CMPT 413/713: Natural Language Processing

Transformers and Self-Attention

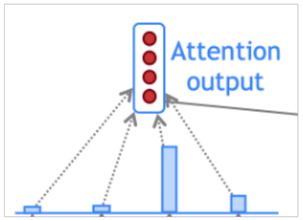
Spring 2026

2026-02-02

Adapted from slides from Danqi Chen and Karthik Narasimhan
(with some content from slides from Chris Manning and Abigail See)

Review of attention in sequence to sequence models

Attentive machine translation summary

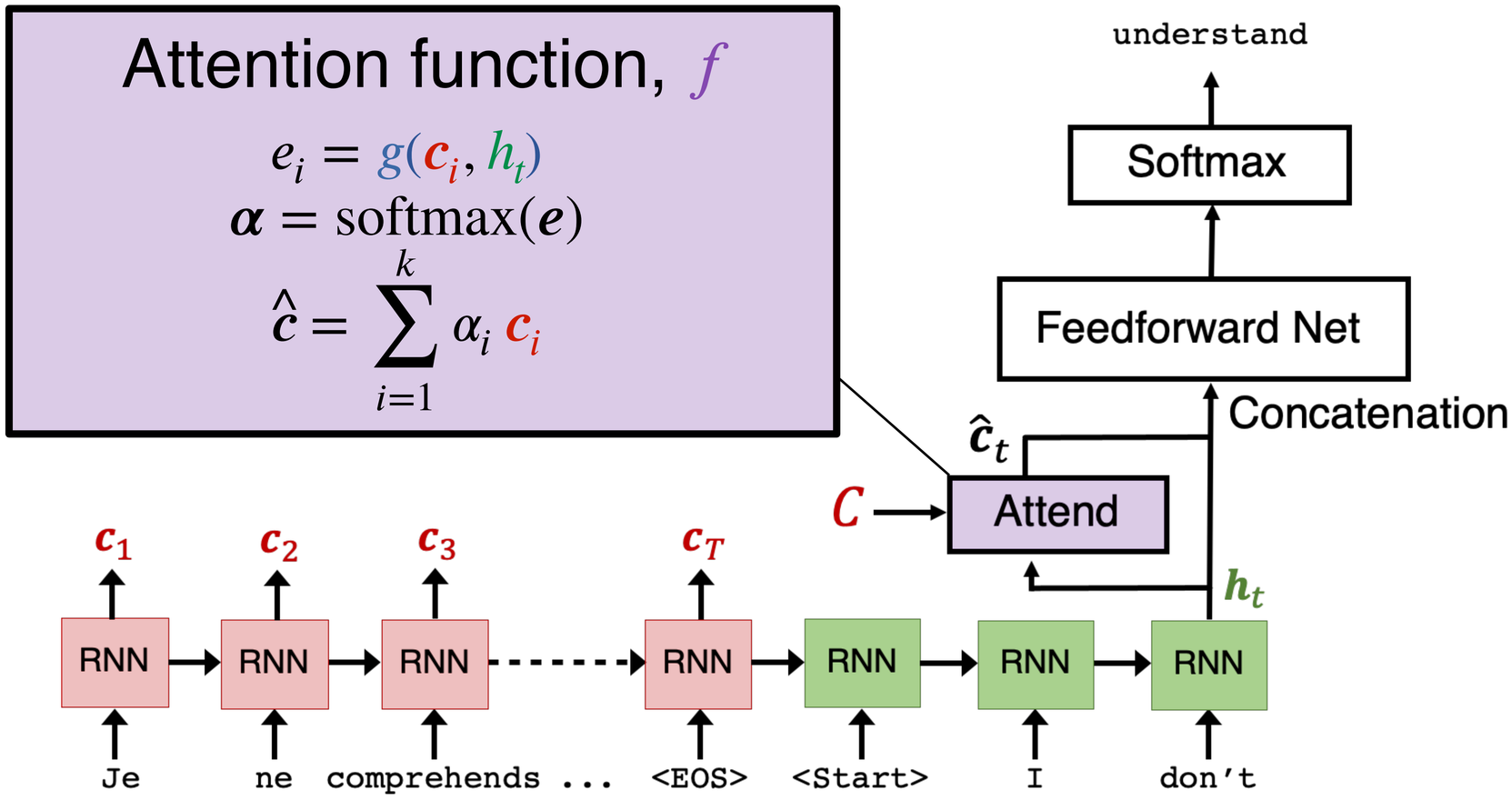


$\hat{c}_t$ is the weighted sum of encoder hidden states

$$C = \{c_1, \ldots, c_T\}$$

(slide credit: Peter Anderson)

Attentive machine translation summary



(slide credit: Peter Anderson)

Summary of attention

Attention function, f

$$e_i = g(\boldsymbol{c}_i, \boldsymbol{z})$$
$$\boldsymbol{\alpha} = \text{softmax}(\boldsymbol{e})$$
$$\hat{\boldsymbol{c}} = \sum_{i=1}^{k} \alpha_i \, \boldsymbol{c}_i$$

Attention scores: $\boldsymbol{e}$ (unnormalized)

Attention weights: $\boldsymbol{\alpha}$ (normalized)

Final attention output

Weighted sum of context features (or values)

Attention score $e_i = g(\boldsymbol{c}_i, \boldsymbol{z})$
how well does the attention candidate $\boldsymbol{c}_i$ match the query $\boldsymbol{z}$

- **Dot-product attention**:

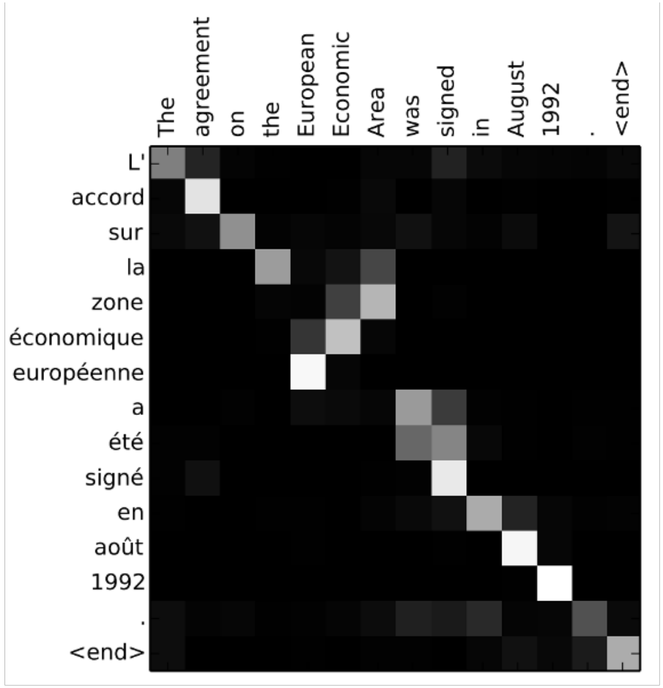
$$g(c_i, z) = z^\top c_i$$

- **Neural network**

$$g(c_i, z) = v^\top \tanh\left(W_1 c_i + W_2 z\right)$$

Motivation of attention

- How much does this attention candidate match the query vector?
- Motivated by biological attention and alignment in machine translation

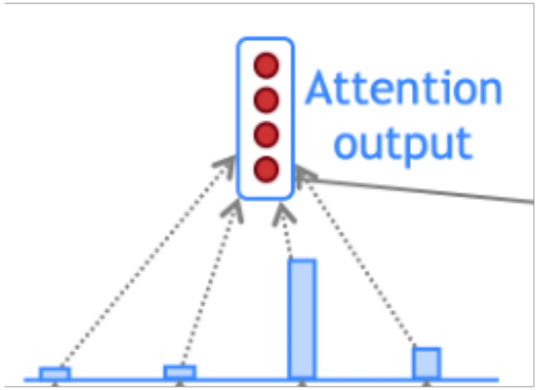


generated (French)

source (English)

Attention weights α_i (0 = black, 1 = white)

get a representation that is a weighted sum over the attention candidates based on a query vector



the | agreement | on | the

Attention is a *general* deep learning technique

- Given a set of value vectors and a query vector, **attention** is a way to compute a **weighted sum** of the values dependent on the query.
  - The query determines what values to focus on,
    - We say: the query "*attends*" to the values
  - In NMT, each decoder hidden state (query) attends to all the encoder hidden state (values)
- A more general form: use a set of keys and values
  - The keys are used to compute the **attention scores**
  - The values are used to compute the **output vector**

Attention is always computed the same way

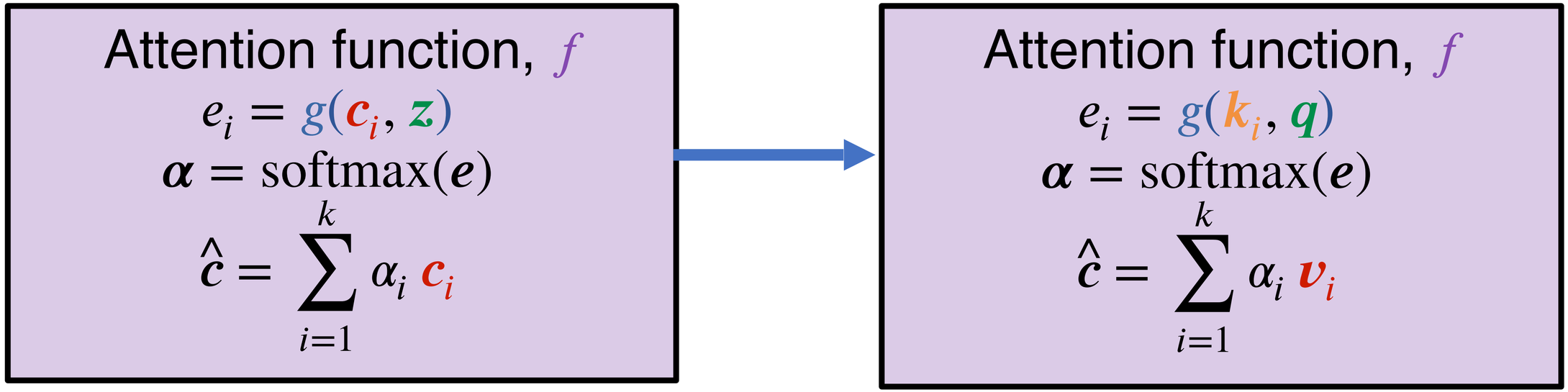
- Assume that we have a set of key-value pairs $\mathbf{k}_1, \ldots, \mathbf{k}_n \in \mathbb{R}^{d_k}$, $\mathbf{v}_1, \ldots, \mathbf{v}_n \in \mathbb{R}^{d_v}$, and a query vector $\mathbf{q} \in \mathbb{R}^{d_q}$
- Computing attention consists of the following steps:
  - Compute the attention scores: $e_i = g(\mathbf{k}_i, \mathbf{q}), \mathbf{e} \in \mathbb{R}^n$
  - Take softmax to get the attention distribution

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

  - Use attention distribution to take weighted sum of values

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

Query-Value-Key view of attention



Projected query,key,value

$$\boldsymbol{q} = W_Q \boldsymbol{z}$$
$$\boldsymbol{k}_i = W_K \boldsymbol{c}_i$$
$$\boldsymbol{v}_i = W_V \boldsymbol{c}_i$$

Matrix form

$$\boldsymbol{q} = W_Q \boldsymbol{z}$$
$$K = W_K C^T$$
$$V = W_V C^T$$

$$C \in \mathbb{R}^{N \times d_C}$$

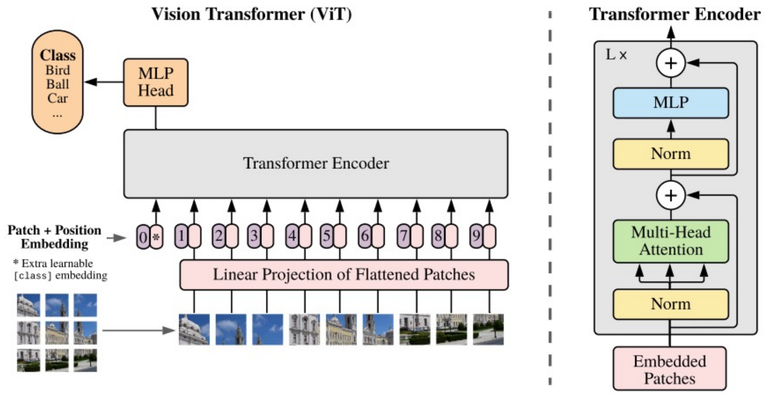
General form of attention: key-value-query

- **Attention** is a way to compute a **weighted sum** of the values dependent on the query and the corresponding keys.
  - All of these (key value query) are represented using **vectors**
    - The query and key are used for addressing (contains partial information). While the values provide more complete information
    - The weighted sum is a **selective summary** of the information found in the values.
    - It is a way to obtain a **fixed-sized representation** of an arbitrary set of representations (values) based on some other representation (the query)

Transformers

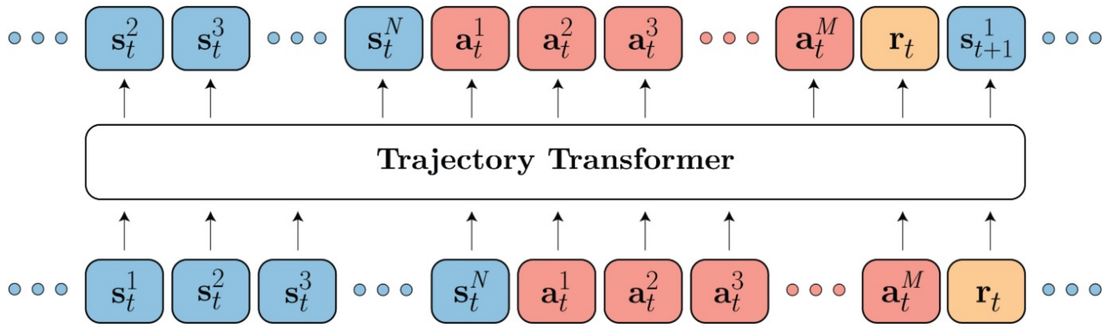
Transformers are everywhere!

- Vision

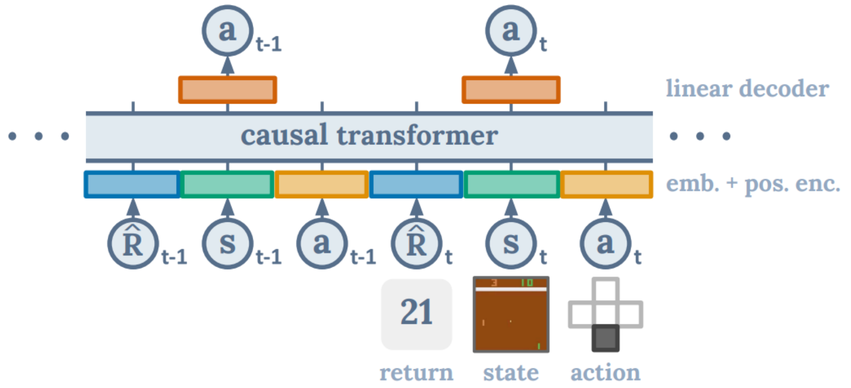


An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale, Dosovitskiy et al, ICLR 2021

