer as the inputs
- …
- The (L+1)-th layer takes outputs from the L-th layer as the inputs

A Tranformer layer: Overview



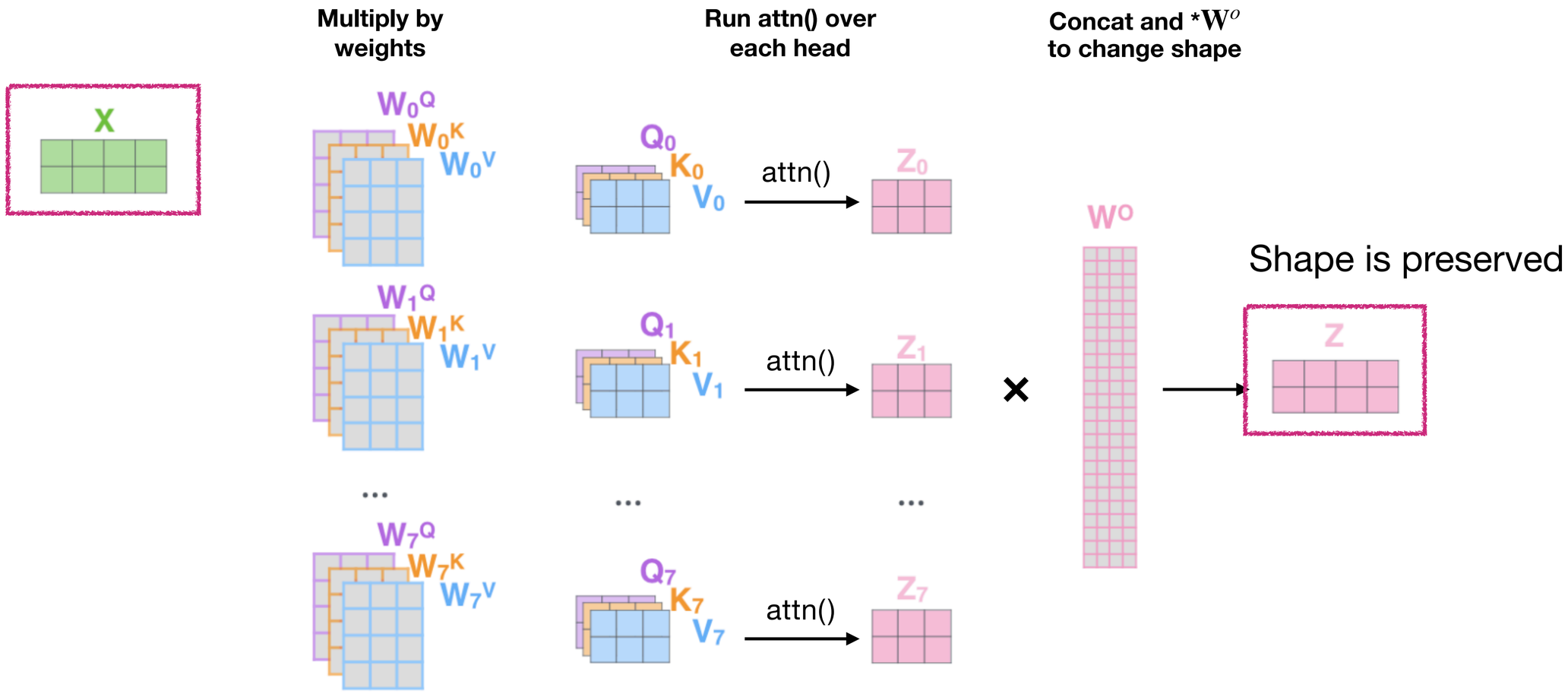
1. (Masked multi-head) Self-attention
2. Add & Normalize
3. Feedforward network (FFN)
4. Add & Normalize

A Tranformer layer: Overview



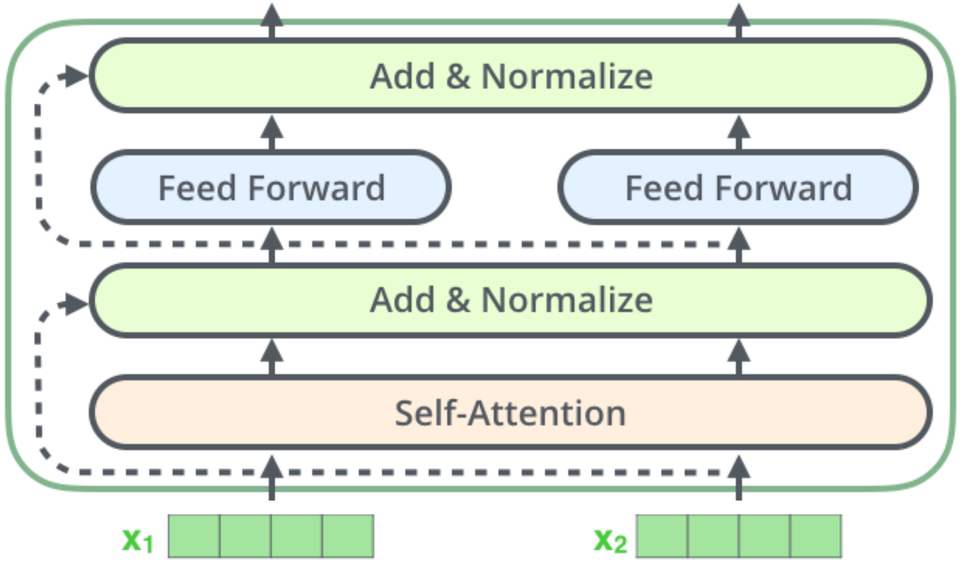
1. (Masked multi-head) Self-attention
2. Add & Normalize
3. Feedforward network (FFN)
4. Add & Normalize

Recap: Multi-head self-attention (MHA)



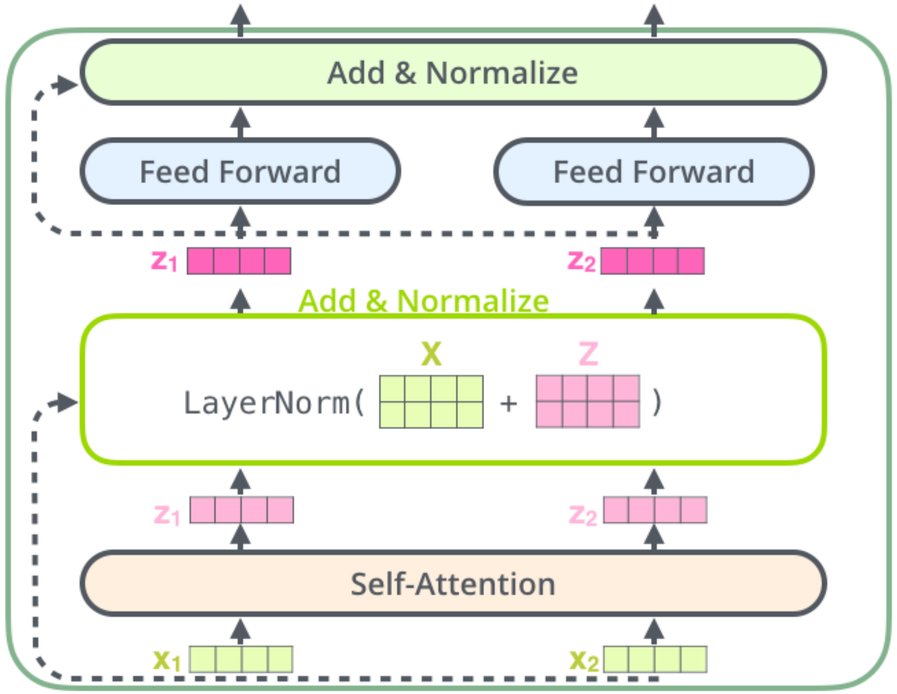
attn(): With masking as this is for the decoder-only!

A Tranformer layer: Overview



1. (Masked multi-head) Self-attention
2. Add & Normalize
3. Feedforward network (FFN)
4. Add & Normalize

A Tranformer layer: Overview



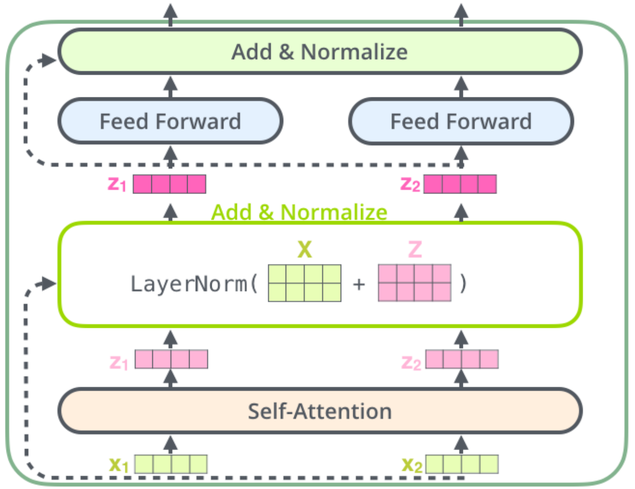
1. (Masked multi-head) Self-attention
2. Add & Normalize
3. Feedforward network (FFN)
4. Add & Normalize

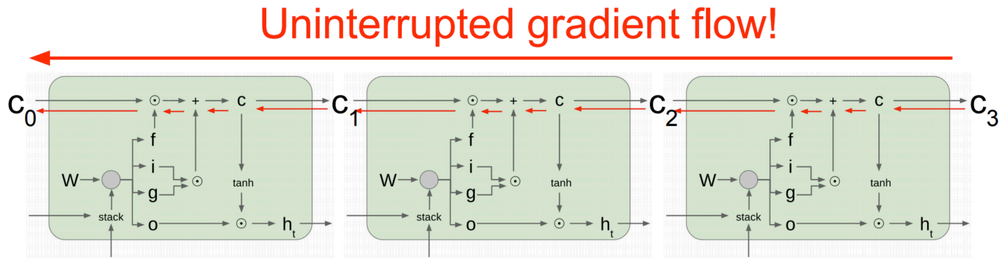
Add: Residual connections

- Add an additive connection between the input and the output

$$\text{Residual}(x, f(x)) = x + f(x)$$

- Intuition:
  - Prevents vanishing gradients (recap LSTMs!)
  - Allows f to learn the difference from the inputs
  - Widely used in Convolutional Neural Networks (CNNs) and adopted in Transformers





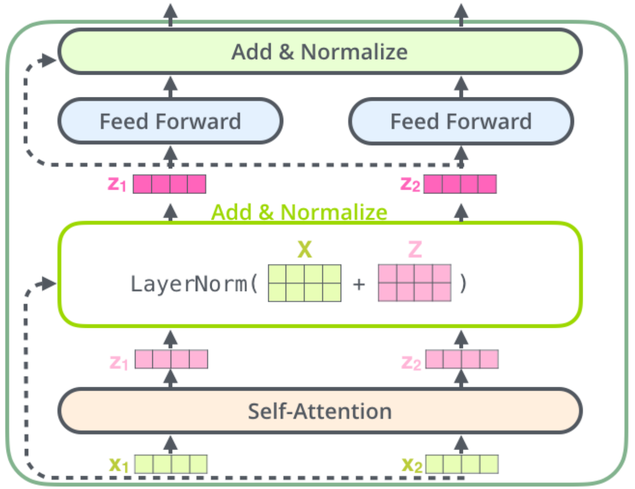
Norm: Normalization

- Normalizes the outputs to be within a consistent range, preventing too much variance in scale of outputs

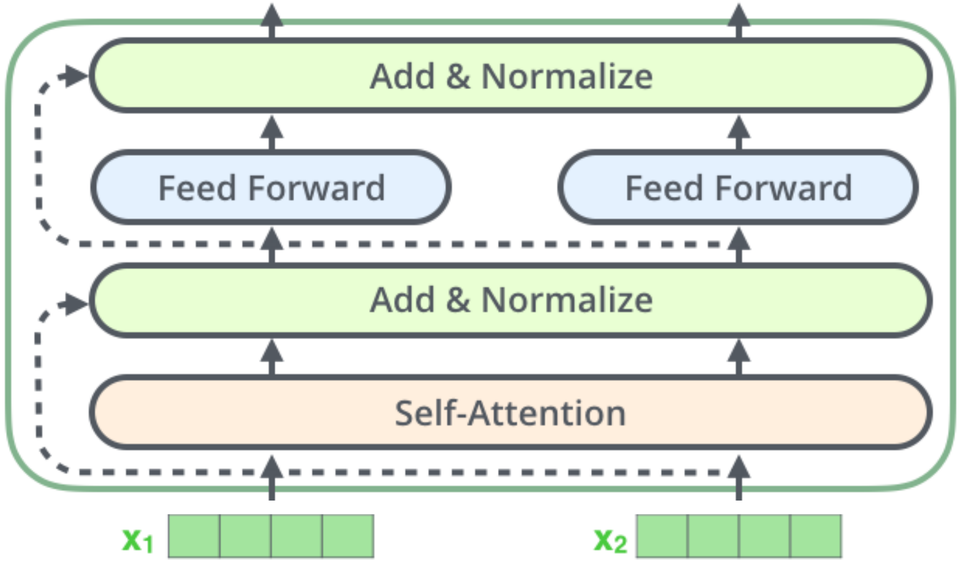
$$y = \frac{x - \mathbb{E}[x]}{\sqrt{\text{Var}[x] + \epsilon}} \times \gamma + \beta$$

- Layer Normalization (LayerNorm; Ba et al. 2016): normalizes the mean and variance

- Apparently the choice of normalization is very important; modern variants use slightly different formulations (more to cover soon)



A Tranformer layer: Overview



1. (Masked multi-head) Self-attention
2. Add & Normalize
3. Feedforward network (FFN)
4. Add & Normalize

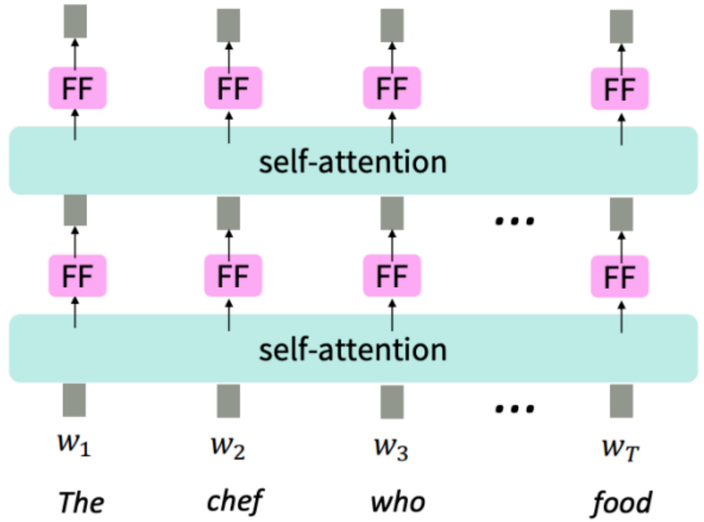
Feed-forward Network (FFN)

- Also called MLP
- There are no element-wise nonlinearities in self-attention; stacking more self-attention layers just re-averages value vectors
- Simple fix: add a feed-forward network to post-process each output vector

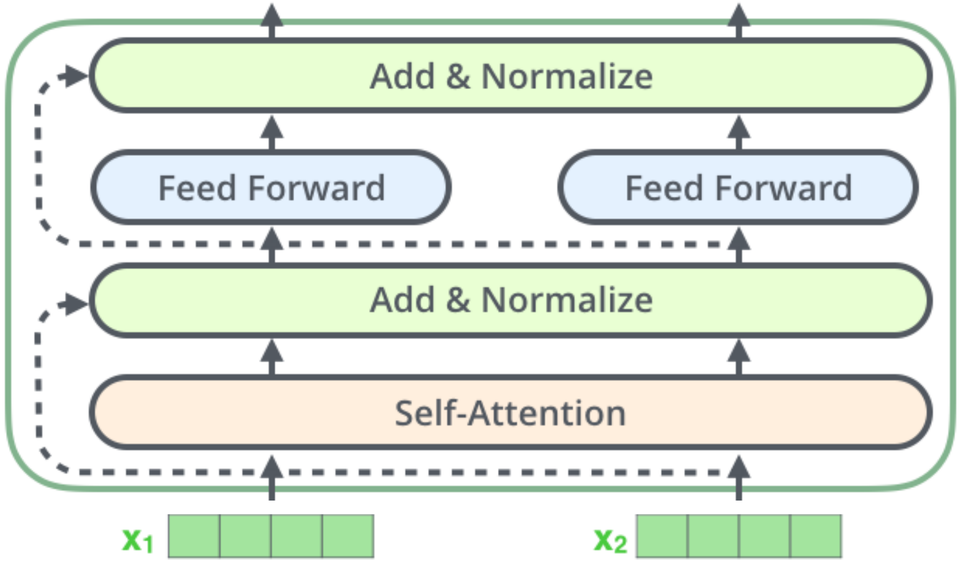
$$\text{FFN}(\mathbf{x}_i) = \text{ReLU}(\mathbf{x}_i\mathbf{W}_1 + \mathbf{b}_1)\mathbf{W}_2 + \mathbf{b}_2$$

$$\mathbf{W}_1 \in \mathbb{R}^{d \times d_{\text{ff}}}, \mathbf{b}_1 \in \mathbb{R}^{d_{\text{ff}}}$$

$$\mathbf{W}_2 \in \mathbb{R}^{d_{\text{ff}} \times d}, \mathbf{b}_2 \in \mathbb{R}^{d}$$



A Tranformer layer: Overview

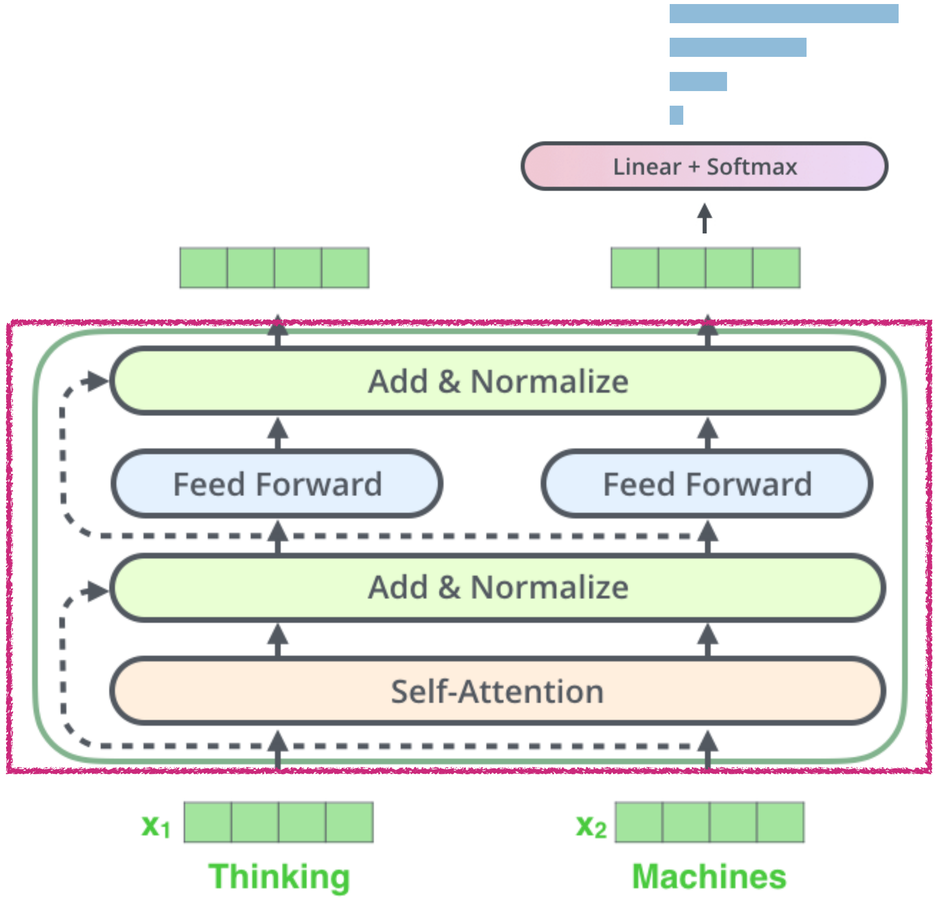


1. (Masked multi-head) Self-attention
2. Add & Normalize
3. Feedforward network (FFN)
4. Add & Normalize

Important reminders:

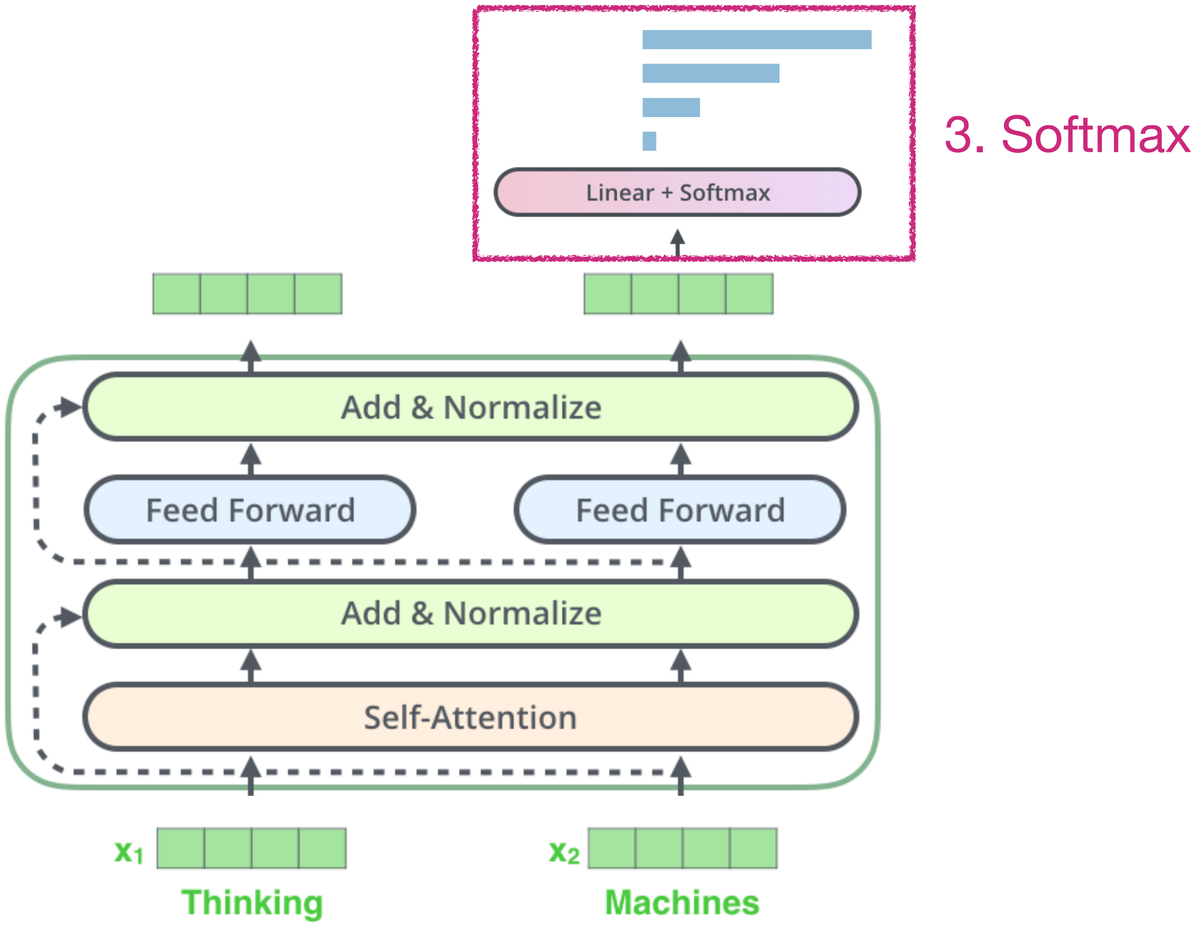
- Add & Normalize and FFN operate independently on each token.
- Self-attention is the only component that models dependencies across tokens.

