History

2014 ~ 2017 :

age of Reccurence (RNN, LSTM)

2021 ~

age of Transformer

2017s : paper << Attention is All You Need >>

↓

2021s : it had became an absolute mainstream

However, different Building blocks, the same goal :

deal with sequence-to-sequence task

Self-Attention

before Transformer discarding RNN/LSTM entirely and using self-attention

① concept

Self-Attention is not attention (learned in lecture 7) !

① Attention is a machanism based on RNN that connects two different modules (encoder and decoder) to solve information bottleneck

② Self-Attention is a machanism used to enhance the internal representation of a single sequence (can be used separately in encoders and decoders) to solve Linear interaction distance and parallelizability problem.

↓↓

Chain of logic

we find traditional encoder-decoder have bottleneck

↓

introduce attention to solve, and find it works well

↓↓

introduce self-attention to replace RNN/LSTM itself

↓↓

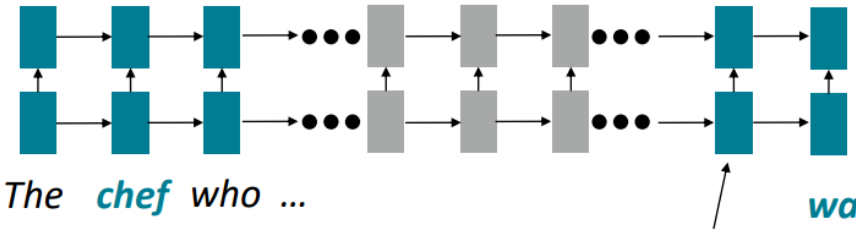
propose Transformer model

② Problems need to solve

⚠ RNNs take O(sequence length) steps for distant word pairs to interact

Issues with recurrent models: **Linear interaction distance**

- **O(sequence length)** steps for distant word pairs to interact means:
  - Hard to learn long-distance dependencies (because gradient problems!)
  - Linear order of words is "baked in"; we already know linear order isn't the right way to think about sentences…

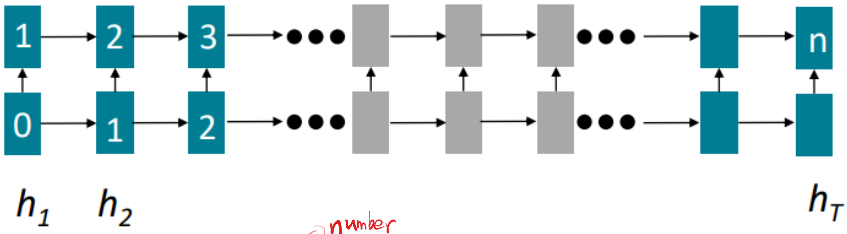


Info of ***chef*** has gone through O(sequence length) many layers!

⚠ Lack of Parallelizability

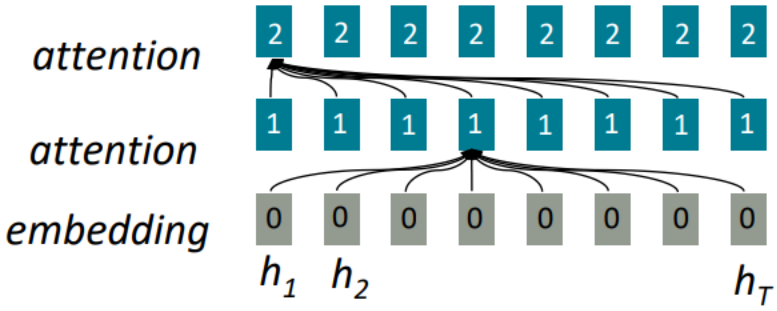
Issues with recurrent models: **Lack of parallelizability**

- Forward and backward passes have **O(sequence length)** unparallelizable operations
  - GPUs can perform a bunch of independent computations at once!
  - But future RNN hidden states can't be computed in full before past RNN hidden states have been computed
  - Inhibits training on very large datasets!



