Transformers and Large Language Models

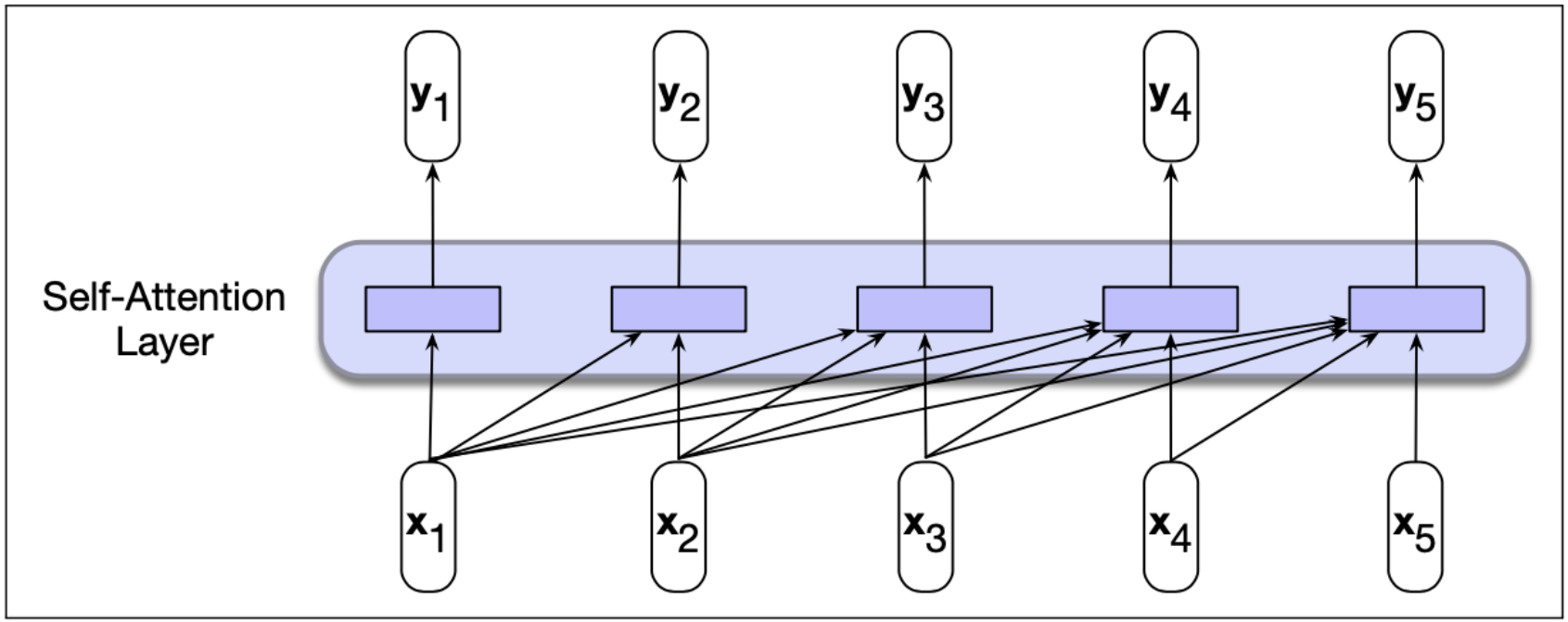
Erhard Hinrichs

Seminar für Sprachwissenschaft
Eberhard-Karls Universität Tübingen

Transformers

- Transformers are non-recurrent networks based on (self-)attention.
- Self-attention allows a network to directly extract and use information from arbitrarily large contexts without the need to pass them through intermediate recurrent connections as in RNNs.
- A self-attention layer maps input sequences to output sequences of the same length, using attention heads.
- Attention heads model how the surrounding words are relevant for the processing of the current word.

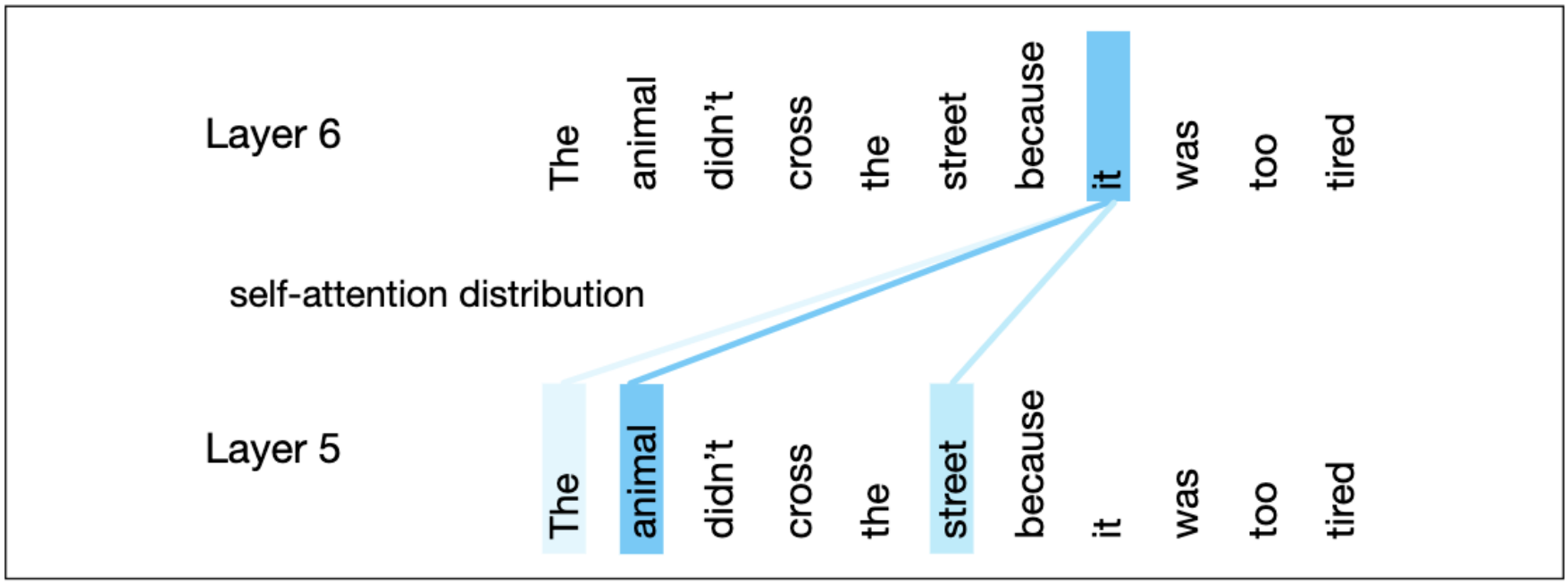
Self-Attention Layer



Relevant Linguistic Examples Motivating Self-Attention

1. The **keys** to the cabinet **are** on the table.
2. The **chicken** crossed the road because **it** wanted to get to the other side.
3. I walked along the **pond**, and noticed that one of the trees long the **bank** had fallen into the **water** after the storm.

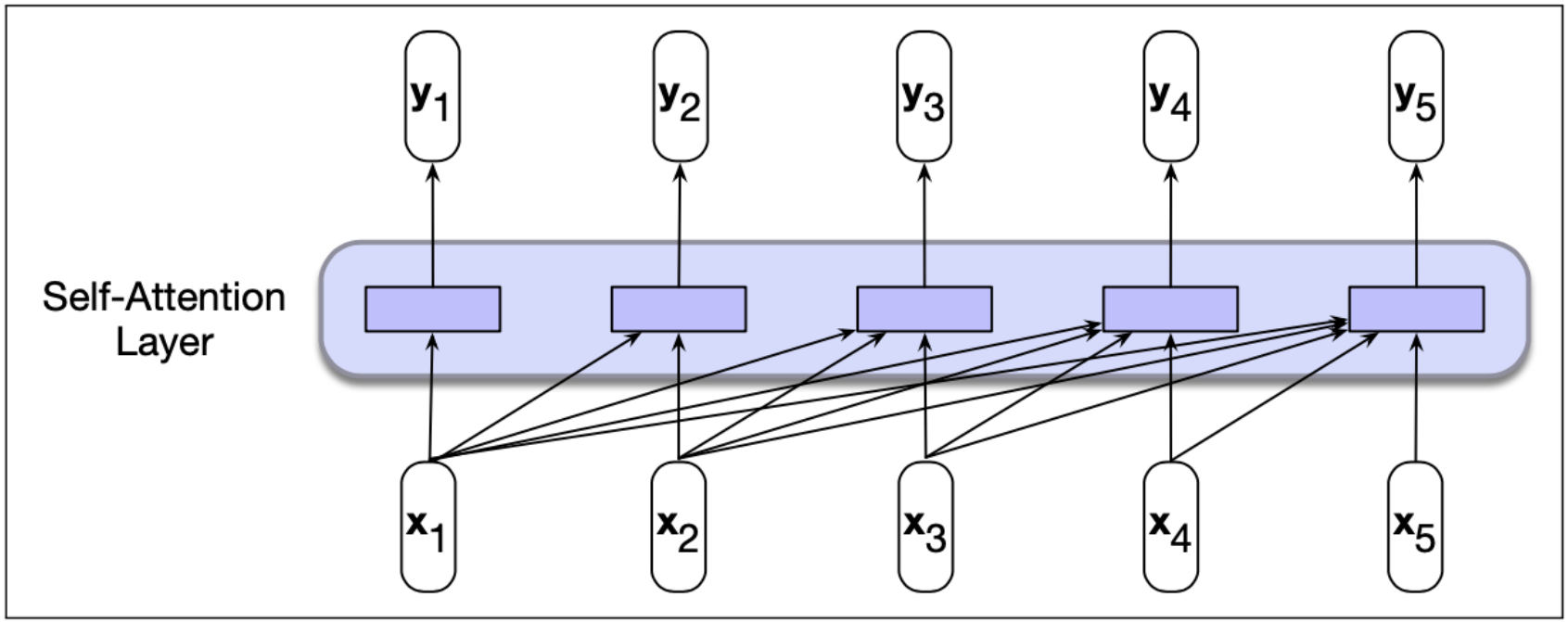
Self-Attention Weight Distribution



Flow of Information in a Self-Attention Layer

- When processing each item in the input of a self-attention layer, the model has access to all inputs up to and including the one under consideration, but no access to information about inputs beyond the current one.
- The computation performed for each item is independent of all the other computations. Hence, forward inference and training can proceed in parallel.