Transformers: Basic idea



2. Stacked Transformer Layers

Transformers: Basic idea



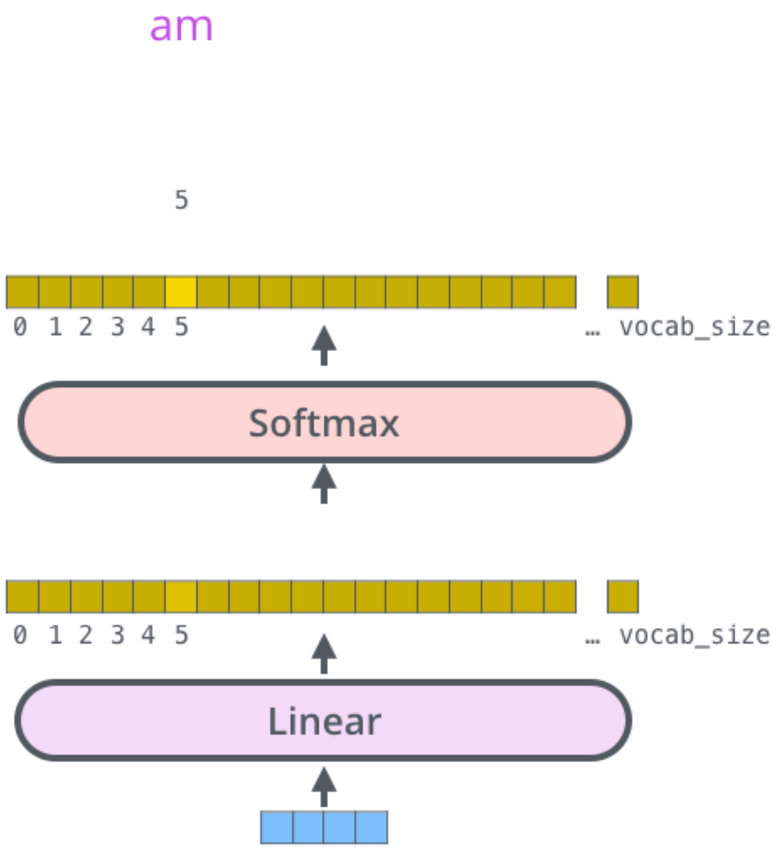
Linear + Softmax

- Recap: our goal is to generate the next word
- The final output should be a probability distribution $\hat{\mathbf{y}} \in \mathbf{R}^V$, where V: vocab size
- First convert the Transformer output $\mathbf{z} \in \mathbf{R}^d$ into the shape we want, then apply softmax to make it probability distribution

$$\hat{\mathbf{y}} = \text{Softmax}\left(\mathbf{W}^O\mathbf{z}\right) \in \mathbb{R}^V, \text{ where } \mathbf{W}^O \in \mathbb{R}^{V \times d}$$

  - If training: Maximize the probability of the ground truth word
  - If inference: Run decoding, e.g., if greedy, take an argmax

Linear + Softmax: Visualization (Inference)



Predicted word (assuming greedy)

$\text{argmax}\left(\hat{\mathbf{y}}\right)$

Probability distribution

$\hat{\mathbf{y}} = \text{Softmax}\left(\mathbf{W}^O\mathbf{z}\right) \in \mathbb{R}^V$

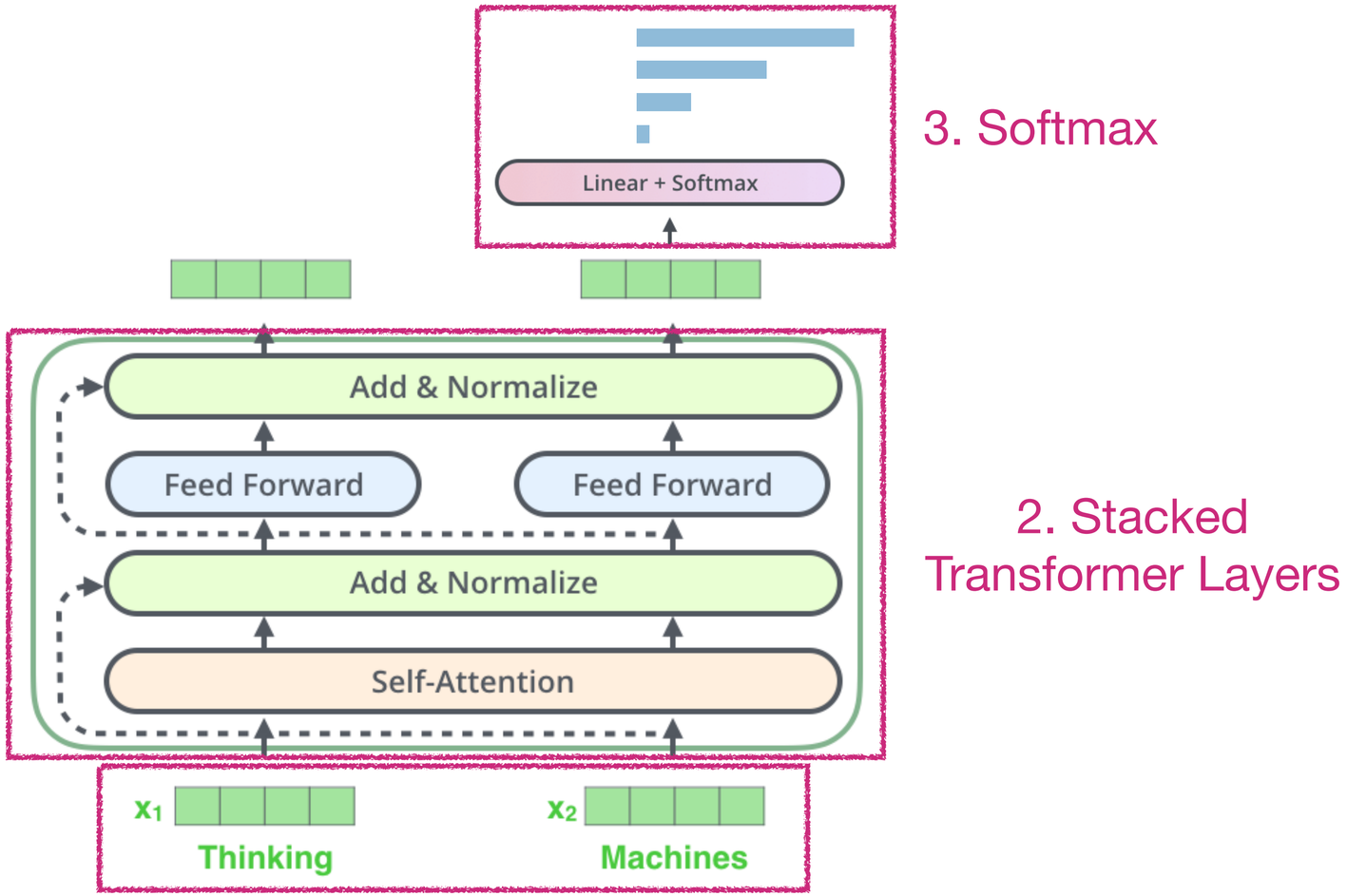
Logits

$\mathbf{W}^O\mathbf{z} \in \mathbb{R}^V$, where $\mathbf{W}^O \in \mathbb{R}^{V \times d}$

Last Transformer layer's output

$\mathbf{z} \in \mathbf{R}^d$

Transformers: Basic idea



3. Softmax

2. Stacked Transformer Layers

1. Input & Embeddings

The Annotated Transformer

The Annotated Transformer

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

- *v2022: Austin Huang, Suraj Subramanian, Jonathan Sum, Khalid Almubarak, and Stella Biderman.*
- *Original: Sasha Rush.*

Quick quiz 1

Which of the following statements is correct?

=Floating-point operations
(Remember this term!)

(a) Transformers have less FLOPs compared to LSTMs.

(b) Transformers have less parameters compared to LSTMs.

(c) Transformers are easier to parallelize compared to LSTMs.

(d) Transformers generalize to longer sequences better than LSTMs.

Depends on hyperparams; not properties of architectures.
(In general, Transformers scale better than LSTMs)

Transformers cannot generalize beyond a pre-defined input length (so-called max_seq_len)

(c) is correct!

Transformers: Pros and cons

- Easier to capture long-range dependencies: we draw attention between every pair of words!
- Easier to parallelize

- Cannot generalize beyond max_seq_len defined before training
  - Due to positional encodings
- Quadratic computation in self-attention
  - Can become very slow and memory-expensive when the sequence length is large

Quick quiz 2

Which one has more parameters (typically), MHA vs. FFN?

(a) MHA

(b) FFN

(c) Equal

(b) is correct!

Quick quiz 3

What is the ratio of FLOPS (floating-point operations) look like for a 175B model (GPT-3)?

MHA : FFN : others

(a) 35% : 44% : 21%
(b) 24% : 65% : 11%
(c) 17% : 80% : 3%

(c) is correct!

FLOP scaling in Transformers

1	description	FLOPs / update	% FLOPS MHA	% FLOPS FFN	% FLOPS attn	% FLOPS logit
8	OPT setups					
9	760M	4.3E+15	35%	44%	14.8%	5.8%
10	1.3B	1.3E+16	32%	51%	12.7%	5.0%
11	2.7B	2.5E+16	29%	56%	11.2%	3.3%
12	6.7B	1.1E+17	24%	65%	8.1%	2.4%
13	13B	4.1E+17	22%	69%	6.9%	1.6%
14	30B	9.0E+17	20%	74%	5.3%	1.0%
15	66B	9.5E+17	18%	77%	4.3%	0.6%
16	175B	2.4E+18	17%	80%	3.3%	0.3%

Transformers in code (1/2)

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

    def __init__(self, d_model, d_ff, num_heads, dropout):
        super().__init__()
        self.mha = MultiHeadAttention(num_heads, d_model, dropout)
        self.ff = nn.Linear(d_model, d_model)
        self.norm1 = nn.LayerNorm(d_model)
        self.norm2 = nn.LayerNorm(d_model)

    def forward(self, x, mask):
        # Multi-head attention + residual connection
        x = x + self.mha(x, x, x, mask)
        # Post-norm
        x = self.norm1(x)
        # Feed-forward + residual connection
        x = x + self.ff(x)
        # Post-norm
        x = self.norm2(x)
        return x
```

Transformers in code (2/2)

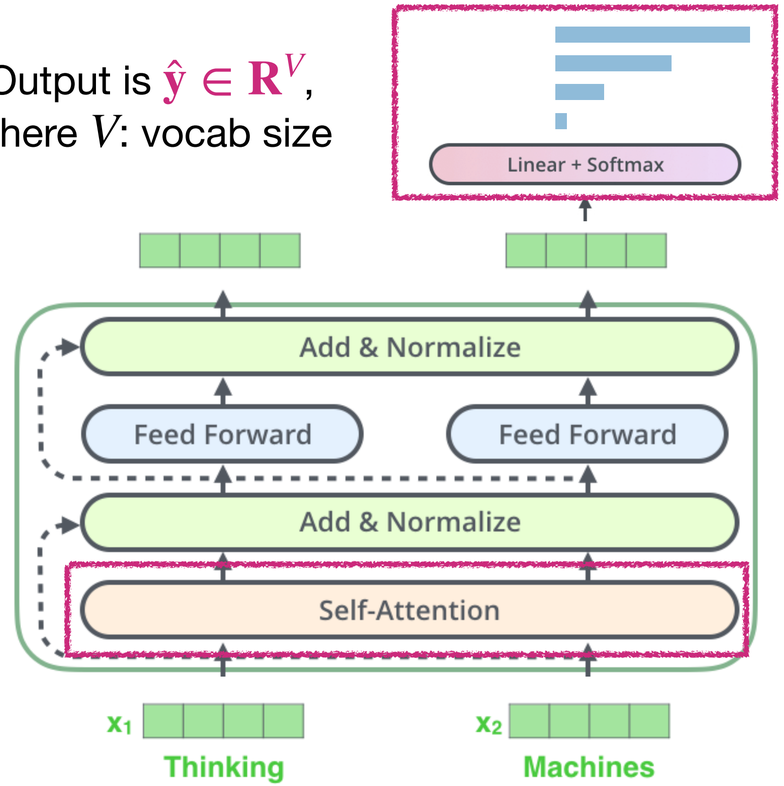
```
class Transformers(nn.Module):

    def __init__(self, vocab_size, num_layers, d_model, d_ff, num_heads, dropout):
        super().__init__()
        self.vocab_size = vocab_size
        self.num_layers = num_layers
        self.d_model = d_model
        self.d_ff = d_ff
        self.num_heads = num_heads
        self.dropout = dropout

        self.embed = nn.Embedding(vocab_size, d_model)
        self.pos_encoder = PositionalEncoding(d_model, dropout)
        self.norm = nn.LayerNorm(d_model)
        self.output = nn.Linear(d_model, vocab_size, bias=False)

        self.input_embed.weight.data = self.output.weight.data

        self.transformer_blocks = nn.ModuleList([
            TransformerBlock(d_model, d_ff, num_heads, dropout) for _ in range(num_layers)
        ])

    def __call__(self, src, src_mask):
        x = self.embed(src)
        pos = torch.arange(x.size(0), device=x.device)
        x = x + self.pos_encoder(pos)
        for block in self.transformer_blocks:
            x = block(x, src_mask)