Numbers indicate min # of steps before a state can be computed

(3) Intuition



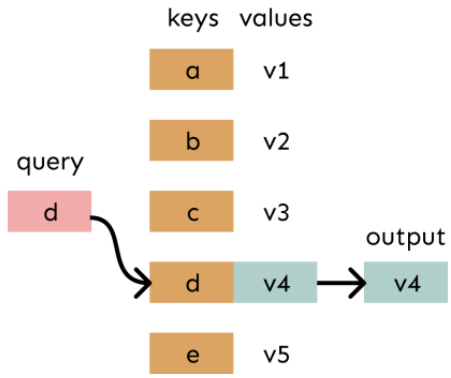
All words attend to all words in previous layer; most arrows here are omitted

Attention as a soft, averaging lookup table

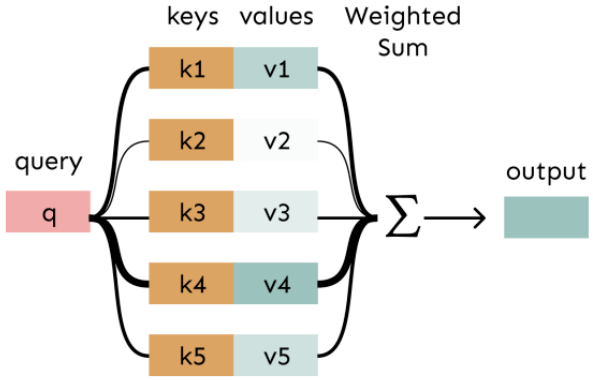
ambiguous 歧义

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

In a **lookup table**, we have a table of keys that map to values. The query matches one of the keys, returning its value.



In **attention**, the query matches all keys *softly*, to a weight between 0 and 1. The keys' values are multiplied by the weights and summed.



(4) In math

Self-Attention: keys, queries, values from the same sequence

Let $\boldsymbol{w}_{1:n}$ be a sequence of words in vocabulary V, like *Zuko made his uncle tea*.

For each $\boldsymbol{w}_i$, let $\boldsymbol{x}_i = E\boldsymbol{w}_i$, where $E \in \mathbb{R}^{d \times |V|}$ is an embedding matrix.

1. Transform each word embedding with weight matrices Q, K, V, each in $\mathbb{R}^{d \times d}$

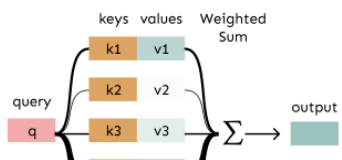
$$\boldsymbol{q}_i = Q\boldsymbol{x}_i \text{ (queries)} \qquad \boldsymbol{k}_i = K\boldsymbol{x}_i \text{ (keys)} \qquad \boldsymbol{v}_i = V\boldsymbol{x}_i \text{ (values)}$$

2. Compute pairwise similarities between keys and queries; normalize with softmax

$$\boldsymbol{e}_{ij} = \boldsymbol{q}_i^\top \boldsymbol{k}_j \qquad \boldsymbol{\alpha}_{ij} = \frac{\exp(\boldsymbol{e}_{ij})}{\sum_{j'} \exp(\boldsymbol{e}_{ij'})}$$

3. Compute output for each word as weighted sum of values

$$\boldsymbol{o}_i = \sum_j \boldsymbol{\alpha}_{ij} \boldsymbol{v}_j$$



$x_i = Ew_i$ is embedding one-hot vector ⇒ word vector $x_i \in R^d$

⑤ Barries and solutions for Self-Attention as a building block

⚠ sequence order

Why have this problem ?

we are using embedding matrix which isn't dependent on the index

Intuition to solve

Fixing the first self-attention problem: **sequence order**

- Since self-attention doesn't build in order information, we need to encode the order of the sentence in our keys, queries, and values.
- Consider representing each **sequence index** as a **vector**

$$\boldsymbol{p}_i \in \mathbb{R}^d, \text{ for } i \in \{1,2,\dots,n\} \text{ are position vectors}$$

- Don't worry about what the p_i are made of yet!
- Easy to incorporate this info into our self-attention block: just add the $\boldsymbol{p}_i$ to our inputs!
- Recall that $\boldsymbol{x}_i$ is the embedding of the word at index i. The positioned embedding is:

$$\tilde{\boldsymbol{x}}_i = \boldsymbol{x}_i + \boldsymbol{p}_i$$

In deep self-attention networks, we do this at the first layer! You could concatenate them as well, but people mostly just add...

