Try this problem

$$\boxed{\underset{1\times 2}{q_i} = \underset{1\times 6}{x_i}\ \underset{6\times 2}{W_Q}}$$

Suppose, you give the following input to your transformer encoder: {flying, arrows} The input embeddings for these two words are **[0,1,1,1,1,0] and [1,1,0,-1,-1,1]**, respectively. Suppose you are trying to represent the first word 'flying' with the help of self-attention in the first encoder. For the first attention head, the query, key and value matrices just take the 2 dimensions from the input each. Thus, the first 2 dimensions define the query vector, and so on. What will be the self-attention output for the word 'flying' corresponding to this attention head. You are using the scaled dot vector.

$$[0\ 1\ 1\ 1\ 1\ 0]\quad W_Q = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \end{bmatrix} = [0\ \ 1]$$

?

$$W_K = \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 1 & 0 \\ 0 & 1 \\ 0 & 0 \\ 0 & 0 \end{bmatrix} \qquad W_V = \begin{bmatrix} , & ' \\ , & ' \\ 0 & 0 \\ 1 & 0 \\ 0 & 1 \end{bmatrix}$$

Try this problem

q1: [0,1], k1: [1,1],, v1: [1,0] ✓
q2: [1,1], k2: [0,-1], v2: [-1,1] ✓
a1?

$$= 0.81\,[1, 0] + 0.19\,[-1, 1]$$

$$= [0.62 \quad 0.19]$$

$$\frac{q1 \cdot k1}{\sqrt{d_k}} = \frac{1}{\sqrt{2}}$$

$d_k = d_q$ always

$$\frac{q_1 \cdot k_2}{\sqrt{d_k}} = \frac{-1}{\sqrt{2}}$$

$$\text{softmax}\left[\frac{1}{\sqrt{2}}, -\frac{1}{\sqrt{2}}\right]$$

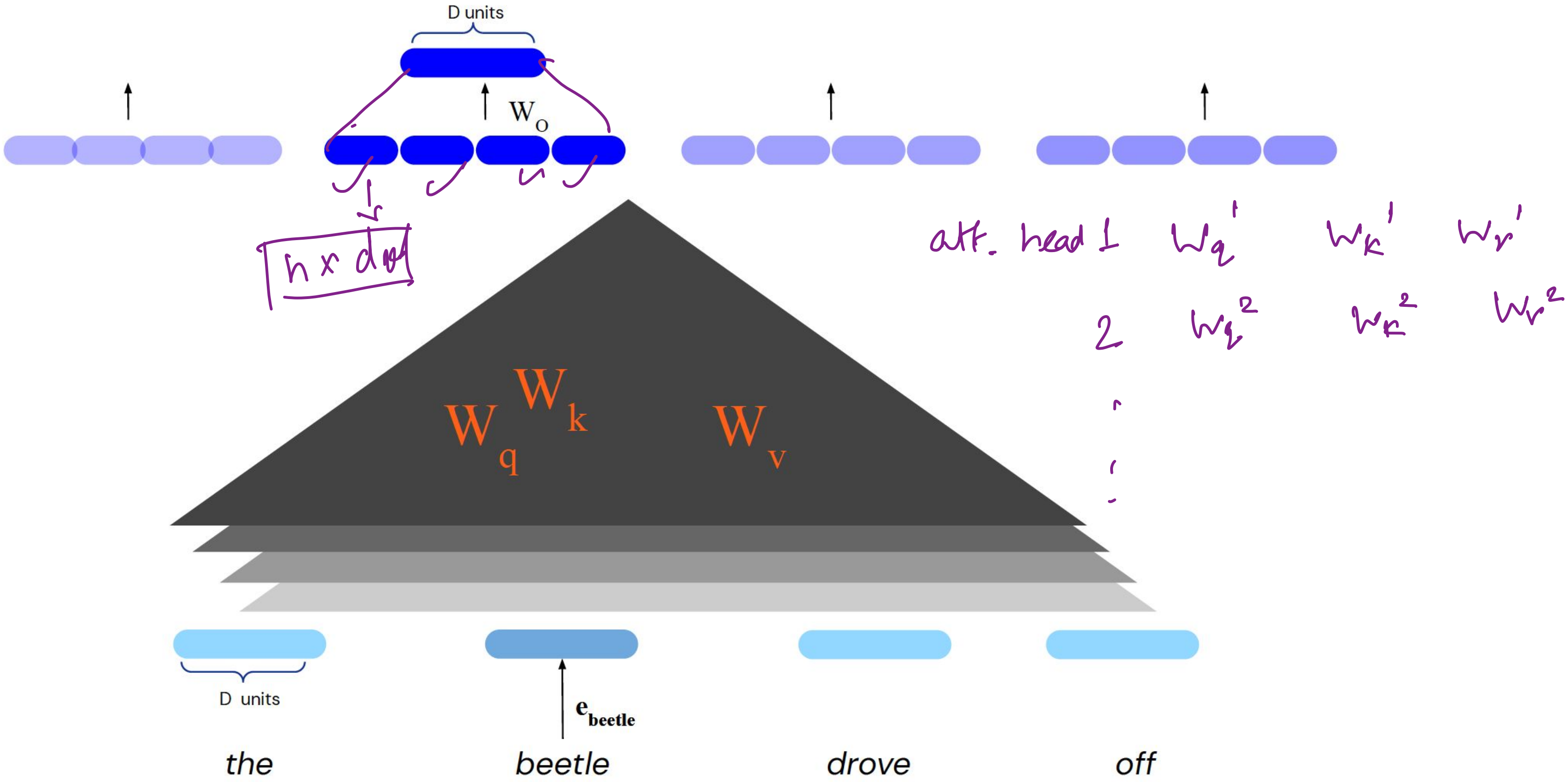
$$= [0.81, 0.19]$$

$h_i^{d}\ W\ h_i^{Te}$

$x_i W_Q \qquad (x_j W_K)^T$

$$x_i \underbrace{W_Q W_K^T}\, x_j$$

Multi-head Attention



Source: *https://deepmind.com/learning-resources/deep-learning-lecture-series-2020*

Why Multi-head attention?

- What if we want to look in multiple places in the sentence at once?
  - For word i, maybe we want to focus on different j for different reasons?

- We'll define multiple attention "heads" through multiple Q,K,V matrices

- Each attention head performs attention independently

- Then the outputs of all the heads are combined!

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

Why Multi-head attention?