        x = self.output(self.norm(x))
        return nn.Softmax(dim=-1)(x)
```

Transformer architectures

Architecture	Use cases	Pretrained models examples (more next lecture)
Transformer Encoder	Classification or sentence-level representation	BERT, RoBERTa, ELECTRA
Transformers Encoder-Decoder	Sequence-to-sequence (e.g., machine translation)	T5, BART
Transformers Decoder	Language modeling	GPT-2, GPT-3, Llama

Decoder-only vs. Encoder-only

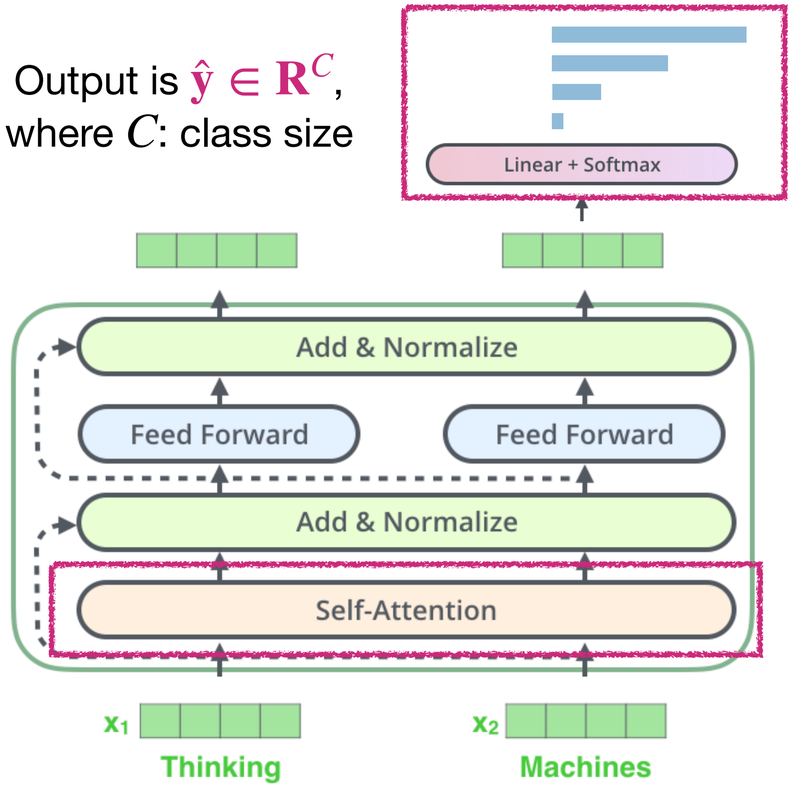
Output is $\hat{\mathbf{y}} \in \mathbf{R}^V$,
where V: vocab size



Masked MHA

Decoder-only

Output is $\hat{\mathbf{y}} \in \mathbf{R}^C$,
where C: class size

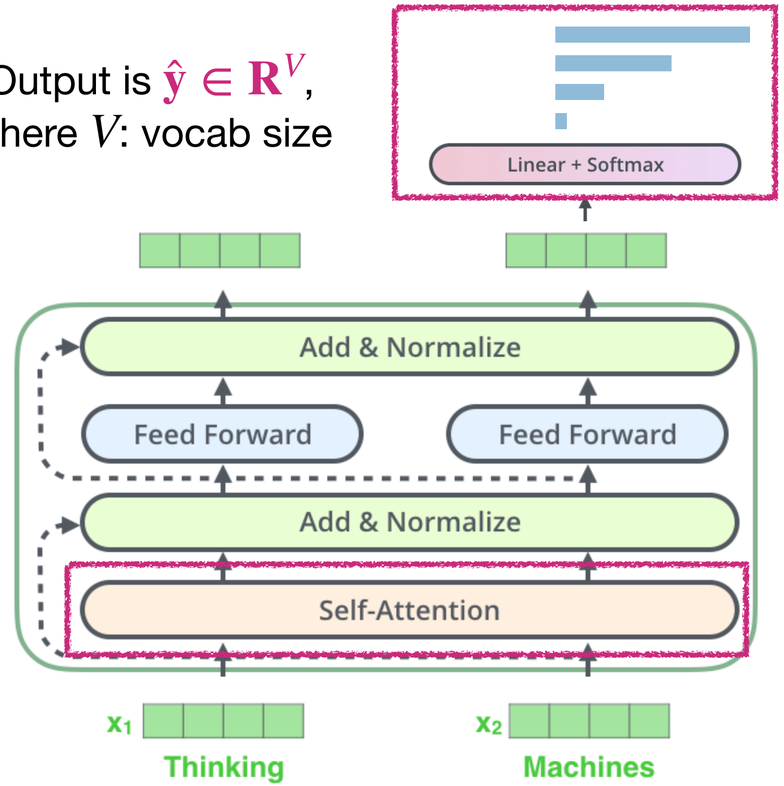


MHA (without masking)

Encoder-only
(for classification)

Decoder-only vs. Encoder-only

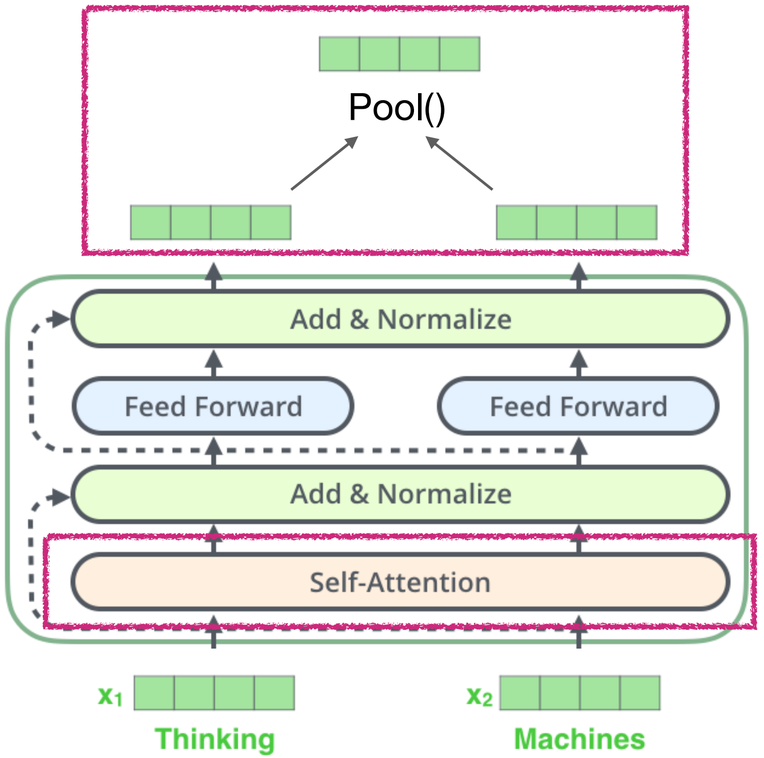
Output is $\hat{\mathbf{y}} \in \mathbf{R}^V$,
where V: vocab size



Masked MHA

Decoder-only

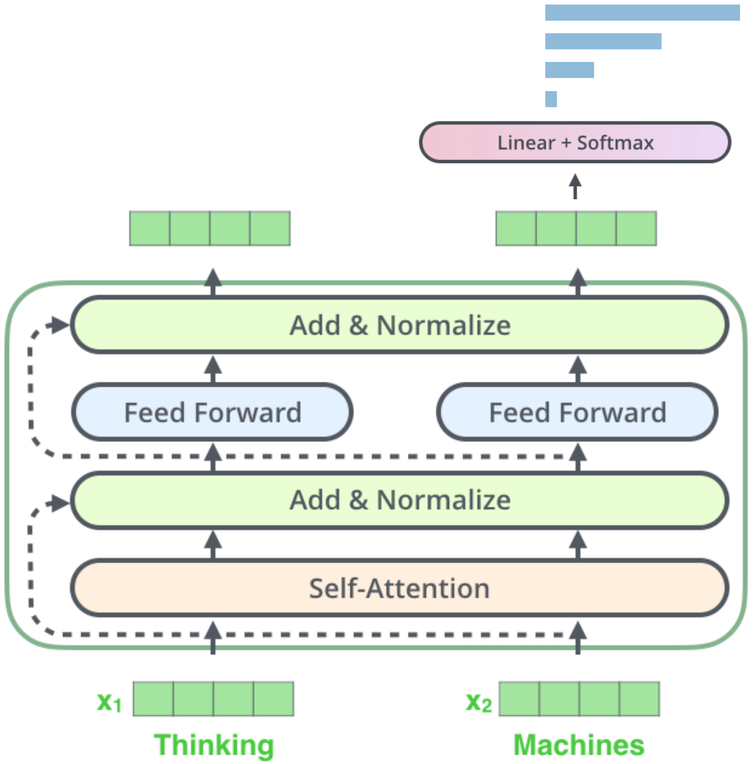
Output is a d-dimensional vector representing the whole input
(no softmax)



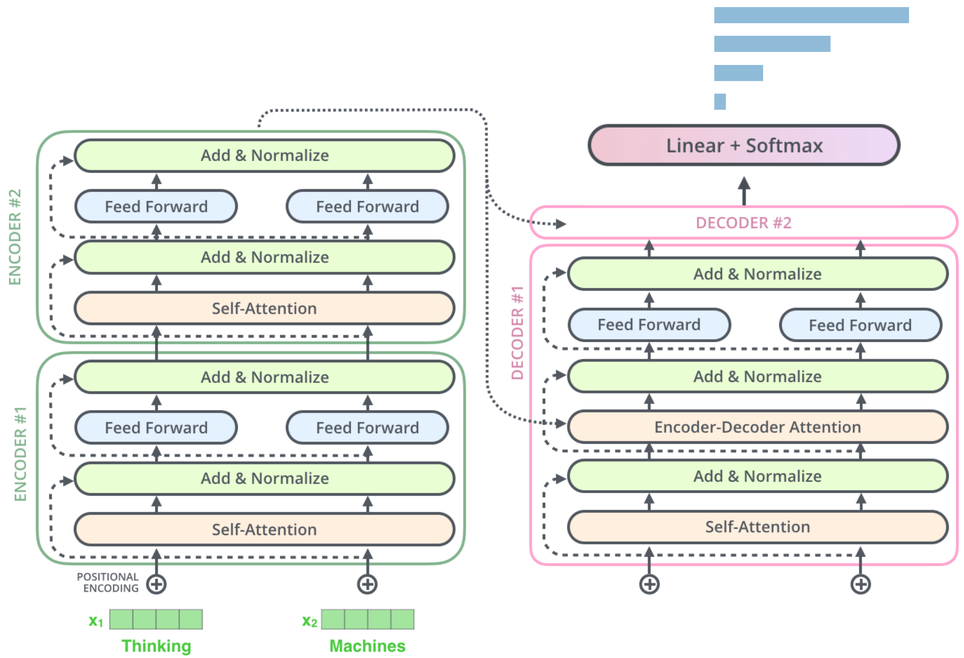
MHA (without masking)

Encoder-only
(for sentence representation)

Decoder-only vs. Encoder-decoder



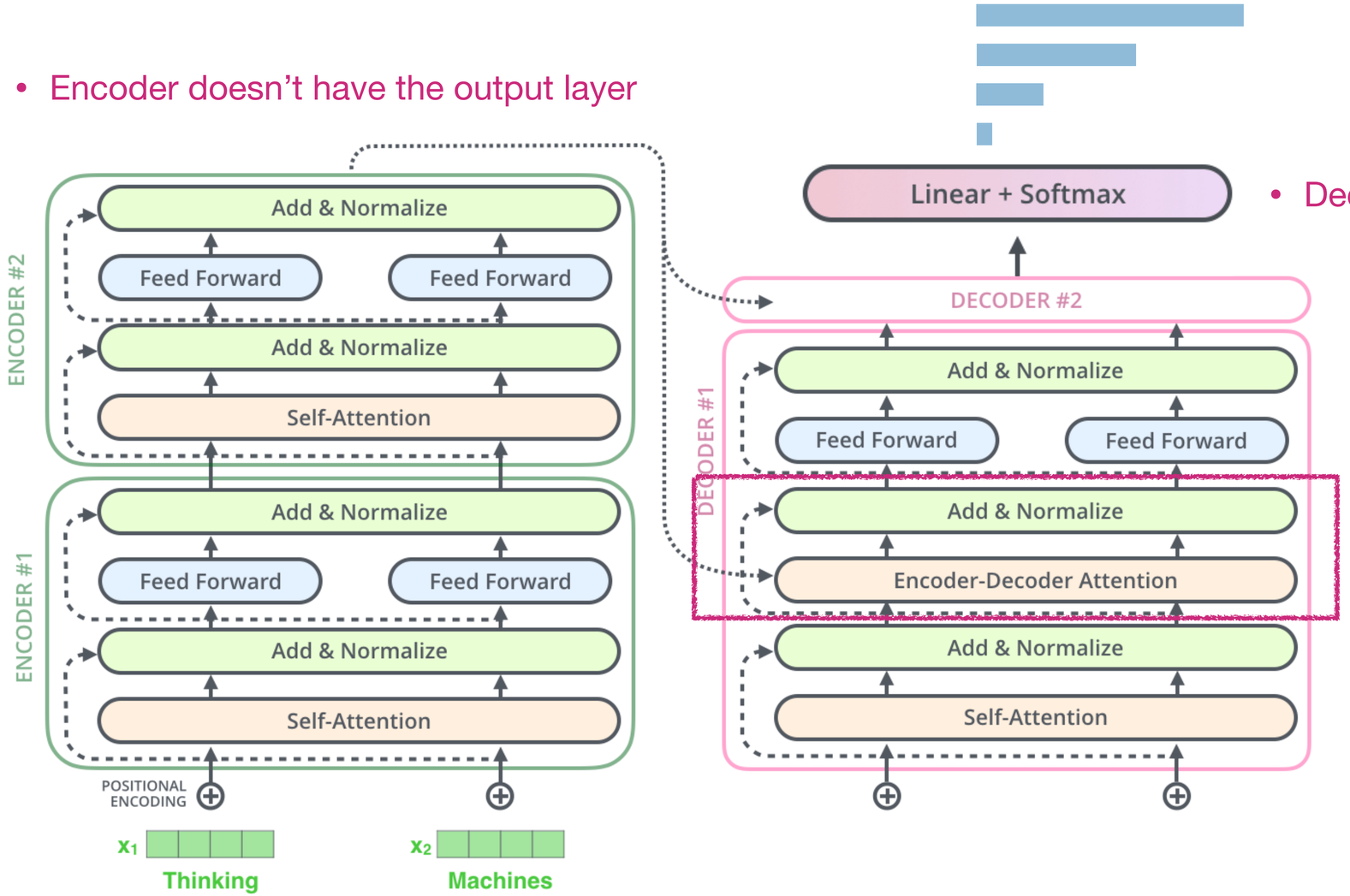
Decoder-only



Encoder-decoder

Encoder-decoder Transformers

- Encoder doesn't have the output layer

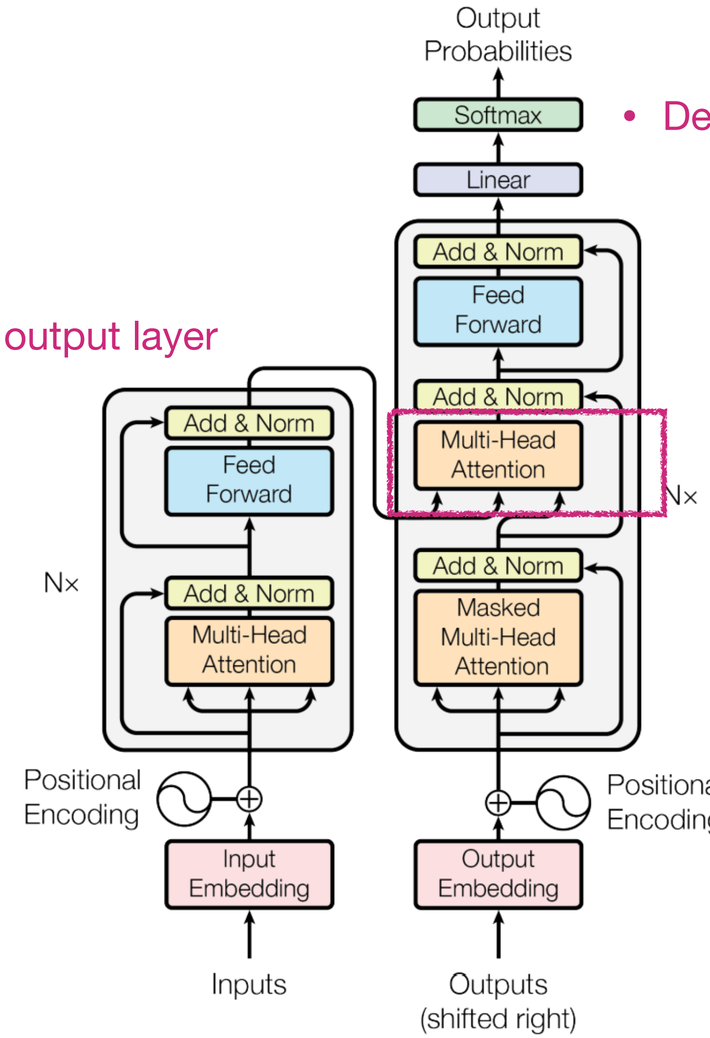


- Decoder output stays the same

- The only new component
- The only layer where the input & output sequences interact

Encoder-decoder Transformers

- Encoder doesn't have the output layer

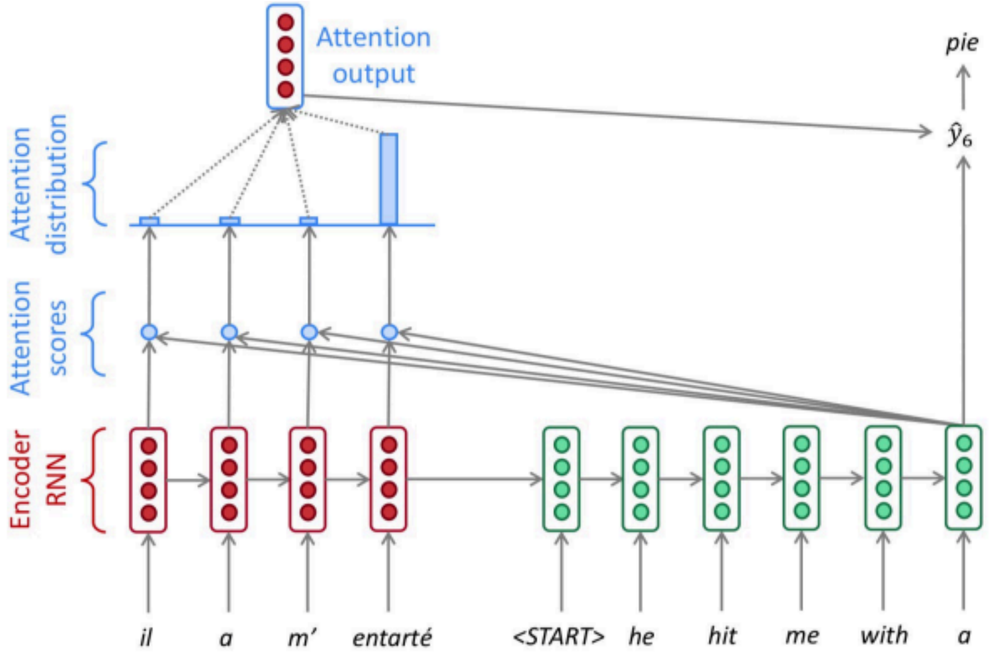


- Decoder output stays the same
- The only new component
- The only layer where the input & output sequences interact

Encoder-decoder cross attention

What is encoder-decoder cross attention?

- Cross-attention between the input and the target sequences
- Similar to attention we saw in the previous lecture
  - Attention in seq2seq: Between the input and the target
  - Self-attention: Within the same sequence



Encoder-decoder cross attention

Self-attention:

$$\mathbf{q}_i = \mathbf{x}_i \mathbf{W}^Q, \mathbf{k}_i = \mathbf{x}_i \mathbf{W}^K, \mathbf{v}_i = \mathbf{x}_i \mathbf{W}^V$$

$$e_{i,j} = \frac{\mathbf{q}_i \cdot \mathbf{k}_j}{\sqrt{d_k}}, \forall j = 1, \ldots, n$$

$$\alpha_i = \text{softmax}(\mathbf{e}_i)$$

$$\mathbf{h}_i = \sum_{j=1}^{n} \alpha_{i,j} \mathbf{v}_j$$

Cross-attention:

(always from the top layer)

$\tilde{\mathbf{x}}_1, \ldots, \tilde{\mathbf{x}}_m$: hidden states from encoder

$\mathbf{x}_1, \ldots, \mathbf{x}_n$: hidden states from decoder

Encoder-decoder cross attention

Self-attention:

$$\mathbf{q}_i = \mathbf{x}_i\mathbf{W}^Q, \mathbf{k}_i = \mathbf{x}_i\mathbf{W}^K, \mathbf{v}_i = \mathbf{x}_i\mathbf{W}^V$$

$$e_{i,j} = \frac{\mathbf{q}_i \cdot \mathbf{k}_j}{\sqrt{d_k}}, \forall j = 1, \ldots, n$$

$$\alpha_i = \text{softmax}(\mathbf{e}_i)$$

$$\mathbf{h}_i = \sum_{j=1}^{n} \alpha_{i,j}\mathbf{v}_j$$

Cross-attention: (always from the top layer)

$\tilde{\mathbf{x}}_1, \ldots, \tilde{\mathbf{x}}_m$: hidden states from encoder

$\mathbf{x}_1, \ldots, \mathbf{x}_n$: hidden states from decoder

$$\mathbf{q}_i = \mathbf{x}_i\mathbf{W}^Q \quad i = 1,2,\ldots,n$$

$$\mathbf{k}_j = \tilde{\mathbf{x}}_j\mathbf{W}^K, \mathbf{v}_j = \tilde{\mathbf{x}}_j\mathbf{W}^V \quad \forall j = 1,2,\ldots,m$$

Encoder-decoder cross attention

Self-attention:

$$\mathbf{q}_i = \mathbf{x}_i\mathbf{W}^Q, \mathbf{k}_i = \mathbf{x}_i\mathbf{W}^K, \mathbf{v}_i = \mathbf{x}_i\mathbf{W}^V$$

$$e_{i,j} = \frac{\mathbf{q}_i \cdot \mathbf{k}_j}{\sqrt{d_k}}, \forall j = 1, \ldots, n$$

$$\alpha_i = \text{softmax}(\mathbf{e}_i)$$

$$\mathbf{h}_i = \sum_{j=1}^{n} \alpha_{i,j}\mathbf{v}_j$$

Cross-attention: (always from the top layer)

$\tilde{\mathbf{x}}_1, \ldots, \tilde{\mathbf{x}}_m$: hidden states from encoder

$\mathbf{x}_1, \ldots, \mathbf{x}_n$: hidden states from decoder

$$\mathbf{q}_i = \mathbf{x}_i\mathbf{W}^Q \quad i = 1,2,\ldots,n$$

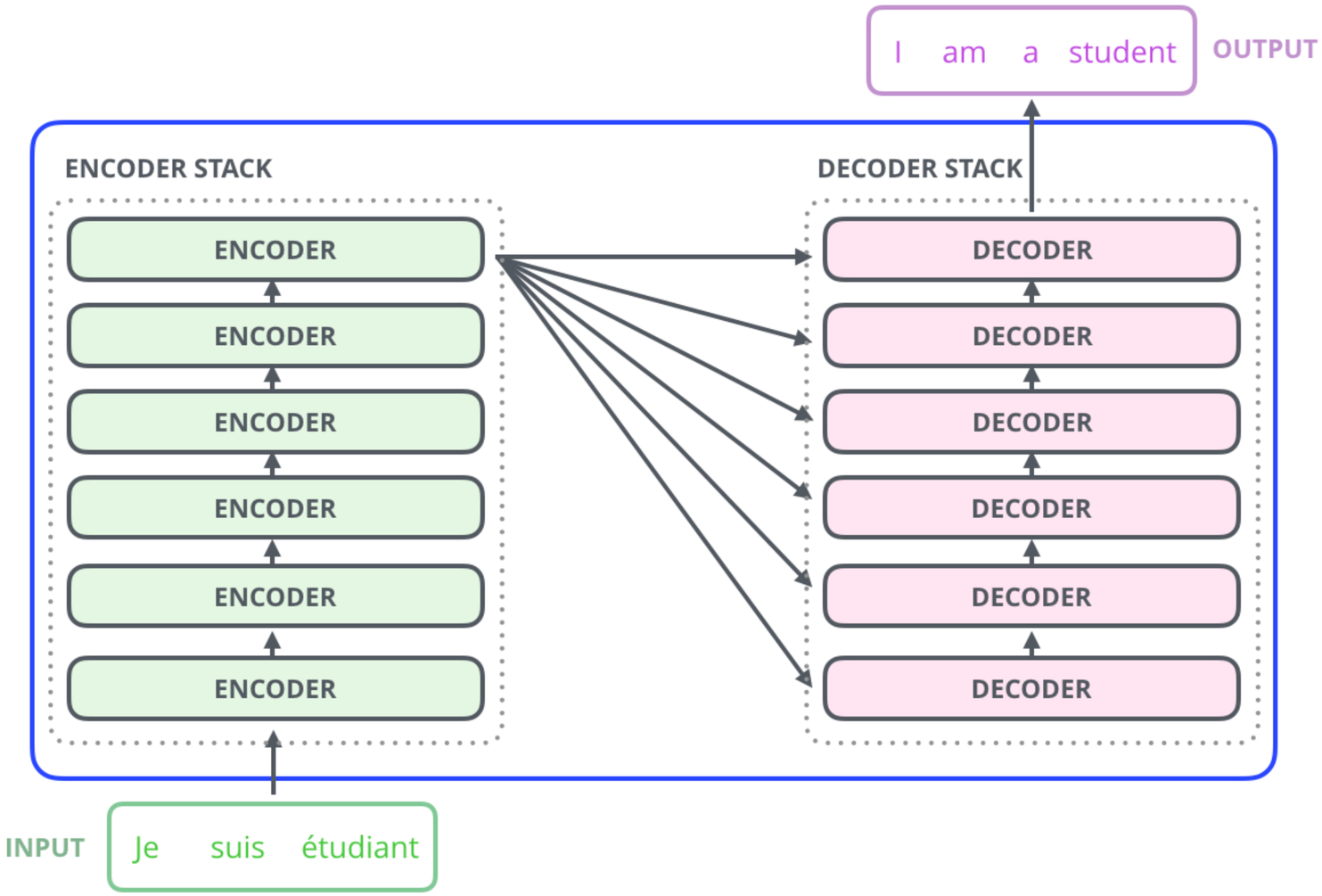
$$\mathbf{k}_j = \tilde{\mathbf{x}}_j\mathbf{W}^K, \mathbf{v}_j = \tilde{\mathbf{x}}_j\mathbf{W}^V \quad \forall j = 1,2,\ldots,m$$

$$e_{i,j} = \frac{\mathbf{q_i} \cdot \mathbf{k}_j}{\sqrt{d_k}}, \forall j = 1, \ldots, m$$

$$\alpha_i = \text{softmax}(\mathbf{e}_i)$$

$$\mathbf{h}_i = \sum_{j=1}^{m} \alpha_{i,j}\mathbf{v}_j$$

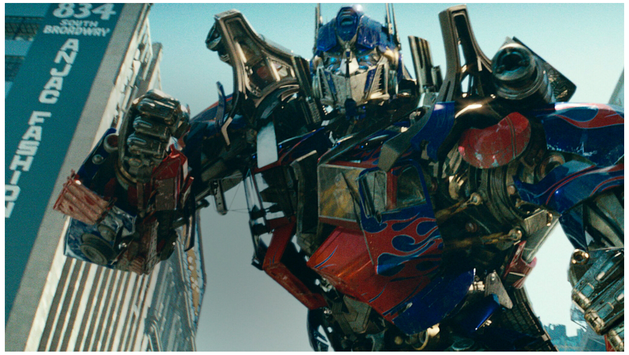
Transformer encoder-decoder: Another visualization



https://jalammar.github.io/illustrated-gpt2/

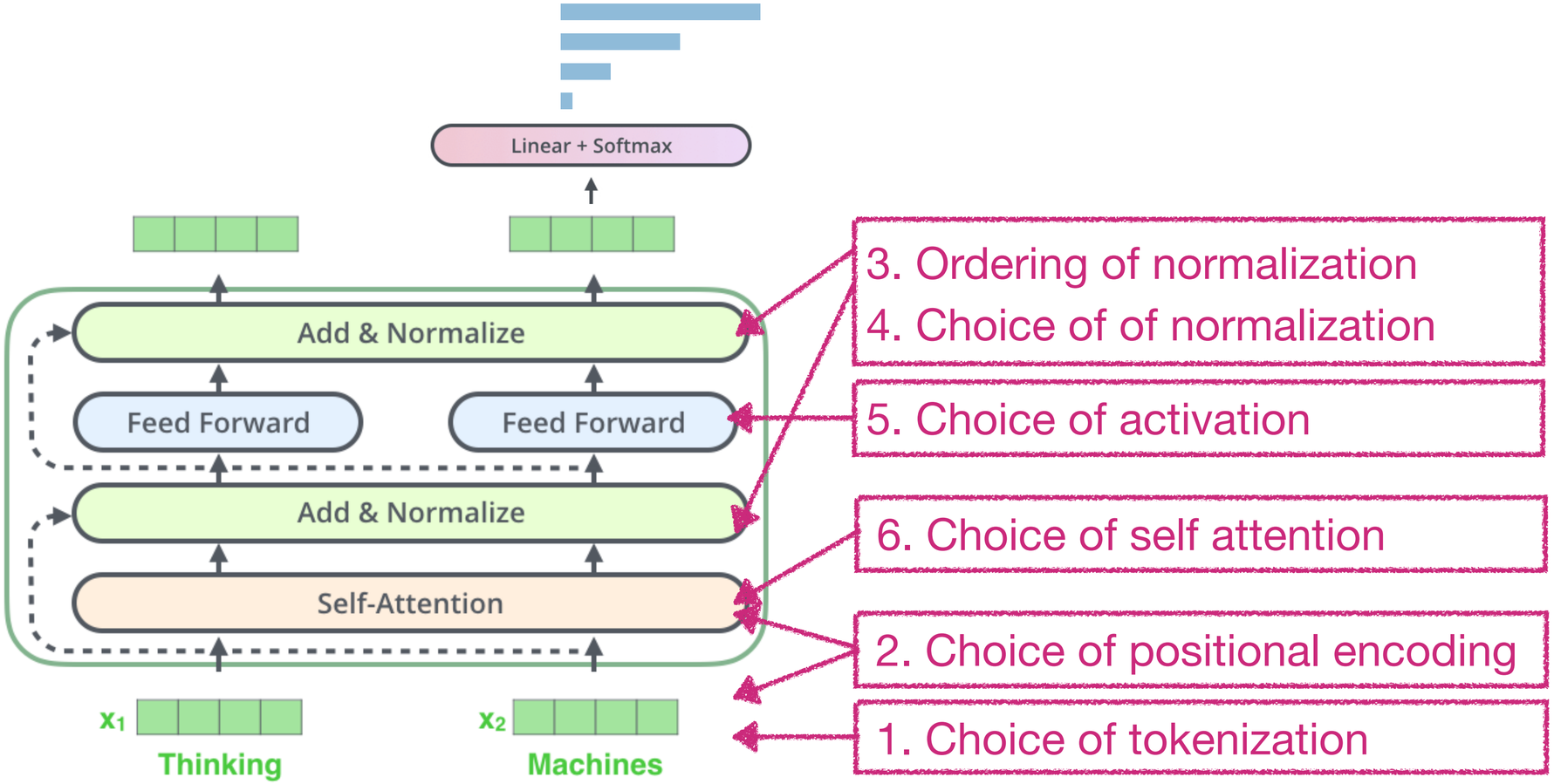
Transformers Variants

Transformers: Context



- Since introduced in 2017 for NMT, Transformers soon benefited from large-scale pre-training—a major breakthrough in NLP
  - Pre-training: Self-supervised learning on massive text data
  - The benefits of pre-training are maximized by scalable architectures
  - Turns out Transformers scale exceptionally well (GPU parallelization, no vanishing gradient issues, etc)
- However, scaling also introduced new small and big challenges, requiring many modifications beyond the original Transformer design
- **Today: Modern Transformers variants for improved scalability**
  - **You will implement this modern variant—rather than the original version—for A2.**

Transformers variants



Transformers variants

	OLMo-1B (0724)
Dimension	2,048
Activation	SwiGLU
FFN dimension	8,192
Vocab size	50,304
Attn heads	16
Num layers	16
Layer norm type	non-parametric
Layer norm eps	1.0E-05
QK-Norm	no
Pos emb.	RoPE
RoPE θ	10,000
Attention variant	full
Biases	-
Weight tying	no
Init dist	normal
Init std	varies
Init trunc	-
MoE layers	-
MoE layer type	-
# Experts	1
# Activated	1
# Vocab params	103M
# Active params	1.3B
# Total params	1.3B
Sequence length	4,096
Batch size (samples)	512
Batch size (tokens)	~2M
warmup steps	2,000
peak LR	4.0E-04
minimum LR	5.0E-05
optimizer	AdamW
weight decay	0.1
beta1	0.9
beta2	0.95
AdamW epsilon	1.0E-05
LR schedule	cosine
gradient clipping	global 1.0
gradient reduce dtype	FP32
optimizer state dtype	FP32
LBL weight	-
Router z-loss weight	-
Pretraining tokens	2,000B
Annealing tokens	50B
Annealing schedule	linear
Annealing min LR	0

In fact, more than six variables that can be done differently

Muennighoff et al. 2025. "OLMoE: Open Mixture-of-Experts Language Models"

Recap: Disclaimer

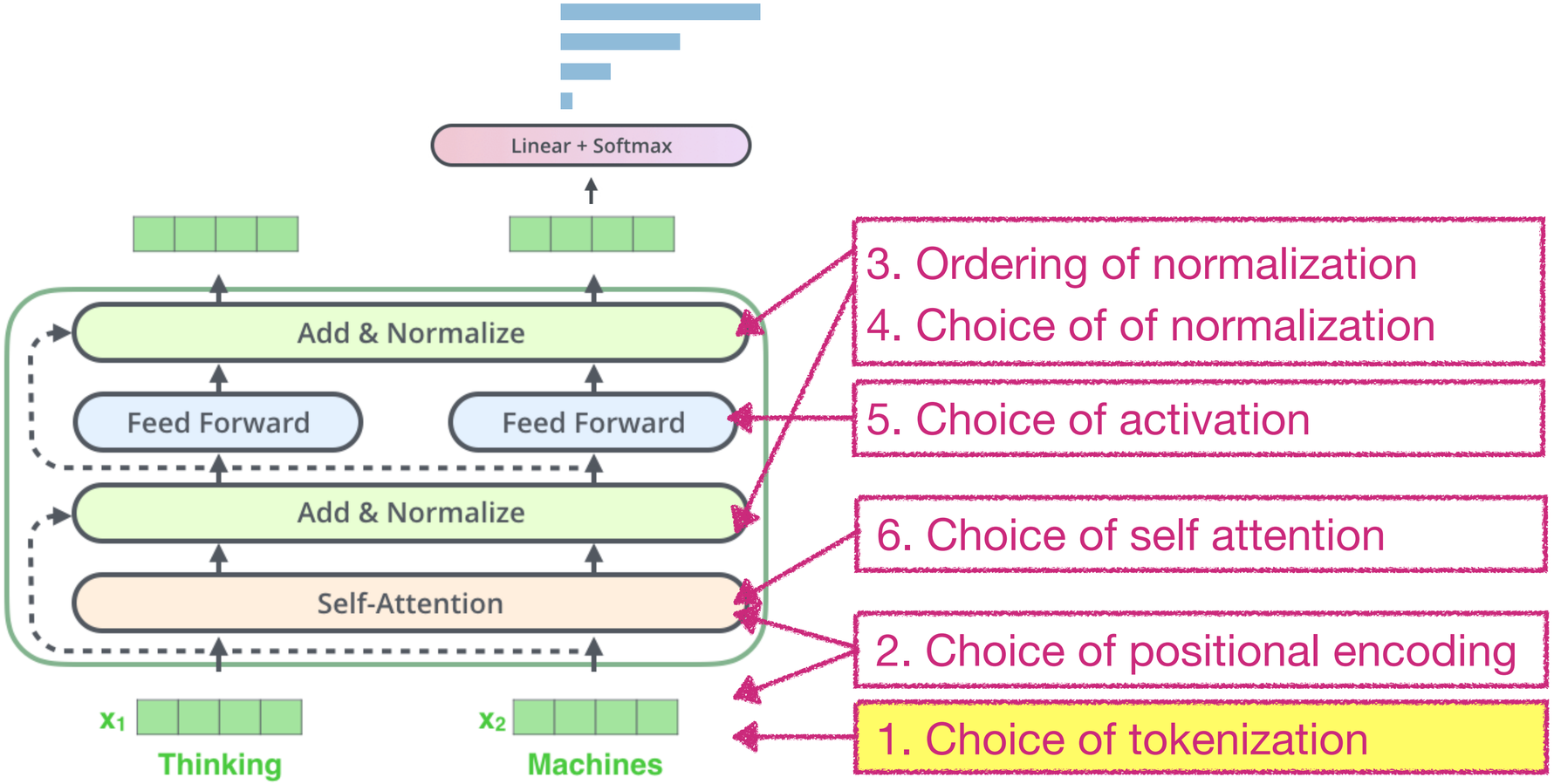
Some design decisions are simply not (yet) justifiable and just come from experimentation.

Example: Noam Shazeer paper that introduced SwiGLU

4 Conclusions

We have extended the GLU family of layers and proposed their use in Transformer. In a transfer-learning setup, the new variants seem to produce better perplexities for the de-noising objective used in pre-training, as well as better results on many downstream language-understanding tasks. These architectures are simple to implement, and have no apparent computational drawbacks. We offer no explanation as to why these architectures seem to work; we attribute their success, as all else, to divine benevolence.

Transformers variants



Tokenization 101 (1/3)

Tokenization: splitting raw text into a sequence of units (called “tokens”)

- “Bridge” between raw text and the numerical vectors (word embeddings) that models can understand

Approach 1: Word-level:

- Splits text based on whitespace (handling punctuation gracefully)
- “I can't go.” → [“I”, “can”, “’t”, “go”, “.”]
- “unbelievable” → [“unbelievable”]
- Examples: .split(), NLTK tokenizer, spaCy tokenizer

Input String	text.split()	NLTK word_tokenize	spaCy
I'm	["I'm"]	["I", "'m"]	["I", "'m"]
U.K.	["U.K."]	["U.K."]	[“U.K."]
end.	["end."]	["end", "."]	["end", "."]
fast-paced	["fast-paced"]	["fast-paced"]	["fast", "-", "paced"]

Tokenization 101 (2/3)

Approach 1: Word-level:

- “I can't go.” → [“I”, “can”, “’t”, “go”, “.”]
- “unbelievable” → [“unbelievable”]
- Issue: **the Out-of-Vocabulary (OOV) problem**
    - Fixed Vocab: Unseen word is treated as [UNK] (Unknown)
    - No Morphology: The model sees "run," "running," and "runs" as three entirely separate things with no shared meaning

Approach 2: Character-level

- “I can't go.” → [“I”, “c”, “a”, “n”, “’”, "t”, “g”, “o”, “.”]
- “unbelievable” → [“u”, “n”, “b”, “e”, “l”, “i”, “e”, “v”, “a”, “b”, “l”, “e”]
- No [UNK] (unless the character is unseen)
- Issue: No inherent meaning in single letters; very long sequences

Approach 3: Subword-level (THE standard)

- “unbelievable” → [“un”, “believ”, “able”]

Tokenization 101 (3/3)

Approach 3: Subword-level (THE standard)

- Intuitions
  - Words are (often) composed of subparts (morphemes)
  - Our vocabulary should have entries for frequent words kept whole, because we have a lot of data about those words
    - “apple” → [“apple”]
  - But it should also have entries for parts of less-frequent words, so our ML models can learn how to compose parts of words into whole words (especially unfamiliar words!)
    - “unbelievable” → [“un”, “believ”, “able”]
- Good compromise between word-level and character-level tokenization
  - No [UNK] (unless the character is unseen)
  - Can consider morphology: “run” → [“run”], “running” → [“run”, “ning”], “runs” → [“run”, “s”]
  - Doesn’t get too long

Original Transformers: WordPiece

WordPiece: A type of subword-level tokenizer

- Start with individual characters, then merge co-occurring characters if **merging increases the likelihood of the training data**

Process:

1. Base vocabulary of every individual character; ## prefix for non-initial subwords, e.g., "hug" → ["h", "##u", "##g"]
2. Iterative merging:
    1. Compute score for all adjacent pairs: $\text{Score}(A, B) = \frac{\text{co-occurrence}(A, B)}{\text{freq}(A) \times \text{freq}(B)}$
    2. Merge the highest-scoring pair
3. Stop when the target vocab size is reached

- Origin: Developed by Google, first for speech recognition in Japanese and Korean, and broadly in language processing
- Used in original Transformers naturally (as it was already a Google standard) and therefore used in subsequent models (BERT, DeBERTa, Electra)

(Running example coming soon, so don't worry if you don't exactly understand the iterative part yet!)

Modern standard: Byte Pair Encoding (BPE)

Byte Pair Encoding: A type of subword-level tokenizer

- **Byte-level** (instead of character), **Frequency-based** (instead of likelihood-based)
- Start with individual bytes, then **merge the most common co-occurring bytes**

Q: What does byte-level mean?

- Text is processed as raw bytes, not characters or words
- Every single character in existence is made of 1, 2, 3, or 4 bytes.
  - ASCII characters (like A, b, ?) are 1 byte each.
  - Complex characters (like é, Ω, or 😄) are 2 to 4 bytes each.
- Key advantage: no [UNK] even with unseen characters!

Modern standard: Byte Pair Encoding (BPE)

Byte Pair Encoding: A type of subword-level tokenizer

- **Byte-level** (instead of character), **Frequency-based** (instead of likelihood-based)
- Start with individual bytes, then **merge the most common co-occurring bytes**

Process:

1. Base vocabulary of every individual bytes, e.g., "hug" → [104, 117, 103]
2. Iterative merging:
    1. Compute co-occurrence for all adjacent byte pairs
    2. Merge the most common pair
3. Stop when the target vocab size is reached

- Origin: Invented for data compression by Phillip Gage in 1994
- Wasn't really used in NLP until researchers later realized this compression logic can be effective for the "unknown word" problem in NMT
- Adapted in GPT-2, now standard in LLMs (RoBERTa, GPT-2/3, Llama, etc)
- Historical detail: Early NLP versions of BPE used Unicode characters; GPT-2 switched back to byte-level (often called Byte-level BPE).

(Running example coming soon, so don't worry if you don't exactly understand the iterative part yet!)

Byte Pair Encoding: Running example

Documents + frequencies: `('hug', 10), ('pug', 5), ('pun', 12), ('bun', 4), ('hugs', 5)`

Target vocab size = 10

vocabulary ***tokenized texts***

Iter 1 `('h', 'u', 'g', 'p', 'n', 'b', 's')` → `('h' 'u' 'g', 10), ('p' 'u' 'g', 5), ('p' 'u' 'n', 12), ('b' 'u' 'n', 4), ('h' 'u' 'g' 's', 5)`

bigrams + frequencies