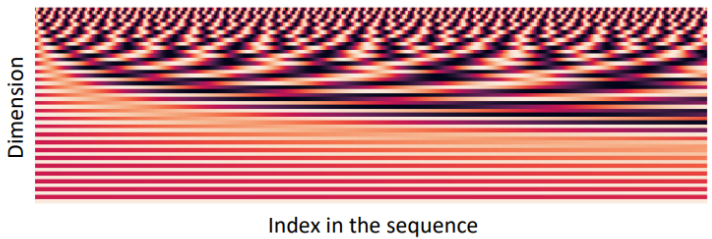
How to reprensente P_i (Position encoding) ?

1

Position representation vectors through sinusoids

- **Sinusoidal position representations:** concatenate sinusoidal functions of varying periods:

$$\boldsymbol{p}_i = \begin{pmatrix} \sin(i/10000^{2*1/d}) \\ \cos(i/10000^{2*1/d}) \\ \vdots \\ \sin(i/10000^{2*\frac{d}{2}/d}) \\ \cos(i/10000^{2*\frac{d}{2}/d}) \end{pmatrix}$$



- Pros:
  - Periodicity indicates that maybe "absolute position" isn't as important
  - Maybe can extrapolate to longer sequences as periods restart!
- Cons:
  - Not learnable; also the extrapolation (外推法) doesn't really work!

$$PE_{(pos,2i)} = \sin\left(\frac{pos}{10000^{2i/d}}\right)$$

$$PE_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{2i/d}}\right)$$

why this formula works?

$$PE_{(pos+k,2i)} = \sin(w_i(pos+k))$$

$$\sin(\alpha+\beta) = \sin\alpha\cos\beta + \cos\alpha\sin\beta$$

$$PE_{(pos+k,2i)} = \sin(w_i \cdot pos)\cos(w_i \cdot k) + \cos(w_i \cdot pos)\sin(w_i \cdot k)$$

$$= PE_{(pos,2i)} \cdot \cos(w_i \cdot k) + PE_{(pos,2i+1)} \cdot \sin(w_i \cdot k)$$

this means PE_{pos+k}'s each element can be PE_{pos}'s linear combination unrelated to absolute position pos

So what only need to learn is "the next word" which is a Relative Relationship pattern

[2]

Position representation vectors learned from scratch

- **Learned absolute position representations:** Let all p_i be learnable parameters! Learn a matrix $\boldsymbol{p} \in \mathbb{R}^{d \times n}$, *and let each* $\boldsymbol{p}_i$ *be a column of that matrix!*

- Pros:
  - Flexibility: each position gets to be learned to fit the data
- Cons:
  - Definitely can't extrapolate to indices outside $1, \ldots, n$. but can't work on n+1 !
- Most systems use this!

- Sometimes people try more flexible representations of position:
  - Relative linear position attention [Shaw et al., 2018]
  - Dependency syntax-based position [Wang et al., 2019]

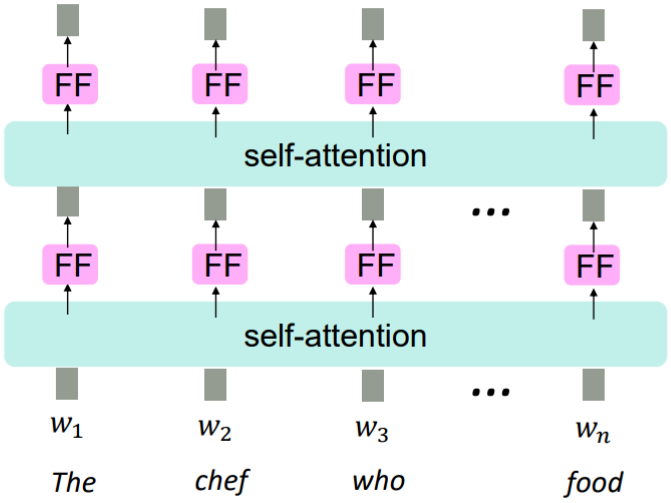
[1] VS [2] : extrapolate or learnable

② nonlinearities

Adding nonlinearities in self-attention

- Note that there are no elementwise nonlinearities in self-attention; stacking more self-attention layers just re-averages **value** vectors (Why? Look at the notes!)
- Easy fix: add a **feed-forward network** to post-process each output vector.

$$m_i = MLP(\text{output}_i)$$
$$= W_2 * \text{ReLU}(W_1\,\text{output}_i + b_1) + b_2$$