- Reinforcement Learning

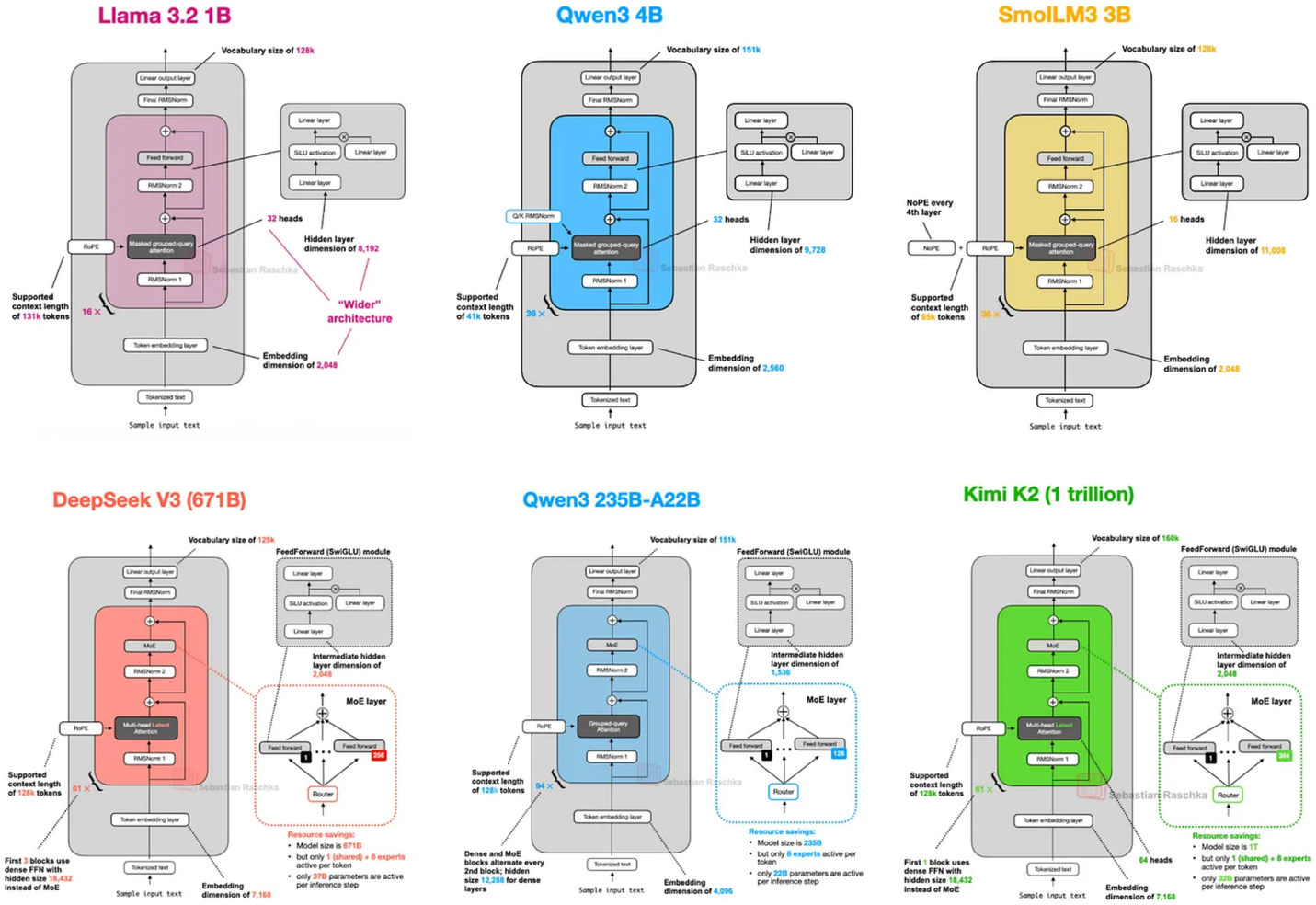


Trajectory Transformer [Janner et al, 2021]



Decision Transformer [Chen et al, 2021]

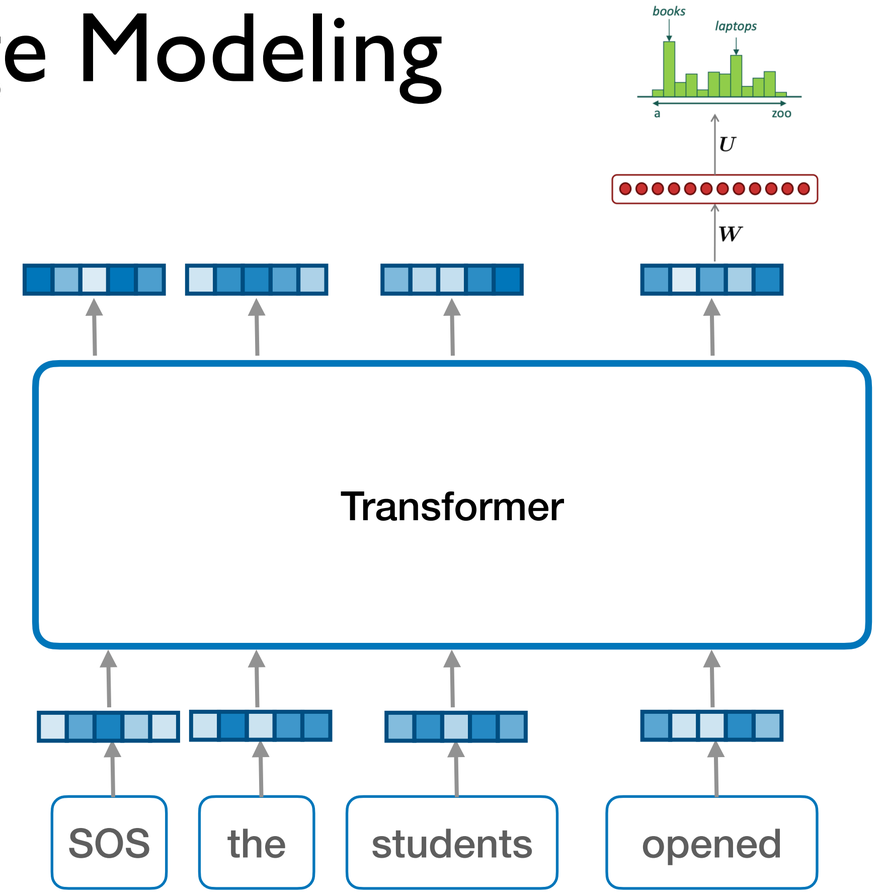
Transformers are the basis of modern LLMs



https://magazine.sebastianraschka.com/p/the-big-llm-architecture-comparison

Transformers for Language Modeling

Self-attention neural module that transforms token representation via many layers



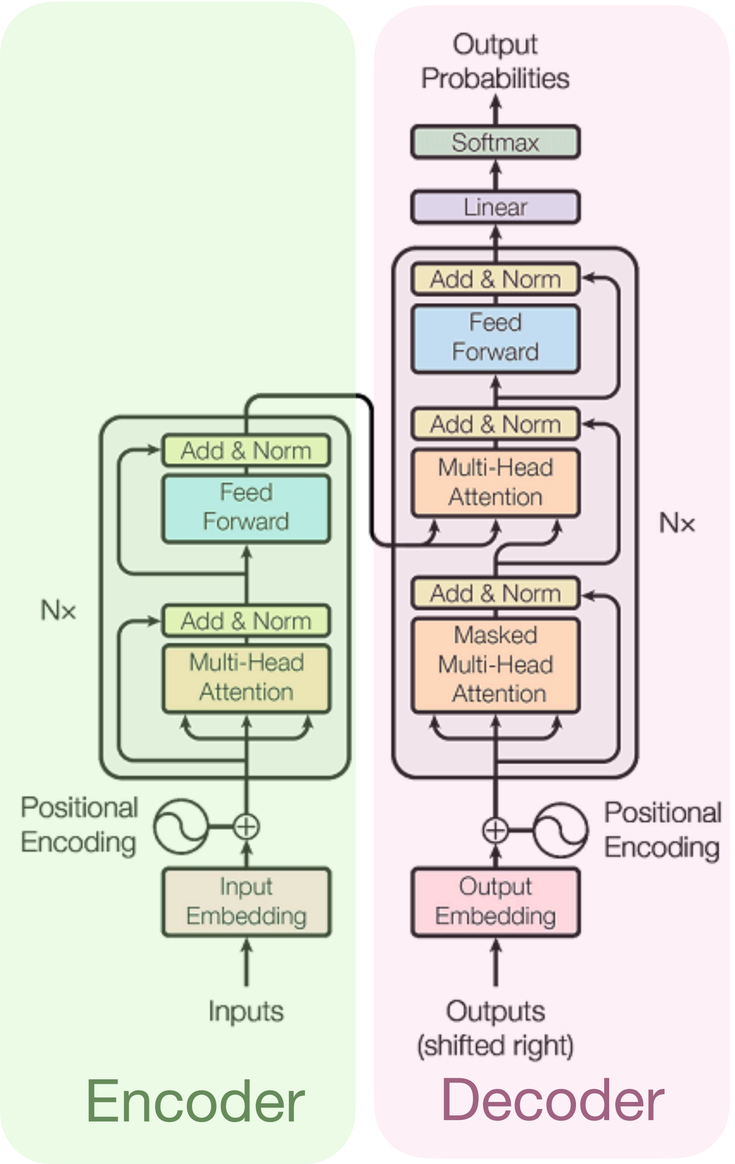
Notes

- Transformers can't actually handle arbitrary length
- But they are built up with modules that share weights
- They have self-attention that allow them to have better representations of context
- They can be parallelized and trained on large amounts of data

$$P(w_t|w_{<t}) \approx P(w_t|\phi(w_{t-n+1,t-1}))$$

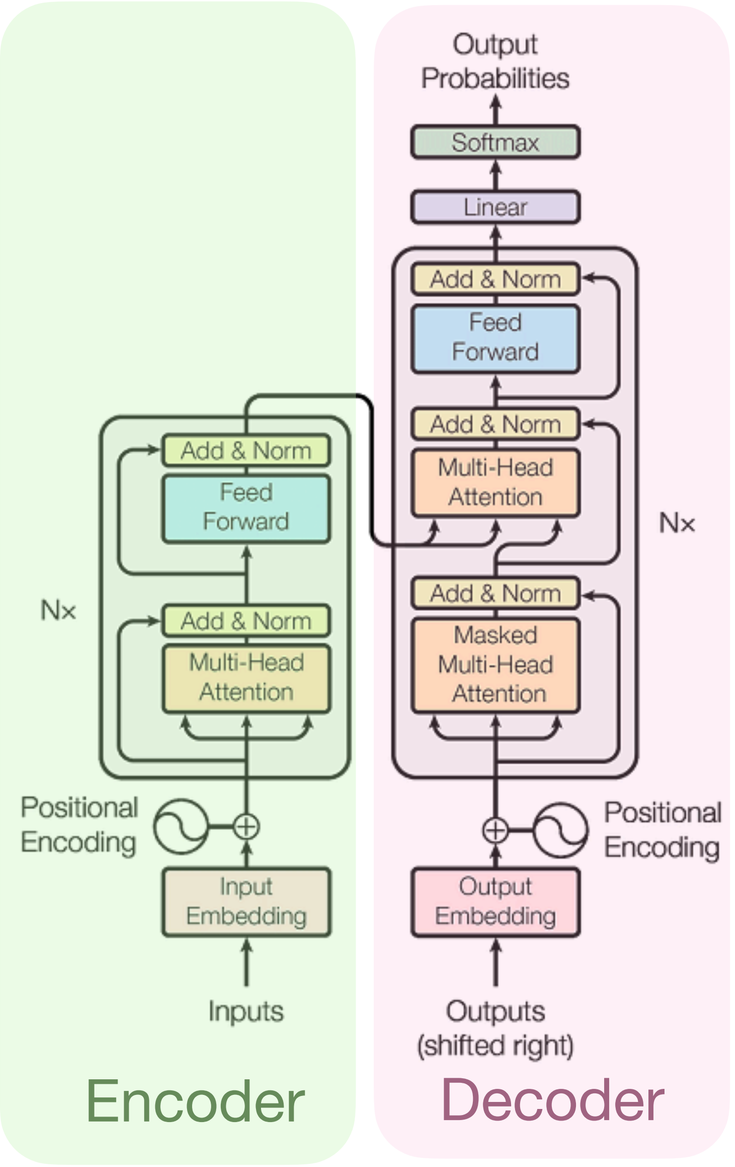
Transformers

- NIPS'17: Attention is All You Need
- Originally proposed for NMT (encoder-decoder framework)
- Used in most LLMs!
- Key idea: **Multi-head self-attention**
- No recurrence structure any more so it trains much faster



Understanding transformers

- From attention to self-attention
- From self-attention to multi-headed self-attention
- Transformer encoder
- Transformer decoder
- Putting the pieces together



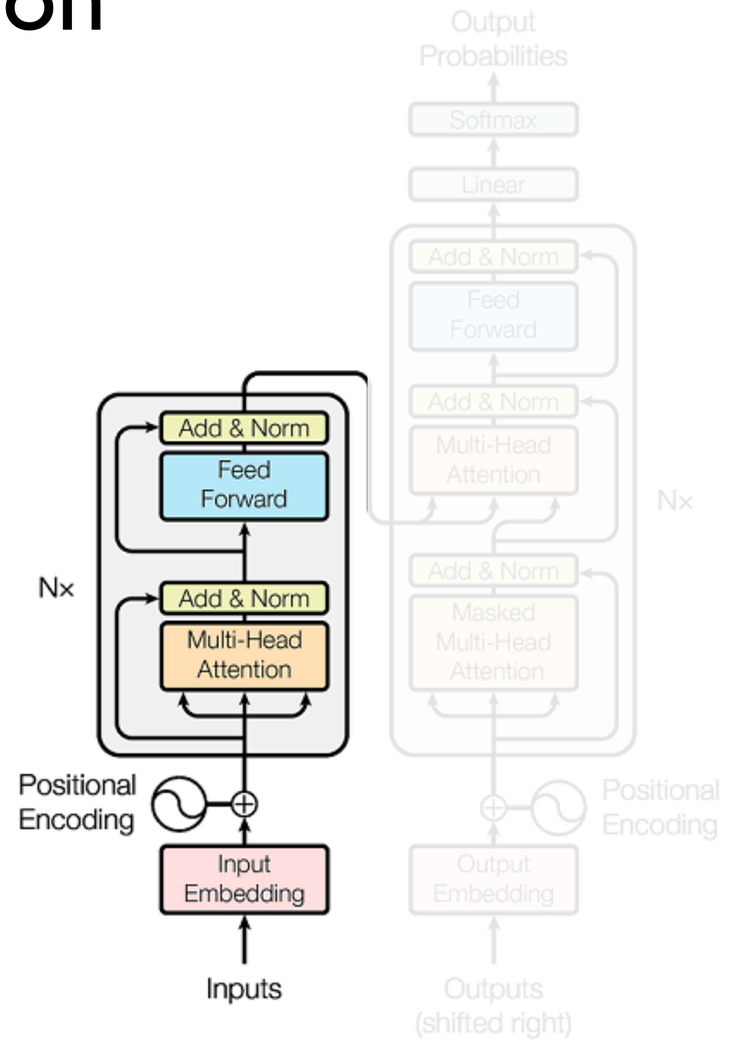
Multi-head self-attention

- Each Transformer block has two-sublayers
  - Multi-Head self-attention
  - 2 layer feedforward NN (with ReLU)

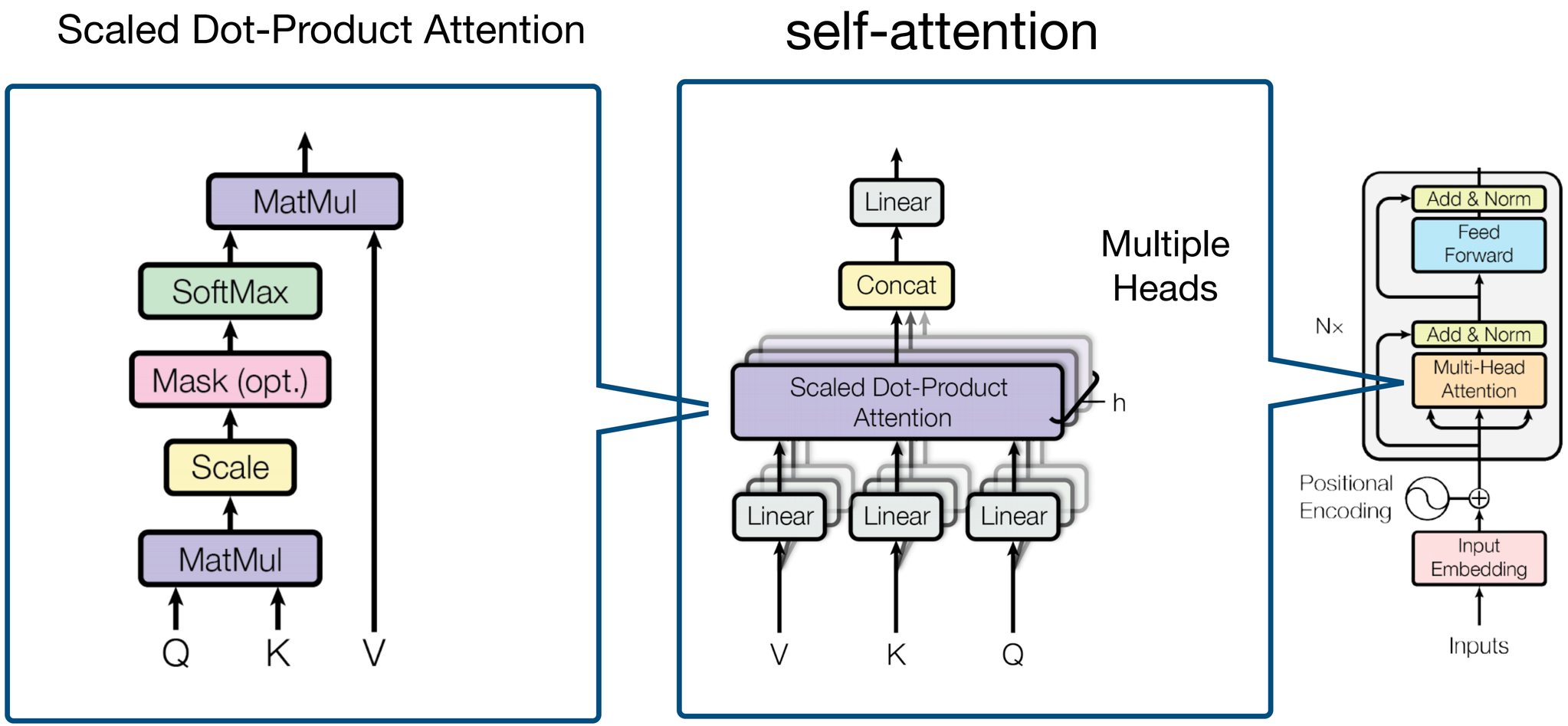
- Each sublayer has a residual connection and a layer normalization
  - LayerNorm(x+SubLayer(x))

Helps the training process!