```
'h' 'u'     15
'u' 'g'     20
'u' 'n'     16
   …
```

Byte Pair Encoding: Running example

Documents + frequencies: `('hug', 10), ('pug', 5), ('pun', 12), ('bun', 4), ('hugs', 5)`

Target vocab size = 10

vocabulary ***tokenized texts***

Iter 1 `('h', 'u', 'g', 'p', 'n', 'b', 's')` → `('h' 'u' 'g', 10), ('p' 'u' 'g', 5), ('p' 'u' 'n', 12), ('b' 'u' 'n', 4), ('h' 'u' 'g' 's', 5)`

Iter 2 `('h', 'u', 'g', 'p', 'n', 'b', 's', 'ug')` → `('h' 'ug', 10), ('p' 'ug', 5), ('p' 'u' 'n', 12), ('b' 'u' 'n', 4), ('h' 'ug' 's', 5)`

bigrams + frequencies

'h' 'u'	15
'u' 'n'	16
…	
'h' 'ug'	15
'p' 'ug'	5
'ug' 's'	5

Byte Pair Encoding: Running example

Documents + frequencies: `('hug', 10), ('pug', 5), ('pun', 12), ('bun', 4), ('hugs', 5)`

Target vocab size = 10

	vocabulary	*tokenized texts*
Iter 1	`('h', 'u', 'g', 'p', 'n', 'b', 's')` →	`('h' 'u' 'g', 10), ('p' 'u' 'g', 5), ('p' 'u' 'n', 12), ('b' 'u' 'n', 4), ('h' 'u' 'g' 's', 5)`
Iter 2	`('h', 'u', 'g', 'p', 'n', 'b', 's', 'ug')` →	`('h' 'ug', 10), ('p' 'ug', 5), ('p' 'u' 'n', 12), ('b' 'u' 'n', 4), ('h' 'ug' 's', 5)`
Iter 3	`('h', 'u', 'g', 'p', 'n', 'b', 's', 'ug', 'un')` →	`('h' 'ug', 10), ('p' 'ug', 5), ('p' 'un', 12), ('b' 'un', 4), ('h' 'ug' 's', 5)`

bigrams + frequencies

'h' 'u'	15
…	
'h' 'ug'	15
'p' 'ug'	5
'ug' 's'	5
…	

Byte Pair Encoding: Running example

Documents + frequencies: `('hug', 10), ('pug', 5), ('pun', 12), ('bun', 4), ('hugs', 5)`

Target vocab size = 10

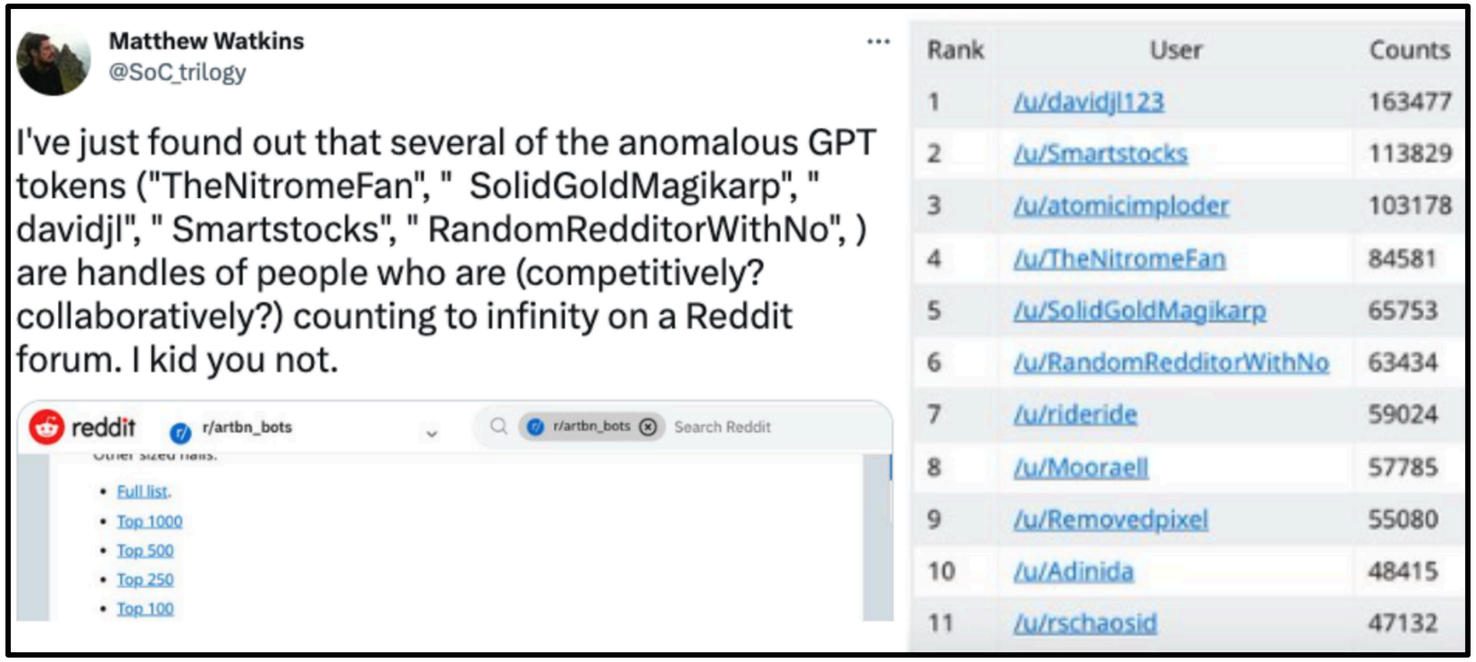
	vocabulary	*tokenized texts*
Iter 1	`('h', 'u', 'g', 'p', 'n', 'b', 's')` →	`('h' 'u' 'g', 10), ('p' 'u' 'g', 5), ('p' 'u' 'n', 12), ('b' 'u' 'n', 4), ('h' 'u' 'g' 's', 5)`
Iter 2	`('h', 'u', 'g', 'p', 'n', 'b', 's', 'ug')` →	`('h' 'ug', 10), ('p' 'ug', 5), ('p' 'u' 'n', 12), ('b' 'u' 'n', 4), ('h' 'ug' 's', 5)`
Iter 3	`('h', 'u', 'g', 'p', 'n', 'b', 's', 'ug', 'un')` →	`('h' 'ug', 10), ('p' 'ug', 5), ('p' 'un', 12), ('b' 'un', 4), ('h' 'ug' 's', 5)`
Iter 4	`('h', 'u', 'g', 'p', 'n', 'b', 's', 'ug', 'un', 'hug')` →	`('hug', 10), ('p' 'ug', 5), ('p' 'un', 12), ('b' 'un', 4), ('hug' 's', 5)`

New word: “puns”

`'p' 'u' 'n' 's'` → `'p' 'un' 's'` ✓

Note on subword tokenization

- Tokenizer is learned directly from data (no rules!).
  - We call the process of building the tokenizer and vocabulary also “training”.
- That means tokenizer and vocabulary could reveal info about the training data (when it’s hidden)!



Why BPE over WordPiece?

Byte-level

- WordPiece: can encounter a character it wasn't trained on (e.g., 😄 or Ω), it defaults to [UNK] (Unknown),
- In a generative chat model, [UNK] is a disaster
- BPE can represent literally any string of text without ever needing an [UNK] token

Speed and Simplicity

- WordPiece: Calculating likelihoods is more expensive
- BPE is a simple "count and merge" algorithm

Token efficiency

- BPE = compression algorithm — it packs the most data into the fewest tokens
- Can generate the same text with fewer tokens = more efficiently

Feb 12 lecture starts from here

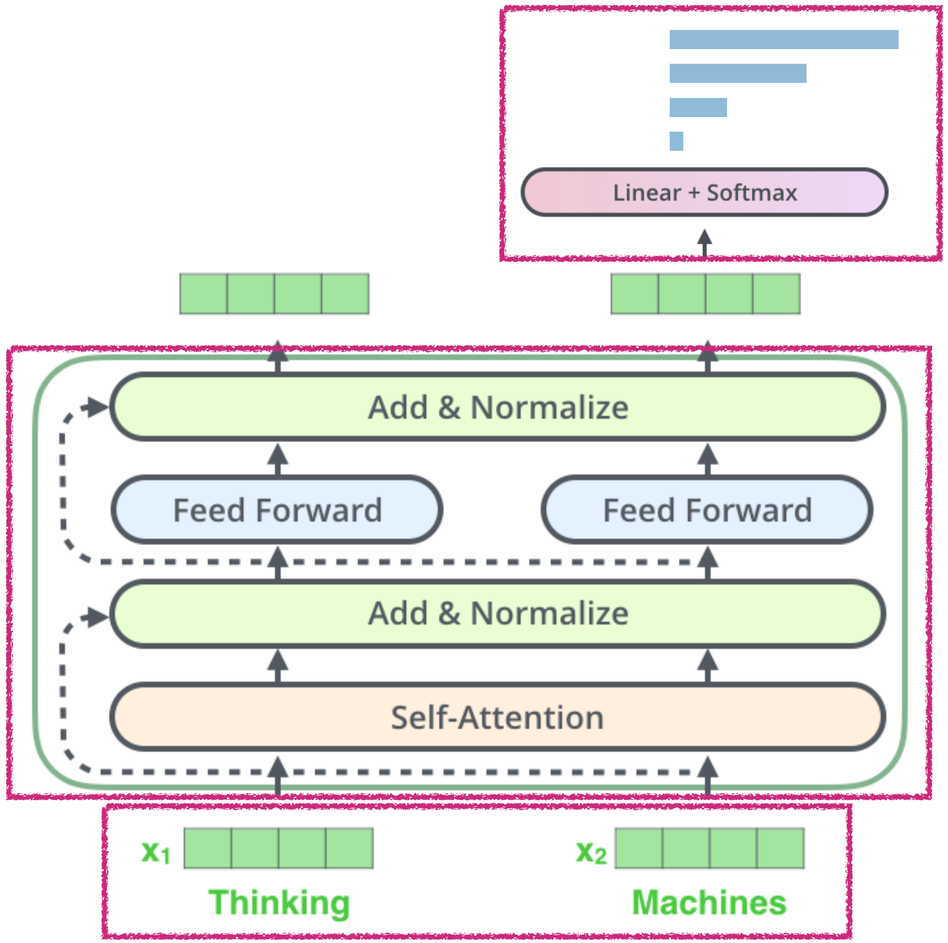
CS 288 Advanced Natural Language Processing

Course website: cal-cs288.github.io/sp26

Ed: edstem.org/us/join/XvztdK

- Class starts at 15:40!
- A1 last possible late day is tomorrow (Friday, Feb 13)
- Team formation results: released by the end of tomorrow (Friday, Feb 13)
- A2 is live (Today's lecture continues to be very relevant)
  - Part 4: You may use GPUs via Google cloud ($50 credits available; see the very first Ed post from Sewon)
- Lecture plan: Complete Transformers (60min) → Pre-training (20min)

Recap: Transformers

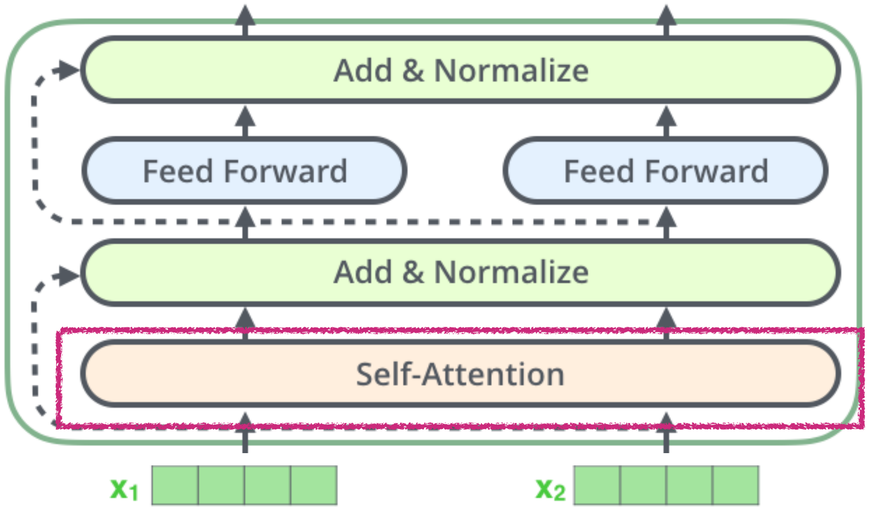


3. Softmax

2. Stacked Transformer Layers

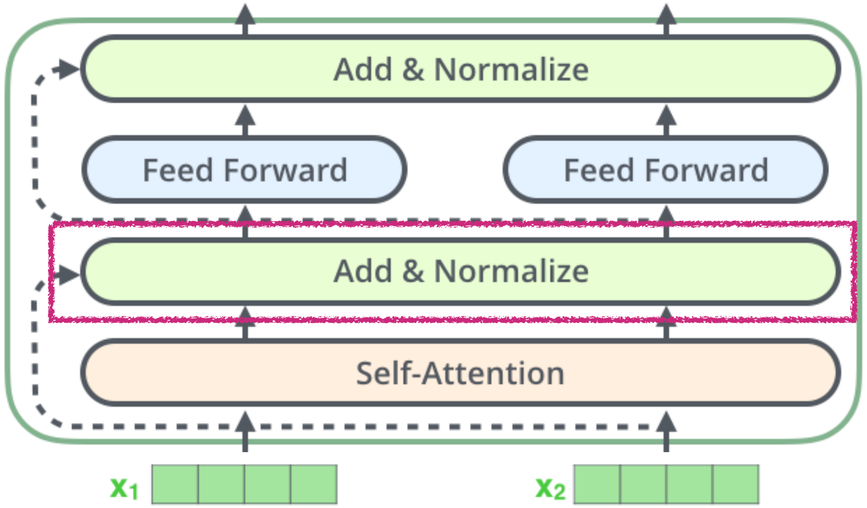
1. Input & Embeddings

Recap in code (1/2)



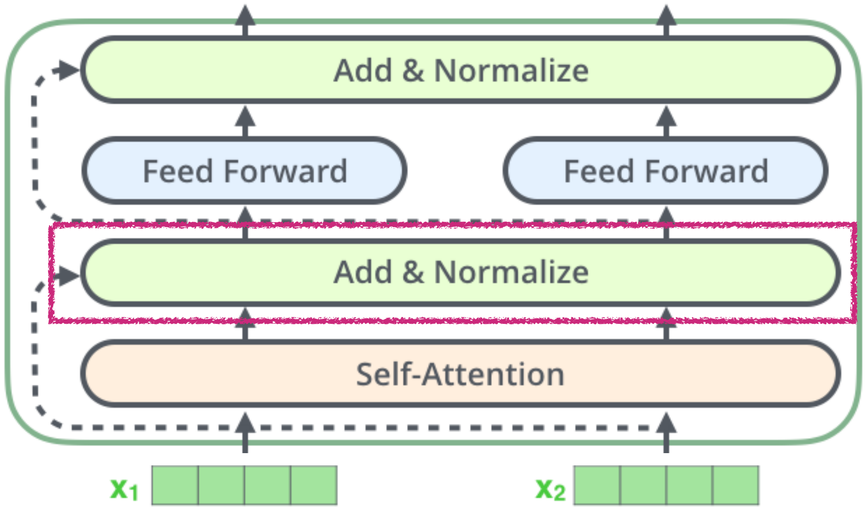

```python
class TransformerBlock(nn.Module):

    def __init__(self, d_model, d_ff, num_heads, dropout):
        super().__init__()
        self.mha = MultiHeadAttention(num_heads, d_model, dropout)
        self.ff = nn.Linear(d_model, d_model)
        self.norm1 = nn.LayerNorm(d_model)
        self.norm2 = nn.LayerNorm(d_model)

    def forward(self, x, mask):
        # Multi-head attention + residual connection
        x = x + self.mha(x, x, x, mask)
        # Post-norm
        x = self.norm1(x)
        # Feed-forward + residual connection
        x = x + self.ff(x)
        # Post-norm
        x = self.norm2(x)
        return x
```

Recap in code (1/2)



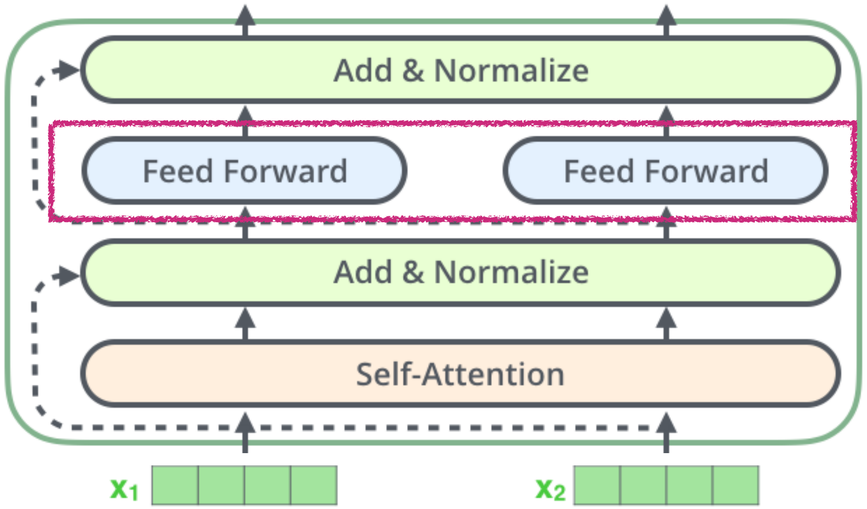

```python
class TransformerBlock(nn.Module):

    def __init__(self, d_model, d_ff, num_heads, dropout):
        super().__init__()
        self.mha = MultiHeadAttention(num_heads, d_model, dropout)
        self.ff = nn.Linear(d_model, d_model)
        self.norm1 = nn.LayerNorm(d_model)
        self.norm2 = nn.LayerNorm(d_model)

    def forward(self, x, mask):
        # Multi-head attention + residual connection
        x = x + self.mha(x, x, x, mask)
        # Post-norm
        x = self.norm1(x)
        # Feed-forward + residual connection
        x = x + self.ff(x)
        # Post-norm
        x = self.norm2(x)
        return x
```

Recap in code (1/2)



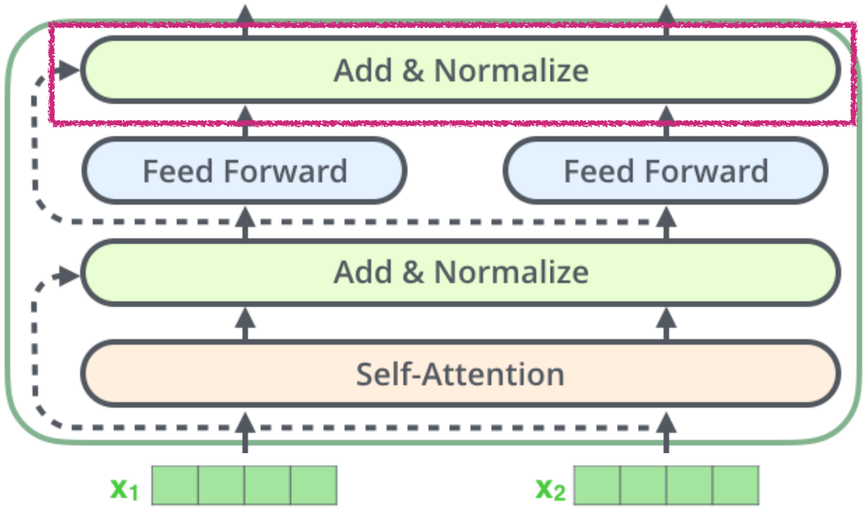

```python
class TransformerBlock(nn.Module):

    def __init__(self, d_model, d_ff, num_heads, dropout):
        super().__init__()
        self.mha = MultiHeadAttention(num_heads, d_model, dropout)
        self.ff = nn.Linear(d_model, d_model)
        self.norm1 = nn.LayerNorm(d_model)
        self.norm2 = nn.LayerNorm(d_model)

    def forward(self, x, mask):
        # Multi-head attention + residual connection
        x = x + self.mha(x, x, x, mask)
        # Post-norm
        x = self.norm1(x)
        # Feed-forward + residual connection
        x = x + self.ff(x)
        # Post-norm
        x = self.norm2(x)
        return x
```

Recap in code (1/2)



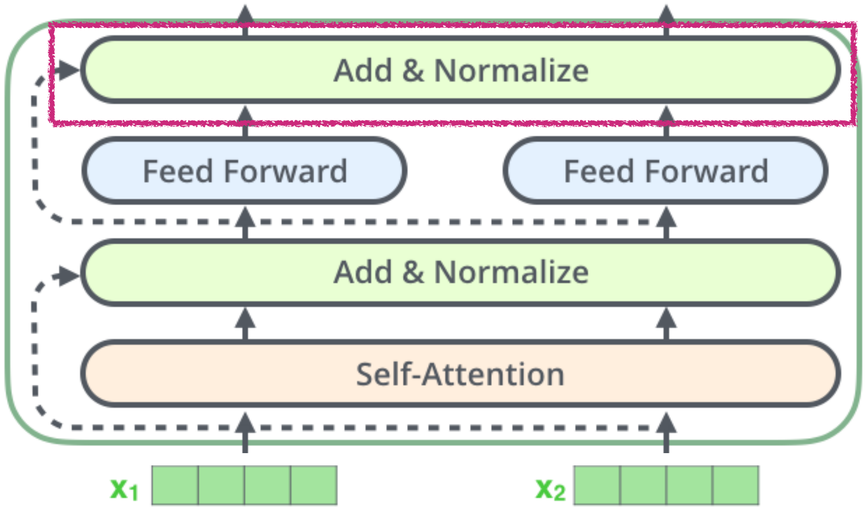

```python
class TransformerBlock(nn.Module):

    def __init__(self, d_model, d_ff, num_heads, dropout):
        super().__init__()
        self.mha = MultiHeadAttention(num_heads, d_model, dropout)
        self.ff = nn.Linear(d_model, d_model)
        self.norm1 = nn.LayerNorm(d_model)
        self.norm2 = nn.LayerNorm(d_model)

    def forward(self, x, mask):
        # Multi-head attention + residual connection
        x = x + self.mha(x, x, x, mask)
        # Post-norm
        x = self.norm1(x)
        # Feed-forward + residual connection
        x = x + self.ff(x)
        # Post-norm
        x = self.norm2(x)
        return x
```

Recap in code (1/2)



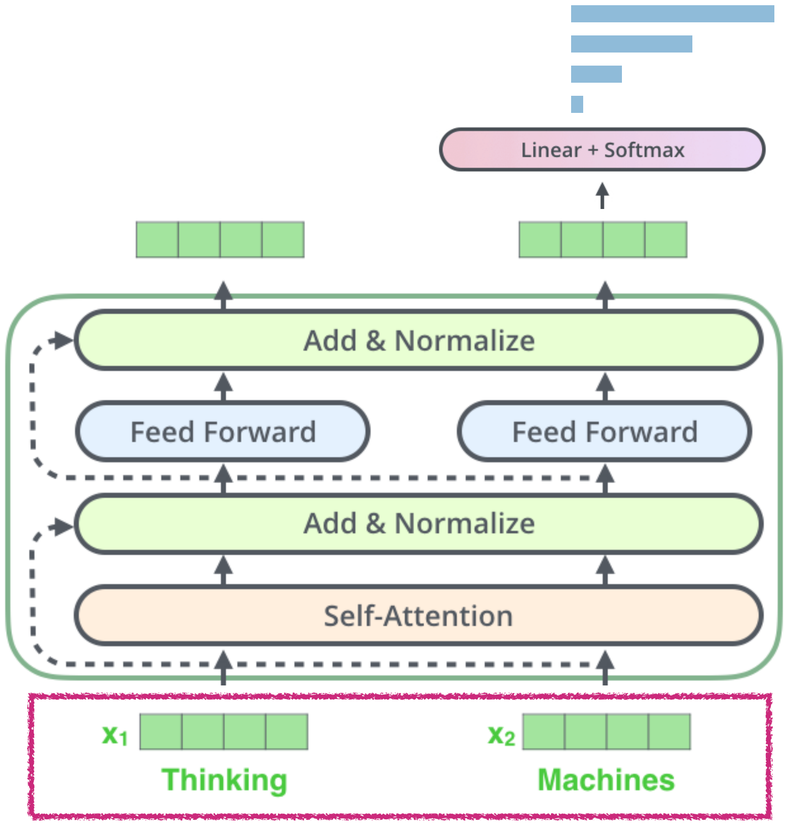