Prior work identified three important types of heads by looking at attention matrices

[Analyzing Multi-Head Self-Attention: Specialized Heads Do the Heavy Lifting, the Rest Can Be Pruned: https://arxiv.org/abs/1905.09418]

1. **Positional** heads that attend mostly to their neighbor.
2. **Syntactic** heads that point to tokens with a specific syntactic relation.
3. Heads that point to **rare words** in the sentence.

object, subject, verb, etc

Source: https://theaisummer.com/self-attention/

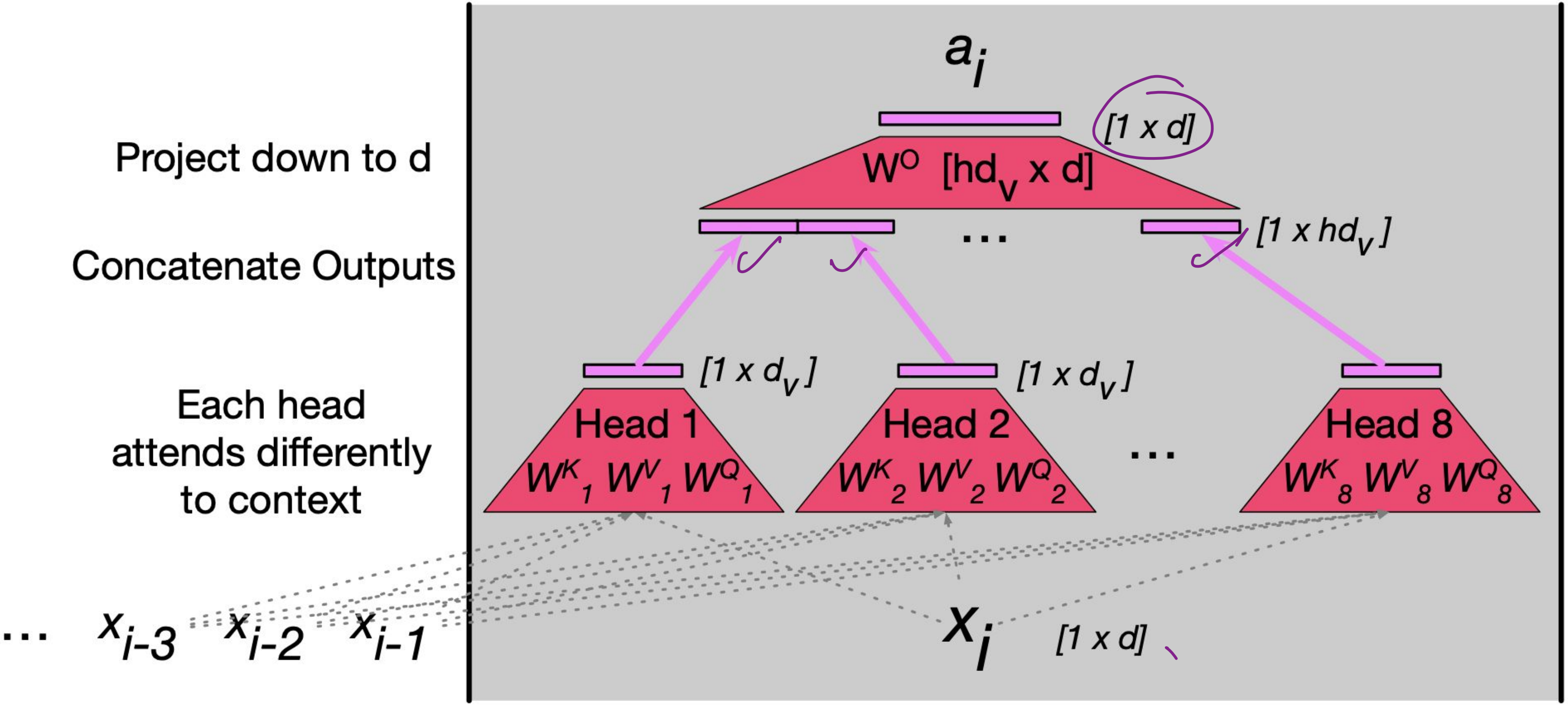
Multi-Head Attention: In Equations

- Each head might be attending to the context for different purposes
  - Different linguistic relationships or patterns in the context

$$
\begin{aligned}
\mathbf{q}_i^c = \mathbf{x}_i\mathbf{W}^{\mathbf{Qc}};\quad \mathbf{k}_j^c = \mathbf{x}_j\mathbf{W}^{\mathbf{Kc}};\quad \mathbf{v}_j^c &= \mathbf{x}_j\mathbf{W}^{\mathbf{Vc}};\quad \forall c \quad 1 \leq c \leq h \\
\text{score}^c(\mathbf{x}_i,\mathbf{x}_j) &= \frac{\mathbf{q}_i^c \cdot \mathbf{k}_j^c}{\sqrt{d_k}} \\
\alpha_{ij}^c &= \text{softmax}(\text{score}^c(\mathbf{x}_i,\mathbf{x}_j)) \\
\mathbf{head}_i^c &= \sum \alpha_{ij}^c \mathbf{v}_j^c \\
\mathbf{a}_i &= \overbrace{(\mathbf{head}^1 \oplus \mathbf{head}^2 \ldots \oplus \mathbf{head}^h)}^{\text{Concatenate}}\mathbf{W}^O \\
\text{MultiHeadAttention}(\mathbf{x}_i,[\mathbf{x}_1,\cdots,\mathbf{x}_N]) &= \mathbf{a}_i
\end{aligned}
$$

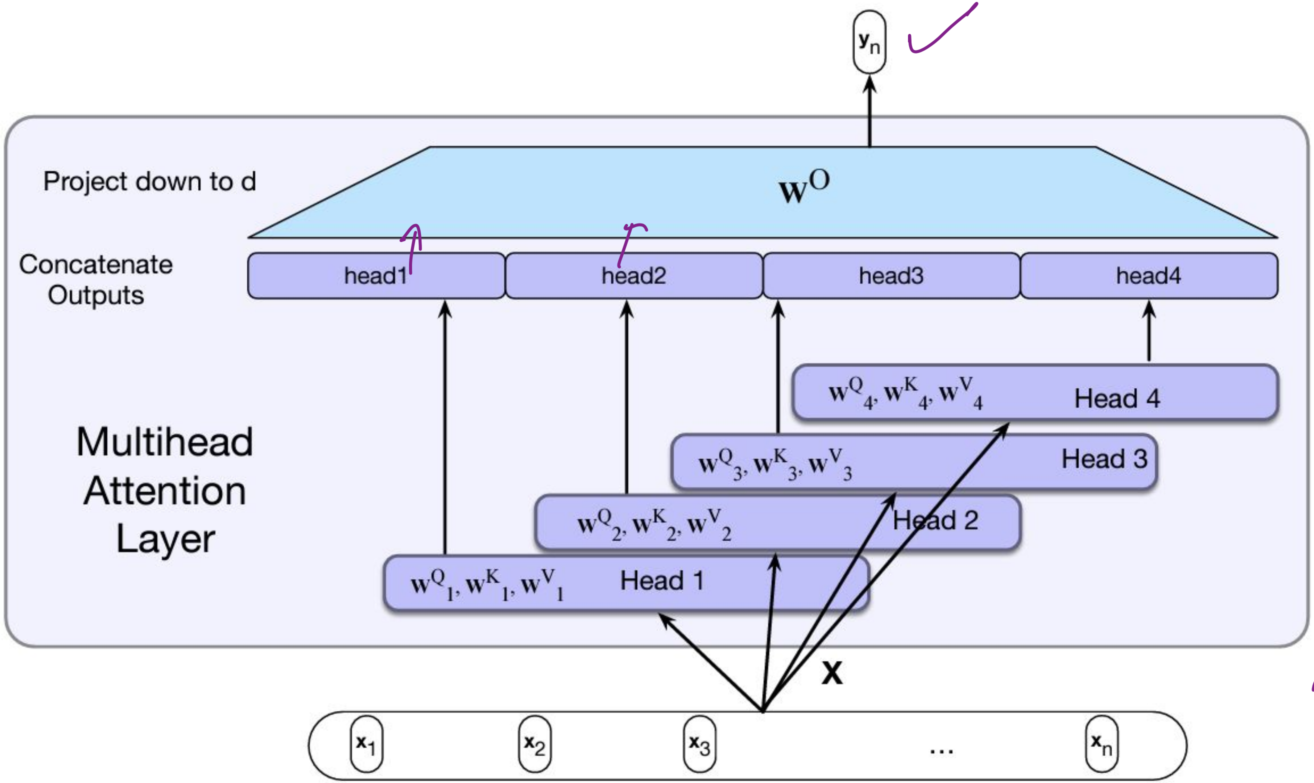
https://web.stanford.edu/~jurafsky/slp3/