- Input layer has a positional encoding



Multi-head self-attention



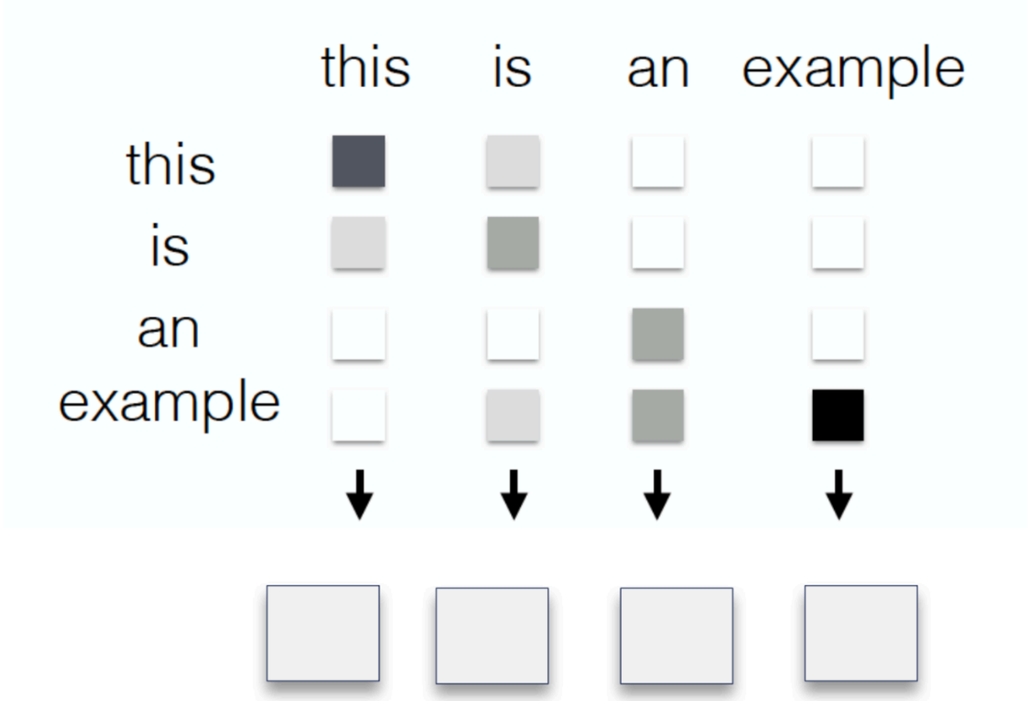
Attention Is All You Need https://arxiv.org/pdf/1706.03762.pdf

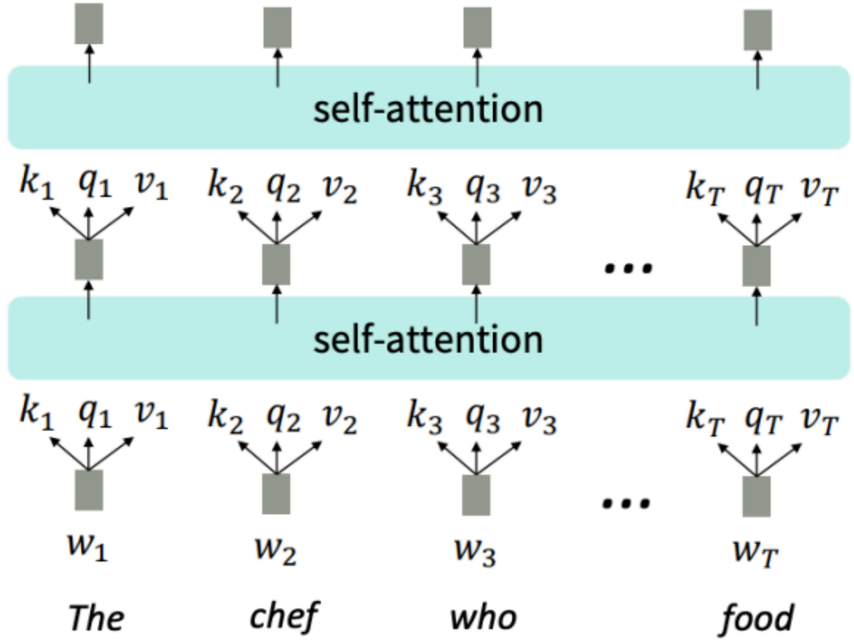
Self Attention

(also referred to as Intra-Attention)

- **Self-attention**: let's use each word as query and compute the attention with all the other words (other words are the keys and values)

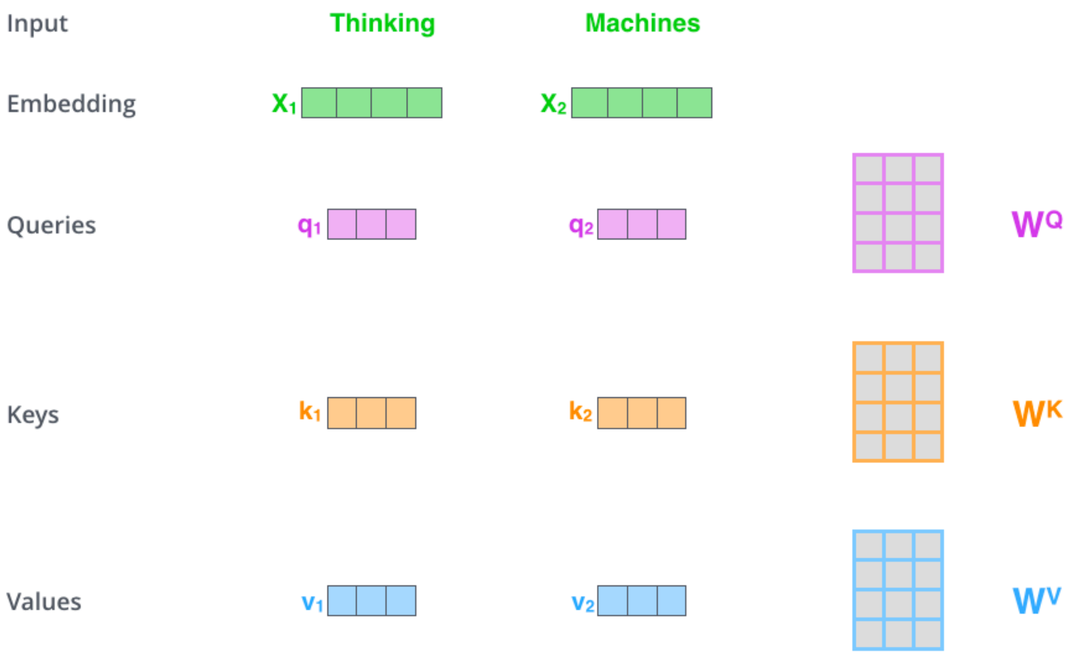
  = the word vectors themselves select each other



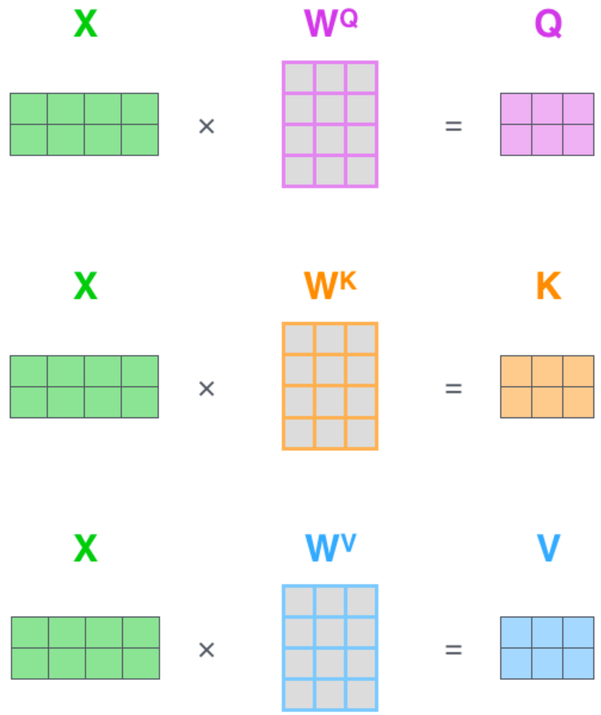


How to get key-value-query for each word?

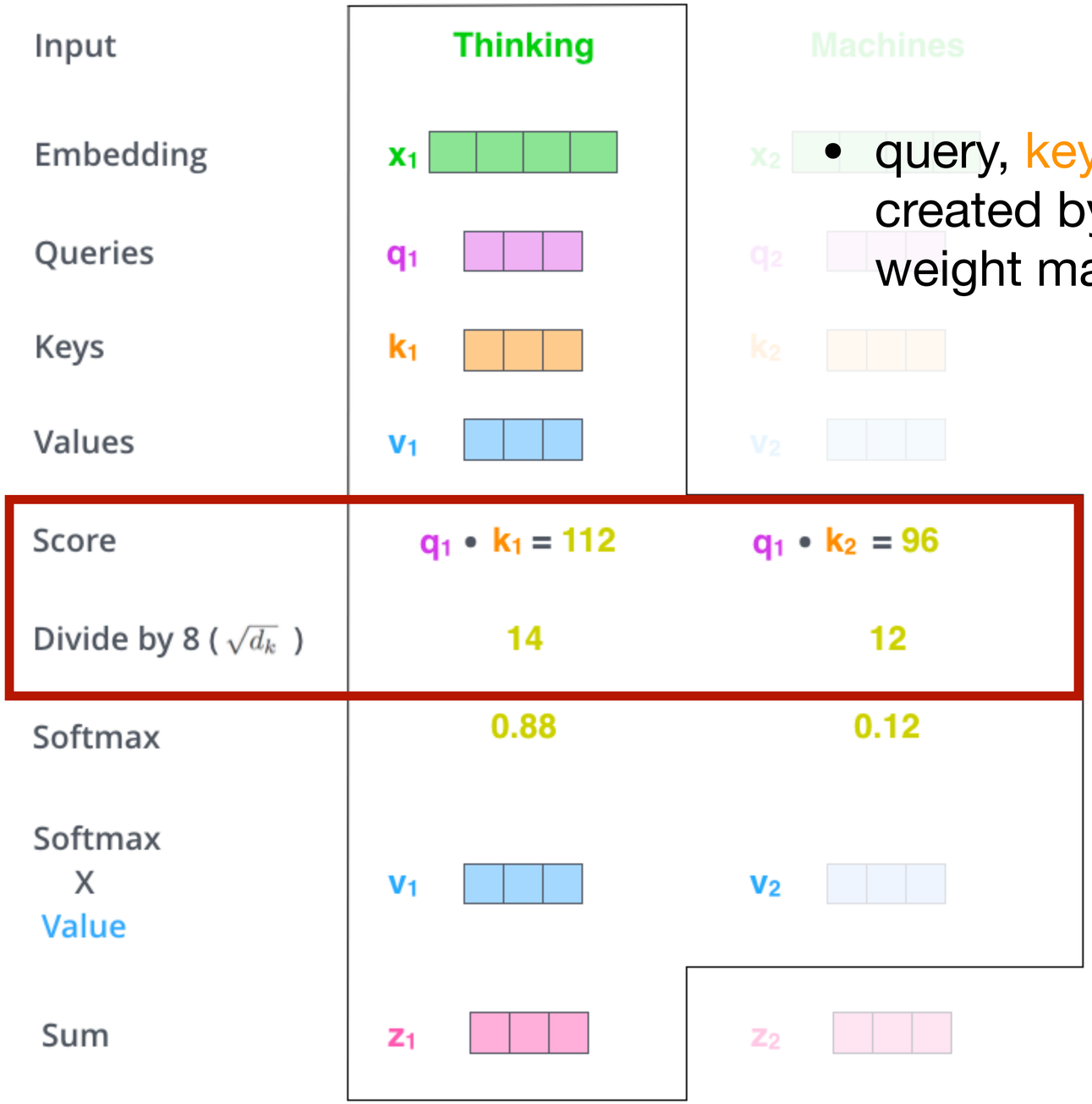
- For each word, we have vectors for the key-value-query
- These vectors are created by multiplying the word embedding by trained weight matrices



Stack into matrices and compute all at once!



(figure credit: Jay Alammar http://jalammar.github.io/illustrated-transformer/)



- query, key, and value vectors created by multiplying learned weight matrices with embedding

- Can be any kind of attention function
- For transformers, this is the scaled dot-product attention

(figure credit: Jay Alammar http://jalammar.github.io/illustrated-transformer/)

Scaled dot-product attention

- Assume keys $\mathbf{k}_1, \mathbf{k}_2, \ldots, \mathbf{k}_n$ and query $\mathbf{q}$

1. **Dot-product attention** (assumes equal dimensions for $\mathbf{k}_i$ and $\mathbf{q}$):

$$g(\mathbf{k}_i, \mathbf{q}) = \mathbf{q}^T \mathbf{k}_i \in \mathbb{R}$$

Scale of dot product (of random vectors) increases (proportional to $\sqrt{d}$) as dimension gets larger

Perform poorly for large *d*

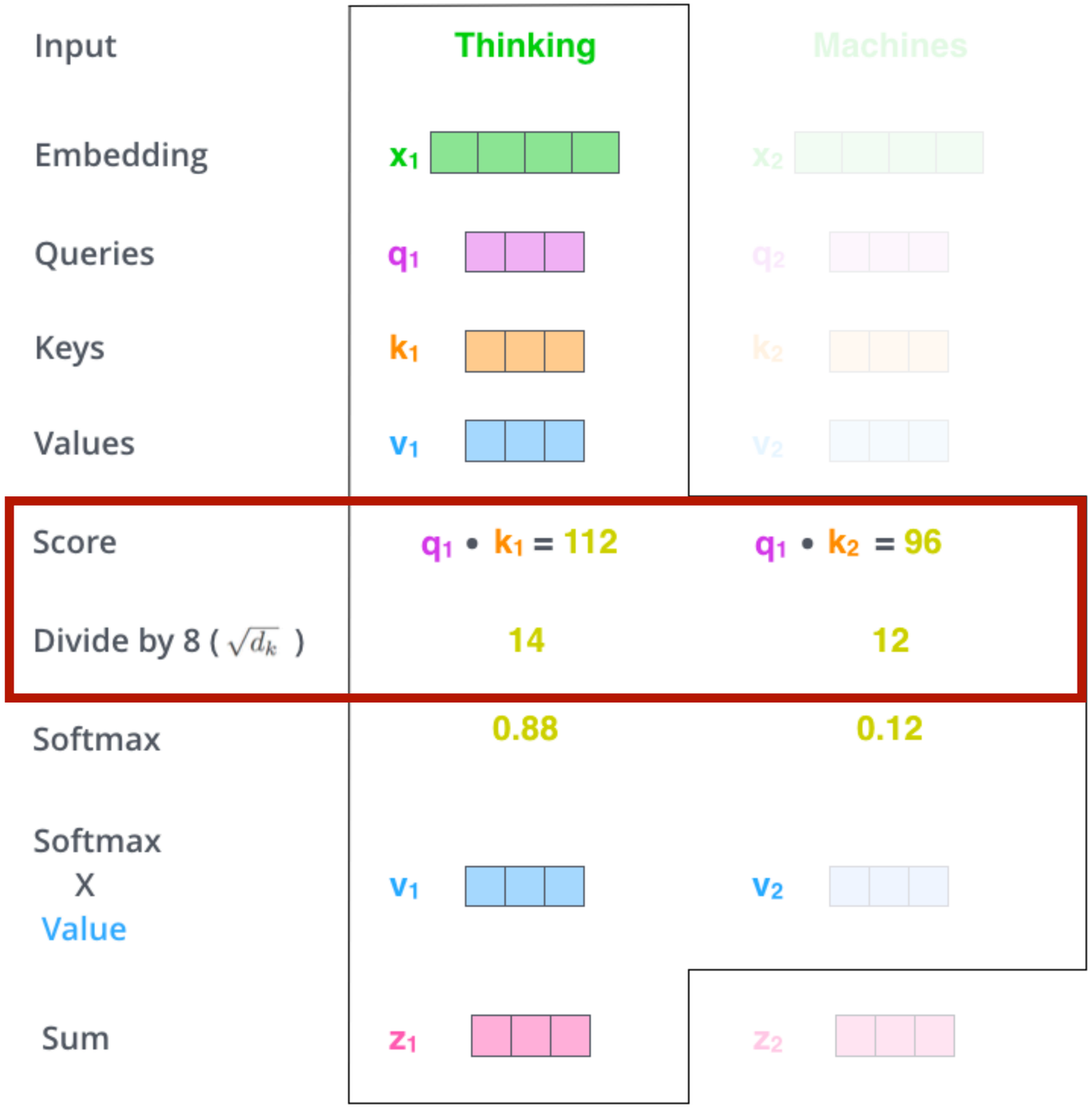
Softmax has small gradient

2. **Scaled dot-product attention:**

$$g(\mathbf{k}_i, \mathbf{q}) = \frac{\mathbf{q}^T \mathbf{k}_i}{\sqrt{d}} \in \mathbb{R}$$