Self-Attention Layer



Self-Attention Networks: Transformers

$$\text{score}(\mathbf{x}_i, \mathbf{x}_j) = \mathbf{x}_i \cdot \mathbf{x}_j \tag{1}$$

$$\alpha_{ij} = \text{softmax}(\text{score}(\mathbf{x}_i, \mathbf{x}_j)) \; \forall j \leq i \tag{2}$$

$$= \frac{\exp(\text{score}(\mathbf{x}_i, \mathbf{x}_j))}{\sum_{k=1}^{i} \exp(\text{score}(\mathbf{x}_i, \mathbf{x}_k))} \; \forall j \leq i \tag{3}$$

$$\mathbf{y}_i = \sum_{j \leq i} \alpha_{ij} \mathbf{x}_j \tag{4}$$

Self-Attention Networks: Different Roles

An input embedding can play three different roles:

- **query**: as the *current focus of attention* that is compared to all of the other preceding inputs.
- **key**: as *a preceding input* that is compared to the current focus of attention.
- **value** that is used to compute the output for the current focus of attention.

Self-Attention Networks: Transformers

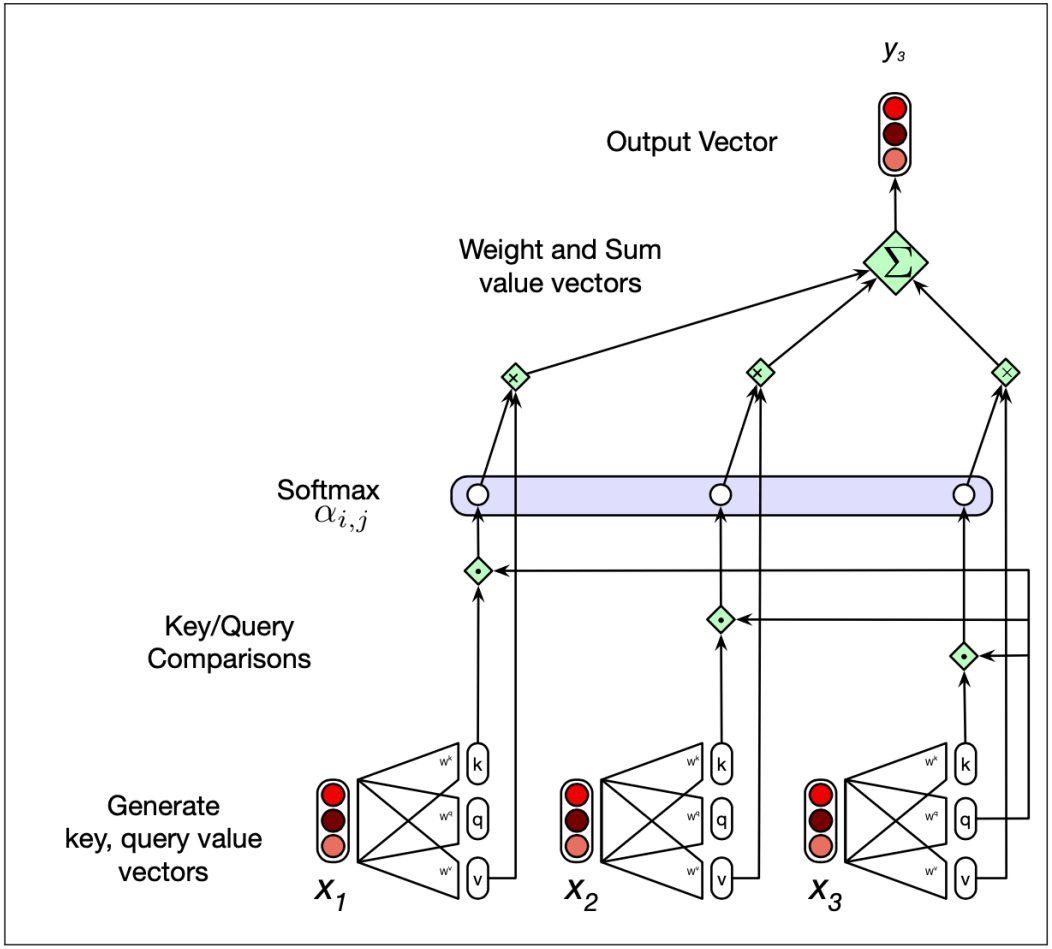
- The roles of query, key, and value are differentiated by three different weight matrices: $\mathbf{W}^{\mathbf{Q}} \in \mathbb{R}^{d \times d'}$, $\mathbf{W}^{\mathbf{K}} \in \mathbb{R}^{d \times d'}$, and $\mathbf{W}^{\mathbf{V}} \in \mathbb{R}^{d \times d''}$.
- The inputs and outputs of transformers, as well as the intermediate vectors after the various layers, all have the same dimensionality $1 \times d$.

$$\mathbf{q}_i = \mathbf{W}^{\mathbf{Q}}\mathbf{x}_i;\ \mathbf{k}_i = \mathbf{W}^{\mathbf{K}}\mathbf{x}_i;\ \mathbf{v}_i = \mathbf{W}^{\mathbf{V}}\mathbf{x}_i \tag{5}$$

$$\text{score}(\mathbf{x}_i, \mathbf{x}_j) = \mathbf{q}_i \cdot \mathbf{k}_j \tag{6}$$

$$\mathbf{y}_i = \sum_{j \leq i} \alpha_{ij}\mathbf{v}_j \tag{7}$$

Calculation of an Output Embedding in a Self-Attention Layer



Self-Attention Networks: Further Adjustments

Scaling the dot product by the square root of the dimensionality

$$\text{score}(\mathbf{x}_i, \mathbf{x}_j) = \frac{\mathbf{q}_i \cdot \mathbf{k}_j}{\sqrt{d_k}} \tag{8}$$

Parallelizing the Cmputation: packing the input embeddings of the N tokens into a single matrix

$$\mathbf{Q} = \mathbf{X}\mathbf{W}^{\mathbf{Q}};\ \mathbf{K} = \mathbf{X}\mathbf{W}^{\mathbf{K}};\ \mathbf{V} = \mathbf{X}\mathbf{W}^{\mathbf{V}}; \tag{9}$$

Final Result: Reducing the Self-Attention Step for an Entire Sequence of N Tokens

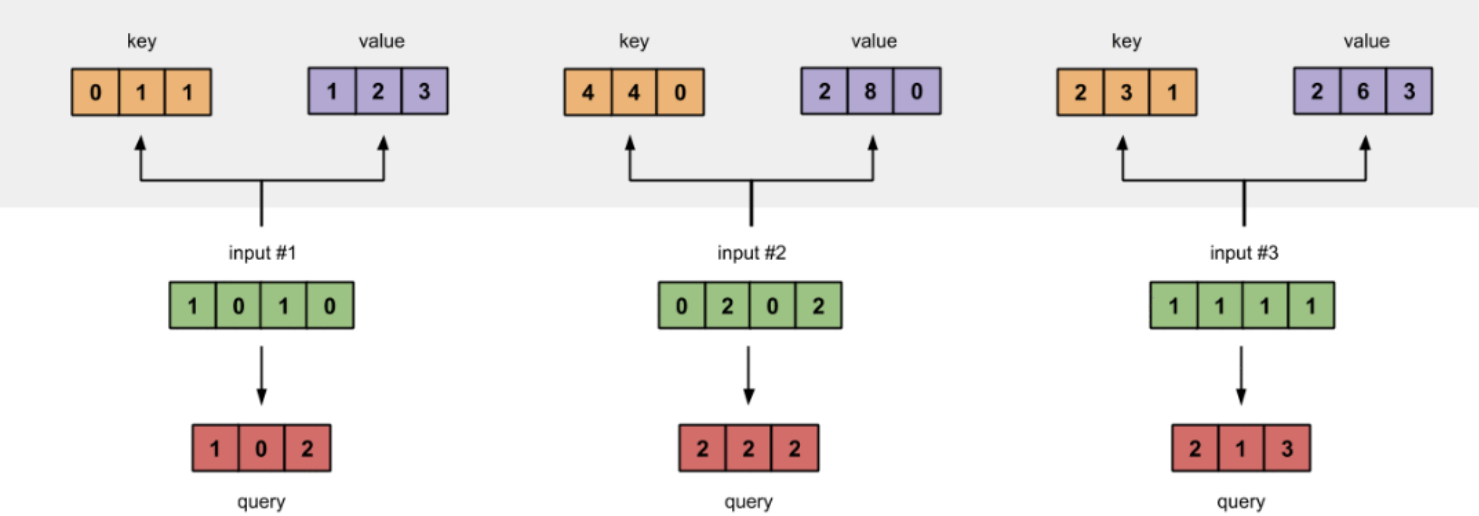
$$\textit{SelfAttention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \mathrm{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_k}}\right)\mathbf{V} \tag{10}$$