Multi-head attention



https://web.stanford.edu/~jurafsky/slp3/

Multi-head Attention Layer



More on dimensions

Model dimension: d (=512)

Query, Key, Value dimensions: d_q, d_k, d_v (=64 each)

Projection matrices: $d \times d_k$ (= 512 x 64 each)

The output at each head: d_v

For "h" (=8) multi-heads: hd_v

To project it back to model dimension: W^O: $d \times hd_v$

W^O: $hd_v \times d$

512x512

https://web.stanford.edu/~jurafsky/slp3/

Single att head:

$$[3 \times 512 \times 64] \times 8 + 512 \times 512$$

$$= 512 \times 512 \times (3+1)$$

$$= \underline{4d^2}$$

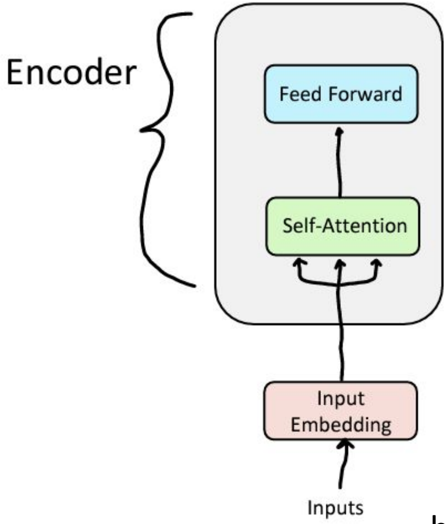
$4 \times 512 \times 512$

$\boxed{\approx 1m}$

Feed-forward Layer

Problem: Since there are no element-wise non-linearities, self-attention is simply performing a re-averaging of the value vectors.

Easy fix: Apply a feedforward layer to the output of attention, providing non-linear activation (and additional expressive power).

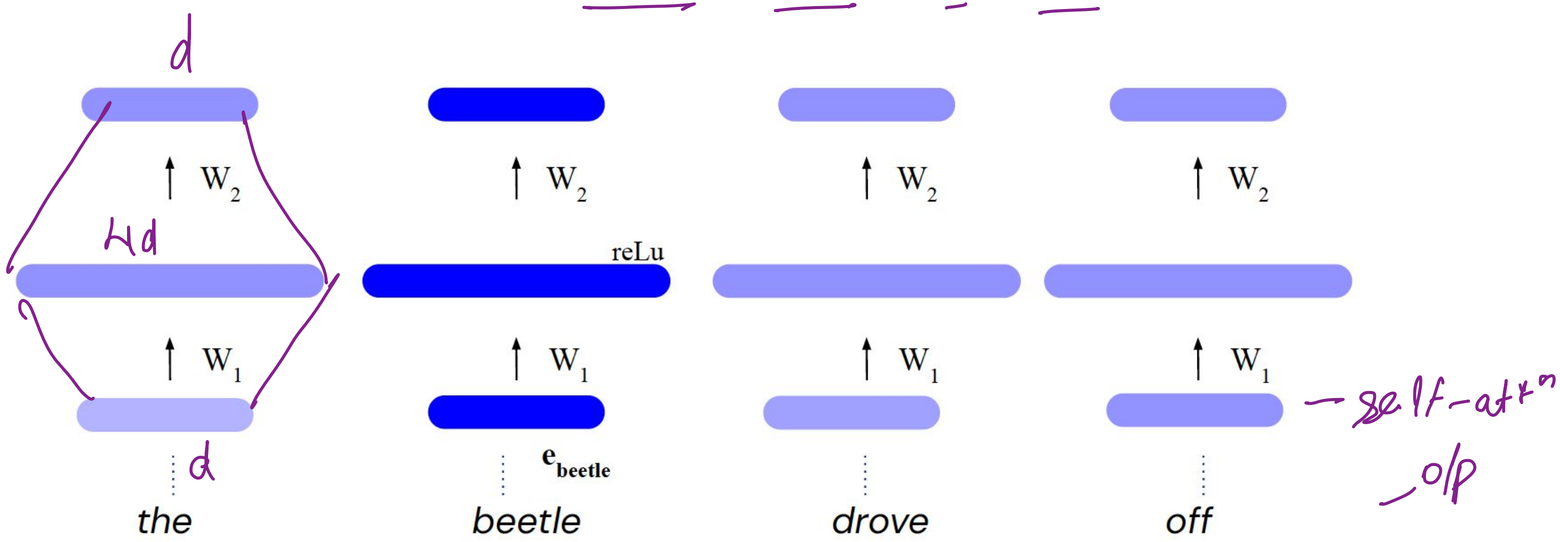


https://web.stanford.edu/class/cs224n/

Feed-forward Layer

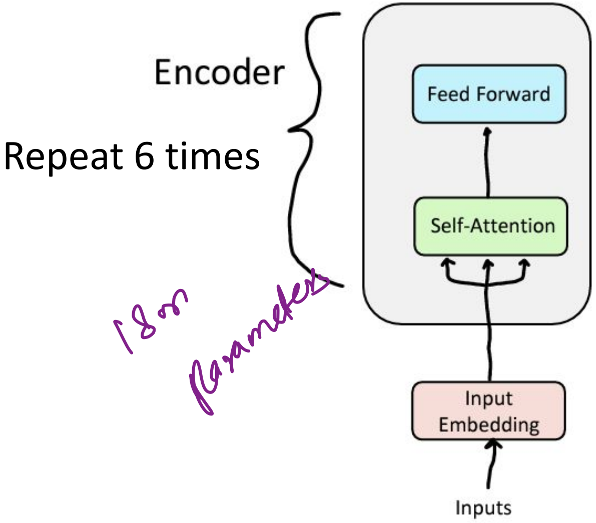
$8d^2$ $\simeq 2m$

$$\mathrm{FFN}(\mathbf{x}_i) = \underline{\mathrm{ReLU}}(\mathbf{x}_i \underline{\mathbf{W}_1} + \underline{b_1})\underline{\mathbf{W}_2} + b_2$$



Source: *https://deepmind.com/learning-resources/deep-learning-lecture-series-2020*

How to make this work for deep networks?