Scaled dot product will perform well for larger dimensions

Scaling factor: d = dimension of hidden state



- Can be any kind of attention function
- For transformers, this is the scaled dot-product attention

- z_1 is the final vector of attended values for “Thinking” as the query

(figure credit: Jay Alammar http://jalammar.github.io/illustrated-transformer/)

Self-attention in equations

- A self-attention layer maps a sequence of input vectors $\mathbf{x}_1, \ldots, \mathbf{x}_n \in \mathbb{R}^{d_1}$ to a sequence of n vectors: $\mathbf{y}_1, \ldots, \mathbf{y}_n \in \mathbb{R}^{d_2}$
  - Note: this is similar as an RNN layer and can be used to replace an RNN layer
- First, construct a set of queries, keys, and values:

$$\mathbf{q}_i = W^Q \mathbf{x}_i, W^Q \in \mathbb{R}^{d_q \times d_1}$$
$$\mathbf{k}_i = W^K \mathbf{x}_i, W^K \in \mathbb{R}^{d_k \times d_1}$$
$$\mathbf{v}_i = W^V \mathbf{x}_i, W^V \in \mathbb{R}^{d_v \times d_1}$$

- Second, for each $\mathbf{q}_i$, compute attentions scores and attention distribution

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

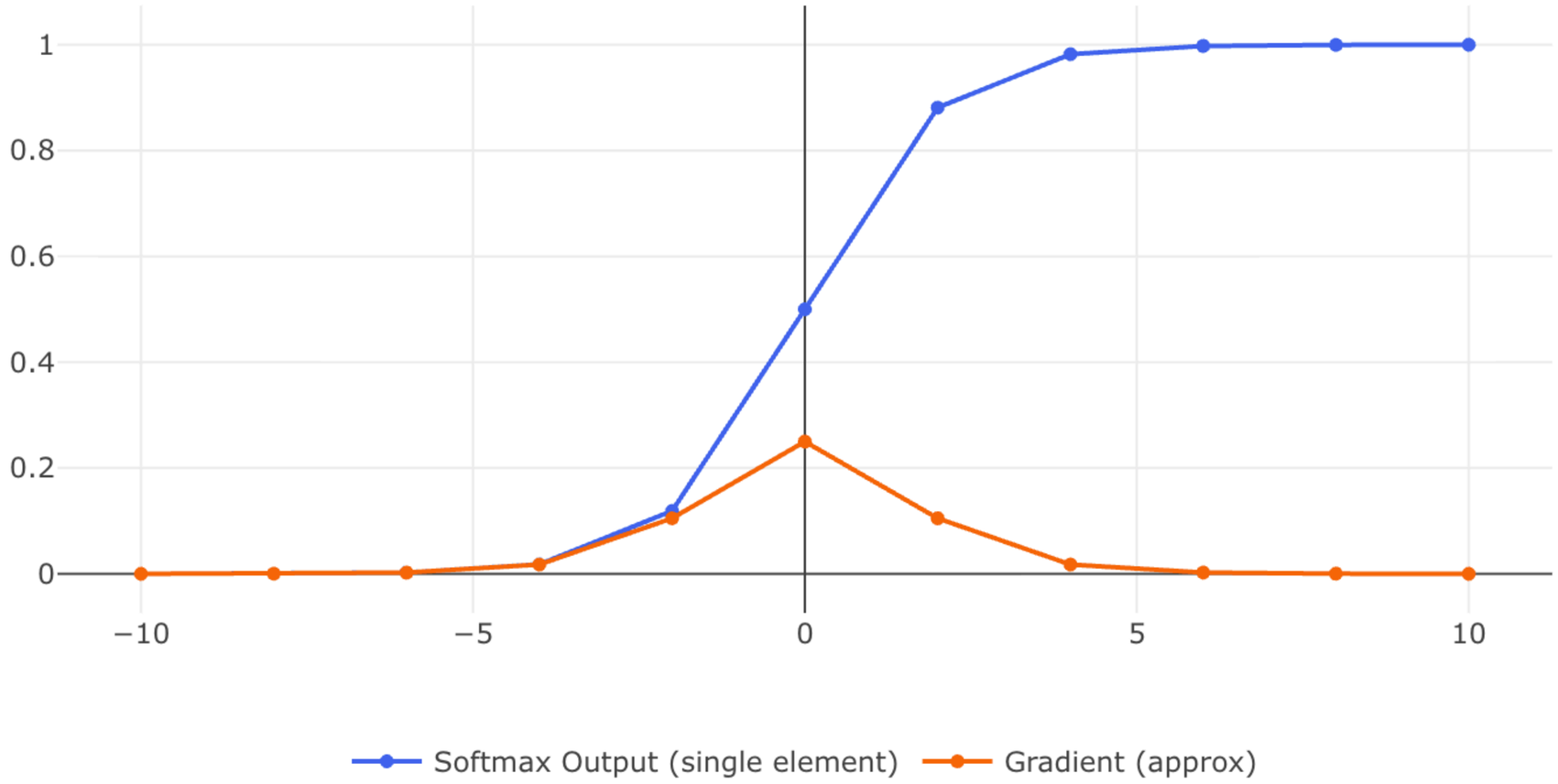
Scaled dot-product so $d_k = d_q$

- Finally, compute the weighted sum:

$$\mathbf{y}_i = \sum_{j=1}^{n} \alpha_{i,j} \mathbf{v}_j \in \mathbb{R}^{d_v} \qquad d_v = d_2$$

Why scaled?

- Large values can cause softmax to saturate
- Large dimensions, more likely to have outlier values (high variance)
- Scaling helps to bring the variance down



https://apxml.com/courses/foundations-transformers-architecture/chapter-2-attention-mechanism-core-concepts/scaled-dot-product-attention

Self-attention: matrix notation

$$X \in \mathbb{R}^{n \times d_1}$$

$$Q = XW^Q, W^Q \in \mathbb{R}^{d_1 \times d_q}$$

$$K = XW^K, W^K \in \mathbb{R}^{d_1 \times d_k}$$

$$V = XW^V, W^V \in \mathbb{R}^{d_1 \times d_v}$$

Note: the notation on this slide are following the original paper
(= the transpose of the matrices in the previous slide)

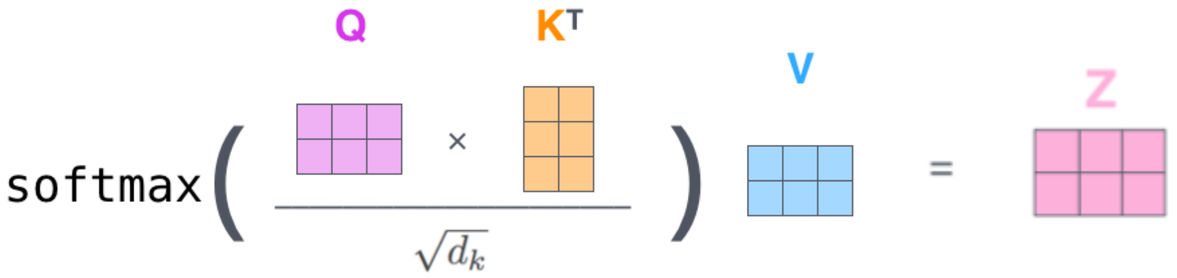
Each row corresponds a token

$n \times d_q$ $d_k \times n$

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

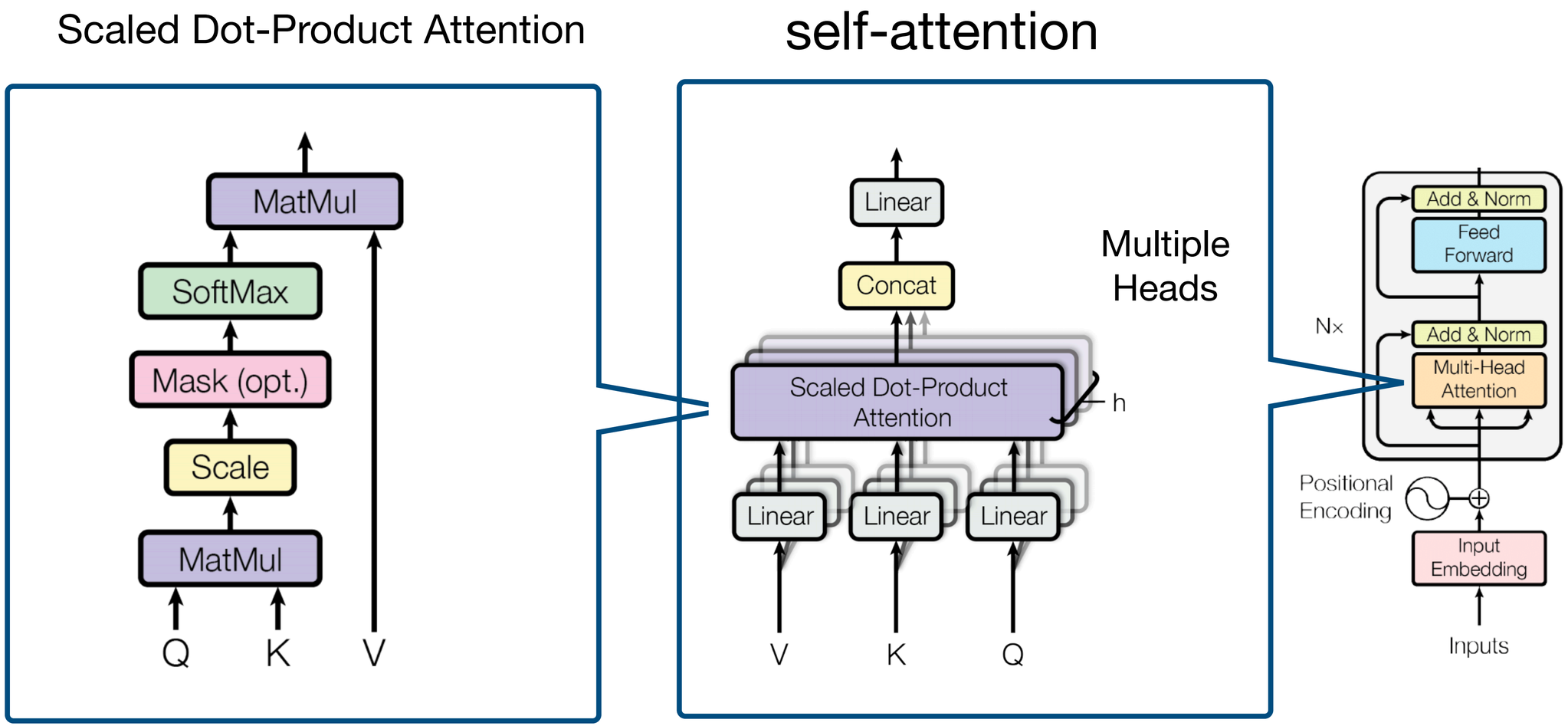
$n \times d_v$

Be careful to make sure
the softmax is over the correct dimension



(figure credit: Jay Alammar
http://jalammar.github.io/illustrated-transformer/)

Multi-head self-attention



Attention Is All You Need https://arxiv.org/pdf/1706.03762.pdf

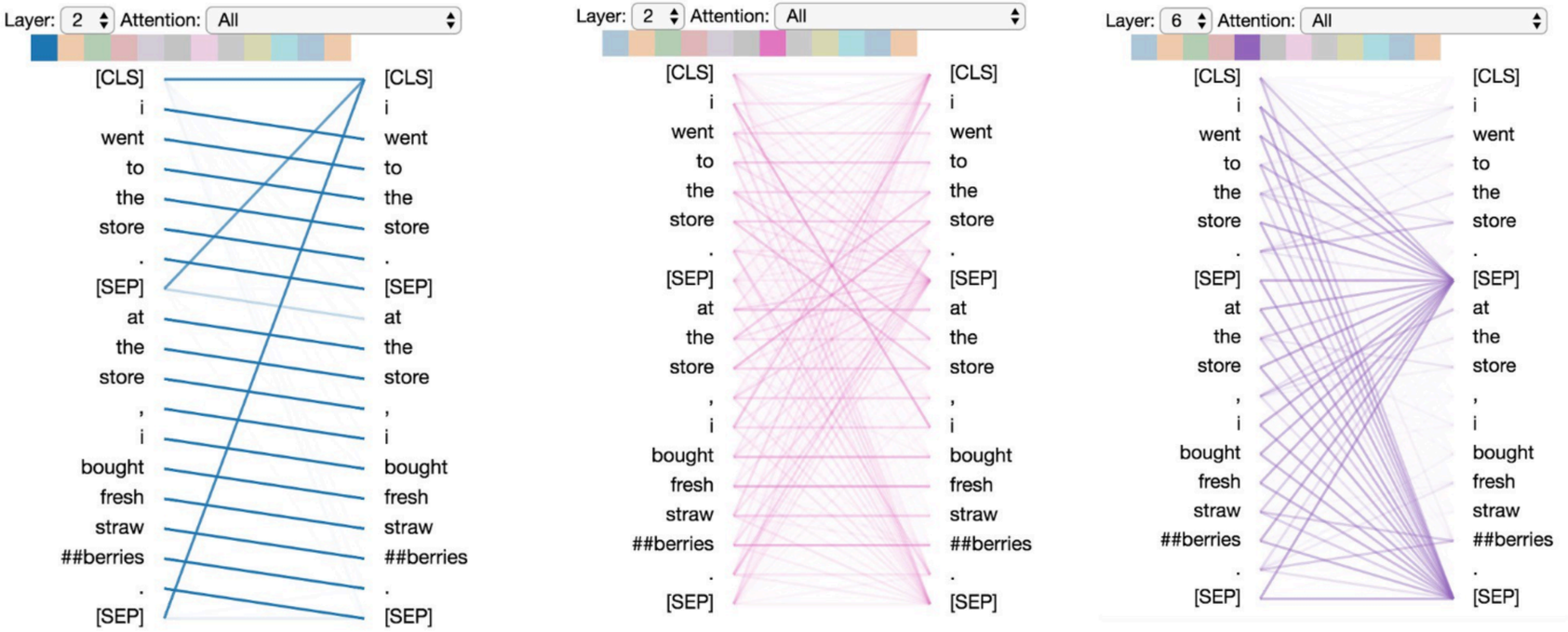
Multi-head self-attention

One head is not expressive enough. Let's have multiple heads!

$$\text{Multihead}(Q, K, V) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h)W^O$$

$$\text{head}_i = A(XW_i^Q, XW_i^K, XW_i^V)$$

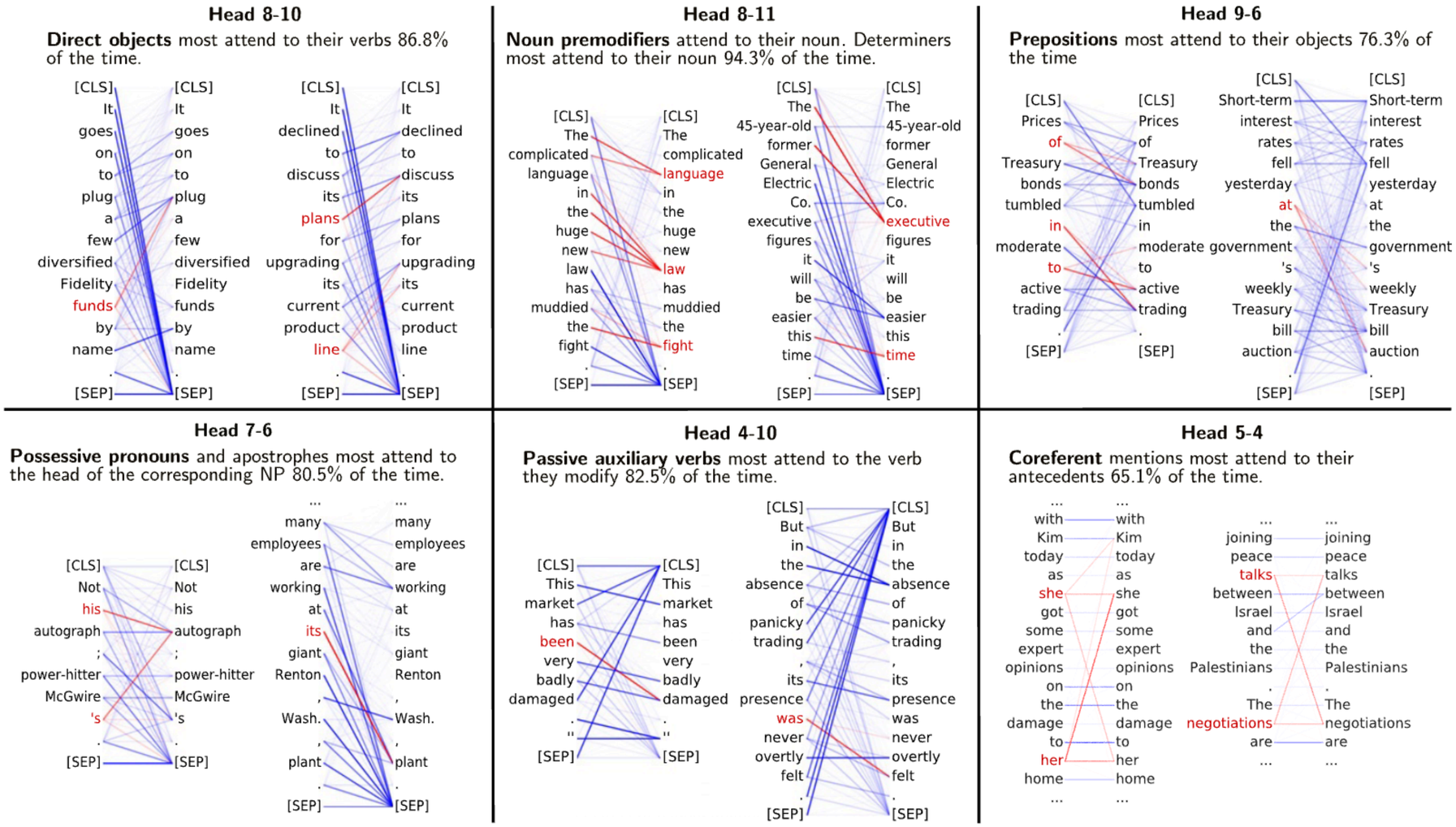
In practice, $h = 8$,
$d = d_{out}/h$, $W^O \in \mathbb{R}^{d_{out} \times d_{out}}$



https://github.com/jessevig/bertviz

Why different heads?