Working through an Example with 3 Input Vectors

1. Prepare inputs
2. Initialise weights
3. Derive key, query and value
4. Calculate attention scores for Input 1
5. Calculate softmax
6. Multiply scores with values
7. Sum weighted values to get Output 1
8. Repeat steps 4–7 for Input 2 and Input 3

This workflow and the illustrations in the next three slides are due to `https://towardsdatascience.com/illustrated-self-attention-2d627e33b20a`

Step 1-3


key
value
0 1 1
1 2 3
input #1
1 0 1 0
1 0 2
query
key
value
4 4 0
2 8 0
input #2
0 2 0 2
2 2 2
query
key
value
2 3 1
2 6 3
input #3
1 1 1 1
2 1 3
query

Initialize Weights for Query, Key, and Value

Weights for **key**:

```
[[0, 0, 1],
 [1, 1, 0],
 [0, 1, 0],
 [1, 1, 0]]
```

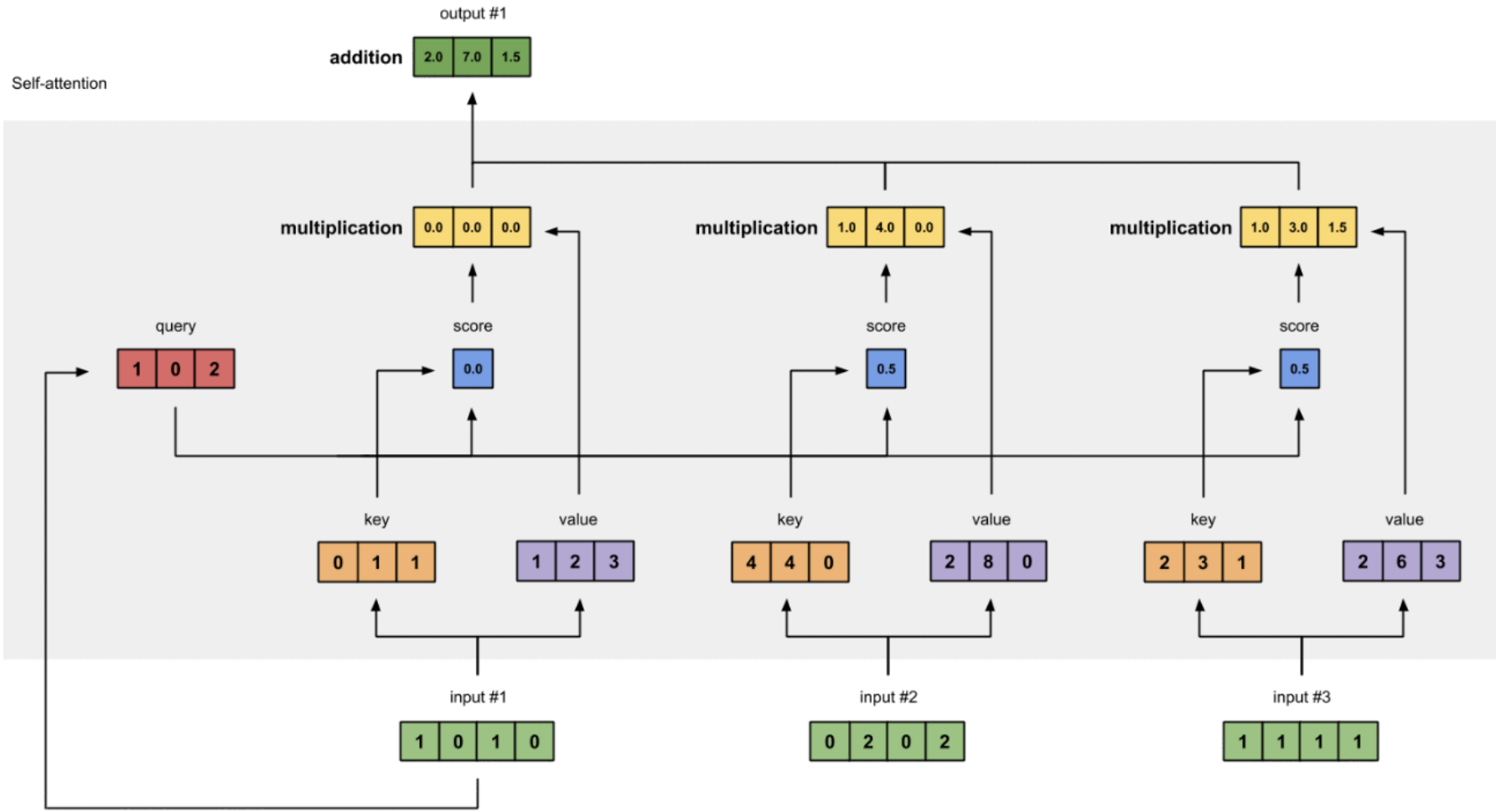
Weights for **query**:

```
[[1, 0, 1],
 [1, 0, 0],
 [0, 0, 1],
 [0, 1, 1]]
```

Weights for **value**:

```
[[0, 2, 0],
 [0, 3, 0],
 [1, 0, 3],
 [1, 1, 0]]
```

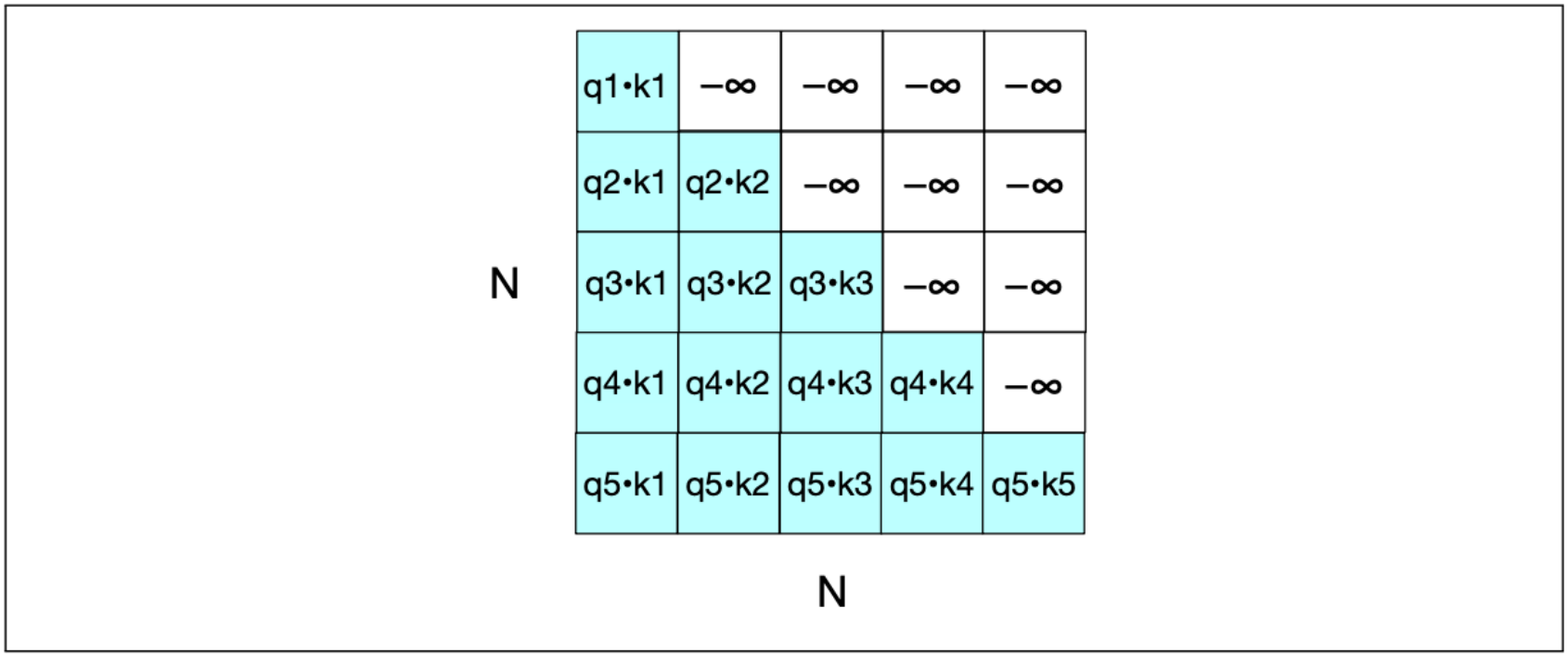
Calculate Attention Scores



Calculate Softmax

$$softmax(z_i) = \frac{exp(z_i)}{\sum_{j=1}^{k} exp(z_j)} \quad 1 \leq i \leq k \tag{11}$$