Training Trick #1: Residual Connections
Training Trick #2: LayerNorm
Training Trick #3: Scaled Dot Product Attention

$\sqrt{d_k}$

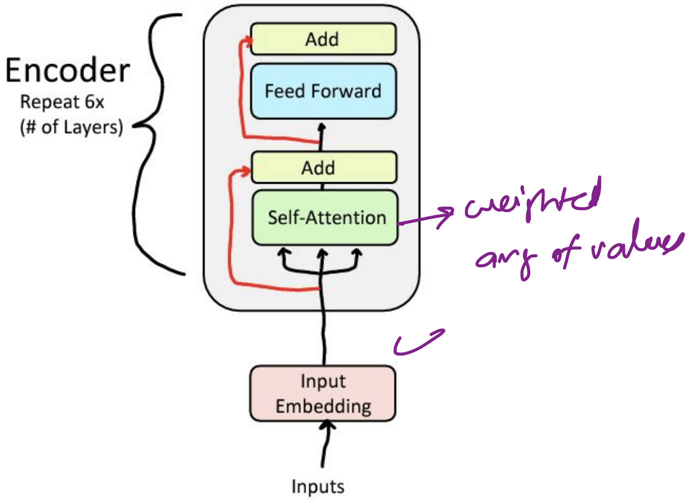
https://web.stanford.edu/class/cs224n/

Training Trick #1: Residual Connections

- Residual connections are a simple but powerful technique from computer vision.
- Deep networks are surprisingly bad at learning the identity function!
- Therefore, directly passing "raw" embeddings to the next layer can actually be very helpful!

  $$x_\ell = F(x_{\ell-1}) + x_{\ell-1}$$

- This prevents the network from "forgetting" or distorting important information as it is processed by many layers.



https://web.stanford.edu/class/cs224n/

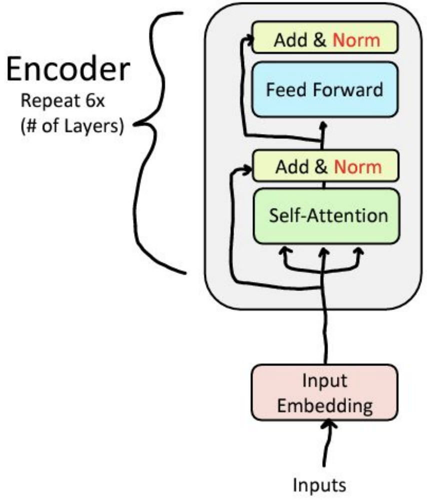
Training Trick #2: Layer Normalization

- **Problem:** Difficult to train the parameters of a given layer because its input from the layer beneath keeps shifting.
- **Solution:** Reduce variation by **normalizing** to zero mean and standard deviation of one within each **layer**.

H-dim (model-dim)

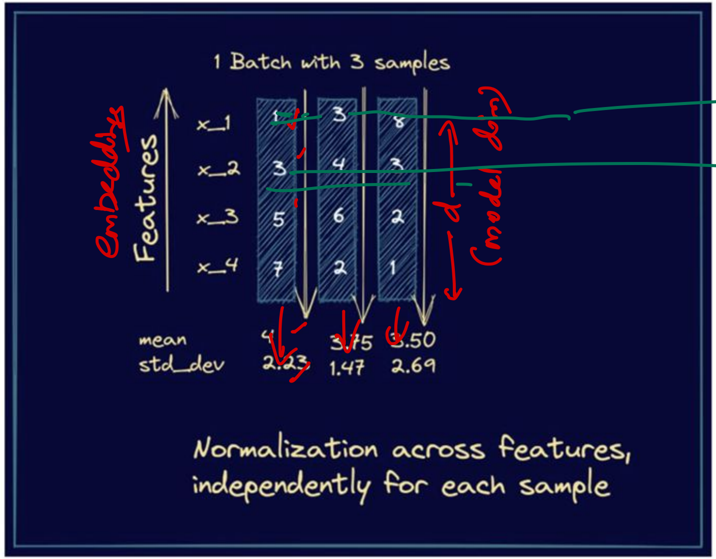
Mean: $\mu^l = \frac{1}{H}\sum_{i=1}^{H} a_i^l$ Standard Deviation: $\sigma^l = \sqrt{\frac{1}{H}\sum_{i=1}^{H}\left(a_i^l - \mu^l\right)^2}$

$$\boxed{x^{\ell\prime} = \frac{x^{\ell} - \mu^{\ell}}{\sigma^{\ell} + \epsilon}}$$



https://web.stanford.edu/class/cs224n/

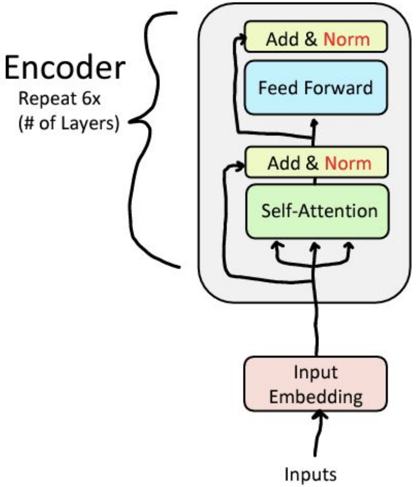
Training Trick #2: Layer Normalization



An Example of How LayerNorm Works (Image by Bala Priya C, Pinecone)

batchnorm

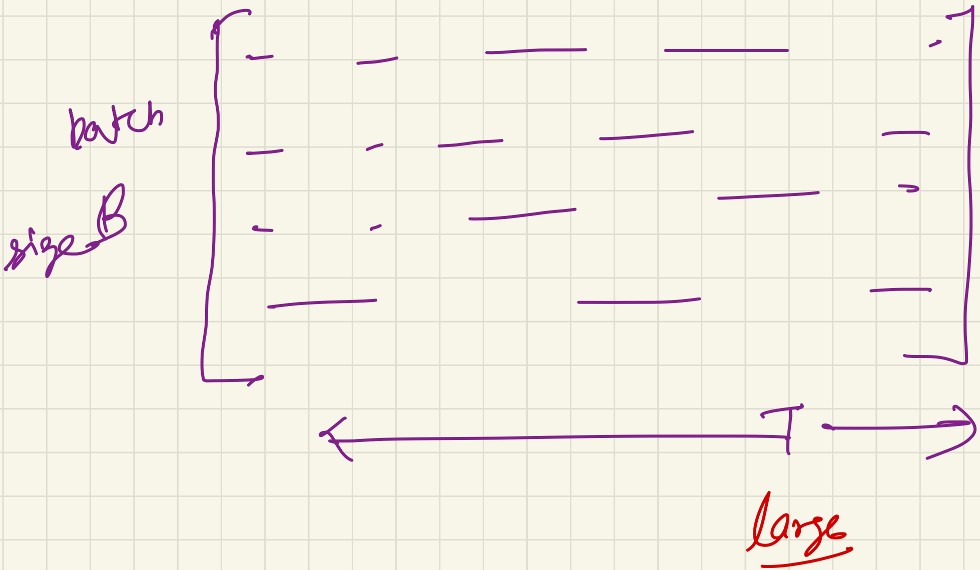
1. much more efficient
2. Can't do batchnorm in decoder (sequences)



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

$$x^{\ell\prime} = \frac{x^\ell - \mu^\ell}{\sigma^\ell + \epsilon}$$

https://web.stanford.edu/class/cs224n/

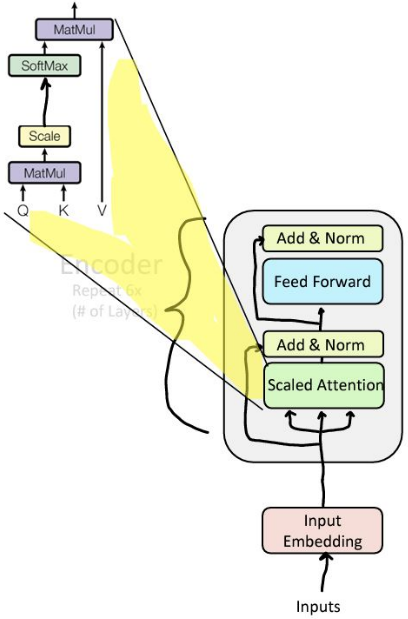


distributed training
↓
communication cost

$B \times T \times d$

Training Trick #3: Scaled Dot Product Attention

- After LayerNorm, the mean and variance of vector elements is 0 and 1, respectively. (Yay!)
- However, the dot product still tends to take on extreme values, as its variance scales with dimensionality d_k