Intuition: the FF network processes the result of attention

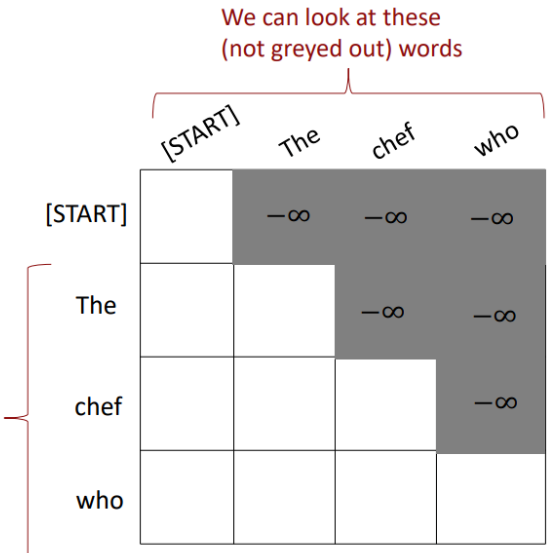
MLP: Multilayer Perceptron, a feed-forward network model

③ Mask the future in decoders

Masking the future in self-attention

- To use self-attention in **decoders**, we need to ensure we can't peek at the future.
- At every timestep, we could change the set of **keys and queries** to include only past words. (Inefficient!)
- To enable parallelization, we **mask out attention** to future words by setting attention scores to $-\infty$.

$$e_{ij} = \begin{cases} q_i^\top k_j, j \le i \\ -\infty, j > i \end{cases}$$



Why $-\infty$?

In softmax(), $\exp(-\infty) = 0$

Only in decoder?

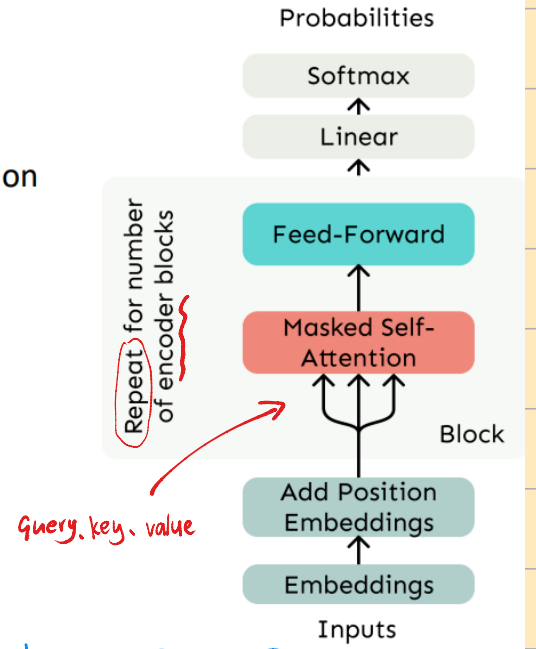
Yes, in practice

In encoder, sequence know all the elements

Summary

Necessities for a self-attention building block:

- **Self-attention**:
  - the basis of the method.
- **Position representations**:
  - Specify the sequence order, since self-attention is an unordered function of its inputs.
- **Nonlinearities**:
  - At the output of the self-attention block
  - Frequently implemented as a simple feed-forward network.
- **Masking**:
  - In order to parallelize operations while not looking at the future.
  - Keeps information about the future from "leaking" to the past.



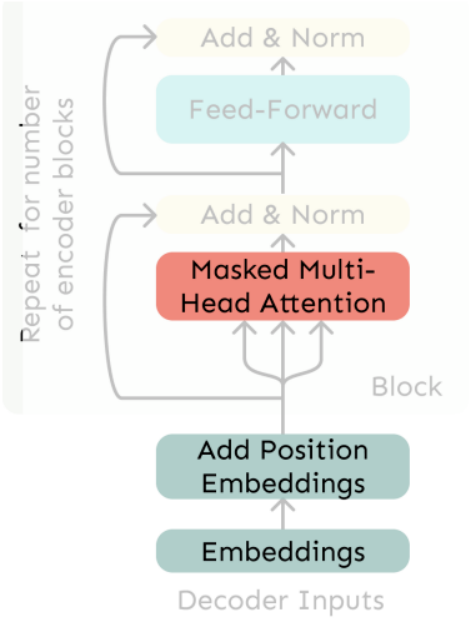
Self-attention architecture

Transformer

① architecture

The Transformer Decoder

- A Transformer decoder is how we'll build systems like **language models**.
- It's a lot like our minimal self-attention architecture, but with a few more components.
- The embeddings and position embeddings are identical.
- We'll next replace our self-attention with **multi-head self-attention.**