Zeroing Out the Upper Triangular Portion of a QT^T Matrix



Multihead Self-Attention

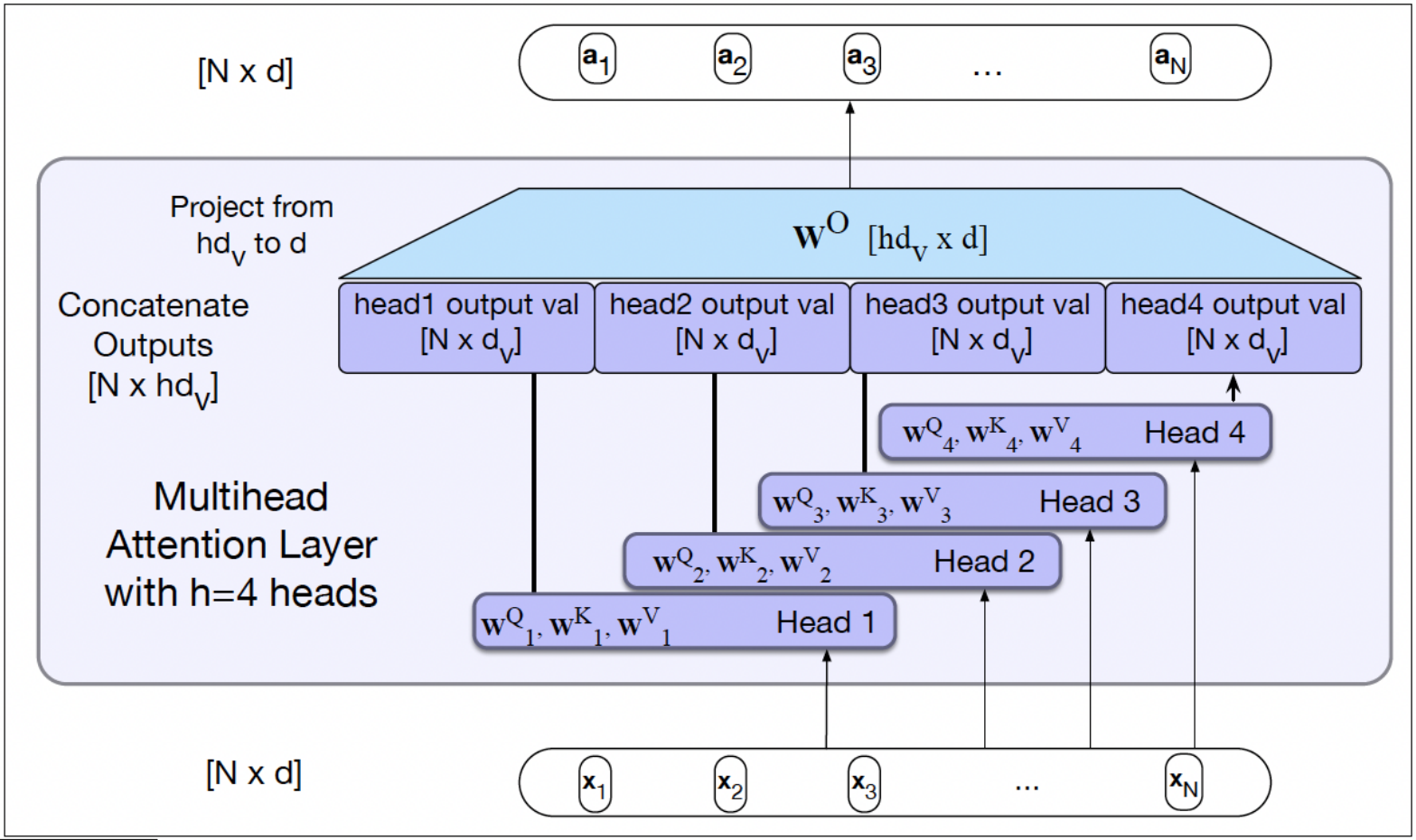


Figure 10.5 Multihead self-attention: Each of the multihead self-attention layers is provided with its own set of key, query and value weight matrices. The outputs from each of the layers are concatenated and then projected to d, thus producing an output of the same size as the input so the attention can be followed by layer norm and feedforward and layers can be stacked.

Multihead Attention Layers

- are sets of self-attention layers, each with its own set of key, query, and value matrices:
  - $\mathbf{W}_i^Q \in \mathbb{R}^{d \times d_k}$
  - $\mathbf{W}_i^K \in \mathbb{R}^{d \times d_k}$
  - $\mathbf{W}_i^V \in \mathbb{R}^{d \times d_v}$
- Each member of such a set of self-attention layers is called a **head**
- Each head gets multiplied by the inputs packed into $\mathbf{X}$ to produce
  - $\mathbf{W}_i^Q \in \mathbb{R}^{N \times d_k}$
  - $\mathbf{W}_i^K \in \mathbb{R}^{N \times d_k}$
  - $\mathbf{W}_i^V \in \mathbb{R}^{N \times d_v}$

Multihead Attention Layers

- The output of a multi-head layer with h heads consists of h vectors of shape $N \times d_v$
- These outputs are concatenated from each head and then, using a linear projection with weight matrix $\mathbf{W}_i^O \in \mathbb{R}^{hd_k \times d}$, reduced to $N \times d$ output.

Multihead Attention

$$\text{MultiHeadAttn}(\mathbf{X}) = (\mathbf{head}_1 \oplus \mathbf{head}_2 \cdots \oplus \mathbf{head}_h)\mathbf{W}^O \tag{12}$$

$$\mathbf{Q} = \mathbf{X}\mathbf{W}_i^Q;\ \mathbf{K} = \mathbf{X}\mathbf{W}_i^K;\ \mathbf{V} = \mathbf{X}\mathbf{W}_i^V \tag{13}$$

$$\mathbf{head}_i = \text{SelfAttention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) \tag{14}$$

Multihead Attention

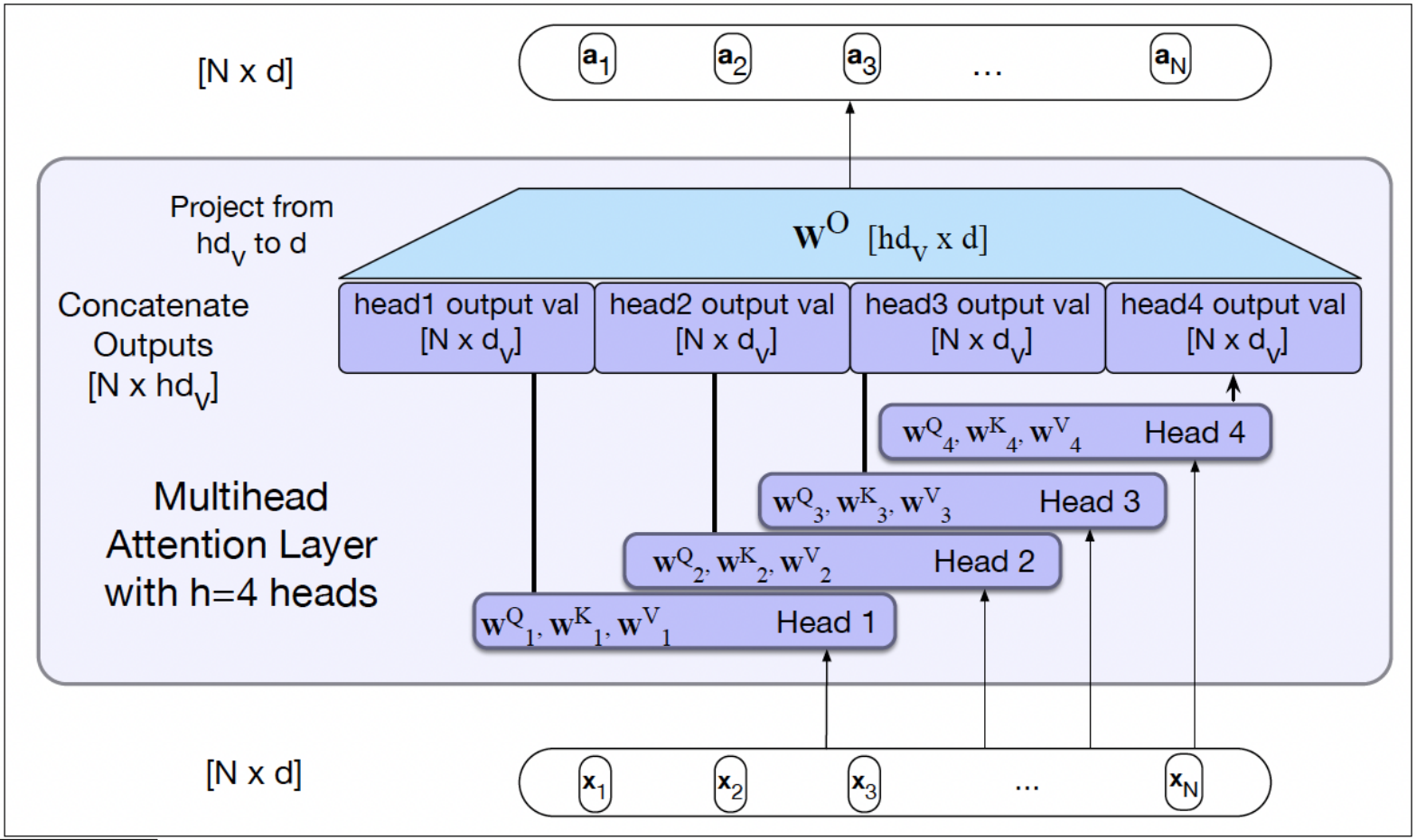
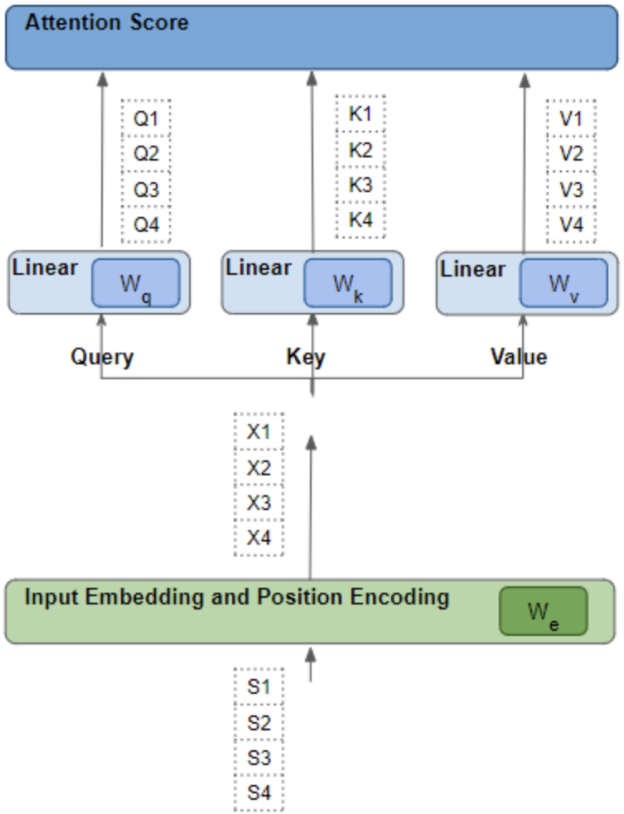