Quick Statistics Review:

- Mean of sum = sum of means = $d_k * 0 = 0$
- Variance of sum = sum of variances = $d_k * 1 = d_k$
- To set the variance to 1, simply divide by $\sqrt{d_k}$!

q, K

https://web.stanford.edu/class/cs224n/

Training Trick #3: Scaled Dot Product Attention

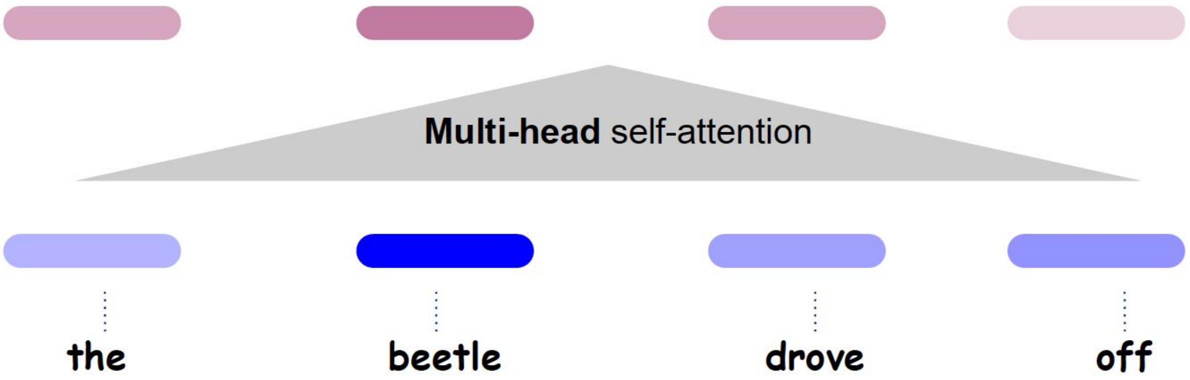
- Assume that the components of q and k are independent random variables with mean 0 and variance 1.
- Then their dot product $q \cdot k = \sum_{i=1}^{d_k} q_i k_i$ has mean 0 and variance d_k.
- Hence the scaling ensures that the resultant dot product has mean 0 and variance 1.

$$\underbrace{\mathrm{var}(q.k)}_{\to\, d_k} = E\left((q.k)^2\right) - \left(\underbrace{E(q.k)}_{0}\right)^2$$

$$1 + 1 + \ldots 1$$

$$E\left[(q_1k_1 + \ldots q_nk_n)^2\right] = E\left[q_1^2k_1^2 + \ldots q_n^2k_n^2 + 2\underbrace{\sum q_iq_jk_ik_j}_{0}\right]$$