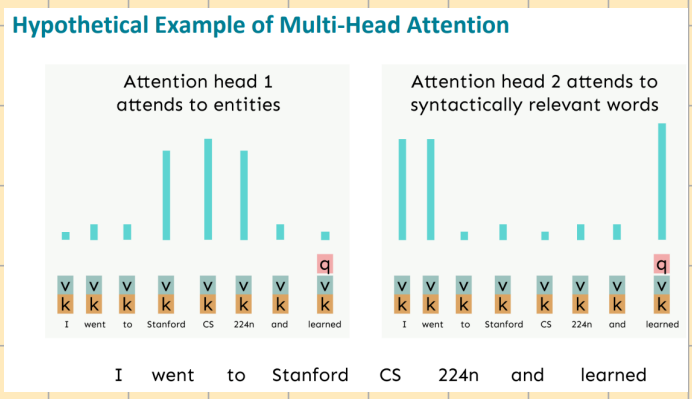


Transformer Decoder

Why use multi-head self-attention ?

Different purpose may focus on different place (eg. pattern, semantic, syntactic structure)

eg:



Core idea of multi-head self-attention

Allow the model to simultaneously focus on different information from different locations in different representation subspaces.

② In math

① Single headed attention

Sequence-Stacked form of Attention

- Let's look at how key-query-value attention is computed, in matrices.
  - Let $X = [x_1; \dots; x_n] \in \mathbb{R}^{n \times d}$ be the concatenation of input vectors.
  - First, note that $XK \in \mathbb{R}^{n \times d}$, $XQ \in \mathbb{R}^{n \times d}$, $XV \in \mathbb{R}^{n \times d}$.
  - The output is defined as output $= \text{softmax}(XQ(XK)^\top)XV \in\in \mathbb{R}^{n \times d}$.

First, take the query-key dot products in one matrix multiplication: $XQ(XK)^\top$

XQ $K^\top X^\top$ $= XQK^\top X^\top \in \mathbb{R}^{n \times n}$

All pairs of attention scores!

Next, softmax, and compute the weighted average with another matrix multiplication.

$\text{softmax}(XQK^\top X^\top)XV = \text{output} \in \mathbb{R}^{n \times d}$

② multi-headed attention

Multi-headed attention

- What if we want to look in multiple places in the sentence at once?
  - For word i, self-attention "looks" where $x_i^\top Q^\top K x_j$ is high, but maybe we want to focus on different j for different reasons?
- We'll define **multiple attention "heads"** through multiple Q,K,V matrices

- Let, $Q_\ell, K_\ell, V_\ell \in \mathbb{R}^{d \times \frac{d}{h}}$, where h is the number of attention heads, and ℓ ranges from 1 to h.
- Each attention head performs attention independently:
  - $\text{output}_\ell = \text{softmax}(XQ_\ell K_\ell^\top X^\top) * XV_\ell$, where $\text{output}_\ell \in \mathbb{R}^{d/h}$
- Then the outputs of all the heads are combined!
  - $\text{output} = [\text{output}_1; \ldots; \text{output}_h]Y$, where $Y \in \mathbb{R}^{d \times d}$

- Each head gets to "look" at different things, and construct value vectors differently.

data stream

input X

⇓

project each head onto a low dimensional subspace

⇓

Each head independently calculates attention

⇓

splicing the outputs of all heads together

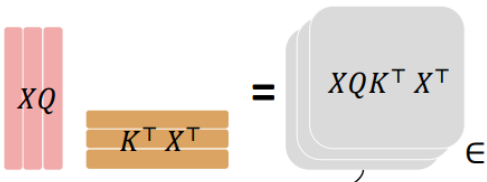
⇓

Integrating information through linear layer Y

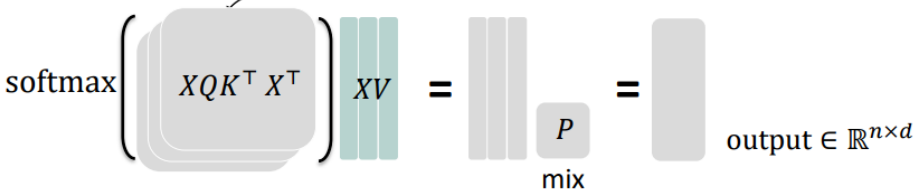
Multi-head self-attention is computationally efficient