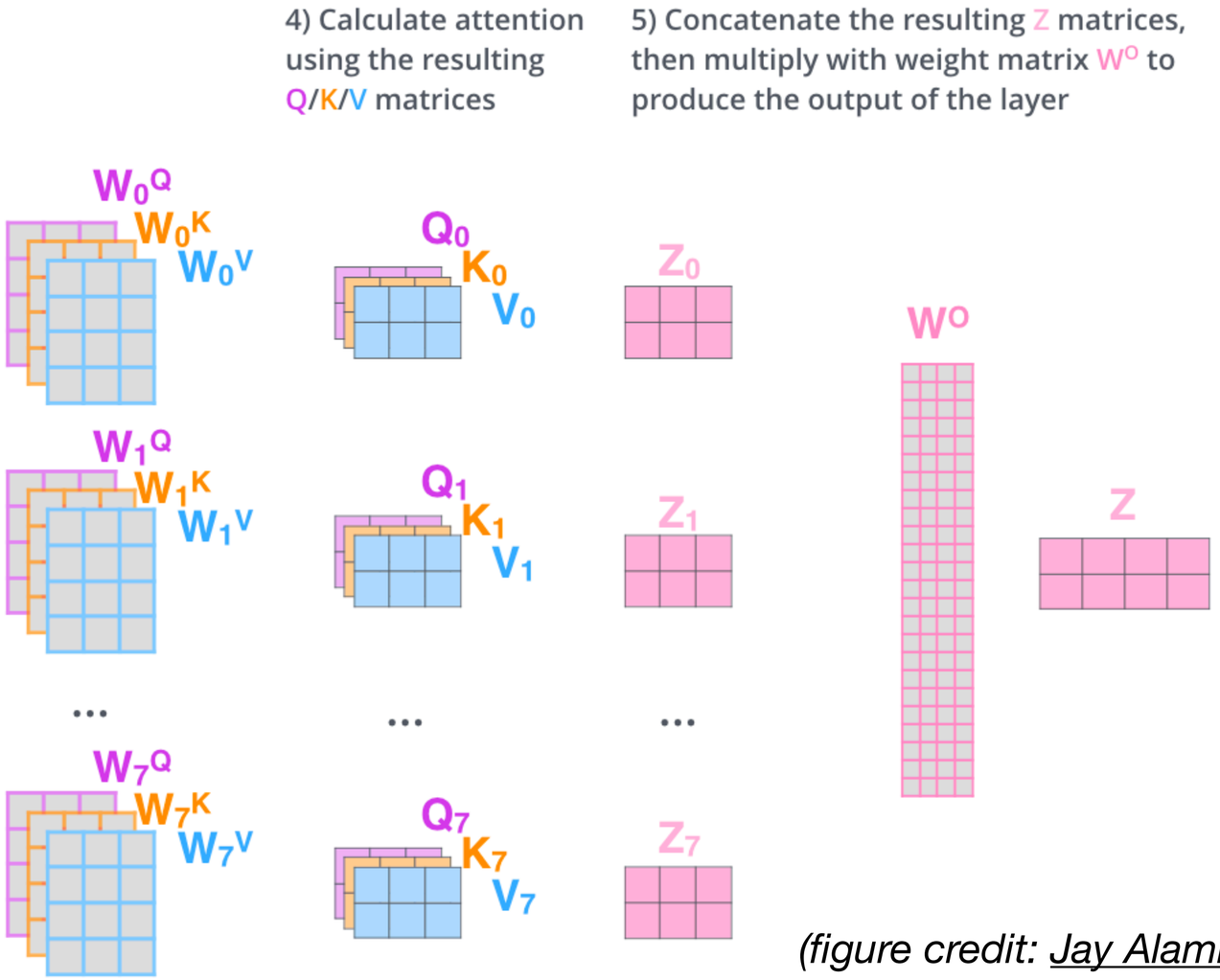
- Different heads learn to attend to different things



Emergent linguistic structure in artificial neural networks trained by self-supervision, Manning et al, PNAS 2019

Multiple heads

- Multiple (different) representations for each query, key, and values
- Different weight matrices —> different vectors
- Different ways for the words to interact with each other



(figure credit: Jay Alammar http://jalammar.github.io/illustrated-transformer/)

Multi-head attention

$$\text{Multihead}(Q, K, V) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h)W^O$$

$$\text{head}_i = A(XW_i^Q, XW_i^K, XW_i^V)$$

- In practice, we use a reduced dimension for each head.

$$W_i^Q \in \mathbb{R}^{d_1 \times d_q}, W_i^K \in \mathbb{R}^{d_1 \times d_k}, W_i^V \in \mathbb{R}^{d_1 \times d_v}$$

$$d_q = d_k = d_v = d/h$$

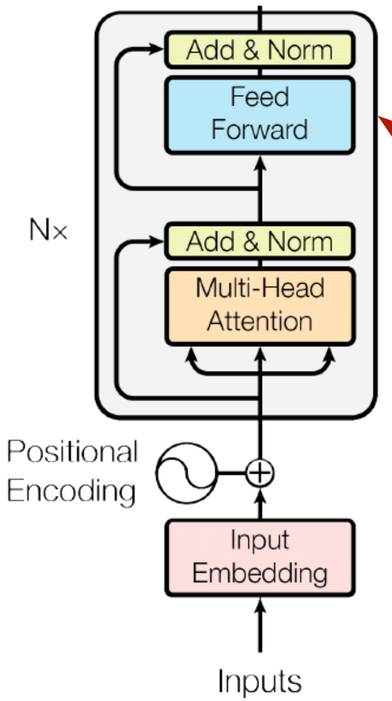
d = hidden size, h = # of heads

$$W^O \in \mathbb{R}^{d \times d_2}$$

If we stack multiple layers, usually $d_1 = d_2 = d$

- The total computational cost is similar to that of single-head attention with full dimensionality

Transformer Encoder



- Each Transformer block has two sub-layers
  - Multi-head attention
  - 2-layer feedforward NN (with ReLU)

Without FFNN: No non-linearity!

Adding nonlinearities

- There is no elementwise nonlinearities in self-attention; stacking more self-attention layers just re-averages value vectors

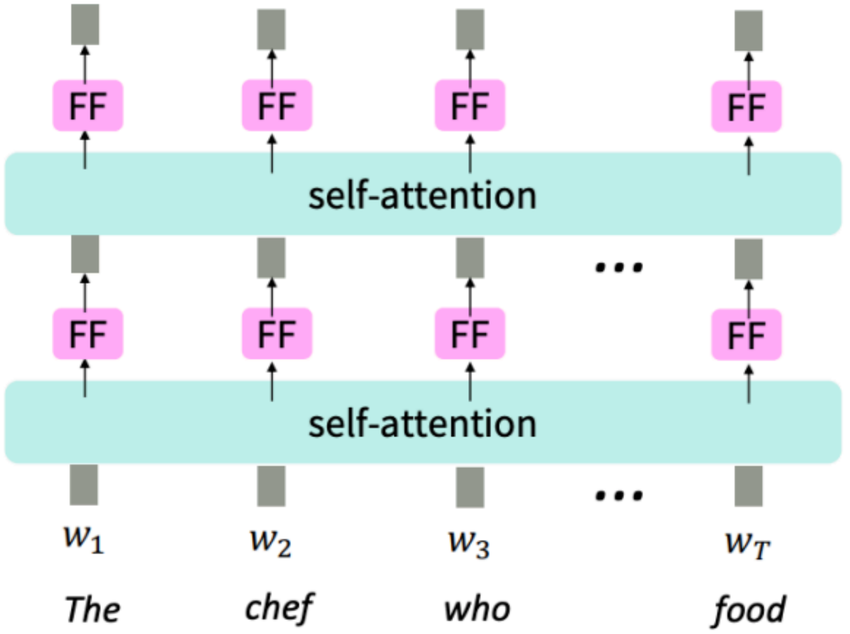
- Simple fix: add a feed-forward network to post-process each output vector

$$\text{FFN}(\mathbf{x}_i) = W_2 \text{ReLU}(W_1 \mathbf{x}_i + \mathbf{b}_1) + \mathbf{b}_2$$

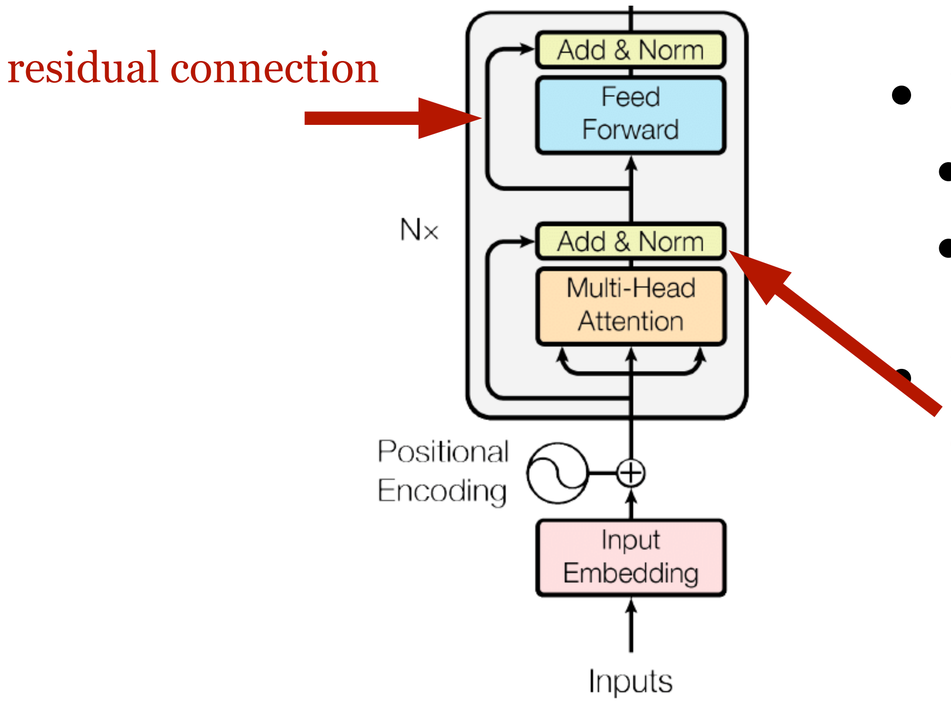
$$W_1 \in \mathbb{R}^{d_{ff} \times d}, \mathbf{b}_1 \in \mathbb{R}^{d_{ff}}$$

$$W_2 \in \mathbb{R}^{d \times d_{ff}}, \mathbf{b}_2 \in \mathbb{R}^{d}$$

In practice, they use $d_{ff} = 4d$



Transformer Encoder



- Each Transformer block has two sub-layers
  - Multi-head attention
  - 2-layer feedforward NN (with ReLU)
- Each sublayer has a residual connection and a layer normalization

$$\text{LayerNorm}(x + \text{SubLayer}(x))$$

(He et al, 2016): Residual connections

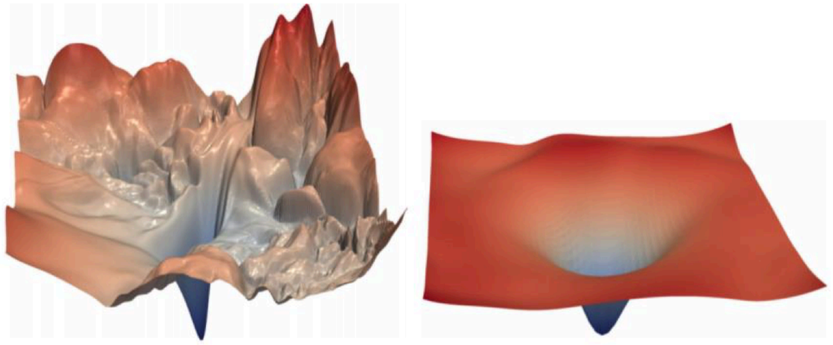
(Ba et al, 2016): Layer Normalization

He+ 2015 https://arxiv.org/abs/1512.03385

Residual Connections

Add input of a layer to output of that layer

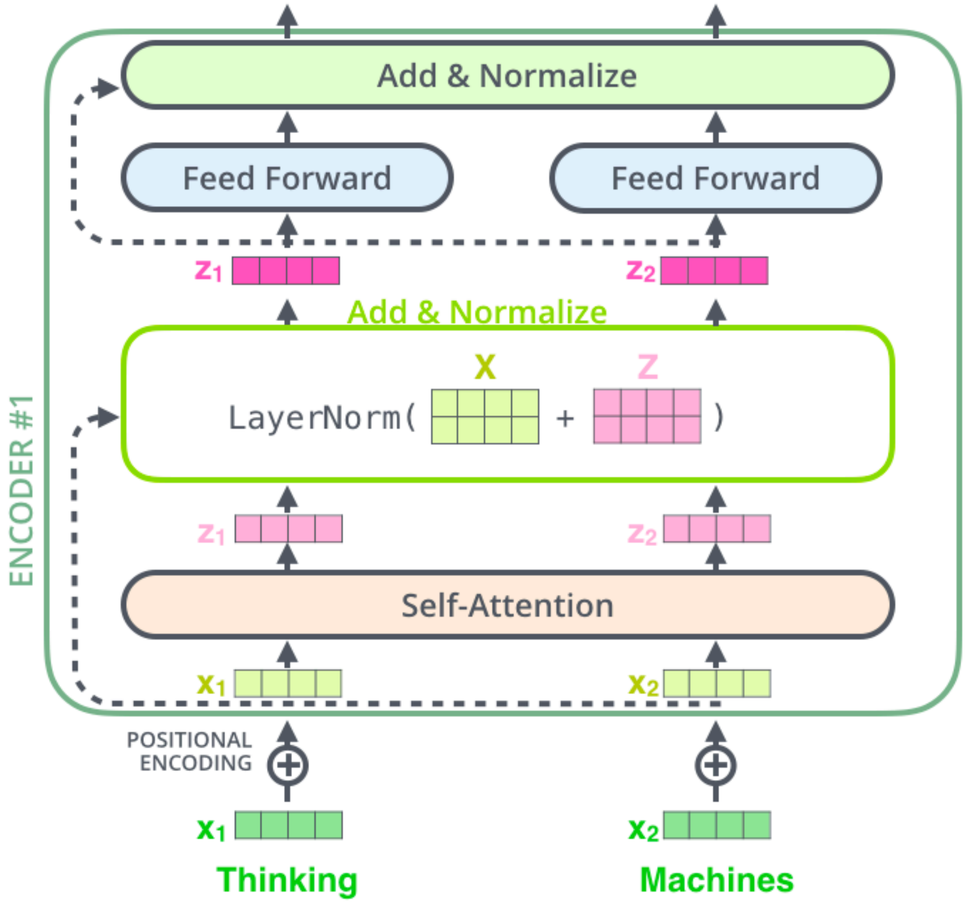
- $\mathbf{z}^{\ell+1} = f(\mathbf{z}^{\ell}) + \mathbf{z}^{\ell}$
- Local gradient is 1 for the identity function
- Easier to learn the difference from the identity function than to learn the function from scratch.