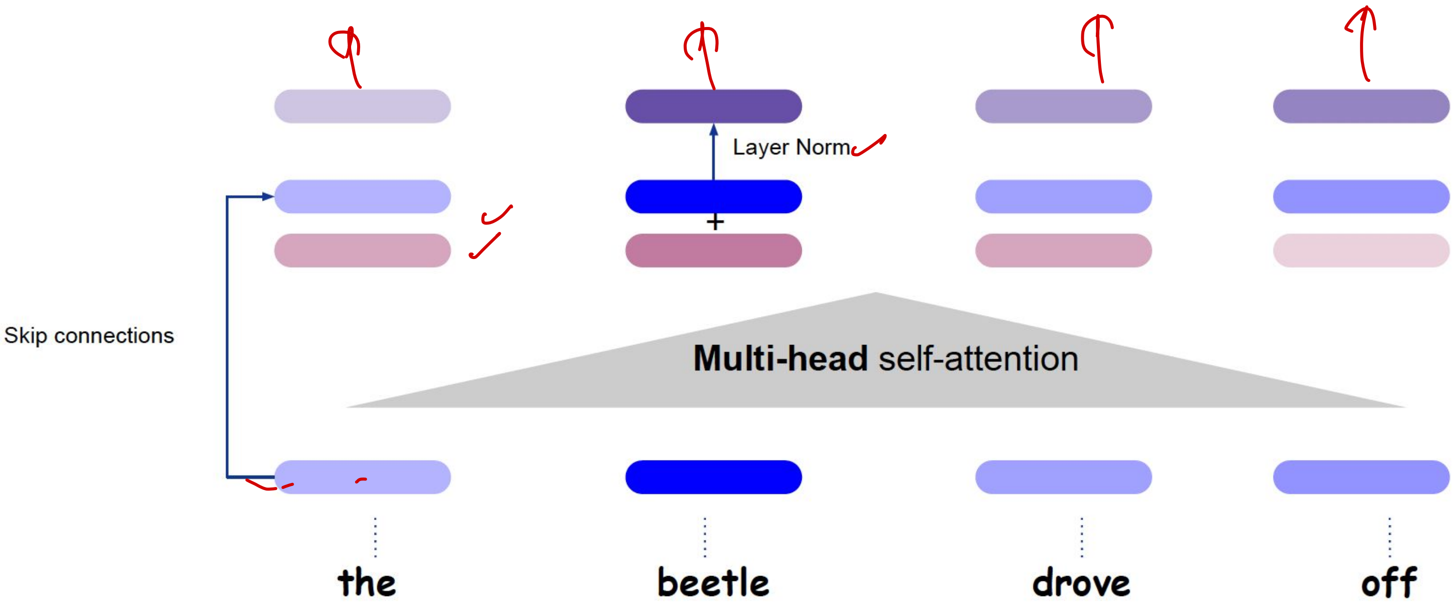
A Complete Transformer Block



embedding

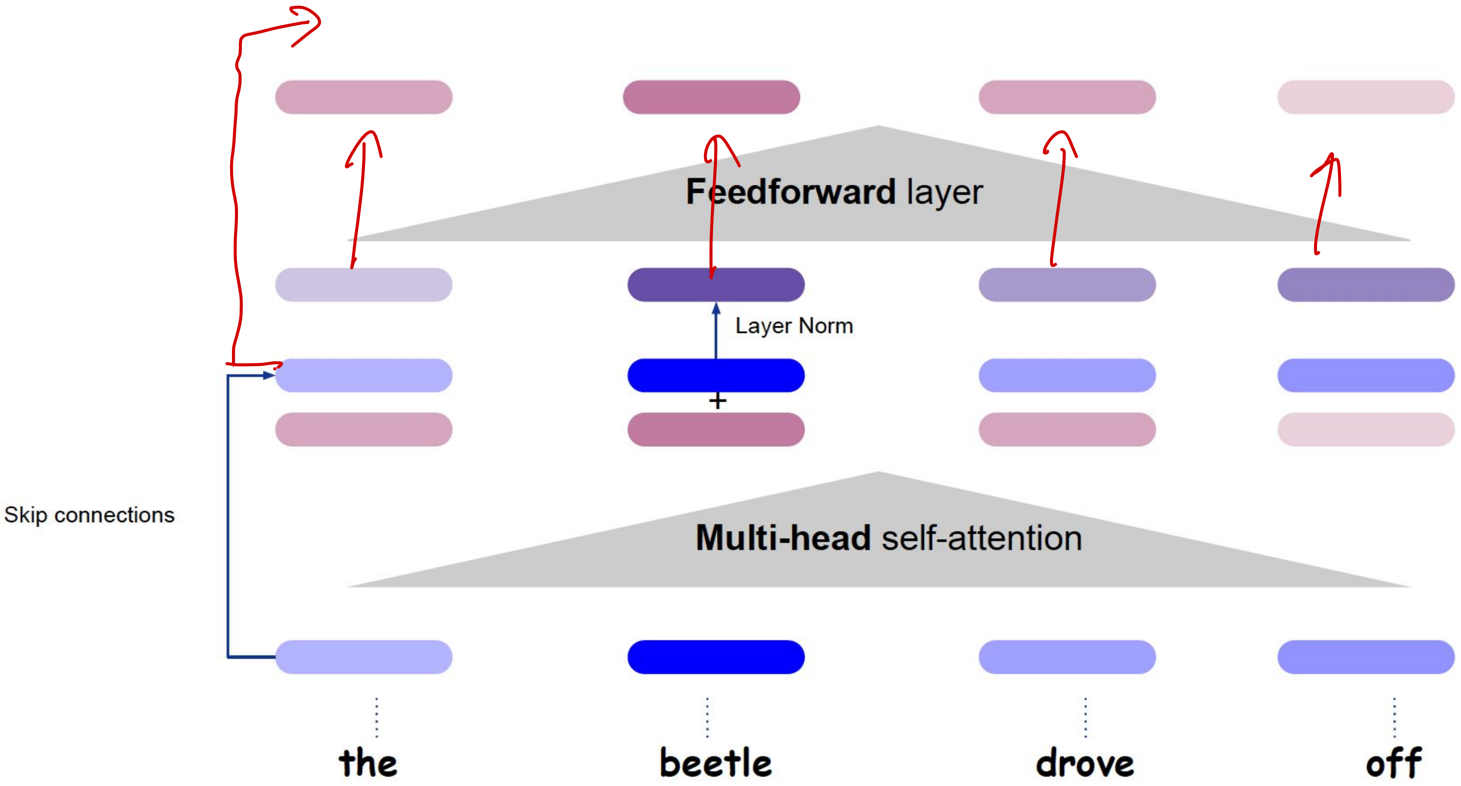
Source: *https://deepmind.com/learning-resources/deep-learning-lecture-series-2020*

A Complete Transformer Block



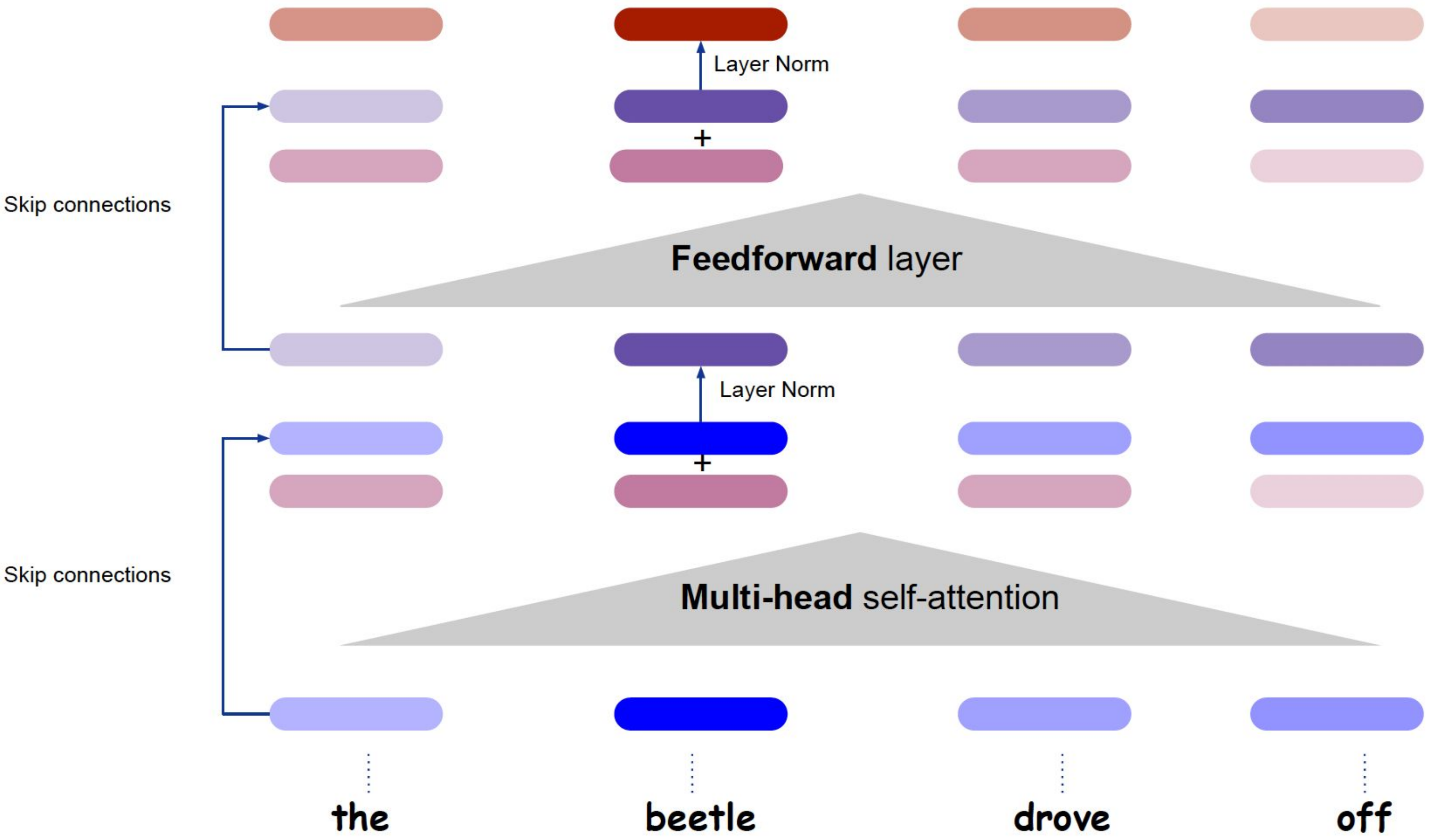
Source: *https://deepmind.com/learning-resources/deep-learning-lecture-series-2020*