```python
class TransformerBlock(nn.Module):

    def __init__(self, d_model, d_ff, num_heads, dropout):
        super().__init__()
        self.mha = MultiHeadAttention(num_heads, d_model, dropout)
        self.ff = nn.Linear(d_model, d_model)
        self.norm1 = nn.LayerNorm(d_model)
        self.norm2 = nn.LayerNorm(d_model)

    def forward(self, x, mask):
        # Multi-head attention + residual connection
        x = x + self.mha(x, x, x, mask)
        # Post-norm
        x = self.norm1(x)
        # Feed-forward + residual connection
        x = x + self.ff(x)
        # Post-norm
        x = self.norm2(x)
        return x
```

Recap in code (1/2)



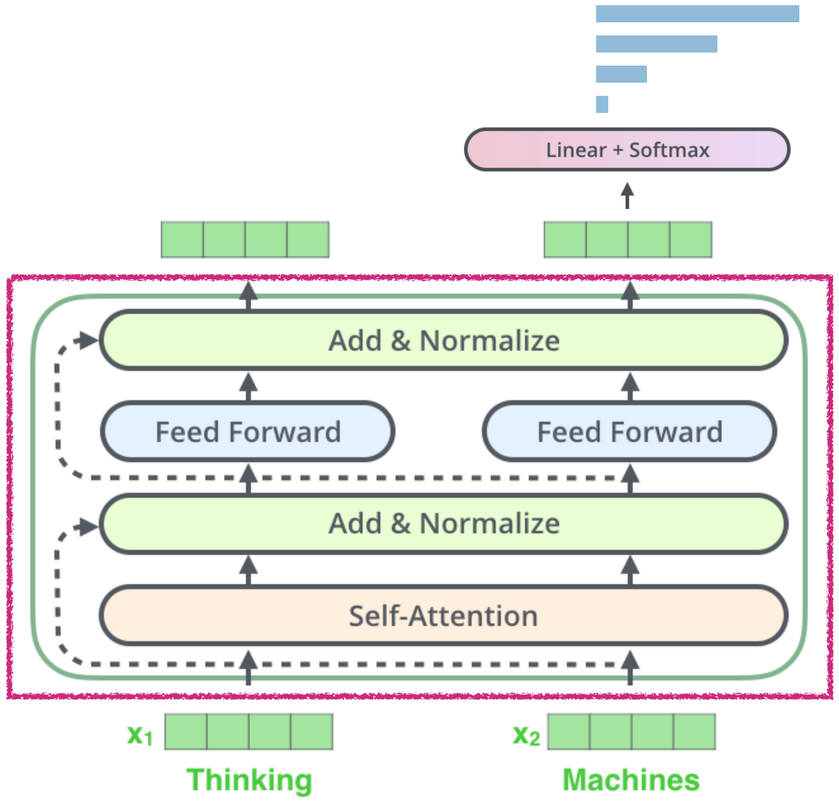

```python
class TransformerBlock(nn.Module):

    def __init__(self, d_model, d_ff, num_heads, dropout):
        super().__init__()
        self.mha = MultiHeadAttention(num_heads, d_model, dropout)
        self.ff = nn.Linear(d_model, d_model)
        self.norm1 = nn.LayerNorm(d_model)
        self.norm2 = nn.LayerNorm(d_model)

    def forward(self, x, mask):
        # Multi-head attention + residual connection
        x = x + self.mha(x, x, x, mask)
        # Post-norm
        x = self.norm1(x)
        # Feed-forward + residual connection
        x = x + self.ff(x)
        # Post-norm
        x = self.norm2(x)
        return x
```

Recap in code (2/2)



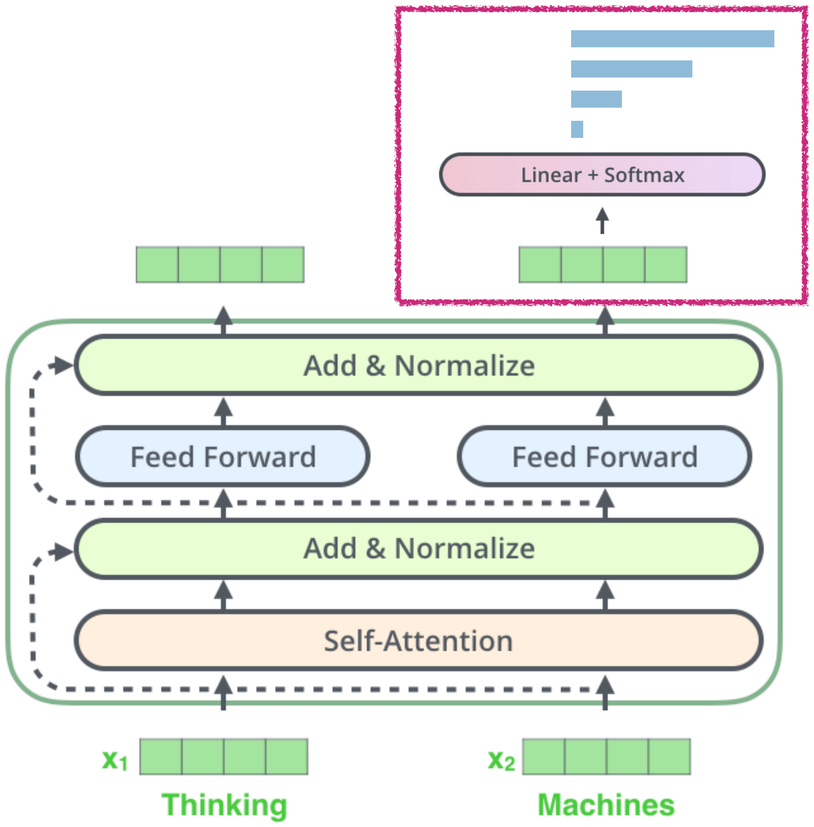

```python
class Transformers(nn.Module):

    def __init__(self, vocab_size, num_layers, d_model, d_ff, num_heads, dropout):
        super().__init__()
        self.vocab_size = vocab_size
        self.num_layers = num_layers
        self.d_model = d_model
        self.d_ff = d_ff
        self.num_heads = num_heads
        self.dropout = dropout

        self.embed = nn.Embedding(vocab_size, d_model)
        self.pos_encoder = PositionalEncoding(d_model, dropout)
        self.norm = nn.LayerNorm(d_model)
        self.output = nn.Linear(d_model, vocab_size, bias=False)

        self.input_embed.weight.data = self.output.weight.data

        self.transformer_blocks = nn.ModuleList([
            TransformerBlock(d_model, d_ff, num_heads, dropout) for _ in range(num_layers)
        ])

    def __call__(self, src, src_mask):
        x = self.embed(src)
        pos = torch.arange(x.size(0), device=x.device)
        x = x + self.pos_encoder(pos)
        for block in self.transformer_blocks:
            x = block(x, src_mask)

        x = self.output(self.norm(x))
        return nn.Softmax(dim=-1)(x)
```

Recap in code (2/2)




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

    def __init__(self, vocab_size, num_layers, d_model, d_ff, num_heads, dropout):
        super().__init__()
        self.vocab_size = vocab_size
        self.num_layers = num_layers
        self.d_model = d_model
        self.d_ff = d_ff
        self.num_heads = num_heads
        self.dropout = dropout

        self.embed = nn.Embedding(vocab_size, d_model)
        self.pos_encoder = PositionalEncoding(d_model, dropout)
        self.norm = nn.LayerNorm(d_model)
        self.output = nn.Linear(d_model, vocab_size, bias=False)

        self.input_embed.weight.data = self.output.weight.data

        self.transformer_blocks = nn.ModuleList([
            TransformerBlock(d_model, d_ff, num_heads, dropout) for _ in range(num_layers)
        ])

    def __call__(self, src, src_mask):
        x = self.embed(src)
        pos = torch.arange(x.size(0), device=x.device)
        x = x + self.pos_encoder(pos)
        for block in self.transformer_blocks:
            x = block(x, src_mask)

        x = self.output(self.norm(x))
        return nn.Softmax(dim=-1)(x)
```

Recap in code (2/2)




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

    def __init__(self, vocab_size, num_layers, d_model, d_ff, num_heads, dropout):
        super().__init__()
        self.vocab_size = vocab_size
        self.num_layers = num_layers
        self.d_model = d_model
        self.d_ff = d_ff
        self.num_heads = num_heads
        self.dropout = dropout

        self.embed = nn.Embedding(vocab_size, d_model)
        self.pos_encoder = PositionalEncoding(d_model, dropout)
        self.norm = nn.LayerNorm(d_model)
        self.output = nn.Linear(d_model, vocab_size, bias=False)

        self.input_embed.weight.data = self.output.weight.data

        self.transformer_blocks = nn.ModuleList([
            TransformerBlock(d_model, d_ff, num_heads, dropout) for _ in range(num_layers)
        ])

    def __call__(self, src, src_mask):
        x = self.embed(src)
        pos = torch.arange(x.size(0), device=x.device)
        x = x + self.pos_encoder(pos)
        for block in self.transformer_blocks:
            x = block(x, src_mask)