- Even though we compute h many attention heads, it's not really more costly.
  - We compute $XQ \in \mathbb{R}^{n \times d}$, and then reshape to $\mathbb{R}^{n \times h \times d/h}$. (Likewise for XK, XV.)
  - Then we transpose to $\mathbb{R}^{h \times n \times d/h}$; now the head axis is like a batch axis.
  - Almost everything else is identical, and the **matrices are the same sizes.**

First, take the query-key dot products in one matrix multiplication: $XQ(XK)^\top$



Next, softmax, and compute the weighted average with another matrix multiplication.



how to compute ?

compute each attention head independently

$Q_i K_i^T \ (i = 1, 2, \ldots h)$

for the ith head

$Q_i \in R^{n \times \frac{d}{h}}$ $K_i \in R^{n \times \frac{d}{h}}$

multiply them

$Q_i K_i^T \in R^{n \times n}$

Parallel compute of all heads at once

$Q' K'^T \in R^{h \times n \times n}$

But why not $R^{(h \times h) \times n \times n}$?

Each head of multi head attention is independently calculated, with completely different parameters that do not interfere with each other

Q.A. Do all the blocks an equal number of heads?

↳ In practice, Yes

Because it's not really cost much. But (set to 0, just like deleting some heads) in a trained Transformer model, certain attention heads may become "ineffective" or "highly specialized" to the point where their effectiveness is almost zero under certain inputs

Scaled Dot Product [Vaswani et al., 2017]

- **"Scaled Dot Product"** attention aids in training.
- When dimensionality d becomes large, dot products between vectors tend to become large.
  - Because of this, inputs to the softmax function can be large, making the gradients small.
- Instead of the self-attention function we've seen:

$$\text{output}_\ell = \text{softmax}(XQ_\ell K_\ell^\top X^\top) * XV_\ell$$

- We divide the attention scores by $\sqrt{d/h}$, to stop the scores from becoming large just as a function of d/h (The dimensionality divided by the number of heads.)

$$\text{output}_\ell = \text{softmax}\left(\frac{XQ_\ell K_\ell^\top X^\top}{\sqrt{d/h}}\right) * XV_\ell$$

assume there are d dimension random vector q, k (q, k IID, avg = 0, var = 1)

dot product

$$q \cdot k = \sum_{i=1}^{d} q_i k_i$$

Expected value

$$E(q \cdot k) = \sum_{i=1}^{d} E(q_i k_i) = \sum_{i=1}^{d} E(q_i) E(k_i) = 0$$

Variance

$$Var(q_i k_i) = E((q_i k_i)^2) - (E(q_i k_i))^2 = E(q_i^2) \cdot E(k_i^2) = 1 \times 1 = 1$$

$$Var(q \cdot k) = Var\left(\sum_{i=1}^{d} q_i k_i\right) = d \times Var(q_i k_i) = d$$

⇓

this means: var of dot product is proportional to the dimension d

⇓

So, when d gets bigger, fluctuation range become wider, more probable to take extreme value (far from 0)

⇓

Softmax() is sensitive to the size of input, if it's big, Softmax output will become very "sharp", which close to one-hot vector

⇓

To one-hot vector, its gradient is very small which may lead to vanishing gradient

$$Var\left(\frac{u \cdot v}{\sqrt{\frac{d}{n}}}\right) = \frac{1}{\frac{d}{n}} Var(u \cdot v) = \frac{1}{\frac{d}{n}} \times \frac{d}{n} = 1$$

So our solution is keeping variance be 1

(3) Optimization methods

(1)

The Transformer Encoder: **Residual connections** [He et al., 2016]

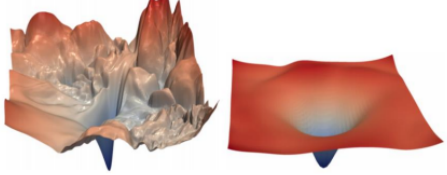
- **Residual connections** are a trick to help models train better.
  - Instead of $X^{(i)} = \text{Layer}(X^{(i-1)})$ (where i represents the layer)

$$X^{(i-1)} \longrightarrow \boxed{\text{Layer}} \longrightarrow X^{(i)}$$

  - We let $X^{(i)} = X^{(i-1)} + \text{Layer}(X^{(i-1)})$ (so we only have to learn "the residual" from the previous layer)

Layer

- Gradient is **great** through the residual connection; it's 1!
- Bias towards the identity function!