A Complete Transformer Block



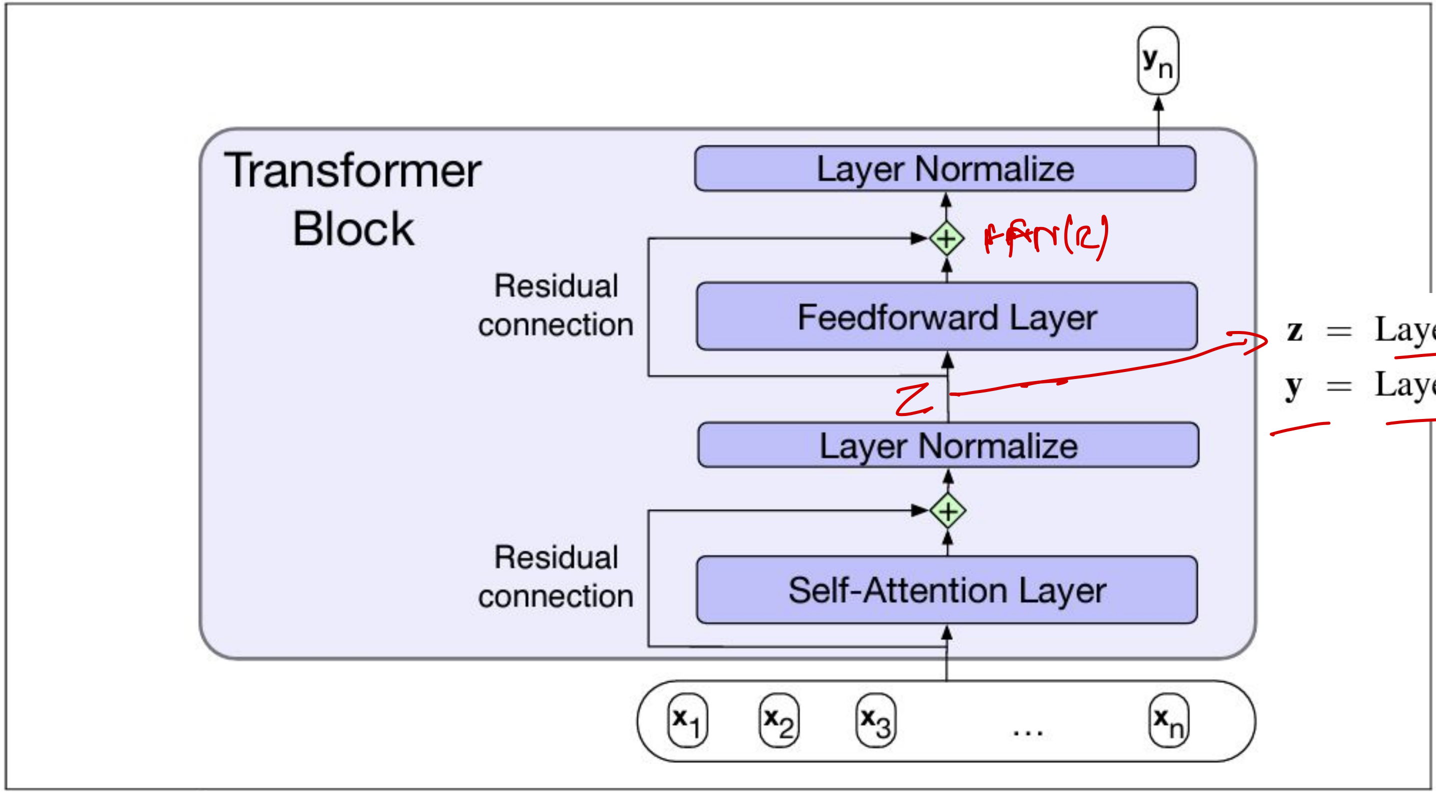
Source: *https://deepmind.com/learning-resources/deep-learning-lecture-series-2020*

A Complete Transformer Block



Source: *https://deepmind.com/learning-resources/deep-learning-lecture-series-2020*

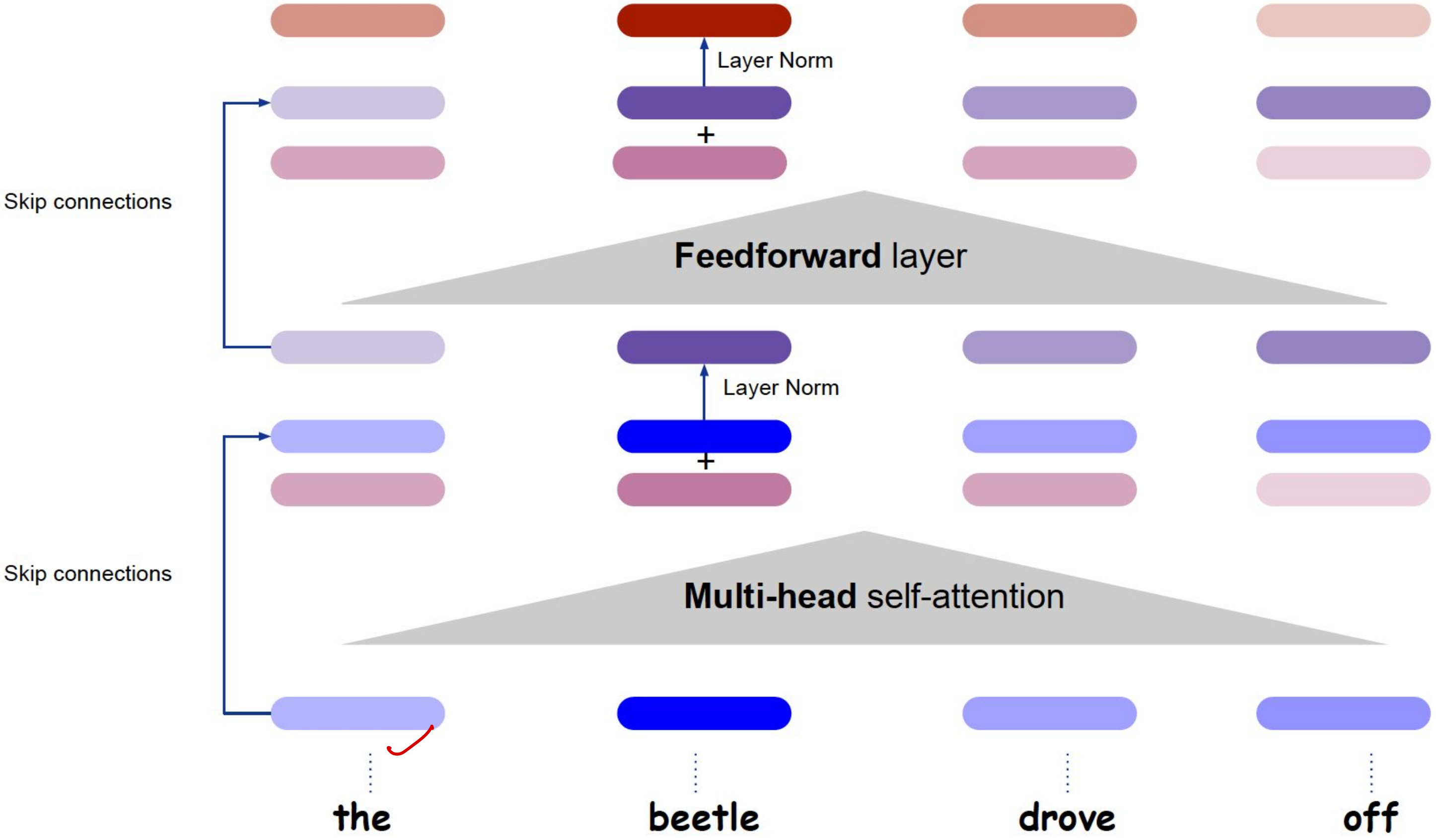
Transformer Block: Another visualization



$$
\begin{aligned}
\mathbf{z} &= \mathrm{LayerNorm}(\mathbf{x} + \mathrm{SelfAttention}(\mathbf{x})) \\
\mathbf{y} &= \mathrm{LayerNorm}(\mathbf{z} + \mathrm{FFN}(\mathbf{z}))
\end{aligned}
$$

https://web.stanford.edu/~jurafsky/slp3/

A Complete Transformer Block



There is still a major problem!