        x = self.output(self.norm(x))
        return nn.Softmax(dim=-1)(x)
```

Recap: Transformers quiz

Which of the following statements is correct?

=Floating-point operations
(Remember this term!)

(a) Transformers have less FLOPs compared to LSTMs.

(b) Transformers have less parameters compared to LSTMs.

(c) Transformers are easier to parallelize compared to LSTMs.

(d) Transformers generalize to longer sequences better than LSTMs.

Depends on hyperparams; not properties of architectures.
(In general, Transformers scale better than LSTMs)

Transformers cannot generalize beyond a pre-defined input length (so-called max_seq_len)
— Even if the positional encoding is valid, the model has never seen how to interpret unseen positional encoding

(c) is correct!

Transformers: Advantages

- Easier to capture long-range dependencies: we draw attention between every pair of words!
- Easier to parallelize
- Empirically, people found Transformers scale very well (compared to RNNs/LSTMs), and therefore, pre-training (next lecture) benefits significantly from Transformers!
    - "Scale well" = You can grow the model size (hidden dimension and number of layers) in a way that it is stable (it trains well) and leads to performance gains
    - Still, people did have to make modification over the original Transformers (today's lecture is largely about that!)

Quick quiz 2

Which one has more parameters (typically), MHA vs. FFN?

(a) MHA

(b) FFN

(c) Equal

1	description	FLOPs / update	% FLOPS MHA	% FLOPS FFN	% FLOPS attn	% FLOPS logit
8	OPT setups					
9	760M	4.3E+15	35%	44%	14.8%	5.8%
10	1.3B	1.3E+16	32%	51%	12.7%	5.0%
11	2.7B	2.5E+16	29%	56%	11.2%	3.3%
12	6.7B	1.1E+17	24%	65%	8.1%	2.4%
13	13B	4.1E+17	22%	69%	6.9%	1.6%
14	30B	9.0E+17	20%	74%	5.3%	1.0%
15	66B	9.5E+17	18%	77%	4.3%	0.6%
16	175B	2.4E+18	17%	80%	3.3%	0.3%

(b) is correct!

Extra context about FLOPs (1/2)

Q: How do you get that FLOP? & Why do FLOPs (and # params) grow faster in FFNs than MHA?

Please refer to yesterday's Ed post! A short answer is

- MHA is $O(n \cdot d^2) + O(n^2 \cdot d)$, FFN is $O(n \cdot d \cdot d_{\text{ff}})$

  (n: sequence length, d: hidden dimension, d_{ff}: hidden dimension in FFN)

- When scaling the model, we typically increase d, d_{ff}, and # of layers (n stays constant)
  - # of layers affect MHA and FFN equally
  - People (typically) scale d_{ff} more rapidly than d
    - Why? More memory-efficient & empirically better
    - Making predictions about *how* to grow the model size = Scaling laws (two weeks from now!)

★This is an advanced topic, so don't worry if you can't exactly follow. You are always welcome to come to OH.★

Extra context about FLOPs (2/2)

Q: Does the FLOP breakdown change as we increase the sequence length limit (*n*)?

Q: If that's true, why do researchers work hard on making attention efficient?

Yes!

- Earlier calculations assume n=2048 and 4096.
- With long-context models, n can be 100K+, and attention becomes a larger fraction of total FLOPs.

However (important practical detail)

- In most LLM training, models are trained with shorter context lengths, and n is increased late in training.
- For most training, the earlier trend still holds: FFN dominates FLOPs.

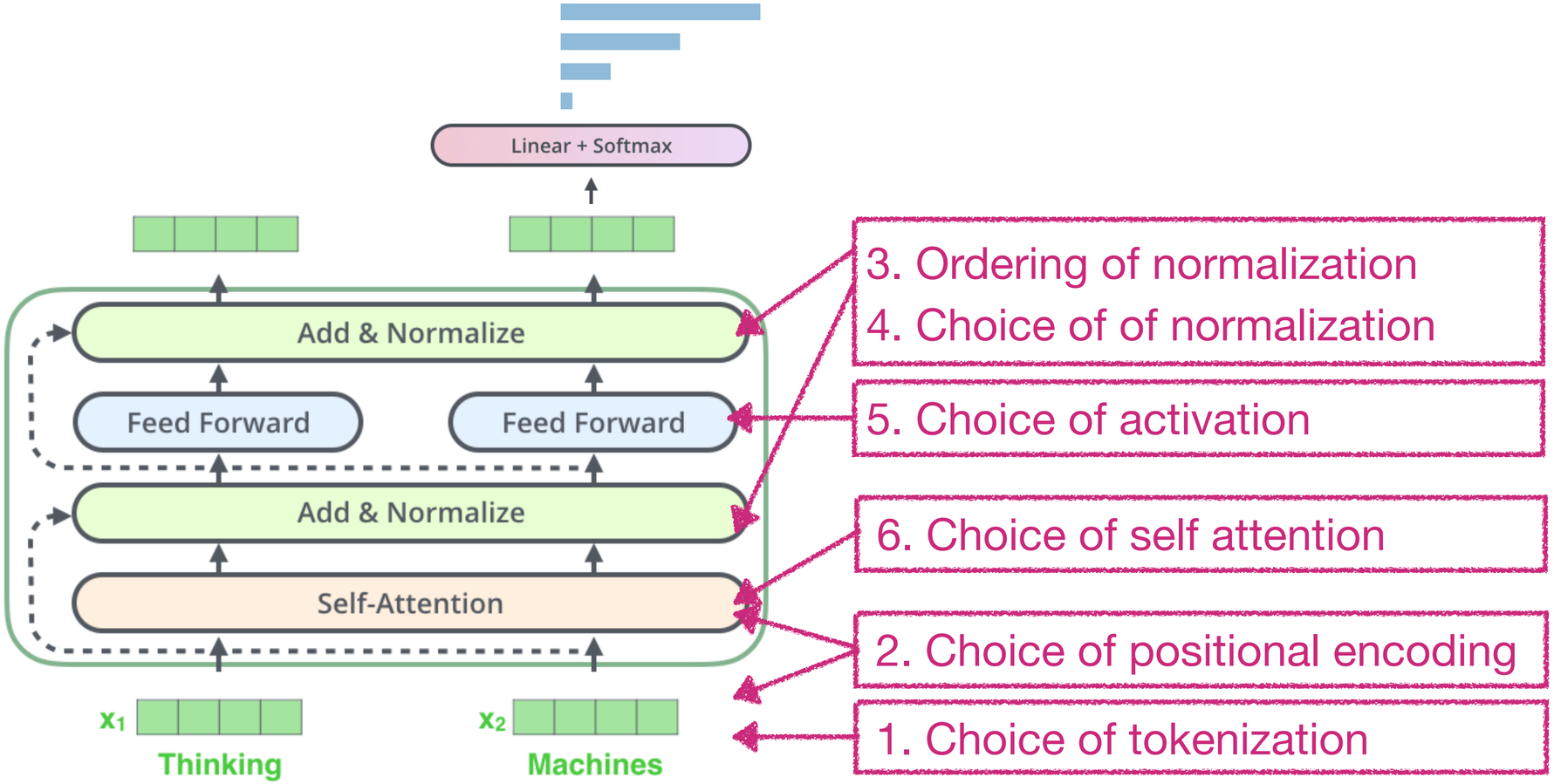
Why researchers make attention efficient? Because FLOP ≠ Runtime efficiency

1. **Memory capacity bound** (both training and inference): Attention needs $O(n^2)$ memory. Even if FLOPs and parameter count are fine, you may be run out of GPU memory.
2. **Memory bandwidth bound** (Inference time; most important): During generation, GPUs repeatedly load the KV cache from HBM; the GPU processor needs to wait. This limits tokens/sec seen by the users.

- Multi-query attention or grouped-query attention (covered today)
- FlashAttention

★This is an advanced topic, so don't worry if you can't exactly follow. You are always welcome to come to OH.★

Transformers variants



Recap: Disclaimer

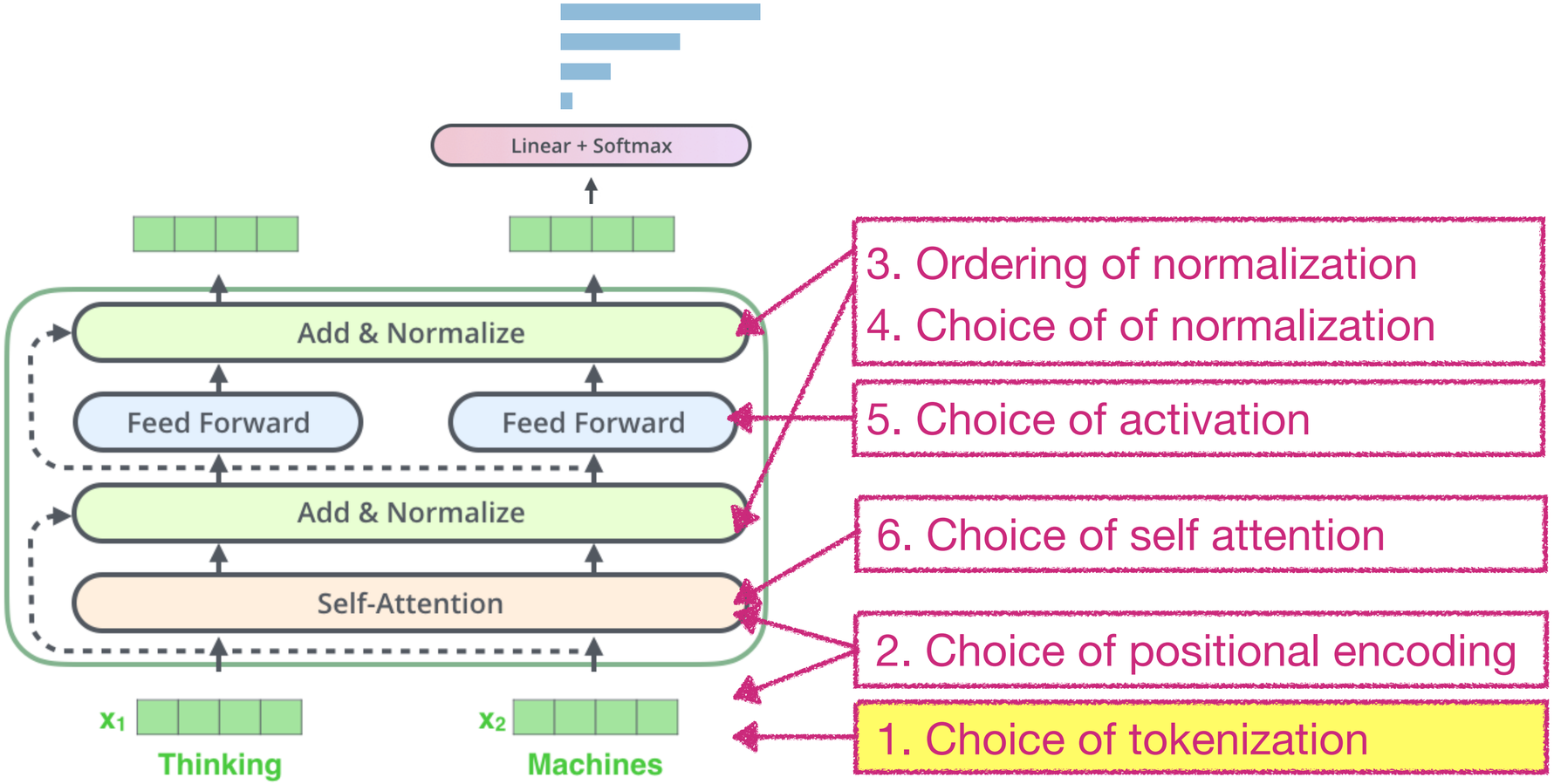
Some design decisions are simply not (yet) justifiable and just come from experimentation.

Example: Noam Shazeer paper that introduced SwiGLU

4 Conclusions

We have extended the GLU family of layers and proposed their use in Transformer. In a transfer-learning setup, the new variants seem to produce better perplexities for the de-noising objective used in pre-training, as well as better results on many downstream language-understanding tasks. These architectures are simple to implement, and have no apparent computational drawbacks. We offer no explanation as to why these architectures seem to work; we attribute their success, as all else, to divine benevolence.

Transformers variants



Recap: Subword-level Tokenization

Approach 1: Word-level

- “I can't go.” → [“I”, “can”, “’t”, “go”, “.”]
- “unbelievable” → [“unbelievable”]
- Issue: **the Out-of-Vocabulary (OOV) problem**

Approach 2: Character-level

- “I can't go.” → [“I”, “c”, “a”, “n”, “’”, "t”, “g”, “o”, “.”]
- “unbelievable” → [“u”, “n”, “b”, “e”, “l”, “i”, “e”, “v”, “a”, “b”, “l”, “e”]
- No [UNK] (unless the character is unseen)
- Issue: No inherent meaning in single letters; very long sequences

Approach 3: Subword-level (THE standard)

- “apple” → [“apple”]
- “unbelievable” → [“un”, “believ”, “able”]
- No [UNK] (unless the character is unseen), Can consider morphology, Doesn’t get too long

Recap: WordPiece vs. BPE

Original Transformers: WordPiece

- Start with individual characters, then merge co-occurring characters if **merging increases the likelihood of the training data**
- History: Developed by Google; naturally used in Transformers (as it was already a Google standard) and therefore used in subsequent models (BERT, DeBERTa, Electra)

Today's standard: Byte Pair Encoding (BPE)

- **Byte-level** (instead of character), **Frequency-based** (instead of likelihood-based)
- Start with individual **bytes**, then **merge the most common co-occurring bytes**
- History: Originally invented for data compression by Phillip Gage in 1994; started to use in NMT; GPT-2 picked it up, and now standard in LLMs (RoBERTa, GPT-2/3, Llama, etc)

(New) Why BPE over WordPiece?

Byte-level

- WordPiece: can encounter a character it wasn't trained on (e.g., 😄 or Ω), it defaults to [UNK] (Unknown),
- In a generative chat model, [UNK] is a disaster
- BPE can represent literally any string of text without ever needing an [UNK] token

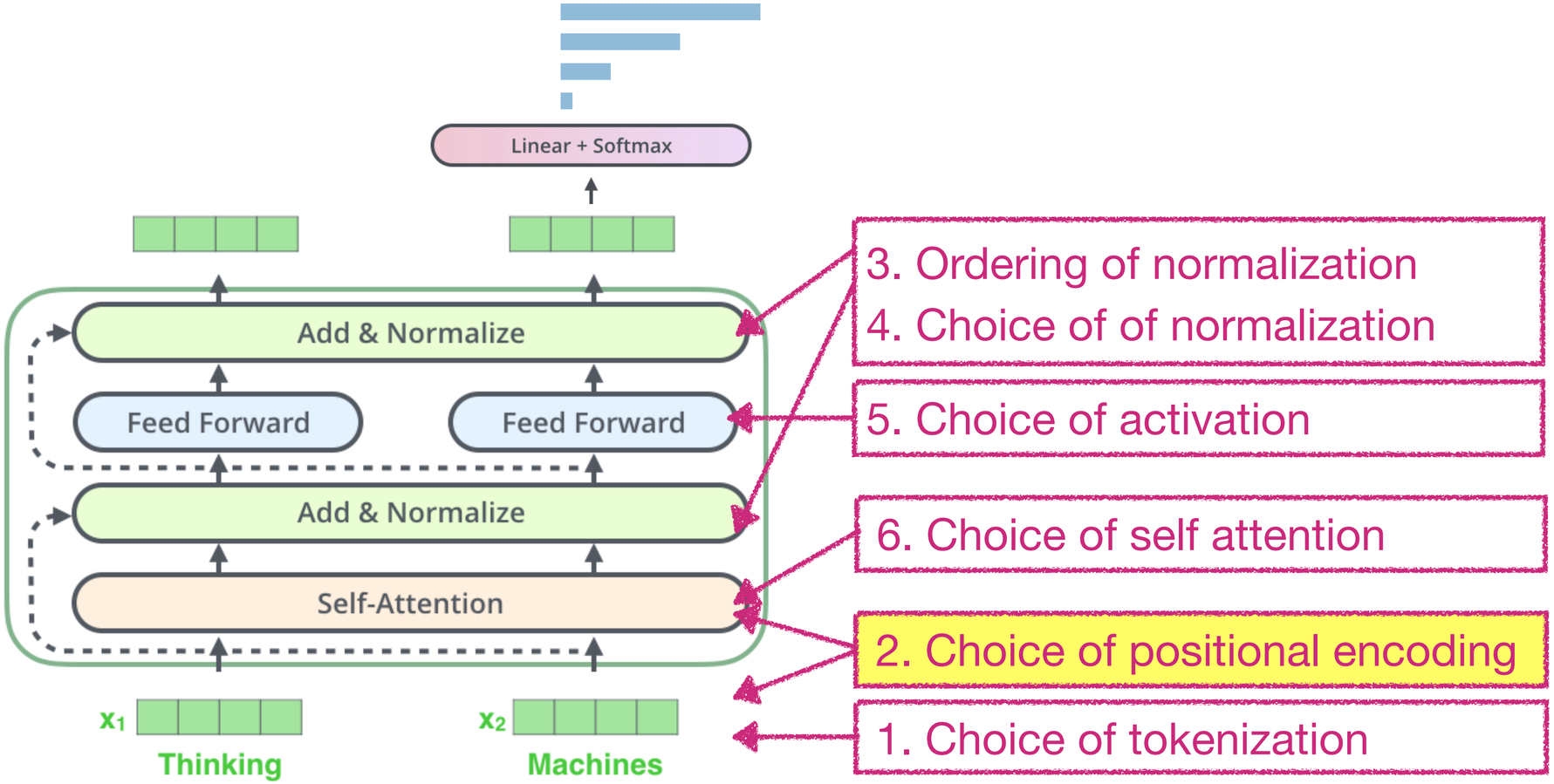
Speed and Simplicity

- WordPiece: Calculating likelihoods is more expensive
- BPE is a simple "count and merge" algorithm

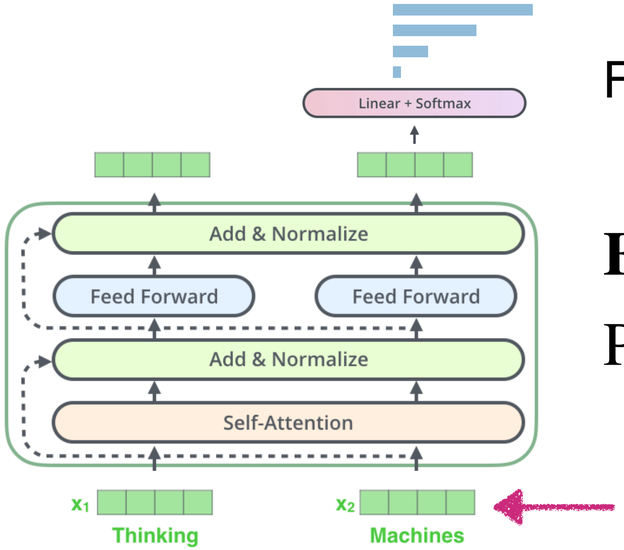
Token efficiency

- BPE = compression algorithm — it packs the most data into the fewest tokens
- Can generate the same text with fewer tokens = more efficiently

Transformers variants



Recap: Positional encoding



For a token x at a position p in the sequence, the input layer outputs

$$\mathbf{E}_x + \mathrm{PE}(p) \in \mathbb{R}^d,$$

$\mathbf{E}_x$: x's word embedding

$\mathrm{PE}(p)$: a positional encoding for the position p

Why? Otherwise, self-attention is **"order-blind"**, as mathematically it is **permutation invariant!**

- A: "The dog bit the man."
- B: "The man bit the dog."
- RNNs know whether "dog" came after "man"
- Self-Attention doesn't distinguish them

Positional encodings: Variants

Original Transformers: Sinusoidal Encoding

- Also called sine embeddings
- Add sines and cosines that enable localization
- Properties
  - Every position has a unique signature
  - Normalized range
  - Relative distance is preserved, e.g., the angle between position 1 and 2 is the same as the angle between position 5 and 6

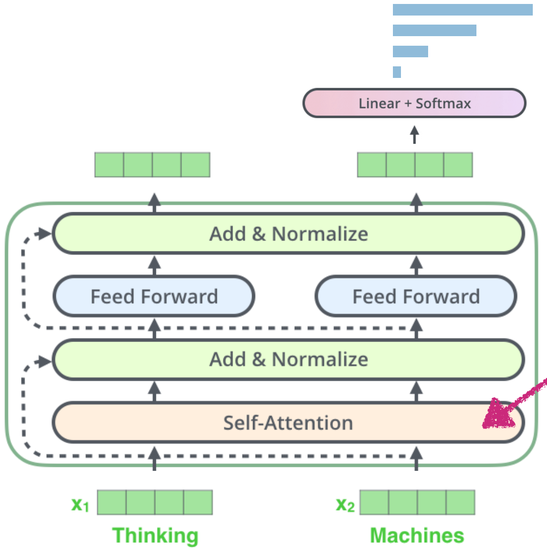
$$\mathrm{PE}(p)_{2i} = \sin\left(\frac{p}{10000^{\frac{2i}{d}}}\right)$$
$$\mathrm{PE}(p)_{2i+1} = \cos\left(\frac{p}{10000^{\frac{2i}{d}}}\right)$$

Variant 1: Absolute embeddings: add a position vector to the embedding

Notable models: GPT-1/2/3, OPT

$$\mathrm{PE}(p) = \mathbf{u}_p$$

Positional encodings: Variants



Positional encoding

Modern variants:

- Add position encoding in the **attention computation (!!)**, and
- Model the **relative** positional information

Version 2: Shaw et al.

Notable models: XLNet, DeBERTa

Version 3: Relative bias (also called T5 Relative Bias)

Notable models: T5, Gopher, Chinchilla

Version 4: RoPE

Notable models: GPT-J, PaLM, Llama, most 2024+ models

Shaw et al. & T5 relative bias

$\alpha_{i,j}$: attention distribution between the i-th and j-th token

Original Transformers

$$\alpha_{i,j} = \text{softmax}\left(\frac{\mathbf{q}_i \mathbf{k}_j^\intercal}{\sqrt{d_k}}\right)$$

Shaw et al.

$$\alpha_{i,j} = \text{softmax}\left(\frac{\mathbf{q}_i (\mathbf{k}_j + a_{i,j})^\intercal}{\sqrt{d_k}}\right)$$

T5 relative bias

$$\alpha_{i,j} = \text{softmax}\left(\frac{\mathbf{q}_i \mathbf{k}_j^\intercal}{\sqrt{d_k}} + b_{i-j}\right)$$

- $a_{i,j}$, b_{i-j}: A learned scalar representing the relative distance between the i-th and j-th token
- Shaw et al. came out first (more expressive); later people found the relative bias is enough (faster)
- Intuition: the model learns specifically for distances. For instance, the bias for "next neighbor" ($i - j = 1$) is a different learned number than "ten tokens away" ($i - j = 10$)
- Purely additive → Biggest different from RoPE (next slide)

RoPE: Rotary positional encodings

High level thought process:

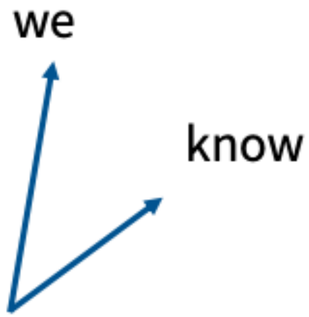
- We want to apply positional encoding to each input while keeping the same attention computation.
- A relative positional encoding should be some $f(x, i)$ s.t.

$$< f(x, i), f(y, j) > = g(x, y, i - j)$$

- That is, the attention function only gets to depend on the relative position $i - j$; invariant to absolute position
- Example:
    - S1: “we know that …”
    - S2: “Of course, we know that …”
    - The relationship between “we” and “know” should be the same in both sentences

RoPE: Rotary positional encodings

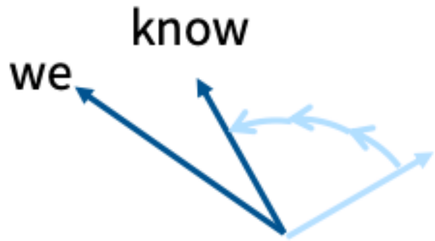
- Rotation satisfies these conditions!



Position independent embeddings

Embedding "we know that"

(Rotate we by "0 positions" and know by "1 positions")



Embedding "of course we know"

(Rotate we by "2 positions" and know by "3 positions")

RoPE: Rotary positional encodings

- RoPE: Actual math — assuming a 2D vector (x_1, x_2)

$$\text{RoPE}(x, i) = \begin{pmatrix} \cos(i\theta) & -\sin(i\theta) \\ \sin(i\theta) & \cos(i\theta) \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}, \text{ where } \theta = 10000^{-\frac{2i}{d}}$$

- Then, if we take a dot product of a rotated query and a rotated key

$$\langle \text{RoPE}(q, j), \text{RoPE}(k, i) \rangle = q^\top \mathscr{R}_{i-j} k$$

- Multiplicative/rotational.

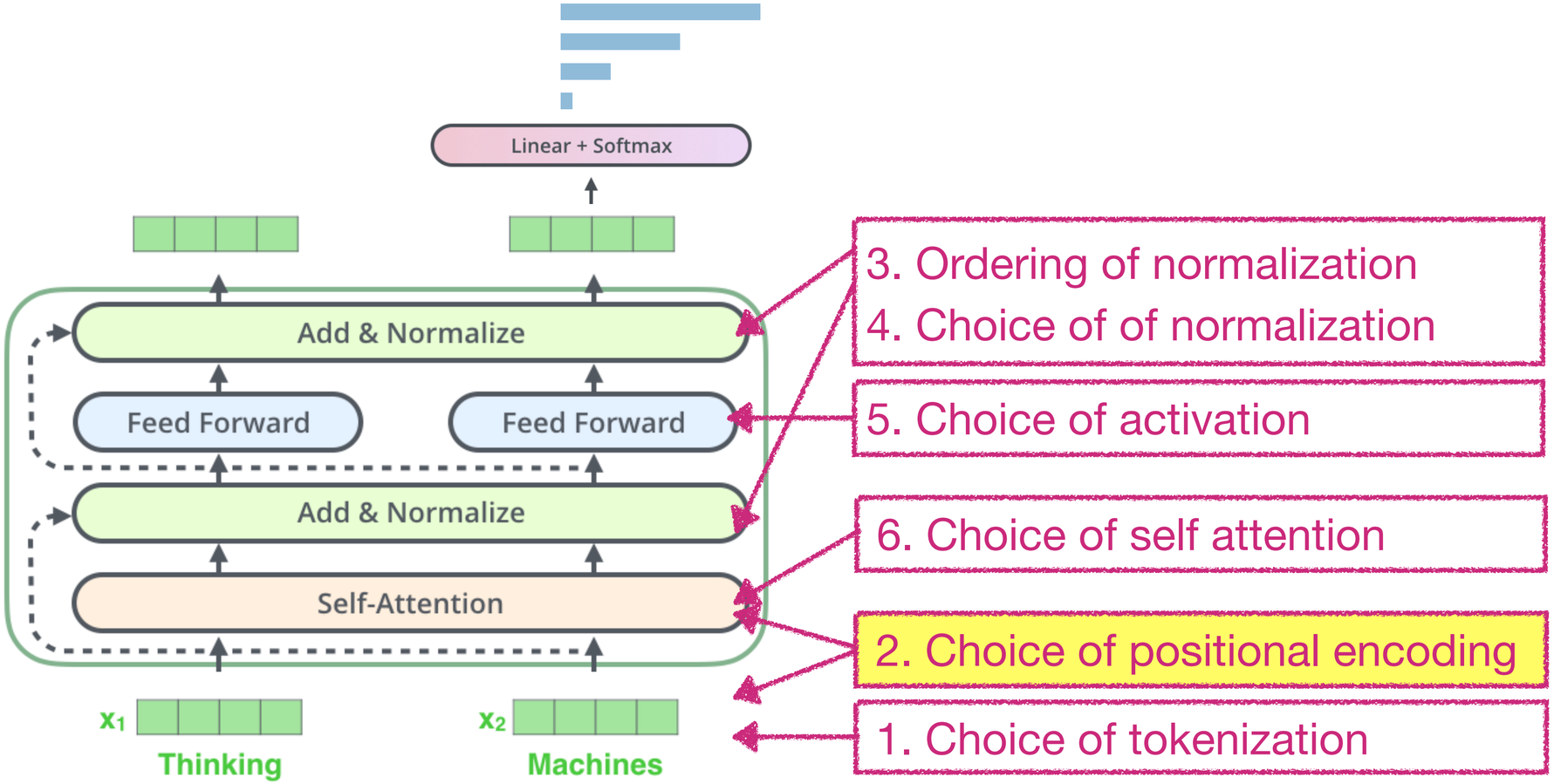
Satisfies our requirement:
$< f(x, i), f(y, j) > = g(x, y, i - j)$

RoPE: Rotary positional encodings

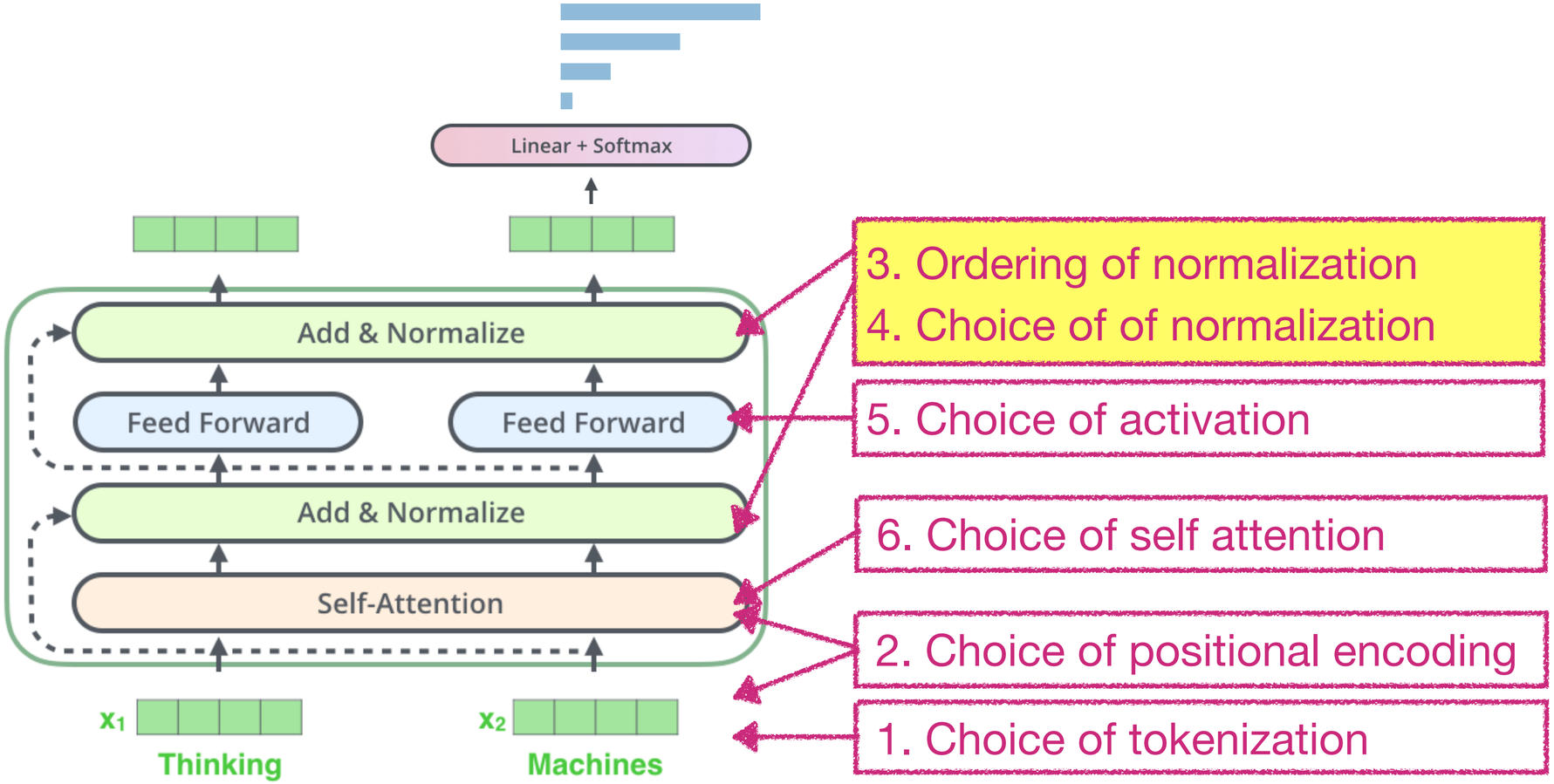
- Actual math — now, with a d-dimensional vector

$$\boldsymbol{R}^d_{\Theta,m} = \begin{pmatrix} \cos m\theta_1 & -\sin m\theta_1 & 0 & 0 & \cdots & 0 & 0 \\ \sin m\theta_1 & \cos m\theta_1 & 0 & 0 & \cdots & 0 & 0 \\ 0 & 0 & \cos m\theta_2 & -\sin m\theta_2 & \cdots & 0 & 0 \\ 0 & 0 & \sin m\theta_2 & \cos m\theta_2 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & \cos m\theta_{d/2} & -\sin m\theta_{d/2} \\ 0 & 0 & 0 & 0 & \cdots & \sin m\theta_{d/2} & \cos m\theta_{d/2} \end{pmatrix}$$

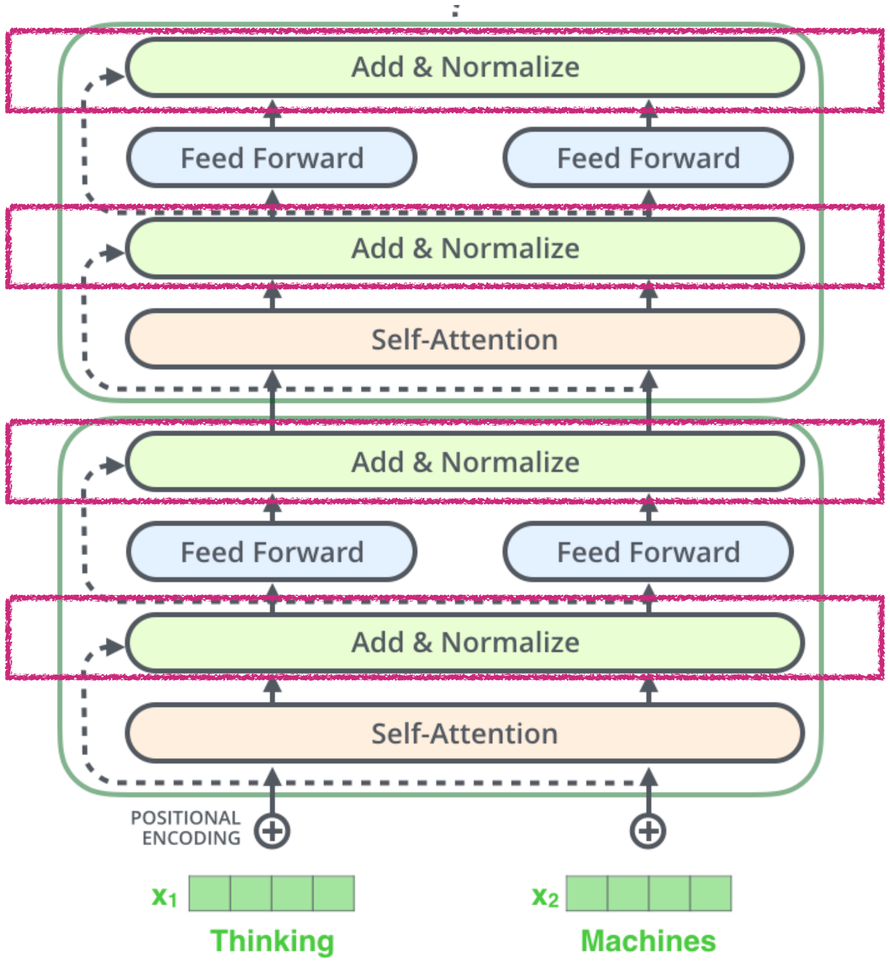
Transformers variants



Transformers variants



Recap: Normalization



Normalizes the outputs to be within a consistent range, preventing too much variance in scale of outputs

Ordering of normalization matters

Post-norm: MHA → Add→Norm → FFN → Add → Norm

- Notable models: Original Transformers, BERT, RoBERTa, etc

Pre-norm: Norm → MHA → Add → Norm → FFN → Add

- Notable models: GPT-2 and GPT-3, Llama, Mistral/Mixtral, PaLM, etc (Most modern LLMs)

Original stated advantage: Removing warmup

Today: stability and larger learning rates for large networks

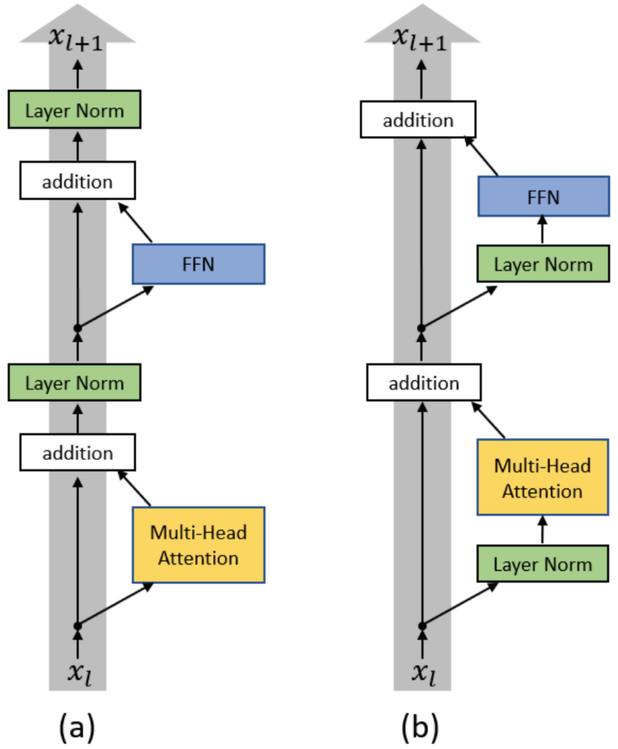


Figure 1. (a) Post-LN Transformer layer; (b) Pre-LN Transformer layer.

Figure from Xiong et al. 2020

Post-norm vs. Pre-nom

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

    def __init__(self, d_model, d_ff, num_heads, dropout):
        super().__init__()
        self.mha = MultiHeadAttention(num_heads, d_model, dropout)
        self.ff = nn.Linear(d_model, d_model)
        self.norm1 = nn.LayerNorm(d_model)
        self.norm2 = nn.LayerNorm(d_model)

    def forward(self, x, mask):
        # Multi-head attention + residual connection
        x = x + self.mha(x, x, x, mask)
        # Post-norm
        x = self.norm1(x)
        # Feed-forward + residual connection
        x = x + self.ff(x)
        # Post-norm
        x = self.norm2(x)
        return x
```

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

    def __init__(self, d_model, d_ff, num_heads, dropout):
        super().__init__()
        self.mha = MultiHeadAttention(num_heads, d_model, dropout)
        self.ff = nn.Linear(d_model, d_model)
        self.norm1 = nn.LayerNorm(d_model)
        self.norm2 = nn.LayerNorm(d_model)