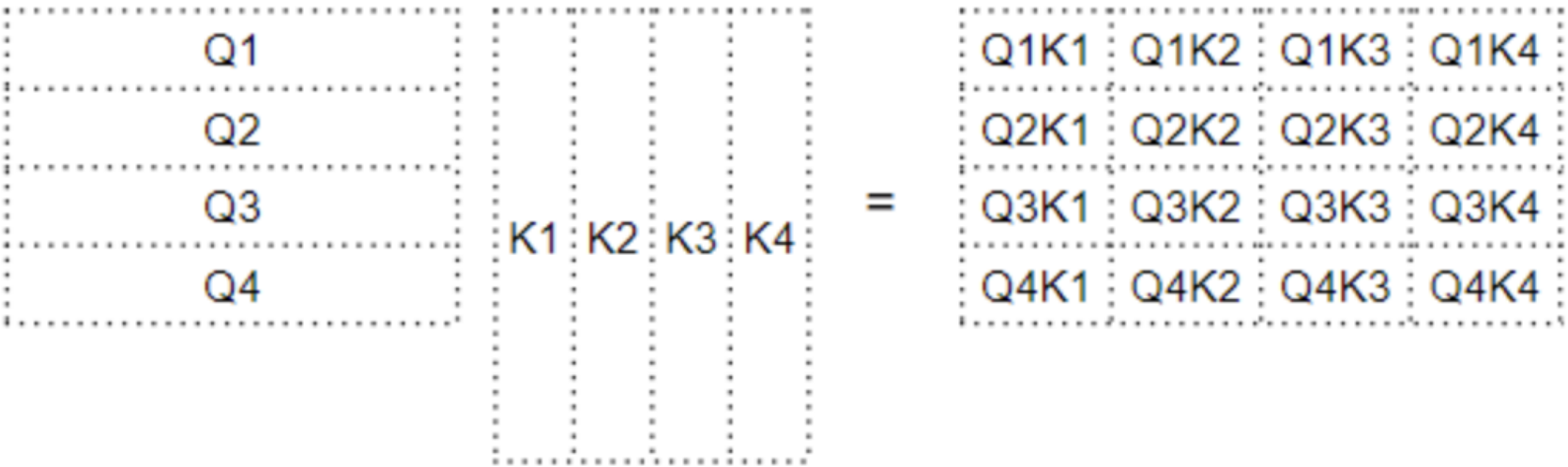
Figure 10.5 Multihead self-attention: Each of the multihead self-attention layers is provided with its own set of key, query and value weight matrices. The outputs from each of the layers are concatenated and then projected to d, thus producing an output of the same size as the input so the attention can be followed by layer norm and feedforward and layers can be stacked.

Multihead Attention: Visual Summary



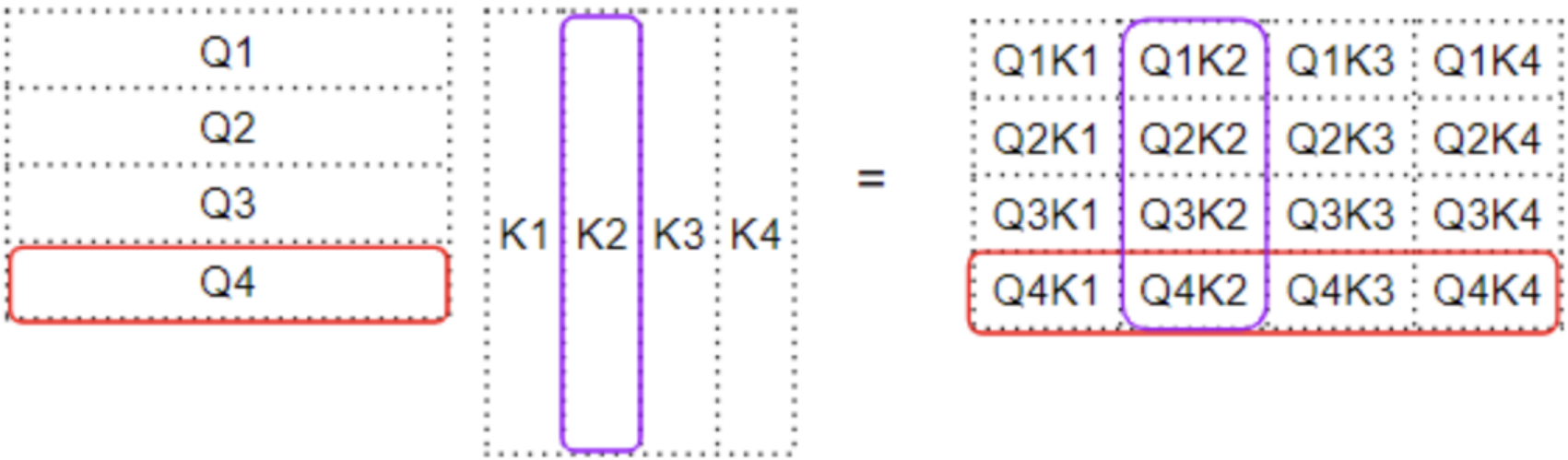
Slide due to Keitan Doshi, https://towardsdatascience.com/transformers-explained-visually-not-just-how-but-why-they-work-so-well-d840bd61a9d3

Multihead Attention: Dot Product between Query and Key matrices



Slide due to Keitan Doshi
https://towardsdatascience.com/transformers-explained-visually-not-just-how-but-why-they-work-so-well-d840bd61a9d3

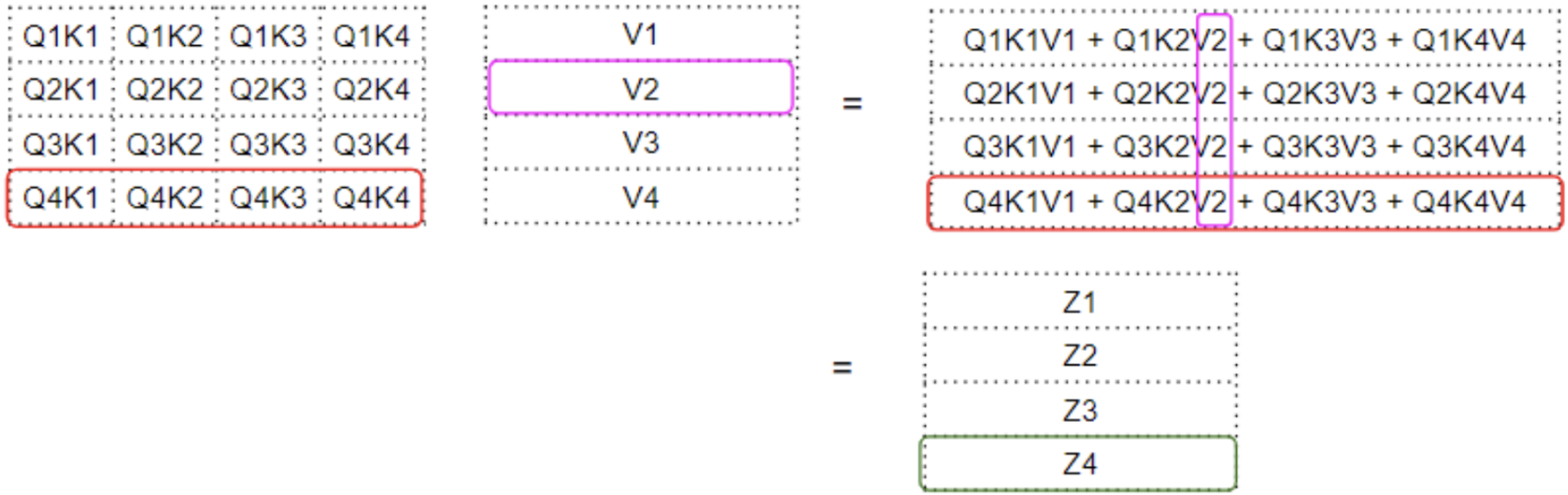
Multihead Attention: Dot Product between Query and Key matrices