Order does not matter!!

Cat kills dog

dog kills Cat

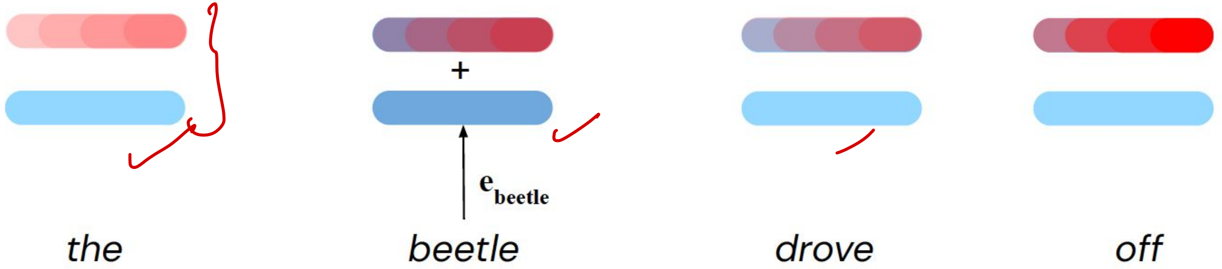
Source: *https://deepmind.com/learning-resources/deep-learning-lecture-series-2020*

Position Encoding of words

- Add fixed quantity to embedding activations
- The quantity added to each input embedding unit ∈ [-1, 1] depends on:
  - The dimension of the unit within the embedding
  - The (absolute) position of the words in the input

$\boxed{\text{max length} \times \text{model dim}}$

i/p to self-attn

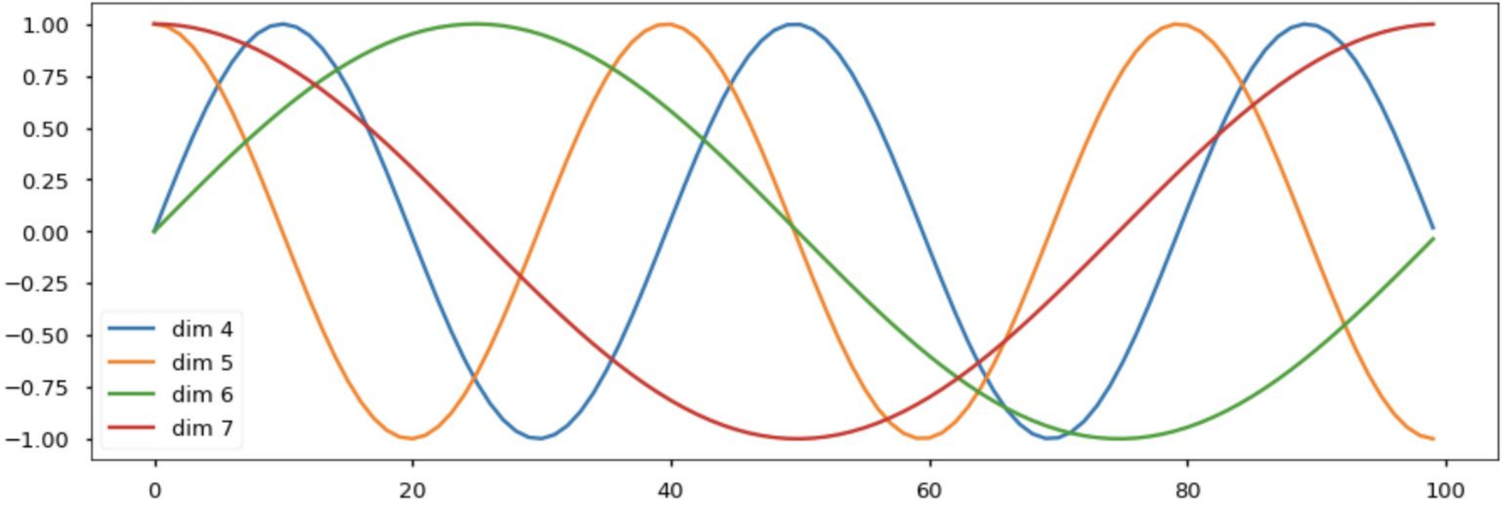


Source: *https://deepmind.com/learning-resources/deep-learning-lecture-series-2020*

Understanding Positional Encoding

Sinusoidal position representations: concatenate sinusoidal functions of varying periods

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



Understanding Positional Encoding

Suppose you have an input sequence of length L (= 512)

Defining the positional encoding of the **k^{th} word** within the sequence. The positional encoding is given by **sine** and **cosine** functions of varying frequencies:

pos:

$i = 1, - - - d/2$

even p.e. $$P(k, 2i) = \sin\left(\frac{k}{n^{2i/d}}\right)$$

10000

odd p.e. $$P(k, 2i+1) = \cos\left(\frac{k}{n^{2i/d}}\right)$$

dim

Here:

k: Position of a word in the input sequence
i: Used for mapping to indices in the positional encoding of a particular word $0 \le i < d/2$
d: Dimension of the output embedding space
P(k,j): Position function for mapping a position k in the input sequence to index (k,j) of the positional matrix
n: User-defined scalar, set to 10,000 by the authors of Attention Is All You Need.

https://rycolab.io/classes/llm-s24/

Understanding Positional Encoding

Positional Encoding Matrix with d=4, n=100

Sequence	Index of token, k	i=0	i=0	i=1	i=1
I	0	P_{00}=sin(0) = 0	P_{01}=cos(0) = 1	P_{02}=sin(0) = 0	P_{03}=cos(0) = 1
am	1	P_{10}=sin(1/1) = 0.84	P_{11}=cos(1/1) = 0.54	P_{12}=sin(1/10) = 0.10	P_{13}=cos(1/10) = 1.0
a	2	P_{20}=sin(2/1) = 0.91	P_{21}=cos(2/1) = -0.42	P_{22}=sin(2/10) = 0.20	P_{23}=cos(2/10) = 0.98
Robot	3	P_{30}=sin(3/1) = 0.14	P_{31}=cos(3/1) = -0.99	P_{32}=sin(3/10) = 0.30	P_{33}=cos(3/10) = 0.96

$$P(k, 2i) = \sin\left(\frac{k}{n^{2i/d}}\right)$$

$$P(k, 2i+1) = \cos\left(\frac{k}{n^{2i/d}}\right)$$

$$\cos\left(\frac{3}{100^{2/4}}\right)$$

$$\cos\left(\frac{3}{100^{1/2}}\right)$$

$$\cos\left(\frac{3}{10}\right)$$

https://rycolab.io/classes/llm-s24/

Other variants: Learned Positional embedding

Goal: learn a position embedding matrix E_{pos} of shape $[1 \times N]$.

Start with randomly initialized embeddings

- one for each integer up to some maximum length.
- i.e., just as we have an embedding for token *fish*, we'll have an embedding for position 3 and position 17.
- As with word embeddings, these position embeddings are learned along with other parameters during training.

https://web.stanford.edu/~jurafsky/slp3/