    def forward(self, x, mask):
        # Pre-norm
        x = self.norm1(x)
        # Multi-head attention + residual connection
        x = x + self.mha(x, x, x, mask)
        # Pre-norm
        x = self.norm2(x)
        # Feed-forward + residual connection
        x = x + self.ff(x)
        return x
```

The choice of normalization matters

Original Transformers: LayerNorm — normalizes the mean and variance across d_{model}

$$y = \frac{x - \mathbb{E}[x]}{\sqrt{\text{Var}[x] + \epsilon}} \times \gamma + \beta$$

Notable models: GPT3/2/1, OPT, GPT-J, BLOOM

Most modern LMs: RMSNorm — does not subtract mean or add a bias term

$$y = \frac{x}{\sqrt{|x|_2^2 + \epsilon}} \times \gamma$$

Notable models: Llama family, PaLM, Chinchilla, T5

LayerNorm v.s RMSNorm

$$y = \frac{x - \mathbb{E}[x]}{\sqrt{\mathrm{Var}[x] + \epsilon}} \times \gamma + \beta \quad \text{vs.} \quad y = \frac{x}{\sqrt{|x|_2^2 + \epsilon}} \times \gamma$$

Modern explanation — it's faster (and just as good)

- Fewer operation (no mean calculation)
- Fewer parameters (no bias term to store)

Q: Isn't stat normalization only a tiny fraction of FLOPS?

Table 1. Proportions for operator classes in PyTorch.

Operator class	% flop	% Runtime
△ Tensor contraction	99.80	61.0
□ Stat. normalization	0.17	25.5
○ Element-wise	0.03	13.5

Ivanov et al. 2020. "Data Movement Is All You Need: A Case Study on Optimizing Transformers"

LayerNorm v.s RMSNorm

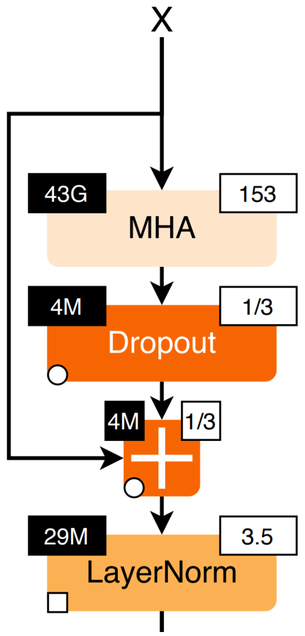
Q: Isn't stat normalization only a tiny fraction of FLOPS?

Table 1. Proportions for operator classes in PyTorch.

Operator class	% flop	% Runtime
△ Tensor contraction	99.80	61.0
□ Stat. normalization	0.17	25.5
○ Element-wise	0.03	13.5

A: FLOPs are not runtime!

RMSNorm can still matter a lot due to the importance of data movement.



Left top ("43G") is FLOPs
Right top ("153") is FLOP-to-memory ratio

Ivanov et al. 2020. "Data Movement Is All You Need: A Case Study on Optimizing Transformers"

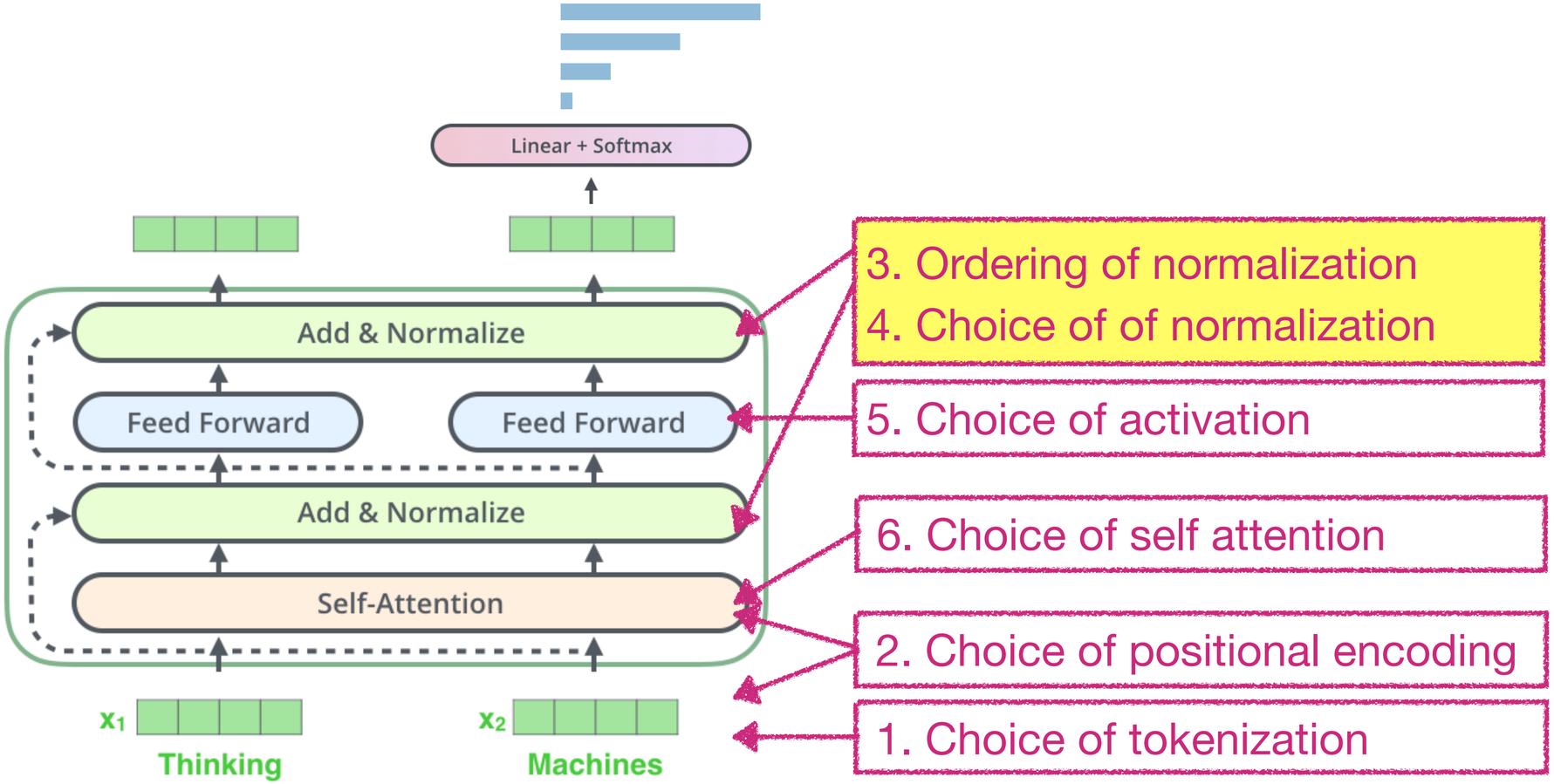
LayerNorm v.s RMSNorm

RMSNorm runtime (and surprisingly, perf) gains have been seen in papers.

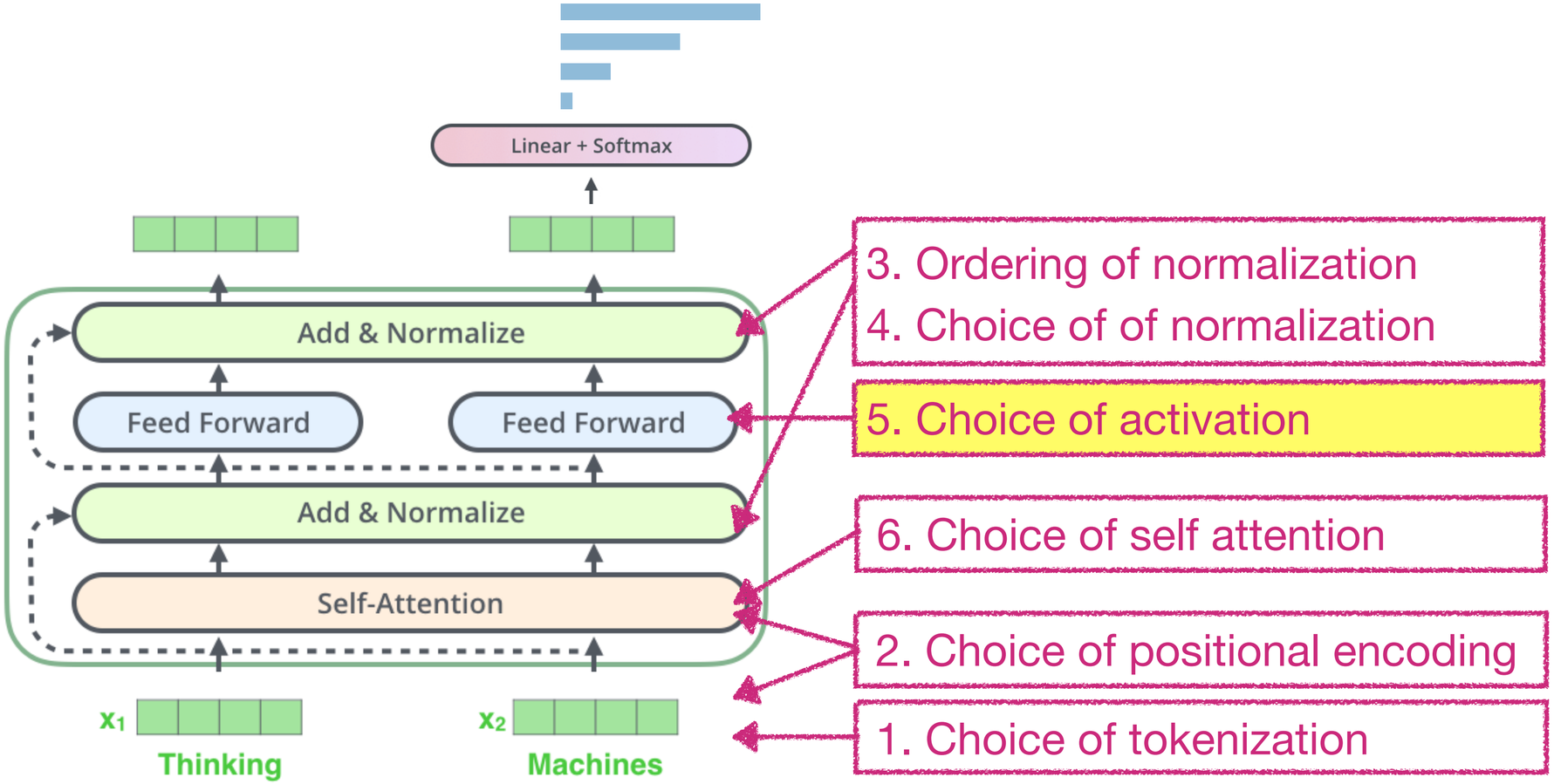
Model	Params	Ops	Step/s	Early loss	Final loss	SGLUE	XSum	WebQ	WMT EnDe
Vanilla Transformer	$223M$	$11.1T$	3.50	2.182 ± 0.005	1.838	71.66	17.78	23.02	26.62
RMS Norm	$223M$	$11.1T$	3.68	2.167 ± 0.008	**1.821**	**75.45**	**17.94**	**24.07**	**27.14**
Rezero	$223M$	$11.1T$	3.51	2.262 ± 0.003	1.939	61.69	15.64	20.90	26.37
Rezero + LayerNorm	$223M$	$11.1T$	3.26	2.223 ± 0.006	1.858	70.42	17.58	23.02	26.29
Rezero + RMS Norm	$223M$	$11.1T$	3.34	2.221 ± 0.009	1.875	70.33	17.32	23.02	26.19
Fixup	$223M$	$11.1T$	2.95	2.382 ± 0.012	2.067	58.56	14.42	23.02	26.31

Narang et al. 2021. "Do Transformer Modifications Transfer Across Implementations and Applications?"

Transformers variants



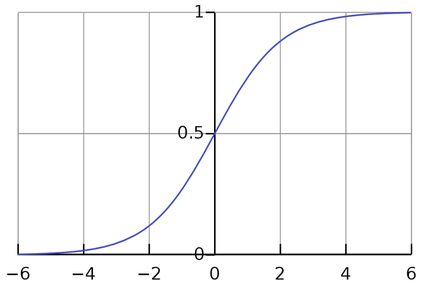
Transformers variants



Activations

sigmoid

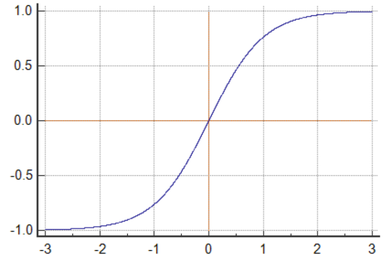
$$f(z) = \frac{1}{1 + e^{-z}}$$



$$f'(z) = f(z) \times (1 - f(z))$$

tanh

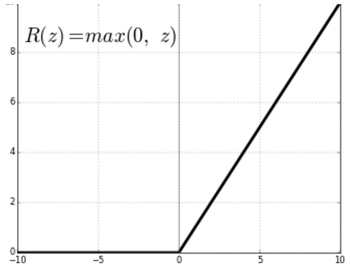
$$f(z) = \frac{e^{2z} - 1}{e^{2z} + 1}$$



$$f'(z) = 1 - f(z)^2$$

ReLU
(rectified linear unit)

$$f(z) = \max(0, z)$$



$$f'(z) = \begin{cases} 1 & z > 0 \\ 0 & z < 0 \end{cases}$$

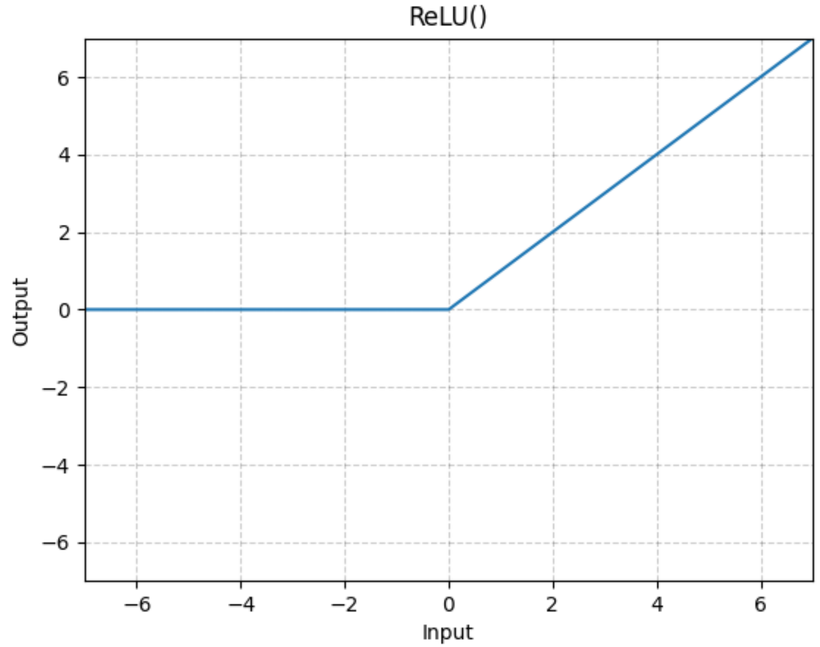
- This is what we learned a couple of weeks ago.
- Today's LLM papers
  - ReLU, GeLU, SiLU, ELU, GLU, GeGLU, ReGLU, SeLU, SwiGLU, LiGLU, …
- **What are these?? What do people use?? Does it matter at all?**

Activations: RELU

RELU (Rectified Linear Units function)

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

Notable models: Original Transformers, T5, Gopher, Chinchilla, OPT



Activations: GELU

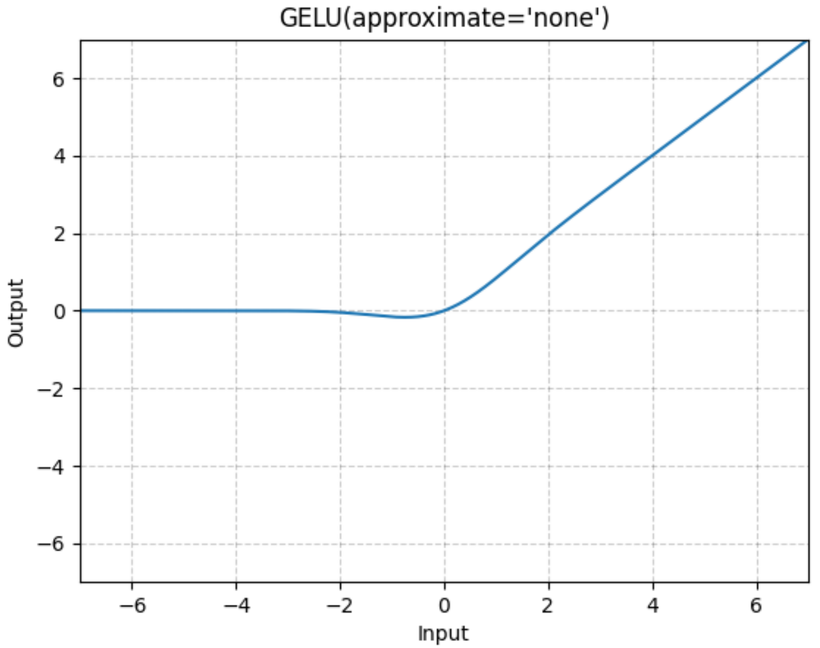
GELU (Gaussian Error Linear Units function)

$$\text{GELU}(x) = x \times \Phi(x)$$

Where $\Phi(x)$ is the Cumulative Distribution Function for Gaussian Distribution

Notable models: GPT1/2/3, GPT-J, GPT-NoeX, BLOOM

Why? Smooth and differentiable. Negative inputs are scaled (not killed completely), Gradient flow never fully dies (In ReLU, can die)

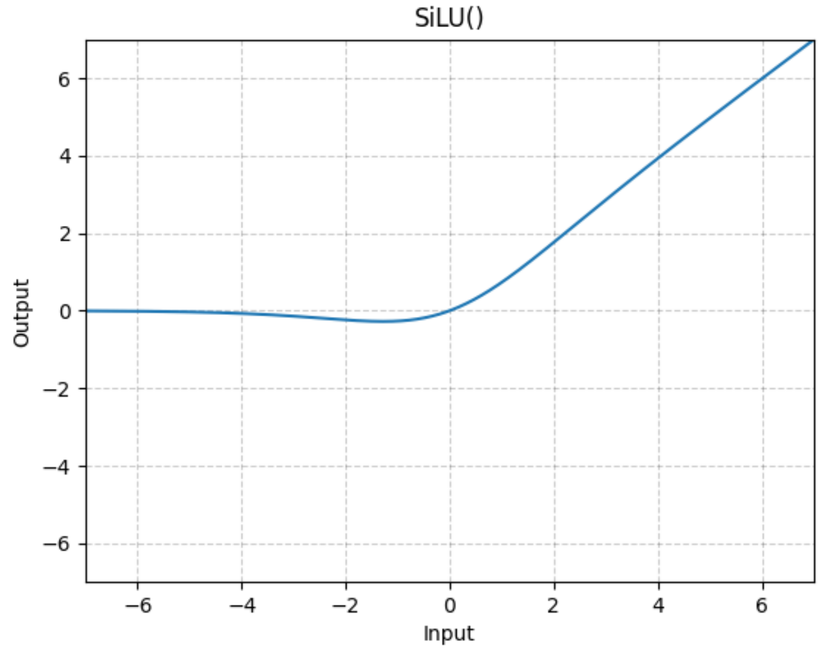


Activations: SiLU

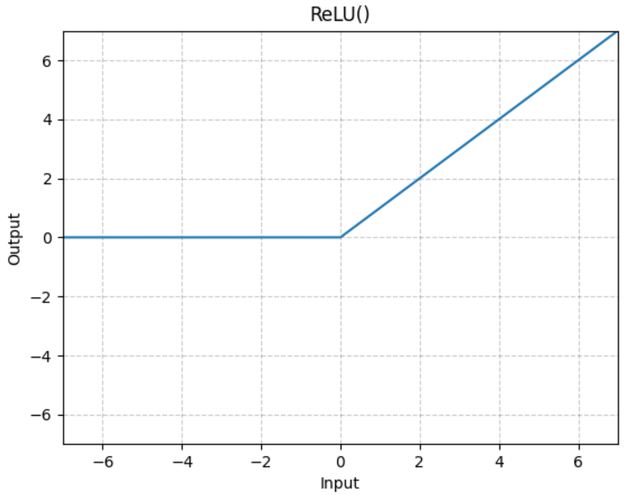
SiLU (Sigmoid Linear Units function)

$$\text{SILU}(x) = x \times \sigma(x)$$

Why? Smooth and differentiable. Negative inputs are scaled (not killed completely), Gradient flow never fully dies (In ReLU, can die)

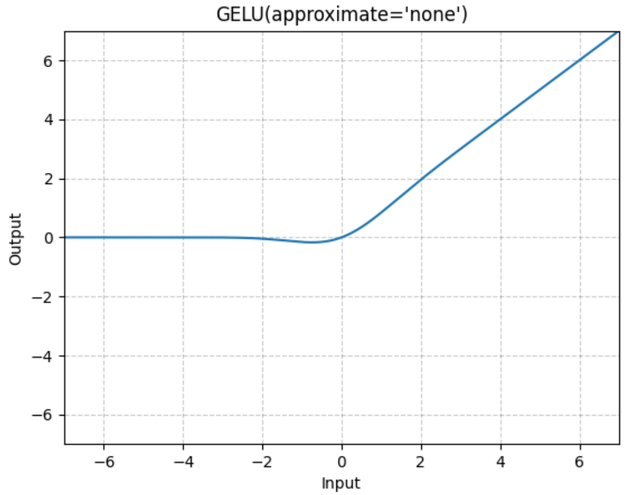


Activations: RELU, GELU, SILU



RELU (Rectified Linear Units function)

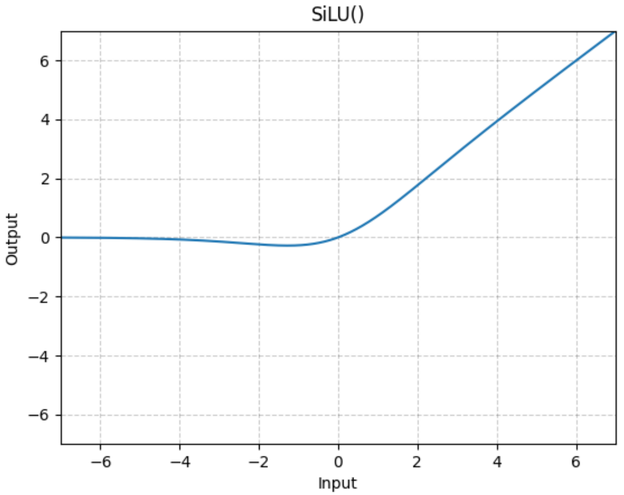
$$\text{RELU}(x) = \max(0,x)$$



GELU (Gaussian Error Linear Units function)

$$\text{GELU}(x) = x \times \Phi(x)$$

$\Phi(x)$: Cumulative Distribution Function for Gaussian Distribution



SiLU (Sigmoid Linear Units function)

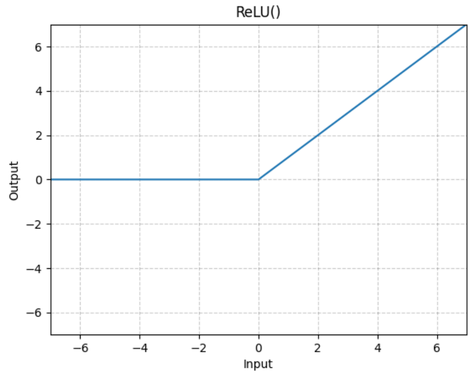
$$\text{SILU}(x) = x \times \sigma(x)$$

“Gated” activations

Augment {RELU, GELU, SILU} with an (entrywise) linear term

$$\max(0, xW_1) \rightarrow \max(0, xW_1) \odot xV$$

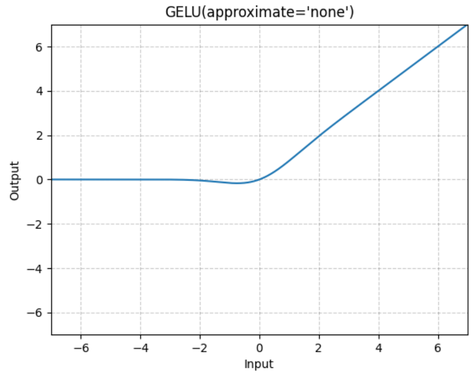
(Now we have an extra parameter V)



RELU (Rectified Linear Units function)

↓

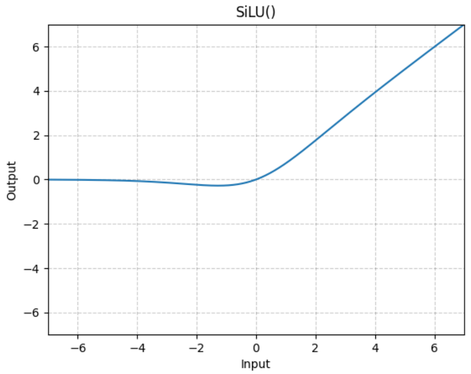
ReGLU (Rectified Gated Linear Units function)



GELU (Gaussian Error Linear Units function)

↓

GeGLU (Gaussian Error Gated Linear Units function)



SiLU (Sigmoid Linear Units function)

↓

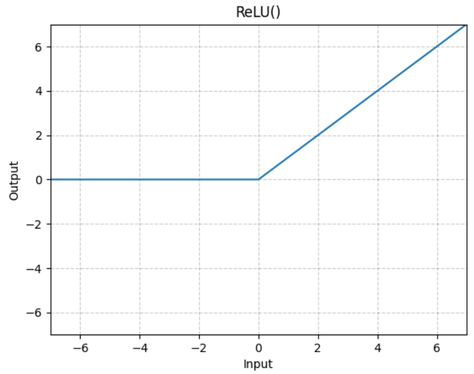
SwiGLU (Swish Gated Linear Unit)

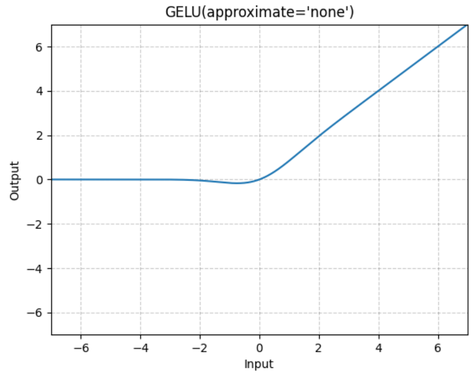
“Gated” activations

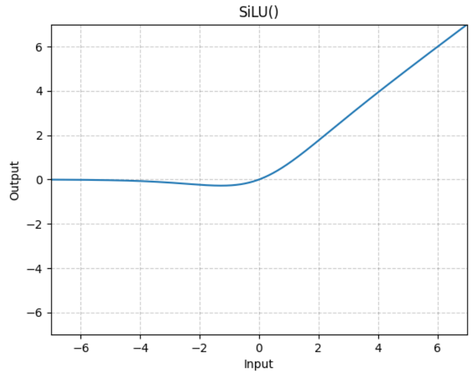
Augment {RELU, GELU, SILU} with an (entrywise) linear term

$$\max(0, xW_1) \rightarrow \max(0, xW_1) \odot xV$$

(Now we have an extra parameter V)







$$\text{FFN}_{\text{RELU}}(x) = \text{RELU}(xW_1)W_2$$

$$\text{FFN}_{\text{ReGLU}}(x) = \left(\text{RELU}(xW_1) \odot xV\right) W_2$$

$$\text{FFN}_{\text{GELU}}(x) = \text{GELU}(xW_1)W_2$$

$$\text{FFN}_{\text{GeGLU}}(x) = \left(\text{GELU}(xW_1) \odot xV\right) W_2$$

(Notable models: T5 v1.1, mT5, LaMDA, Phi3, Gemma 2, Gemma 3)

$$\text{FFN}_{\text{SILU}}(x) = \text{SILU}(xW_1)W_2$$

$$\text{FFN}_{\text{SwiGLU}}(x) = \left(\text{SILU}(xW_1) \odot xV\right) W_2$$

(Notable models: Llama 1/2/3, PaLM, Mistral, Olmo, GPT-OSS, most models post 2023)

“Gated” activations

Results from Shazeer 2020 that introduced SwiGLU

4 Conclusions

We have extended the GLU family of layers and proposed their use in Transformer. In a transfer-learning setup, the new variants seem to produce better perplexities for the de-noising objective used in pre-training, as well as better results on many downstream language-understanding tasks. These architectures are simple to implement, and have no apparent computational drawbacks. We offer no explanation as to why these architectures seem to work; we attribute their success, as all else, to divine benevolence.

On SuperGLUE

On SQuAD

	Score Average	EM	F1
FFN_{ReLU}	72.76	83.18	90.87
FFN_{GELU}	72.98	83.09	90.79
$\text{FFN}_{\text{Swish}}$	72.40	83.25	90.76
FFN_{GLU}	73.95	82.88	90.69
$\text{FFN}_{\text{GEGLU}}$	73.96	83.55	91.12
$\text{FFN}_{\text{Bilinear}}$	73.81	**83.82**	91.06
$\text{FFN}_{\text{SwiGLU}}$	**74.56**	83.42	91.03
$\text{FFN}_{\text{ReGLU}}$	73.66	83.53	**91.18**
[Raffel et al., 2019]	71.36	80.88	88.81
ibid. stddev.	0.416	0.343	0.226

Shazeer et al. 2020. "GLU Variants Improve Transformer"

“Gated” activations

Other work confirming Shazeer 2020.

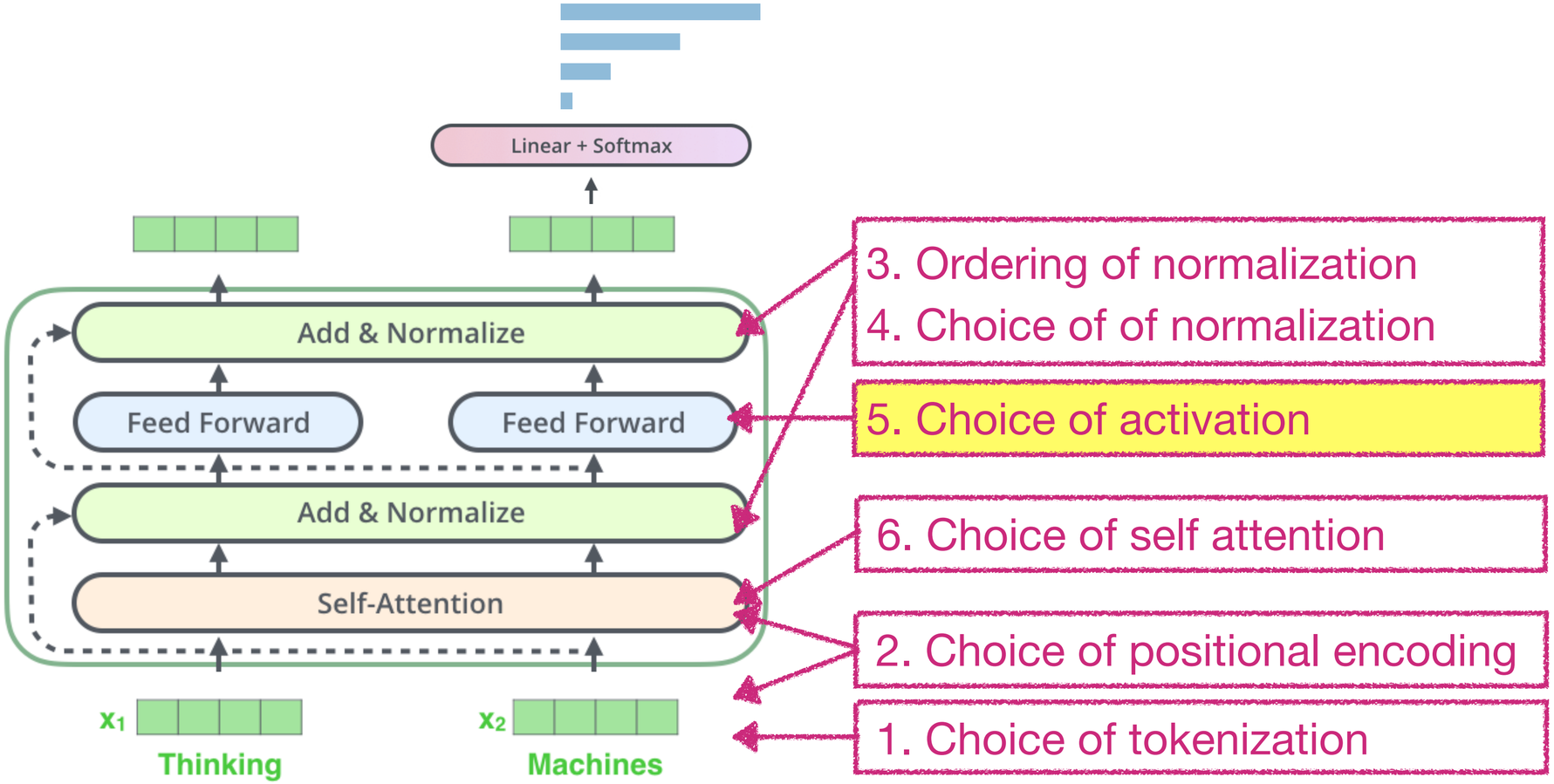
Model	Params	Ops	Step/s	Early loss	Final loss	SGLUE	XSum	WebQ	WMT EnDe
Vanilla Transformer	$223M$	$11.1T$	3.50	2.182 ± 0.005	1.838	71.66	17.78	23.02	26.62
GeLU	$223M$	$11.1T$	3.58	2.179 ± 0.003	1.838	**75.79**	**17.86**	**25.13**	26.47
Swish	$223M$	$11.1T$	3.62	2.186 ± 0.003	1.847	**73.77**	17.74	**24.34**	**26.75**
ELU	$223M$	$11.1T$	3.56	2.270 ± 0.007	1.932	67.83	16.73	23.02	26.08
GLU	$223M$	$11.1T$	3.59	2.174 ± 0.003	**1.814**	**74.20**	**17.42**	24.34	**27.12**
GeGLU	$223M$	$11.1T$	3.55	2.130 ± 0.006	**1.792**	**75.96**	**18.27**	**24.87**	**26.87**
ReGLU	$223M$	$11.1T$	3.57	2.145 ± 0.004	**1.803**	**76.17**	**18.36**	**24.87**	**27.02**
SeLU	$223M$	$11.1T$	3.55	2.315 ± 0.004	1.948	68.76	16.76	22.75	25.99
SwiGLU	$223M$	$11.1T$	3.53	2.127 ± 0.003	**1.789**	**76.00**	**18.20**	**24.34**	**27.02**
LiGLU	$223M$	$11.1T$	3.59	2.149 ± 0.005	**1.798**	**75.34**	**17.97**	**24.34**	26.53
Sigmoid	$223M$	$11.1T$	3.63	2.291 ± 0.019	1.867	**74.31**	17.51	23.02	26.30
Softplus	$223M$	$11.1T$	3.47	2.207 ± 0.011	1.850	**72.45**	17.65	**24.34**	**26.89**

Narang et al. 2021. "Do Transformer Modifications Transfer Across Implementations and Applications?"

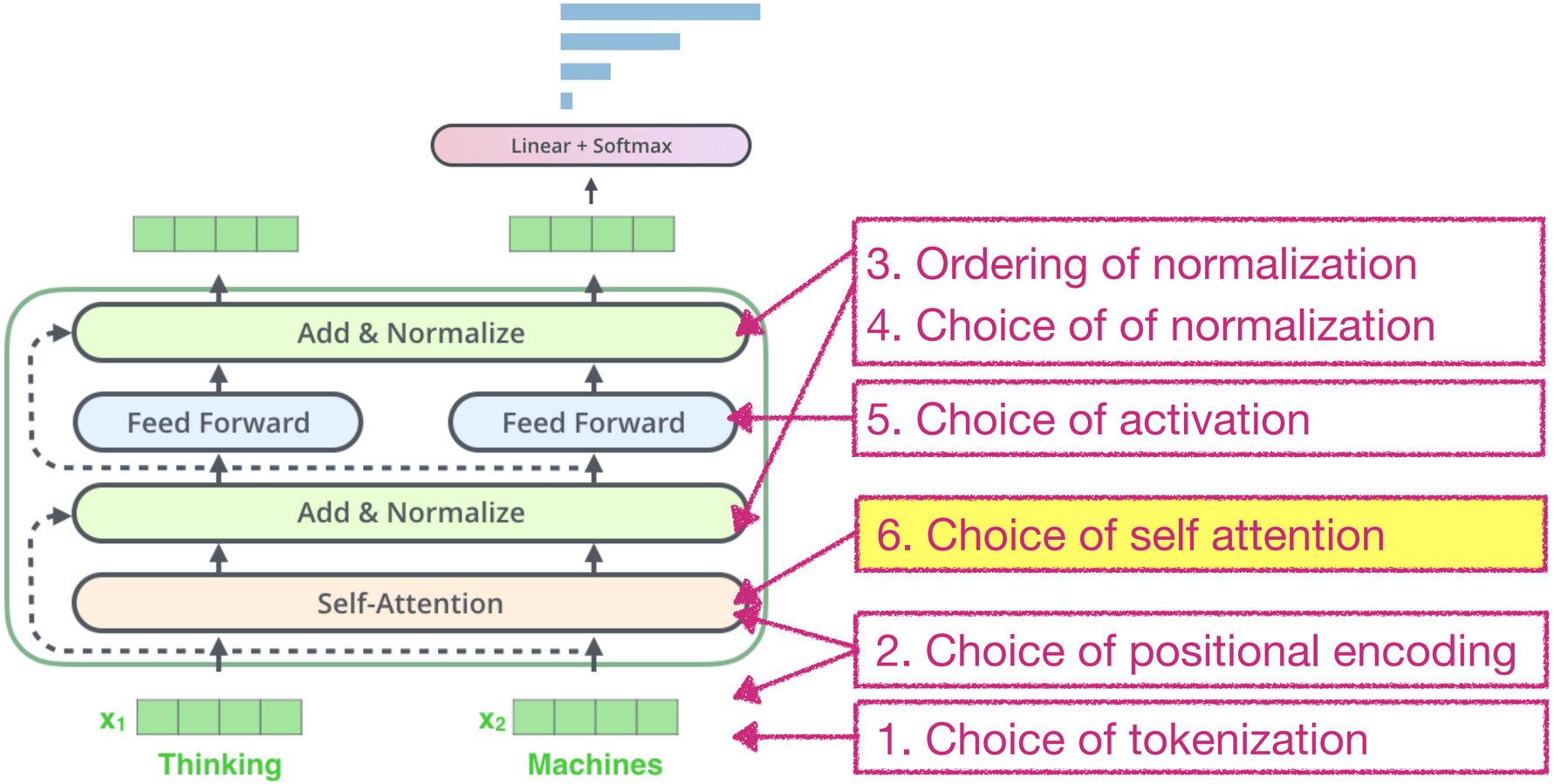
Activations: Summary

- Many variations (ReLU, GeLU, *GLU) across models.
- *GLU isn’t necessary for a good model (e.g., GPT-3), but it’s probably helpful.
- Evidence points toward somewhat consistent gains from Swi/GeGLU.

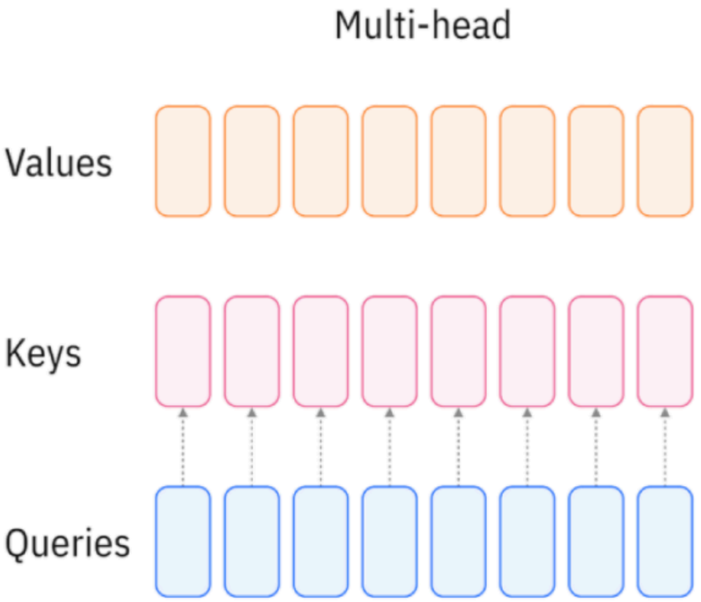
Transformers variants



Transformers variants



Multi-head attention

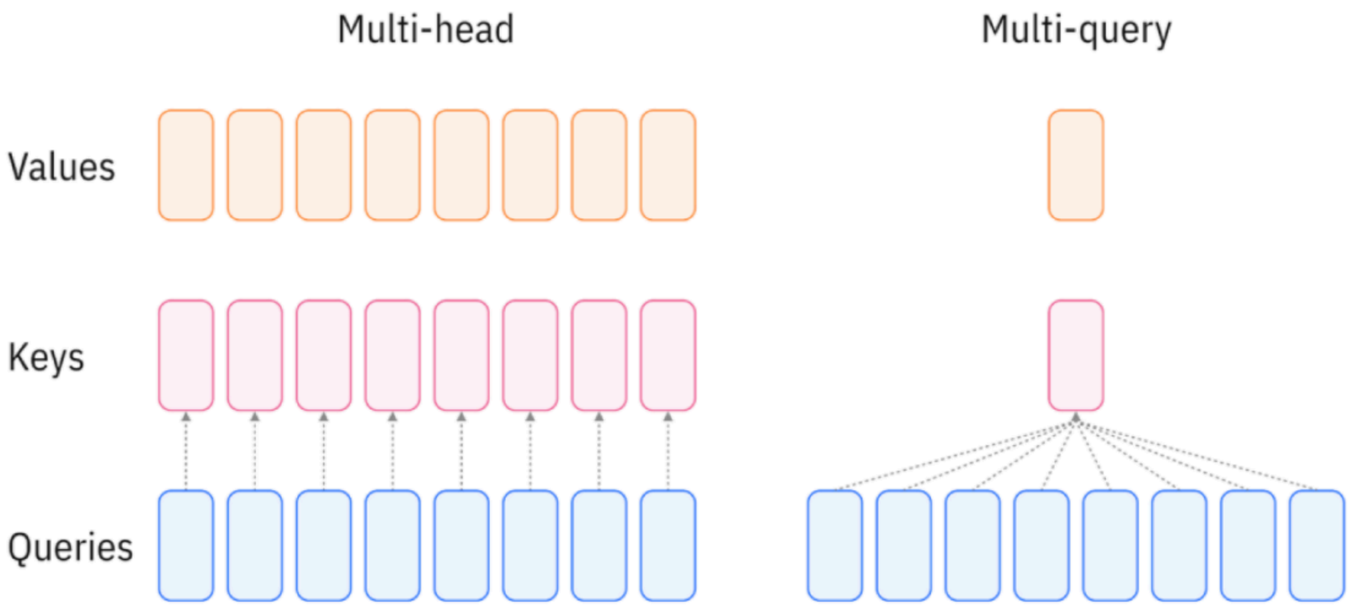


Even though MHA has less FLOPs, it can still be a bottleneck for runtime due to **memory bandwidth bound**

- During generation, GPUs repeatedly load the KV cache from HBM; the GPU processor needs to wait. This limits tokens/sec seen by the users.

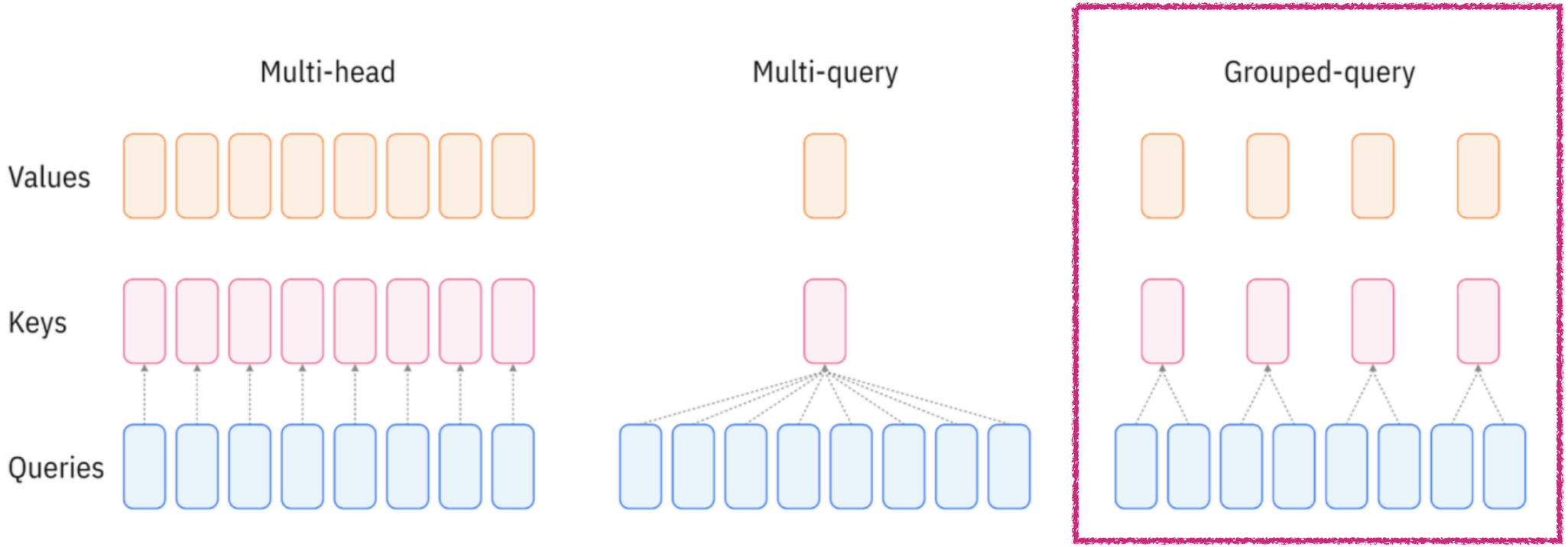
Image credit: www.ibm.com/think/topics/grouped-query-attention

Multi-Query Attention



Shazeer. 2019. "Fast Transformer Decoding: One Write-Head is All You Need"

Grouped-Query Attention



- Shares key and value heads for each ***group*** of query heads
- Saves memory, which leads to faster inference

Shazeer. 2019. "Fast Transformer Decoding: One Write-Head is All You Need"

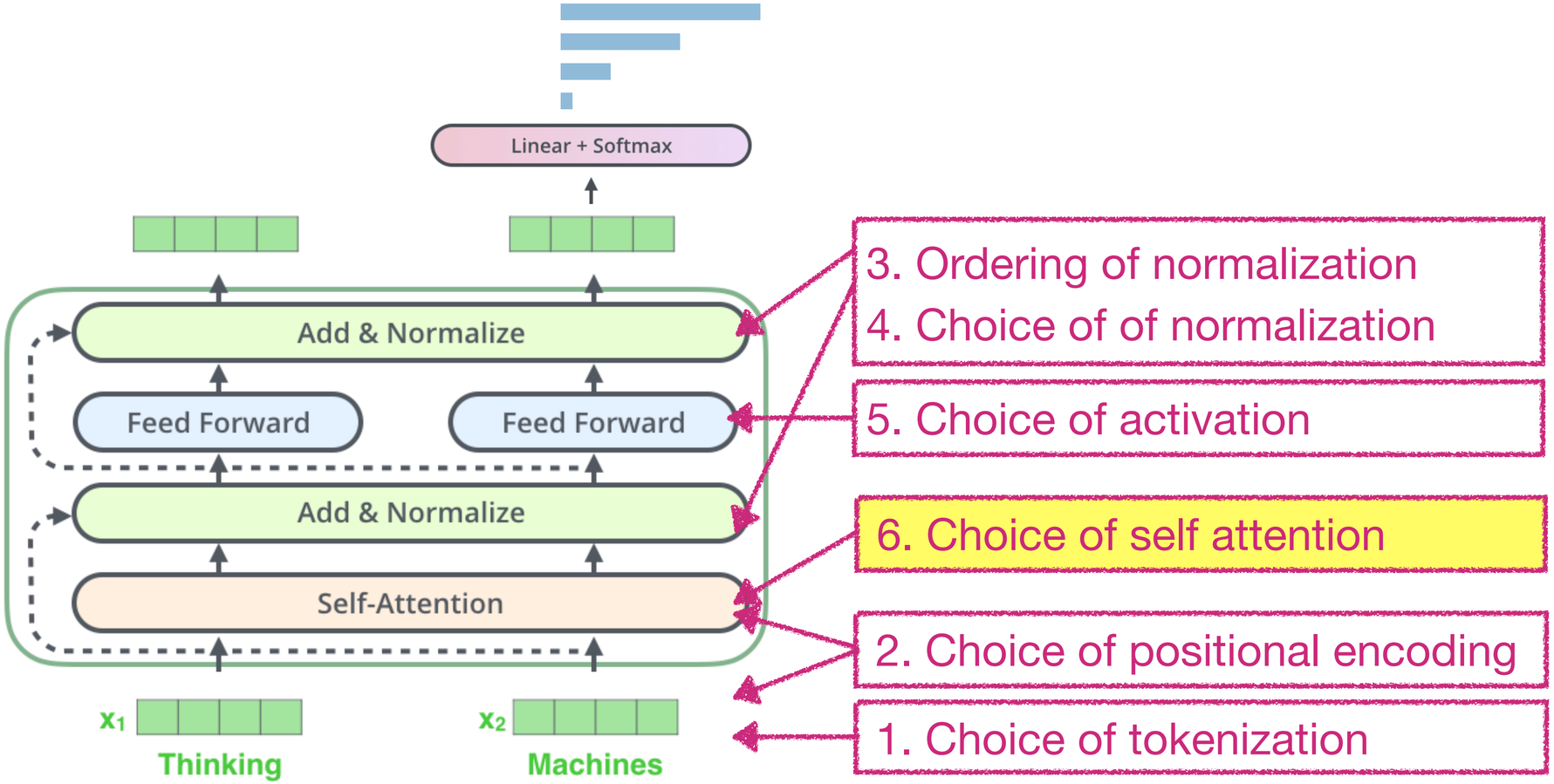
Grouped-Query Attention in code