Slide due to Keitan Doshi

`https://towardsdatascience.com/transformers-explained-visually-not-just-how-but-why-they-work-so-well-d840bd61a9d3`

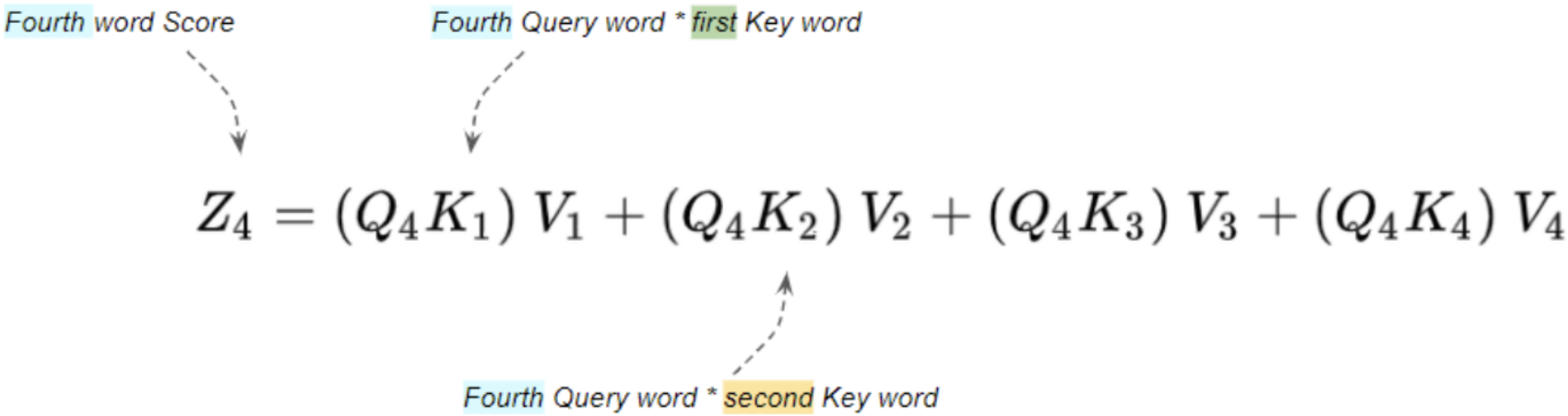
Dot Product between Query-Key and Value Matrices



Slide due to Keitan Doshi

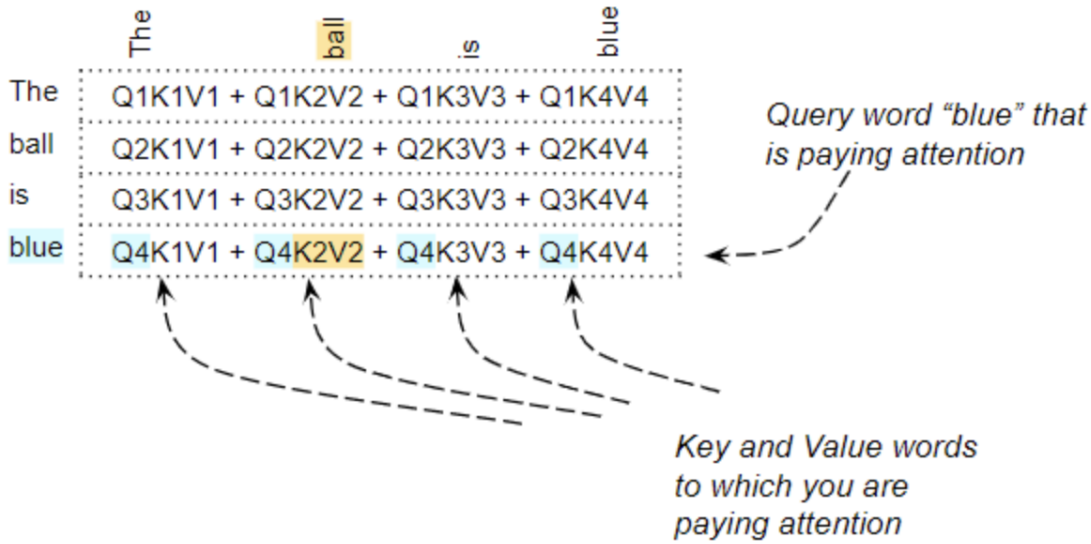
`https://towardsdatascience.com/transformers-explained-visually-not-just-how-but-why-they-work-so-well-d840bd61a9d3`

Attention Score for the word blue pays attention to every other word



Slide due to Keitan Doshi
`https://towardsdatascience.com/transformers-explained-visually-not-just-how-but-why-they-work-so-well-d840bd61a9d3`

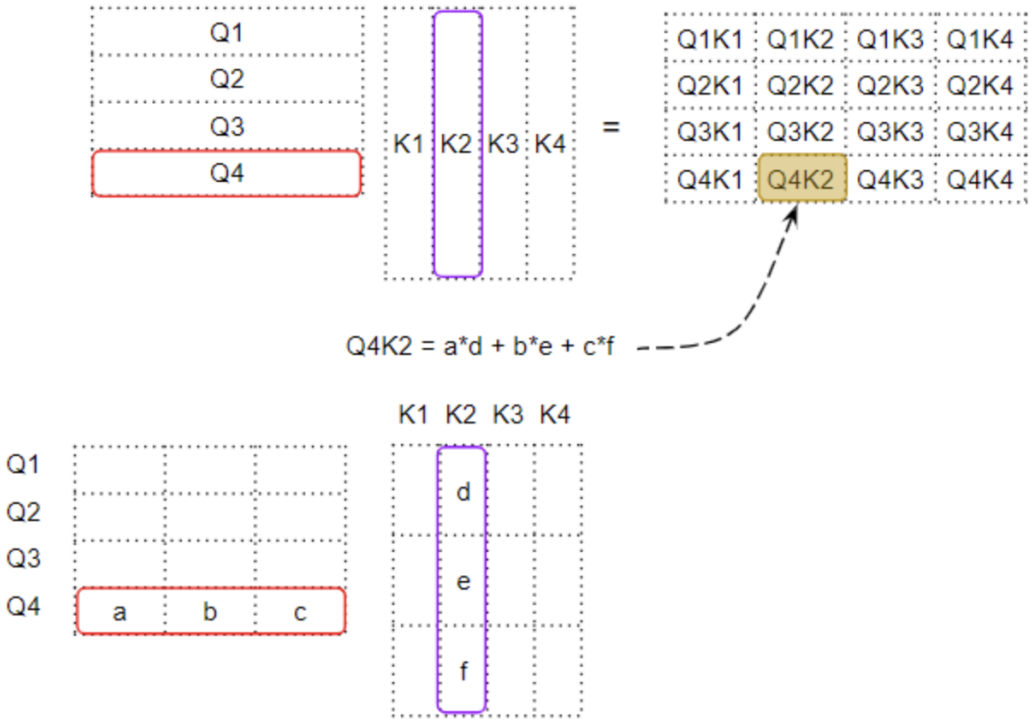
Dot Product between Query-Key and Value Matrices



Slide due to Keitan Doshi

`https://towardsdatascience.com/transformers-explained-visually-not-just-how-but-why-they-work-so-well-d840bd61a9d3`

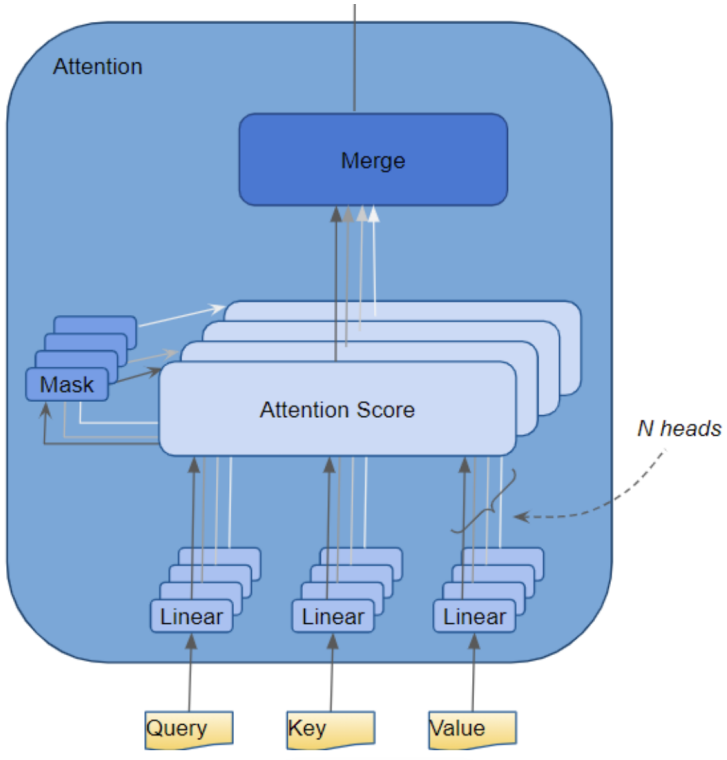
Each cell is a dot product between two word vectors



Slide due to Keitan Doshi

`https://towardsdatascience.com/transformers-explained-visually-not-just-how-but-`

Multihead Attention: Visual Summary



Slide due to Keitan Doshi

`https://towardsdatascience.com/transformers-explained-visually-not-just-how-but-why-they-work-so-well-d840bd61a9d3`

Transformer Blocks

- A transformer block consists of a single attention layer followed by a feed-forward layer with residual connections and layer normalizations following each.
- Transformer blocks can be stacked to make deeper and more powerful networks.