[no residuals] [residuals]

https://arxiv.org/pdf/1712.09913.pdf

Residual connections and Layer Normalization



(figure credit: Jay Alammar http://jalammar.github.io/illustrated-transformer/)

LayerNorm

- changes input features to have mean 0 and variance 1 per layer.
- Adds two more parameters

$$\mu^l = \frac{1}{H}\sum_{i=1}^{H} a_i^l \qquad \sigma^l = \sqrt{\frac{1}{H}\sum_{i=1}^{H}\left(a_i^l - \mu^l\right)^2}$$

$$h_i = \frac{g_i}{\sigma_i}\left(a_i - \mu_i\right) + b_i$$

- For more stable and efficient training

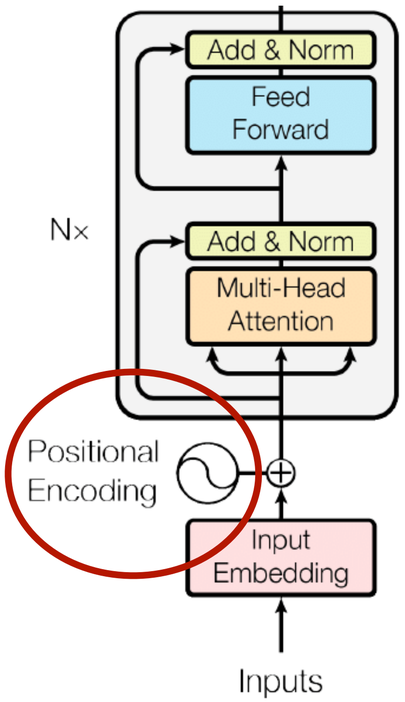
(Ba et al, 2016): Layer Normalization

Add & Norm

Residual Connections and Layer Norm

- Combine residual connection and layer norm into a single "Add & Norm" component
- Two choices:
  - Pre-norm (input): $\mathbf{z}^{\ell+1} = f(\text{LN}(\mathbf{z}^{\ell})) + \mathbf{z}^{\ell}$ https://arxiv.org/abs/2002.04745
  - Res-Post-norm (output): $\mathbf{z}^{\ell+1} = \text{LN}(f(\mathbf{z}^{\ell})) + \mathbf{z}^{\ell}$ https://arxiv.org/abs/2111.09883
  - Post-norm: $\mathbf{z}^{\ell+1} = \text{LN}(f(\mathbf{z}^{\ell}) + \mathbf{z}^{\ell})$ https://arxiv.org/abs/1706.03762
- Pre-norm leads to faster training.

Transformer Encoder



- Each Transformer block has two sub-layers
  - Multi-head attention
  - 2-layer feedforward NN (with ReLU)
- Each sublayer has a residual connection and a layer normalization

$$\text{LayerNorm}(x + \text{SubLayer}(x))$$

- Input layer has a positional encoding

Necessary for the model to know the position of the token

(He et al, 2016): Residual connections

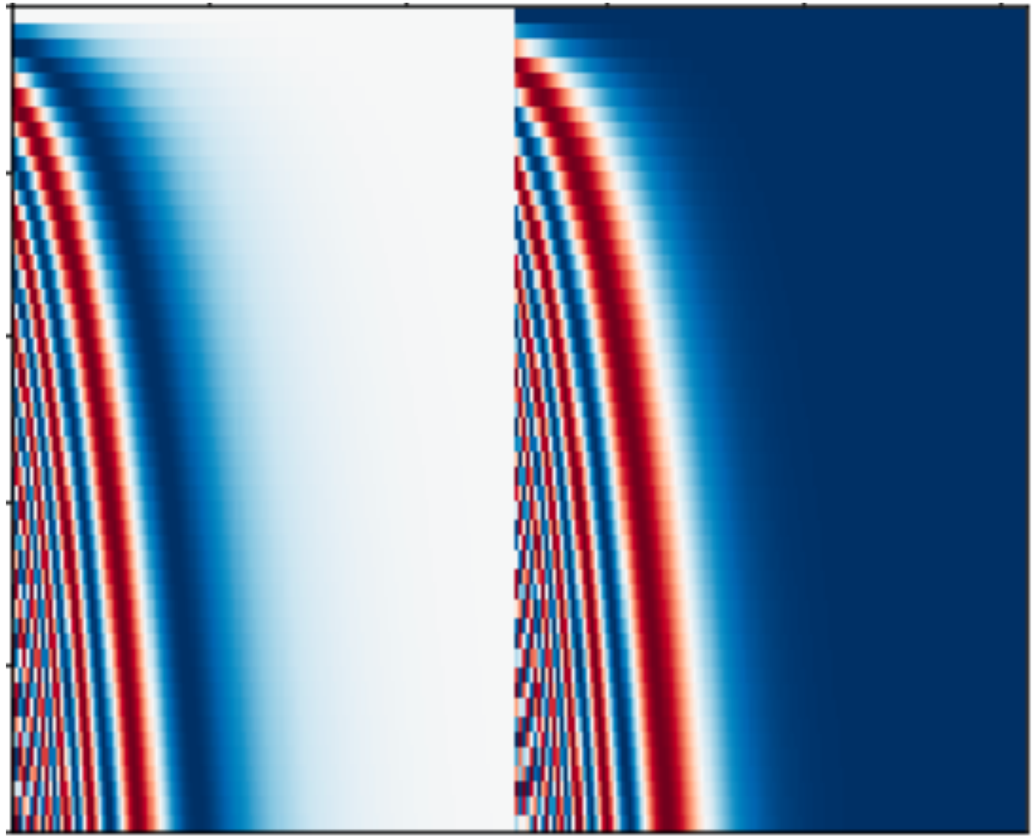
(Ba et al, 2016): Layer Normalization

Positional encoding

$$\overrightarrow{p_t}^{(i)} = f(t)^{(i)} := \begin{cases} \sin(\omega_k . t), & \text{if } i = 2k \\ \cos(\omega_k . t), & \text{if } i = 2k+1 \end{cases}$$

$$\omega_k = \frac{1}{10000^{2k/d}}$$

$$\overrightarrow{p_t} = \begin{bmatrix} \sin(\omega_1 . t) \\ \cos(\omega_1 . t) \\ \\ \sin(\omega_2 . t) \\ \cos(\omega_2 . t) \\ \\ \vdots \\ \\ \sin(\omega_{d/2} . t) \\ \cos(\omega_{d/2} . t) \end{bmatrix}_{d \times 1}$$



Sine Cosine

t = position

d = embedding dimension

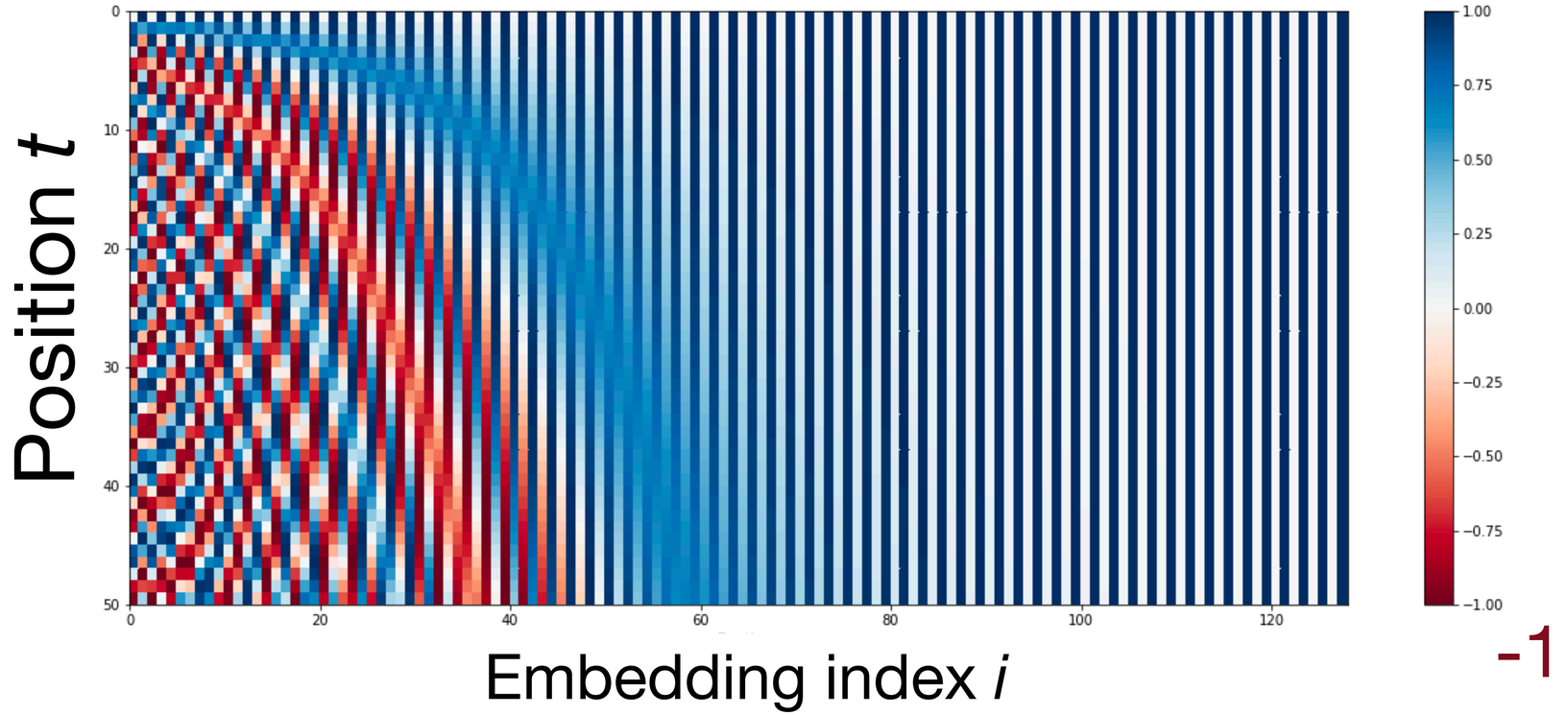
i = embedding index (0 to d-1)

Positional encoding

$$\overrightarrow{p_t}^{(i)} = f(t)^{(i)} := \begin{cases} \sin(\omega_k . t), & \text{if } i = 2k \\ \cos(\omega_k . t), & \text{if } i = 2k+1 \end{cases}$$

$$\omega_k = \frac{1}{10000^{2k/d}}$$

$$\overrightarrow{p_t} = \begin{bmatrix} \sin(\omega_1 . t) \\ \cos(\omega_1 . t) \\ \\ \sin(\omega_2 . t) \\ \cos(\omega_2 . t) \\ \\ \vdots \\ \\ \sin(\omega_{d/2} . t) \\ \cos(\omega_{d/2} . t) \end{bmatrix}_{d \times 1}$$



t = position

d = embedding dimension

i = embedding index (0 to d-1)

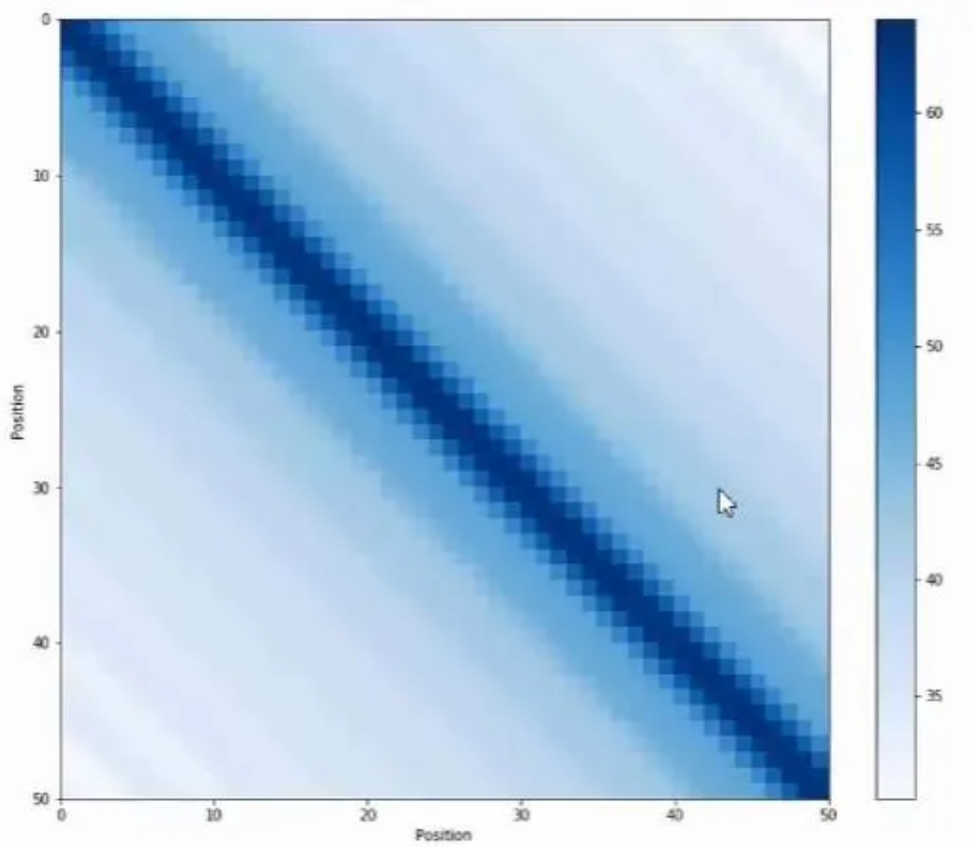
Positional encoding

$$p_t^{(i)} = f(t)^{(i)} := \begin{cases} \sin(\omega_k \cdot t), & \text{if} \quad i = 2k \\ \cos(\omega_k \cdot t), & \text{if} \quad i = 2k+1 \end{cases}$$

$$\omega_k = \frac{1}{10000^{2k/d}}$$

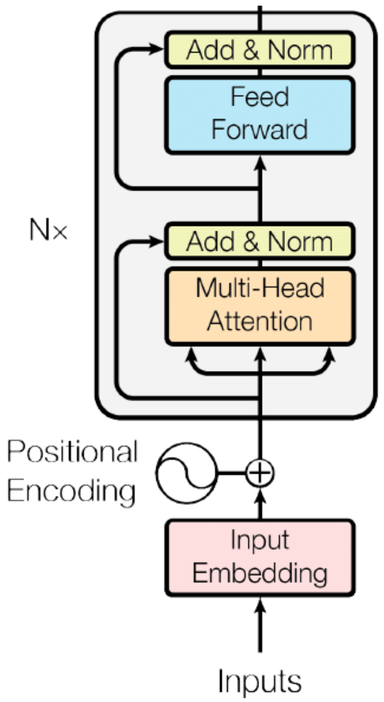
- Fixed absolute encoding
- Words that are closer to each other have higher dot product (relative attention is high)
- Can scale to long sequences

Dot product of positions



Transformer
Non-recurrent, deep model with attention

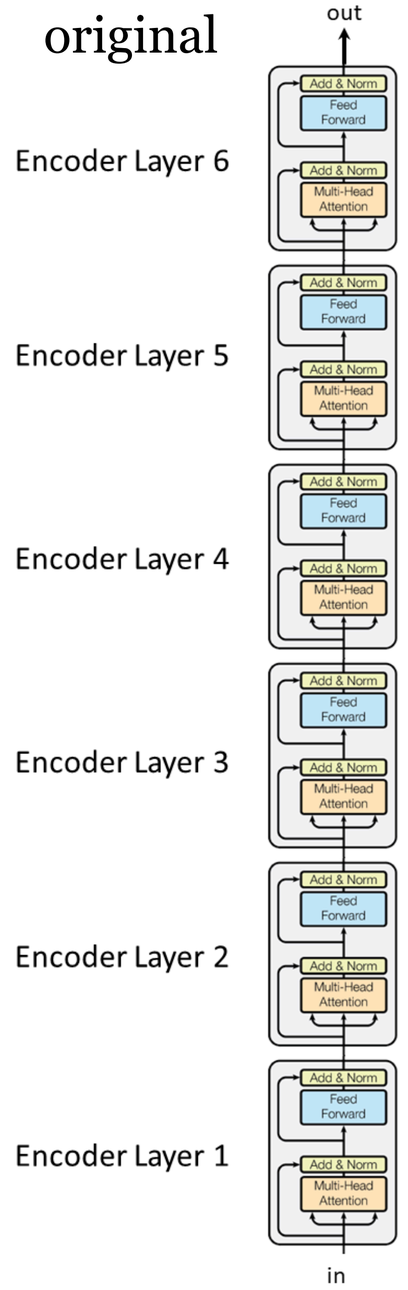
Transformer encoder



- Each Transformer block has two sub-layers
  - Multi-head attention
  - 2-layer feedforward NN (with ReLU)
- Each sublayer has a residual connection and a layer normalization

$$\text{LayerNorm}(x + \text{SubLayer}(x))$$

- Input layer has a positional encoding
- Input embedding is byte pair encoding (BPE)