```
bsz, seqlen, _ = x.shape
xq, xk, xv = self.wq(x), self.wk(x), self.wv(x)

xq = xq.view(bsz, seqlen, self.n_local_heads, self.head_dim)
xk = xk.view(bsz, seqlen, self.n_local_kv_heads, self.head_dim)
xv = xv.view(bsz, seqlen, self.n_local_kv_heads, self.head_dim)
```

```
# repeat k/v heads if n_kv_heads < n_heads
keys = repeat_kv(keys, self.n_rep)  # (bs, cache_len + seqlen, n_local_heads, head_dim)
values = repeat_kv(values, self.n_rep)  # (bs, cache_len + seqlen, n_local_heads, head_dim)
```


https://github.com/meta-llama/llama/blob/main/llama/model.py

Transformers variants



Transformers variants: Summary

- **Tokenization:** Byte Pair Encoding established as the standard
- **Positional Encoding**: Most do Rotary Position Embeddings (RoPE), used every attention layer (instead of just in the embedding layer)
- **Post-norm vs. Pre-norm**: Everyone does pre-norm, likely with good reasons
- **LayerNorm vs. RMSNorm**: RMSNorm has clear compute wins, sometimes even performance
- **Activations**: GLUs seem generally better, though differences are small
- **Attention:** Grouped-query attention instead of multi-head attention or multi-query attention

Original Transformers vs. Llama

	Vaswani et al.	Llama	Llama 2
Tokenization	WordPiece	BPE	BPE
Positional Encoding	Sinusoidal	RoPE	RoPE
Norm Position	Post-norm	Pre-norm	Pre-norm
Norm Type	LayerNorm	RMSNorm	RMSNorm
Activation	ReLU	SwiGLU	SwiGLU
Attention	Multi-head	Multi-head	Grouped-query

How much does it matter?

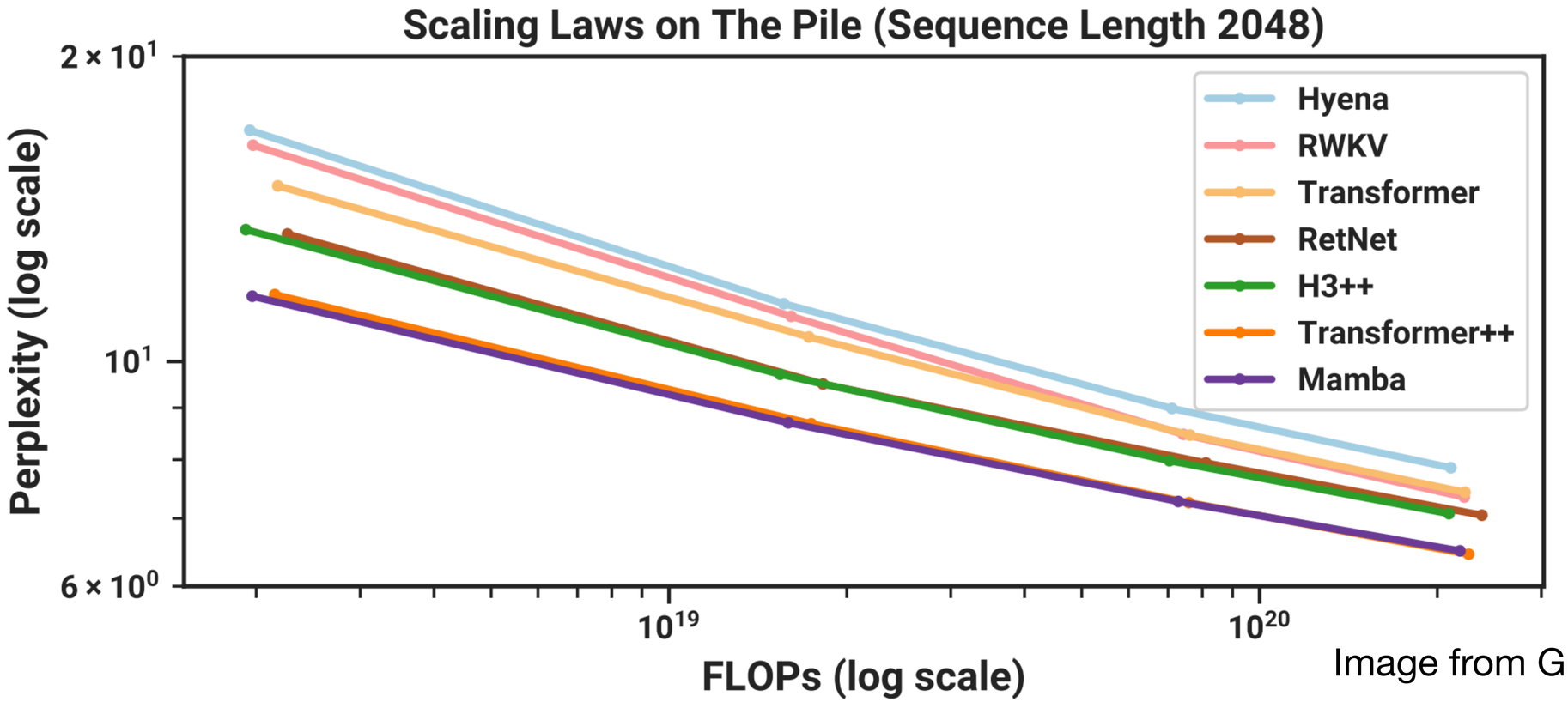


Image from Gu & Dao (2023)

- "Transformer" is Vaswani et al., "Transformer++" is (basically) Llama 2 (but with the same tokenizer)
- Stronger architecture is ~10x more efficient!

Open-ended question

Current LLMs are predominantly based on Transformer architectures.
How essential is it that LLMs use Transformers?

Open-ended question

Argument 1

In 2018, many industry labs tried to do LLM training but with LSTMs. They never worked.

Argument 2

dr. jack morris
@jxmnop

one of the most important things I know about deep learning I learned from this paper: "Pretraining Without Attention"

this what I found so surprising:
these people developed an architecture very different from Transformers called BiGS, spent months and months optimizing it and training different configurations, only to discover that **at the same parameter count, a wildly different architecture produces identical performance to transformers**

this may imply that as long as there are enough parameters, and things are reasonably well-conditioned (i.e. a decent number of nonlinearities and and connections between the pieces) then it really doesn't matter how you arrange them, i.e. any sufficiently good architecture works just fine

Current consensus (somewhat)

- The ***exact*** Transformers architecture may not be ***that*** important, as long as:
  - Enough parameter count
  - A reasonable degree of nonlinearity and other expressive components
  - Can be optimized stably (no exploding/vanishing gradients, etc)
- People have been working on this architecture search for decades (*manually*)
- And they found Transformers
  - The original Transformers didn't have all those properties — but people reached the current form through years of refinements ("twist search" still ongoing)
  - But the architecture does not have to be ***exactly*** this way

- Key question: Now there's architecture (originally for MT), but what enabled LLM?
  - Pre-training (Next lecture!)

Questions?

Acknowledgement

Princeton COS 484 by Danqi Chen, Tri Dao, Vikram Ramaswamy

CMU Advanced NLP by Graham Neubig & Sean Welleck

Stanford CS336 Language Modeling from Scratch by Tatsumori Hashimoto & Percy Liang