Transformer Blocks

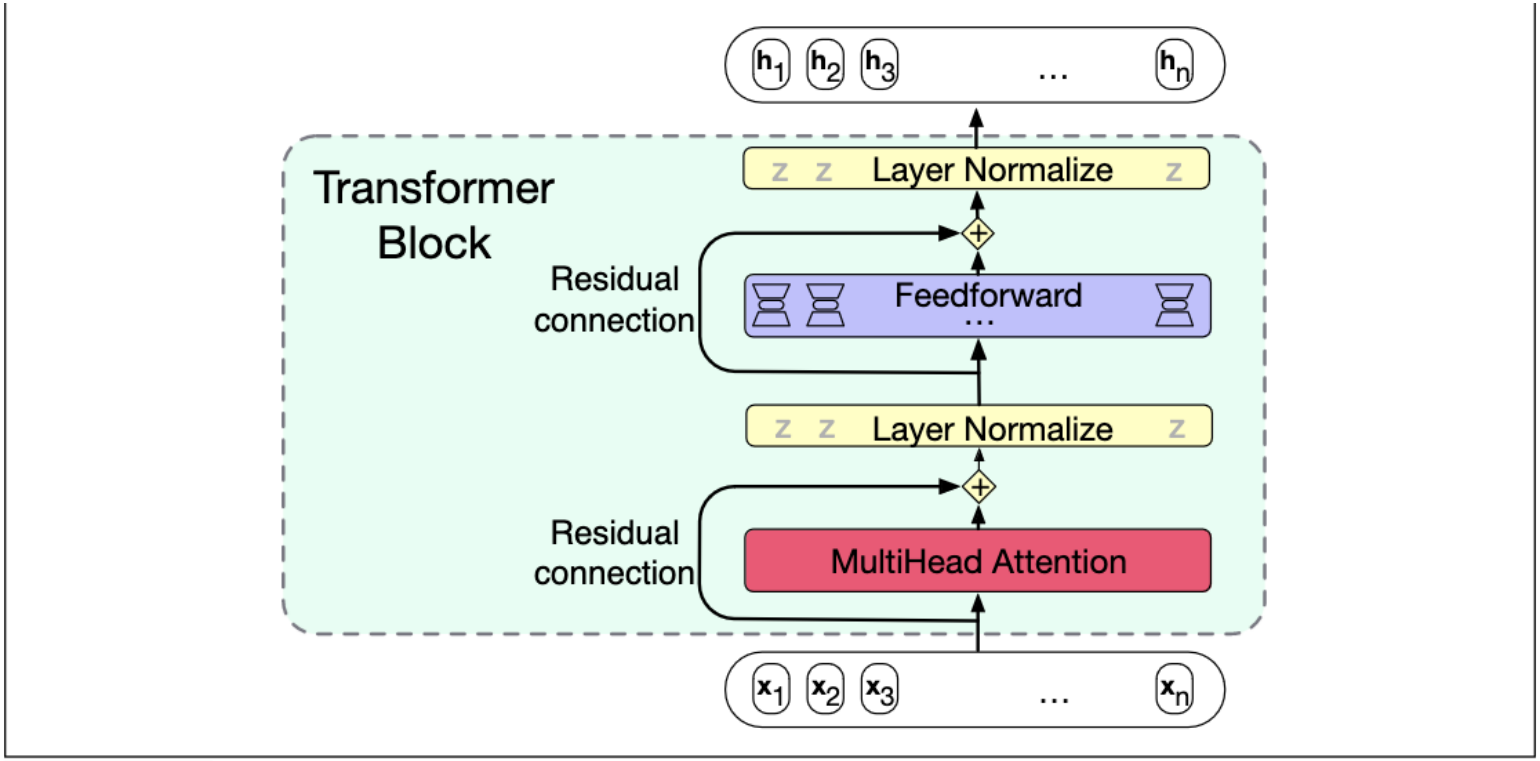


Figure 10.6 A transformer block showing all the layers.

Transformer Blocks: Layer Norm

$$\mathbf{z} = \text{LayerNorm}(\mathbf{x} + \text{SelfAttn}(\mathbf{x})) \tag{15}$$

$$\mathbf{y} = \text{LayerNorm}(\mathbf{z} + \text{FFNN}(\mathbf{z})) \tag{16}$$

$$\mu = \frac{1}{d_h}\sum_{i=1}^{d_h} x_i \tag{17}$$

$$\sigma = \sqrt{\frac{1}{d_h}\sum_{i=1}^{d_h}(x_i - \mu)^2} \tag{18}$$

$$\hat{\mathbf{x}} = \frac{(\mathbf{x} - \mu)}{\sigma} \tag{19}$$

$$\textit{LayerNorm} = \gamma\, \hat{\mathbf{x}} + \beta \tag{20}$$

Putting it all together

$$\mathbf{O} = \text{Layer Norm}(\mathbf{X} + \text{Self-Attention}(\mathbf{X})) \tag{21a}$$

$$\mathbf{H} = \text{Layer Norm}(\mathbf{O} + \text{FFN}(\mathbf{O})) \tag{21b}$$

Or alternatively:

$$\mathbf{T}^1 = \text{Self-Attention}(\mathbf{X}) \tag{22a}$$

$$\mathbf{T}^2 = \mathbf{X} + \mathbf{T}^1 \tag{22b}$$

$$\mathbf{T}^3 = \text{LayerNorm}(\mathbf{T}^2) \tag{22c}$$

$$\mathbf{T}^4 = \text{FFN}(\mathbf{T}^3) \tag{22d}$$

$$\mathbf{T}^5 = \mathbf{T}^4 + \mathbf{T}^3 \tag{22e}$$

$$\mathbf{H} = \text{LayerNorm}(\mathbf{T}^5) \tag{22f}$$

The Residual Stream View

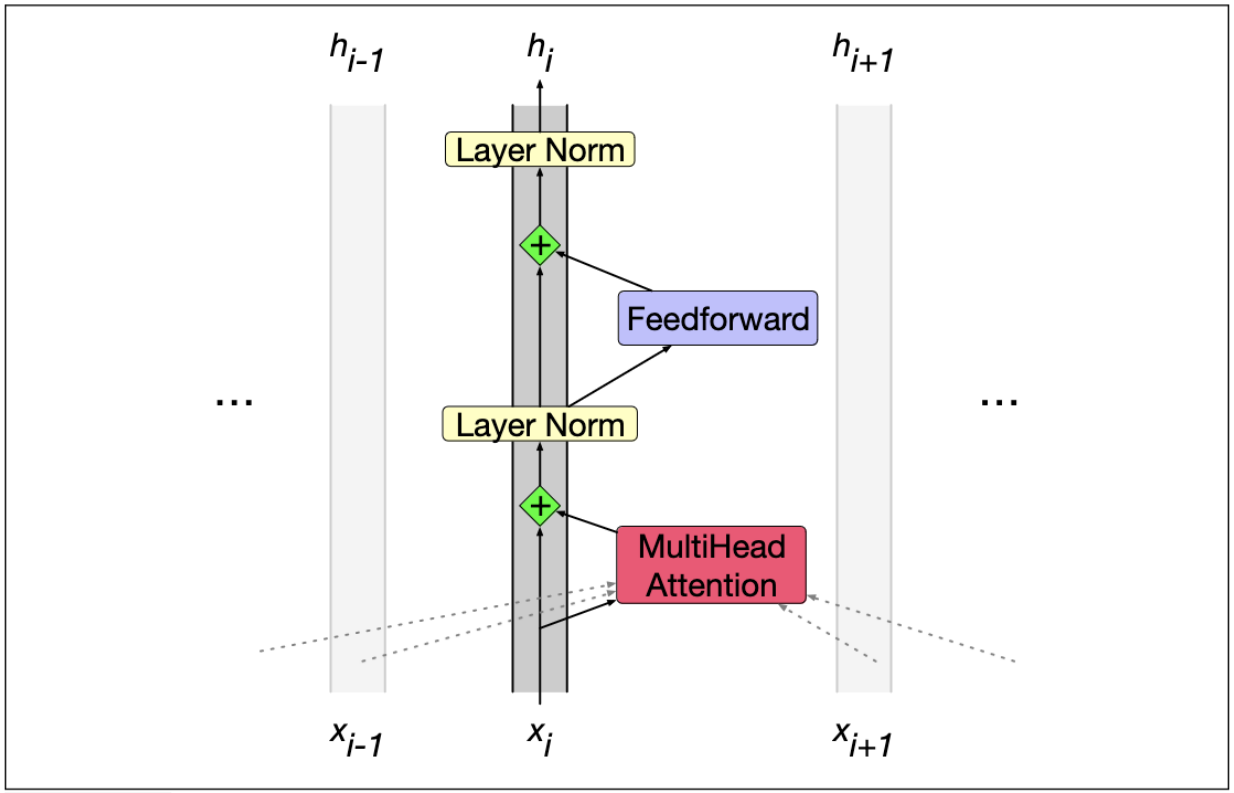


Figure 10.7 The residual stream for token x_i, showing how the input to the transformer block x_i is passed up through residual connections, the output of the feedforward and multi-head attention layers are added in, and processed by layer norm, to produce the output of this block, h_i, which is used as the input to the next layer transformer block. Note that of all the components, only the MultiHeadAttention component reads information from the other residual streams in the context.