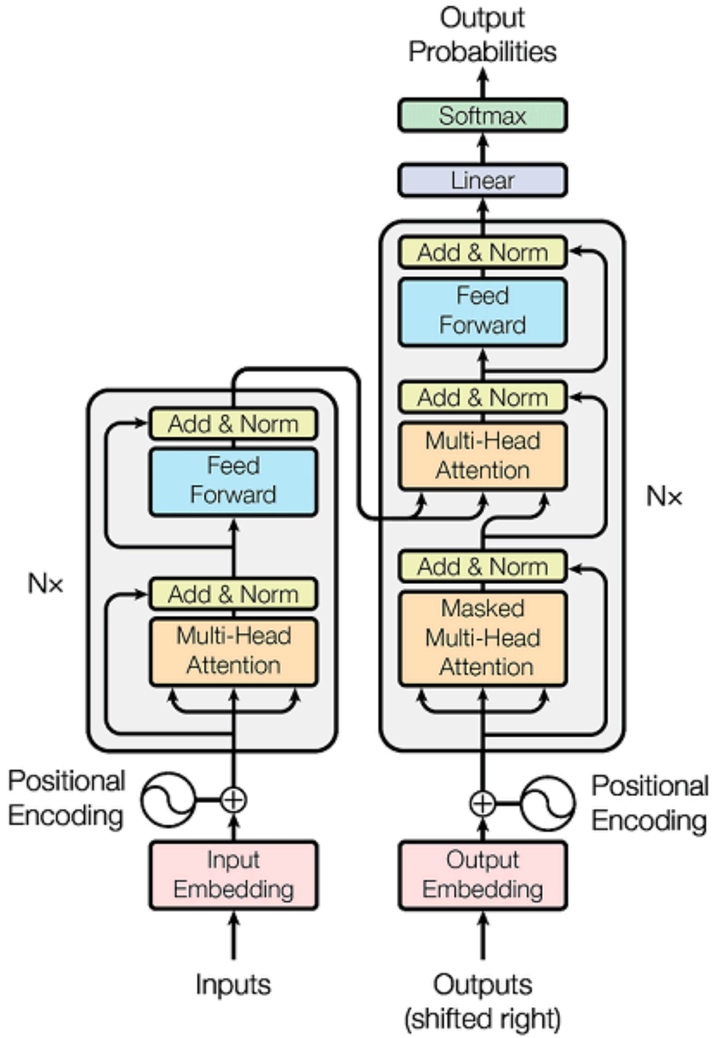


- BERT_base: 12 layers, 12 heads, hidden size = 768, 110M parameters
- BERT_large: 24 layers, 16 heads, hidden size = 1024, 340M parameters

(He et al, 2016): Residual connections

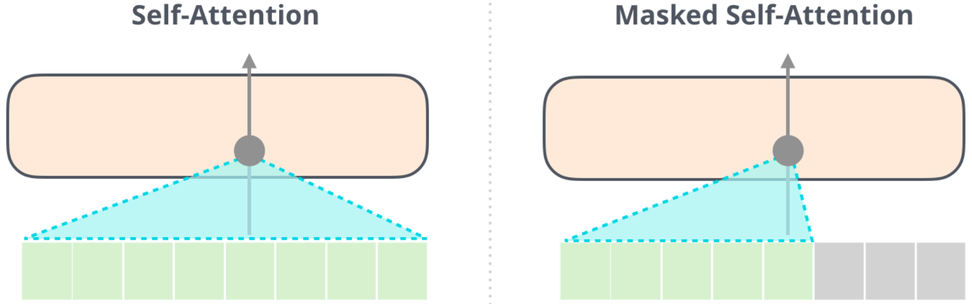
(Ba et al, 2016): Layer Normalization

Transformer decoder



- Encoder-Decoder Attention, where queries come from previous decoder layer and keys and values come from output of encoder

- Masked decoder self-attention on previously generated outputs



(figure credit: Jay Alammar http://jalammar.github.io/illustrated-gpt2/)

- also 6 layers (in original paper)

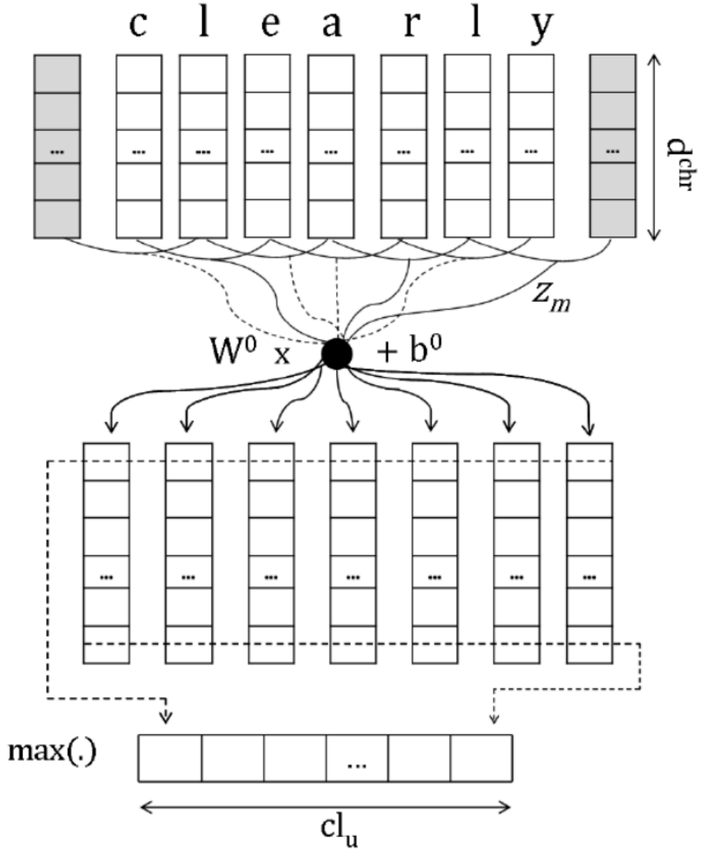
Subword modeling

Subword modeling

- Why subword modeling?
  - Captures morphology
  - Helps with OOV words
    - New words, spelling variants, misspellings, and noisy text
- Ways of incorporating subword modeling
  - Use subwords (word-pieces) as tokens
  - Hybrid architecture where part of the word embeddings come from subword modeling
- Used in most SOTA NLP methods
  - Character CNNs in ELMo
  - BPE (Byte Pair Encoding) in original Transformer paper
  - Wordpiece / sentence piece (in BERT)

NN over characters to build word representations

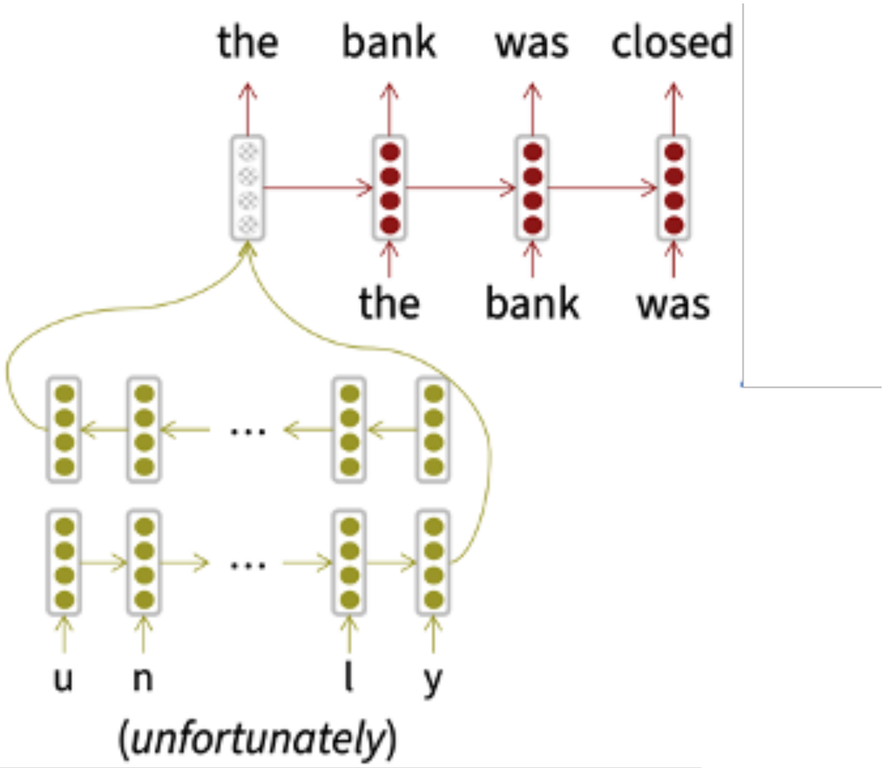
- Convolution over characters to generate word embeddings



Fixed window of word embeddings used for PoS tagging

Learning Character-level Representations for Part-of-Speech Tagging (Dos Santos and Zadrozny 2014)

- Same objective as word2vec but with characters
- Bi-directional LSTM to compute embedding



Char2vec: A joint model for word embedding and word morphology (Cao and Rei, 2016)

Byte Pair Encoding

- Originally a compression algorithm
  - Bottom up clustering
  - Most frequent byte pair -> a new byte
  - For words, replace bytes with character ngrams
- Automatically build vocabulary
  - Vocabulary is pieces of words (or character ngrams)
  - Deterministic algorithm that finds the common longest pieces of words to use in vocabulary

Byte Pair Encoding

count	bigram
5 + 2	l o
5 + 2	o w
2 + 6	w e
2	e r
6	n e
6	e w
6 + 3	e s
6 + 3	s t
3	w i
3	i d
3	d e

- A word (character ngram) segmentation algorithm
  - Start with a vocabulary of characters
  - Take most frequent ngram pair -> add the new ngram the to vocabulary

Dictionary

5 l o w
2 l o w e r
6 n e w e s t
3 w i d e s t

Vocabulary

l, o, w, e, r, n, w, s, t, i, d

Start with all characters in vocabulary

Byte Pair Encoding

count	bigram
5 + 2	l o
5 + 2	o w
2	w e
2	e r
6	n e
6	e w
6	w es
6 + 3	es t
3	w i
3	i d
3	d es

- A word (character ngram) segmentation algorithm
  - Start with a vocabulary of characters
  - Take most frequent ngram pair -> add the new ngram the to vocabulary

Dictionary

5 l o w
2 l o w e r
6 n e w **es** t
3 w i d **es** t

Vocabulary

l, o, w, e, r, n, w, s, t, i, d, **es**

Add a pair **(e,s)** with frequency 9

Byte Pair Encoding

count	bigram
5 + 2	l o
5 + 2	o w
2	w e
2	e r
6	n e
6	e w
6	w est
3	w i
3	i d
3	d est

- A word (character ngram) segmentation algorithm
  - Start with a vocabulary of characters
  - Take most frequent ngram pair -> add the new ngram the to vocabulary

Dictionary

5 l o w
2 l o w e r
6 n e w **est**
3 w i d **est**

Vocabulary

l, o, w, e, r, n, w, s, t, i, d, es, **est**

Add a pair **(es,t)** with frequency 9

Byte Pair Encoding

count	bigram
5 + 2	lo w
2	w e
2	e r
6	n e
6	e w
6	w est
3	w i
3	i d
3	d est

- A word (character ngram) segmentation algorithm
  - Start with a vocabulary of characters
  - Take most frequent ngram pair -> add the new ngram the to vocabulary

Dictionary

5	**lo** w
2	**lo** w e r
6	n e w est
3	w i d est

Vocabulary

l, o, w, e, r, n, w, s, t, i, d, es, est, **lo**

Add a pair **(l,o)** with frequency 7

Byte Pair Encoding

- When to stop
  - Have a target vocabulary size and stop when you reach it
- Deterministic, common longest piece segmentation of words
- Segmentation is only within words already identified by some prior tokenizers
- Automatically decide vocabulary to use (vocabulary is pieces of words - character ngrams)

Wordpiece/Sentencepiece

- Used in Google NMT
  - V1: wordpiece
  - V2: sentencepiece
- Difference way to select what ngram to add
  - Choose n-gram that maximally reduces perplexity
  - Greedy approximation to maximizing the language model log likelihood

Japanese and Korean Voice Search [Schuster and Nakajima 2012]

Wordpiece/Sentencepiece

- Used in Google NMT
  - V1: wordpiece
  - V2: sentencepiece
- Variant of wordpiece model is used in BERT
  - Common words are in vocabulary: at, Fairfax, 1910s
  - Other words built from workpieces: Hypatia = h ##yp ##ati ##a

Wordpiece/Sentencepiece

- Used in Google NMT
  - V1: wordpiece
  - V2: sentencepiece

only tokenizes inside words

works directly on raw text
(use special token _ for whitespace)

Also accommodates Unigram

https://github.com/google/sentencepiece

https://arxiv.org/pdf/1808.06226 [Kudo and Richardson 2018]

Unigram

- Finds vocabulary V that gives the maximum likelihood for a given vocabulary size. For given segmentation $T = (x_1, \ldots x_M)$ for input X, likelihood is given by

$$P(\mathbf{x}) = \prod_{i=1}^{M} p(x_i) \qquad x_i \in V \qquad \sum_{x \in V} p(x) = 1$$

- Need to estimate vocabulary V, possible segmentations, and token probabilities $p(x_i)$ jointly: use EM

- With vocabulary V and input X, can get optimal segmentation using Viterbi, or output multiple segmentations with their probabilities (with sampling with subword regularization).

Unigram Models for Subword Segmentation [Kudo 2018]

Unigram algorithm

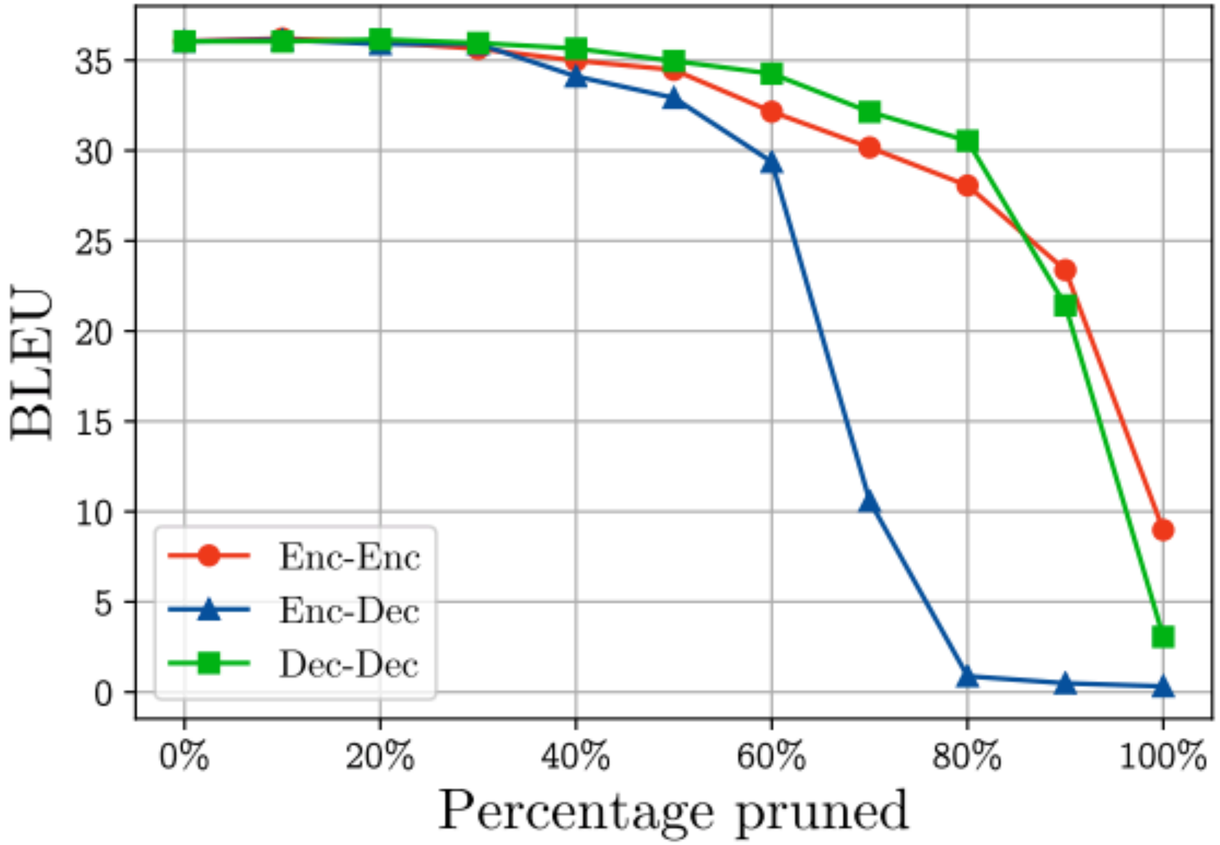
- Start with heuristic seed vocabulary V (that is sufficiently large)
  - For example: all characters + frequent substrings (can also run BPE)
- Repeat until $|V|$ reaches the desired vocabulary size
  - Given V, optimize likelihood of text data ($P(X)$) and tokenization T using EM algorithm
  - Compute $loss(x_i)$ for each token $x_i \in V$ where loss is how much the likelihood is to be reduced if token x_i is removed
  - Sort by loss and keep top 80% tokens in V (note: always keep single character subwords to avoid OOV)

Do we need all these heads?

3 types of attention: Enc-Enc, Enc-Dec, Dec-Dec

6 layers, 16 heads each layer for each type

- Can we prune away some of the heads of a trained model during test time?