[no residuals] [residuals]

[Loss landscape visualization, Li et al., 2018, on a ResNet]

(2)

The Transformer Encoder: **Layer normalization** [Ba et al., 2016]

- **Layer normalization** is a trick to help models train faster.
- Idea: cut down on uninformative variation in hidden vector values by normalizing to unit mean and standard deviation **within each layer**.
  - LayerNorm's success may be due to its normalizing gradients [Xu et al., 2019]
- Let $x \in \mathbb{R}^d$ be an individual (word) vector in the model.
- Let $\mu = \sum_{j=1}^{d} x_j$; this is the mean; $\mu \in \mathbb{R}$.
- Let $\sigma = \sqrt{\frac{1}{d}\sum_{j=1}^{d}(x_j - \mu)^2}$; this is the standard deviation; $\sigma \in \mathbb{R}$.
- Let $\gamma \in \mathbb{R}^d$ and $\beta \in \mathbb{R}^d$ be learned "gain" and "bias" parameters. (Can omit!)
- Then layer normalization computes:

$$\text{output} = \frac{x - \mu}{\sqrt{\sigma + \epsilon}} * \gamma + \beta$$

Normalize by scalar mean and variance

Modulate by learned elementwise gain and bias

$\mu = \frac{1}{d}\sum_{j=1}^{d} x_j$

ϵ here is a very small number (eg. 1e-5), to prevent from divided by 0

Anti normalization – "Preserving expressive power"

which prevent from all vectors being constrained to the same distribution

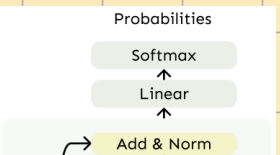
Q.A. Do words share their statistics?

↳ No, they do independently for each word

(4) Total diagram

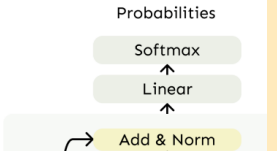
The Transformer Decoder

- The Transformer Decoder is a stack of Transformer Decoder **Blocks**.

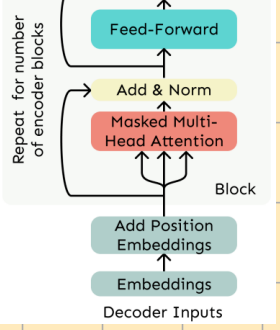


The Transformer Encoder

- The Transformer Decoder constrains to **unidirectional context**, as for **language**

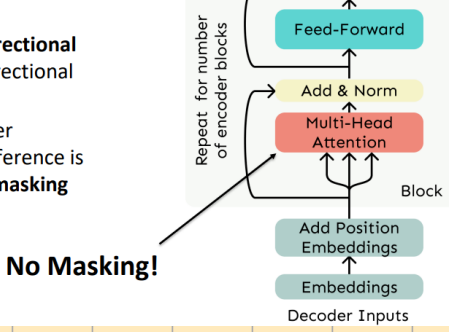


- Each Block consists of:
  - Self-attention
  - Add & Norm
  - Feed-Forward
  - Add & Norm
- That's it! We've gone through the Transformer Decoder.