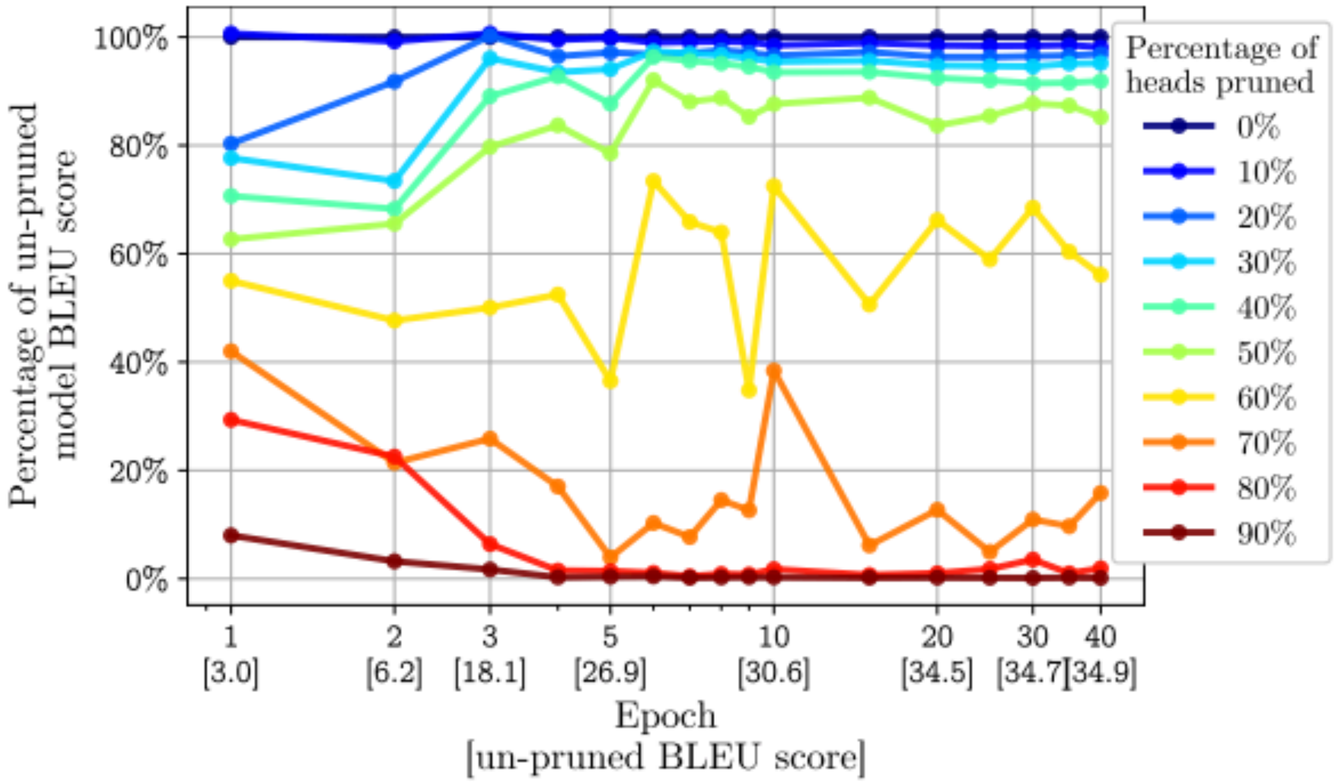
Are Sixteen Heads Really Better than One?
Michel, Levy, and Neubig, NeurIPS 2019

Do we need all these heads?

3 types of attention: Enc-Enc, Enc-Dec, Dec-Dec

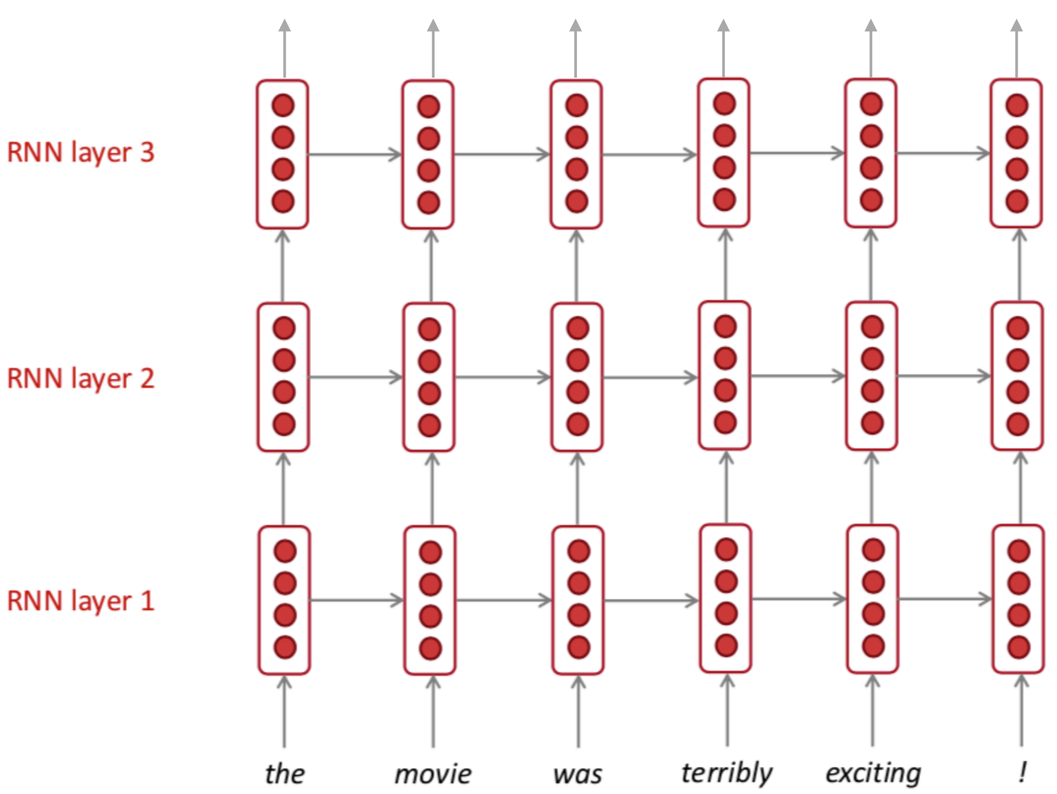
6 layers, 16 heads each layer for each type

- Can we train a good MT model with less heads?

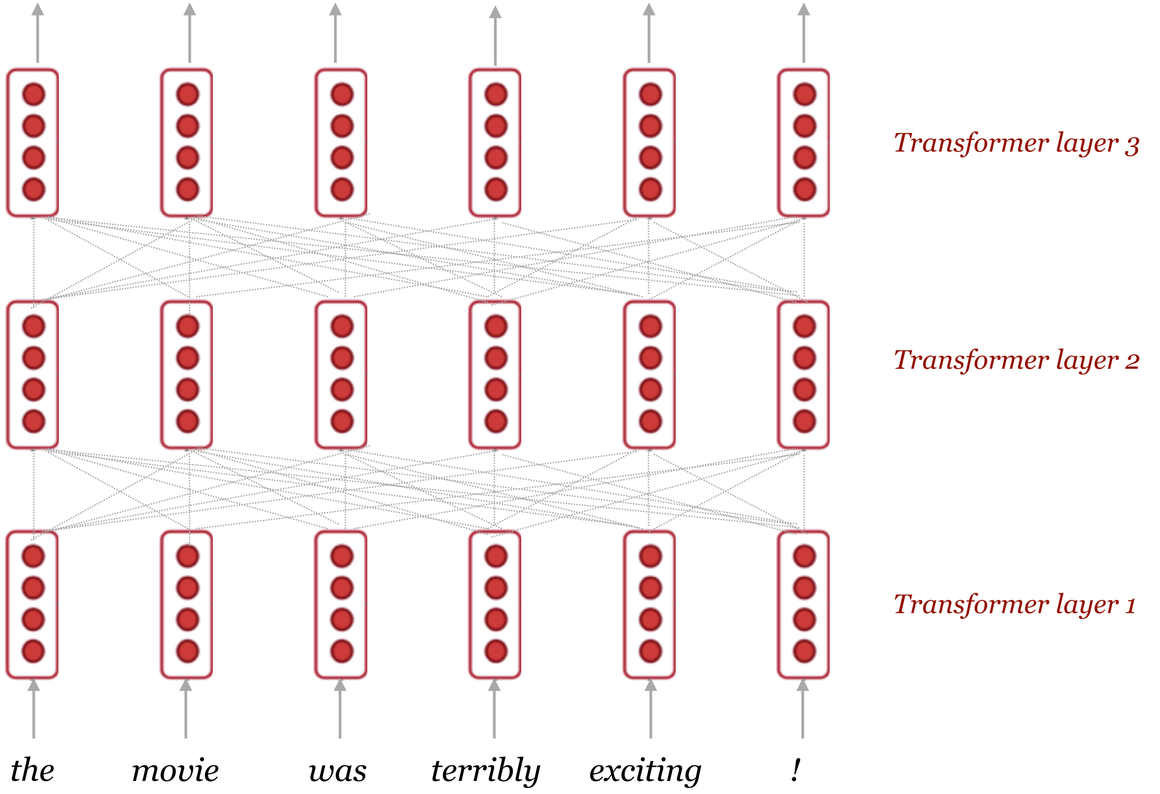


Are Sixteen Heads Really Better than One?
Michel, Levy, and Neubig, NeurIPS 2019

RNNs vs Transformers



RNN



Transformer

Useful Resources

Pytorch (https://pytorch.org/docs/stable/nn.html#transformer-layers)

nn.Transformer:

```
>>> transformer_model = nn.Transformer(nhead=16, num_encoder_layers=12)
>>> src = torch.rand((10, 32, 512))
>>> tgt = torch.rand((20, 32, 512))
>>> out = transformer_model(src, tgt)
```

nn.TransformerEncoder:

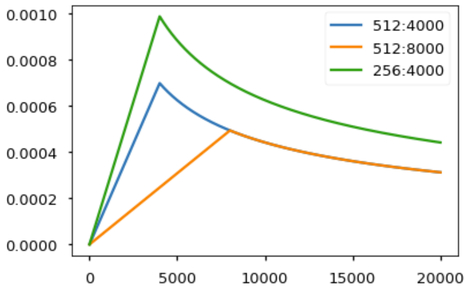
```
>>> encoder_layer = nn.TransformerEncoderLayer(d_model=512, nhead=8)
>>> transformer_encoder = nn.TransformerEncoder(encoder_layer, num_layers=6)
>>> src = torch.rand(10, 32, 512)
>>> out = transformer_encoder(src)
```

🤗 **Transformers**

https://github.com/huggingface/transformers

Other details

- Learning rate with warmup and decay



- Label smoothing

The Annotated Transformer:

http://nlp.seas.harvard.edu/2018/04/03/attention.html

A Jupyter notebook which explains how Transformer works line by line in PyTorch!

Useful resources

- Illustrated transformer: https://jalammar.github.io/illustrated-transformer/
  - 1.5 hour course on transformer-based LLMs
- 3Blue1Brown: https://www.youtube.com/watch?v=eMlx5fFNoYc
- With code: https://d2l.ai/chapter_attention-mechanisms-and-transformers/transformer.html
-

Perfomance on machine translation

Model	BLEU		Training Cost (FLOPs)	
	EN-DE	EN-FR	EN-DE	EN-FR
ByteNet [18]	23.75			
Deep-Att + PosUnk [39]		39.2		$1.0 \cdot 10^{20}$
GNMT + RL [38]	24.6	39.92	$2.3 \cdot 10^{19}$	$1.4 \cdot 10^{20}$
ConvS2S [9]	25.16	40.46	$9.6 \cdot 10^{18}$	$1.5 \cdot 10^{20}$
MoE [32]	26.03	40.56	$2.0 \cdot 10^{19}$	$1.2 \cdot 10^{20}$
Deep-Att + PosUnk Ensemble [39]		40.4		$8.0 \cdot 10^{20}$
GNMT + RL Ensemble [38]	26.30	41.16	$1.8 \cdot 10^{20}$	$1.1 \cdot 10^{21}$
ConvS2S Ensemble [9]	26.36	**41.29**	$7.7 \cdot 10^{19}$	$1.2 \cdot 10^{21}$
Transformer (base model)	27.3	38.1	**$3.3 \cdot 10^{18}$**	
Transformer (big)	**28.4**	**41.8**	$2.3 \cdot 10^{19}$	

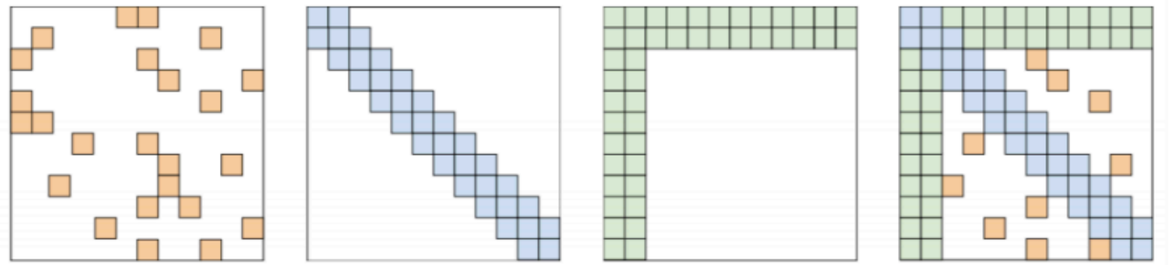
Attention is all you need
Vaswani et al, NeurIPS 2017

Transformer Pros and Cons

- Pros
  - Easier to capture dependencies: we draw attention between every pair of words
  - Easier to parallelize (matrix operations)

$$Q = XW^Q, W^Q \in \mathbb{R}^{d_1 \times d_q}$$
$$K = XW^K, W^K \in \mathbb{R}^{d_1 \times d_k}$$
$$V = XW^V, W^V \in \mathbb{R}^{d_1 \times d_v}$$

- Cons
  - Quadratic computation in self-attention
    - Can become very slow when the sequence length is large

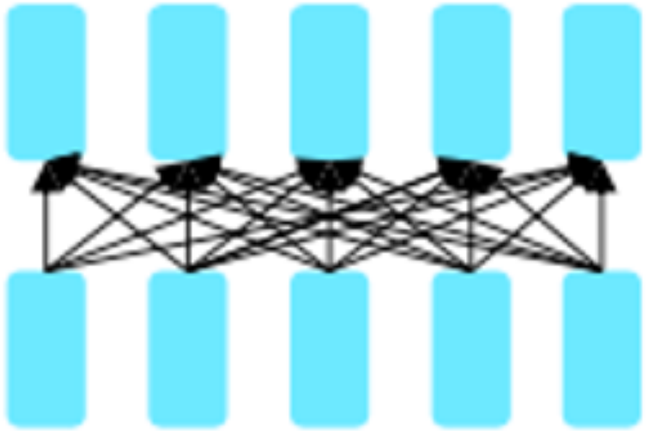


  - Are these positional representations enough to capture positional information?

Transformers for pretraining

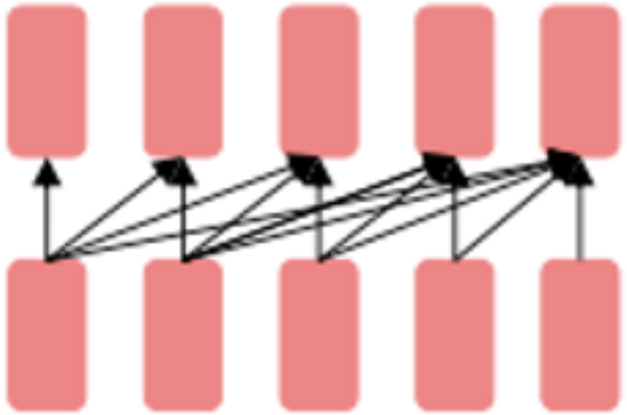
- Self-supervised Transformer based models shattered language understanding benchmarks in NLP in 2018.
- Trained on large text corpus with self-supervised objectives and then transferred.

Encoder only



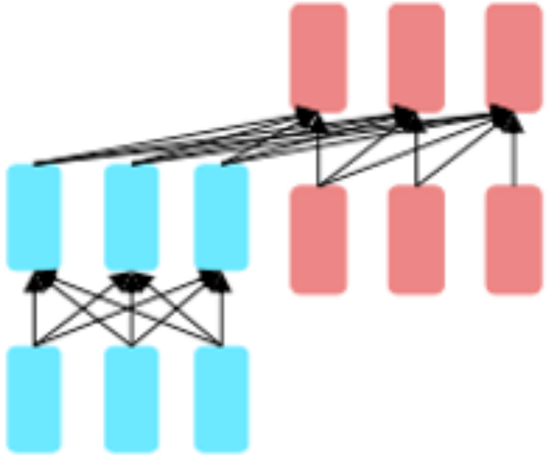
- Masked language models
- Bidirectional context
- BERT + variants (e.g. RoBERTa)

Decoder only



- Language models
- Can't condition on future words, good for generation
- GPT, LLaMa, PaLM

Encoder-Decoder



- Combine benefits of both
- Original Transformer, UniLM, BART, T5

Slide adapted from: Stanford CS224n, John Hewitt