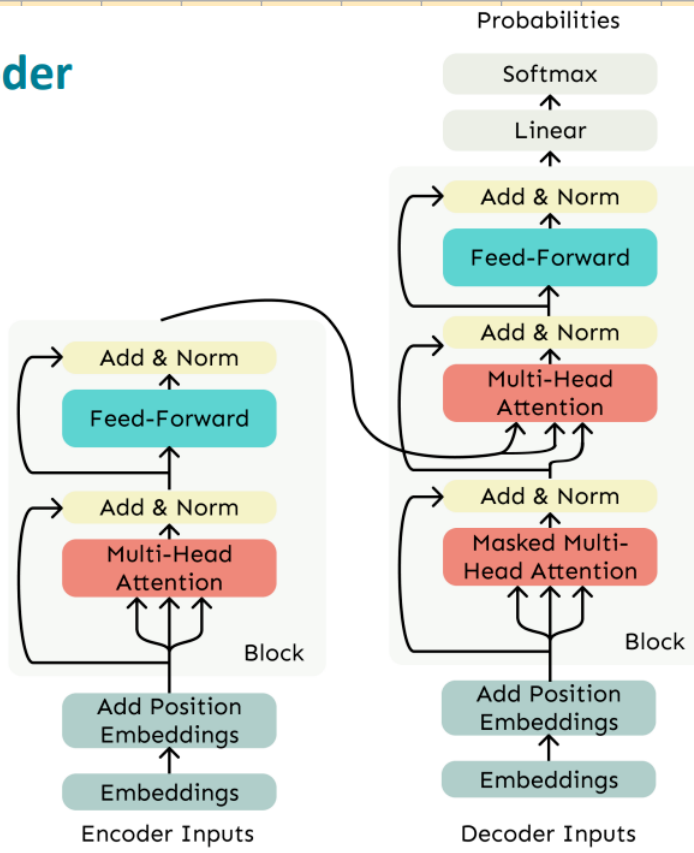
models.

- What if we want **bidirectional context**, like in a bidirectional RNN?
- This is the Transformer Encoder. The only difference is that we **remove the masking** in the self-attention.



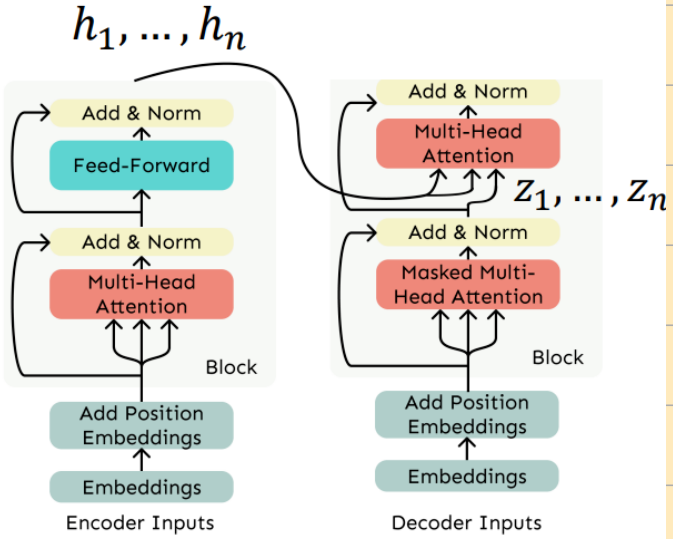
The Transformer Encoder-Decoder

- Recall that in machine translation, we processed the source sentence with a **bidirectional** model and generated the target with a **unidirectional model**.
- For this kind of seq2seq format, we often use a Transformer Encoder-Decoder.
- We use a normal Transformer Encoder.
- Our Transformer Decoder is modified to perform **cross-attention** to the output of the Encoder.



Cross-attention (details)

- We saw that self-attention is when keys, queries, and values come from the same source.
- In the decoder, we have attention that looks more like what we saw last week.
- Let $h_1, \dots, h_n$ be **output** vectors **from** the Transformer **encoder**; $x_i \in \mathbb{R}^d$
- Let $z_1, \dots, z_n$ be input vectors from the Transformer **decoder**, $z_i \in \mathbb{R}^d$
- Then keys and values are drawn from the **encoder** (like a memory):
  - $k_i = Kh_i, v_i = Vh_i$.
- And the queries are drawn from the **decoder**, $q_i = Qz_i$.



(5) Drawbacks and variants of Transformers

What would we like to fix about the Transformer?

- **Quadratic compute in self-attention (today)**:
  - Computing all pairs of interactions means our computation grows **quadratically** with the sequence length!
  - For recurrent models, it only grew linearly!
- **Position representations**:
  - Are simple absolute indices the best we can do to represent position?
  - Relative linear position attention [Shaw et al., 2018]
  - Dependency syntax-based position [Wang et al., 2019]

Do we even need to remove the quadratic cost of attention?

- As Transformers grow larger, a larger and larger percent of compute is **outside** the self-attention portion, despit the quadratic cost.
- In practice, **almost no large Transformer language models use anything but the quadratic cost attention we've presented here.**
  - The cheaper methods tend not to work as well at scale.
- So, is there no point in trying to design cheaper alternatives to self-attention?
- Or would we unlock much better models with much longer contexts (>100k tokens?) if we were to do it right?