Equations for Transfomer Block from Embedding Stream Perspective

$$\mathbf{t_i^1} = \text{MultiHeadAttention}(\mathbf{x_i}, [\mathbf{x_1}, \ldots, \mathbf{x_N}]) \tag{23a}$$

$$\mathbf{t_i^2} = \mathbf{t_i^1} + \mathbf{x_i} \tag{23b}$$

$$\mathbf{t_i^3} = \text{LayerNorm}(\mathbf{t_i^2}) \tag{23c}$$

$$\mathbf{t_i^4} = \text{FFN}(\mathbf{t_i^3}) \tag{23d}$$

$$\mathbf{t_i^5} = \mathbf{t_i^4} + \mathbf{t_i^3} \tag{23e}$$

$$\mathbf{h_i} = \text{LayerNorm}(\mathbf{t_i^5}) \tag{23f}$$

Moving Information between Residual Streams

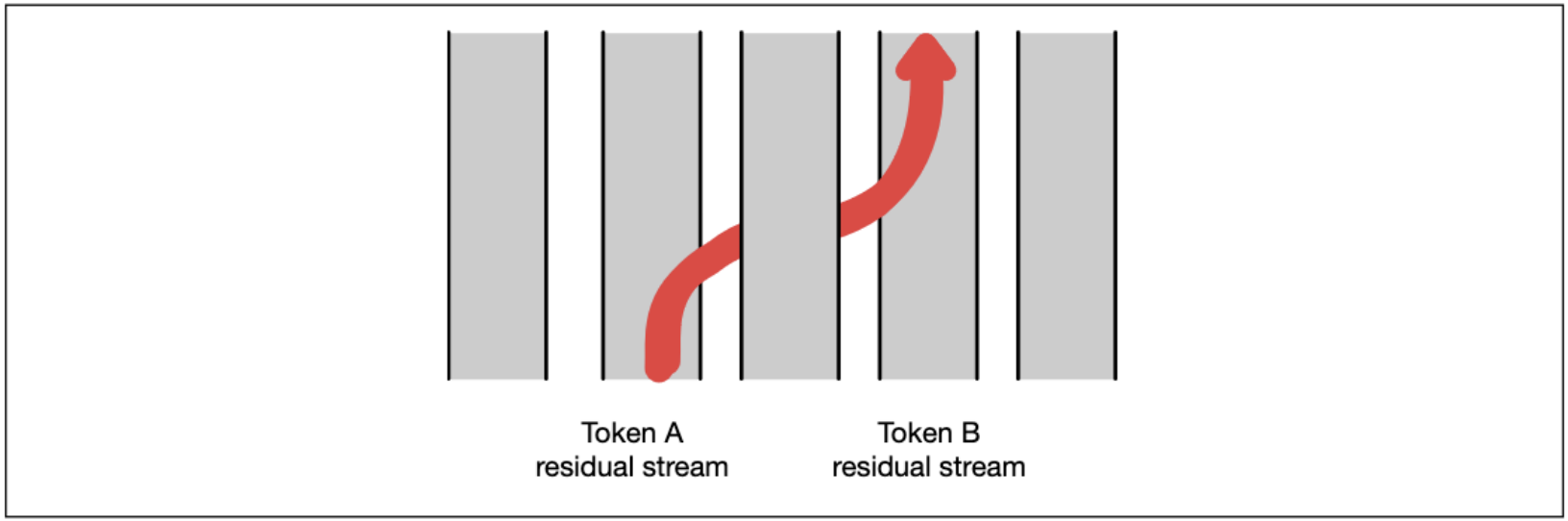


Figure 10.8 An attention head can move information from token A's residual stream into token B's residual stream.

Prenorm View

$$\mathbf{t}_i^1 = \text{LayerNorm}(\mathbf{x}_i) \tag{24a}$$

$$\mathbf{t}_i^2 = \text{MultiHeadAttention}(\mathbf{t}_i^1, [\mathbf{t}_1^1, \ldots, \mathbf{x}_N^1]) \tag{24b}$$

$$\mathbf{t}_i^3 = \mathbf{t}^2 + \mathbf{x}_i \tag{24c}$$

$$\mathbf{t}_i^4 = \text{LayerNorm}(\mathbf{t}_i^3) \tag{24d}$$

$$\mathbf{t}_i^5 = \text{FFN}(\mathbf{t}_i^4) \tag{24e}$$

$$\mathbf{h}_i = \mathbf{t}_i^5 + \mathbf{t}_i^3 \tag{24f}$$

Architecture of Prenorm Transformer Block

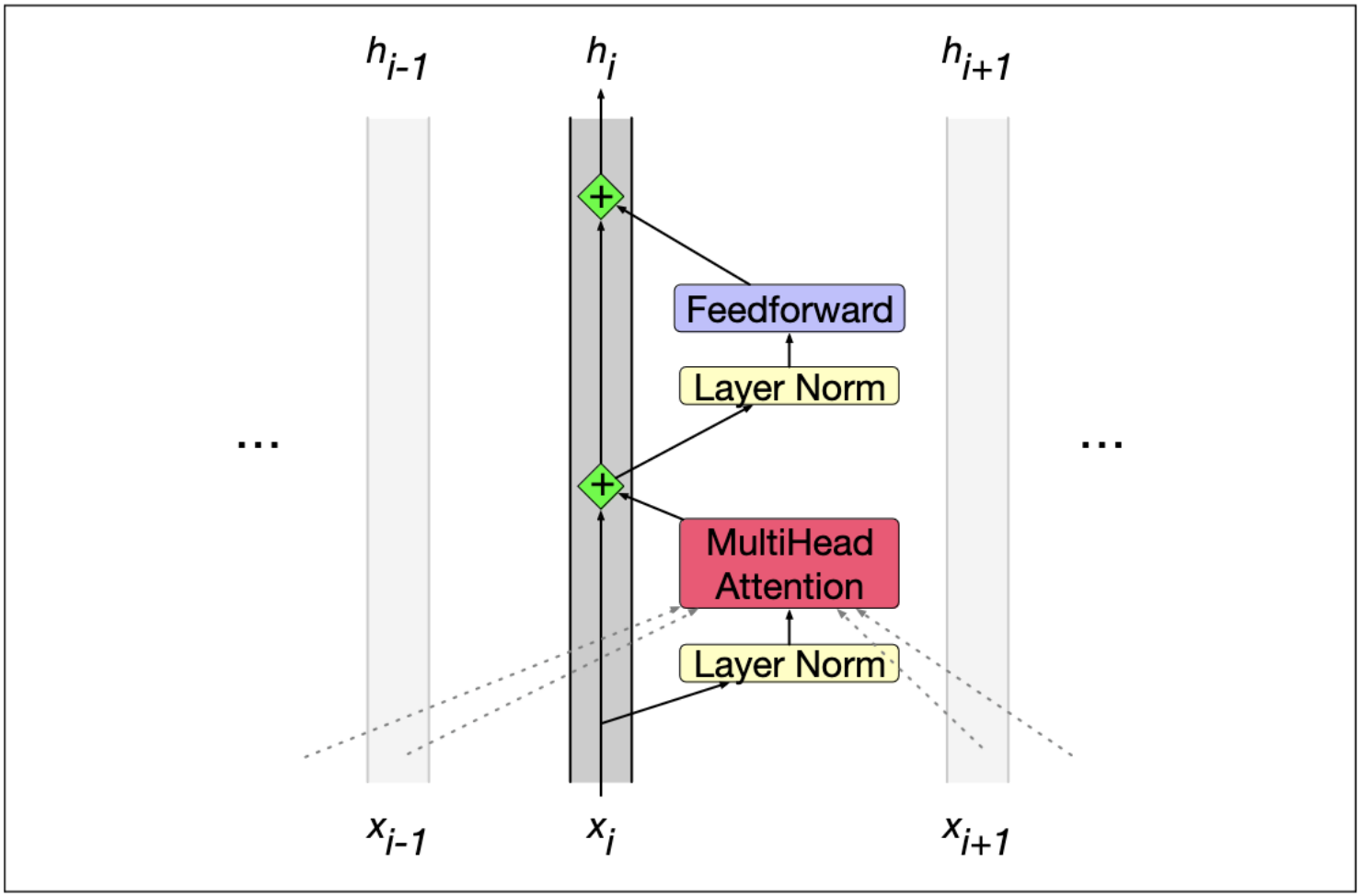


Figure 10.9 The architecture of the prenorm transformer block. Here the nature of the residual stream, passing up information from the input, is even clearer.

Modeling word order: positional embedding

- Transformers models do not come with built-in information about sequence order such as time-step information in an RNN
- Transformers models do not have any notion of relative or absolute positions of the tokens in a sequence.
- In order to model word order, positional embeddings can be combined with input embeddings.
- Positional embeddings that are specific to each position in an input sequence.

Preparing the Transformer Input

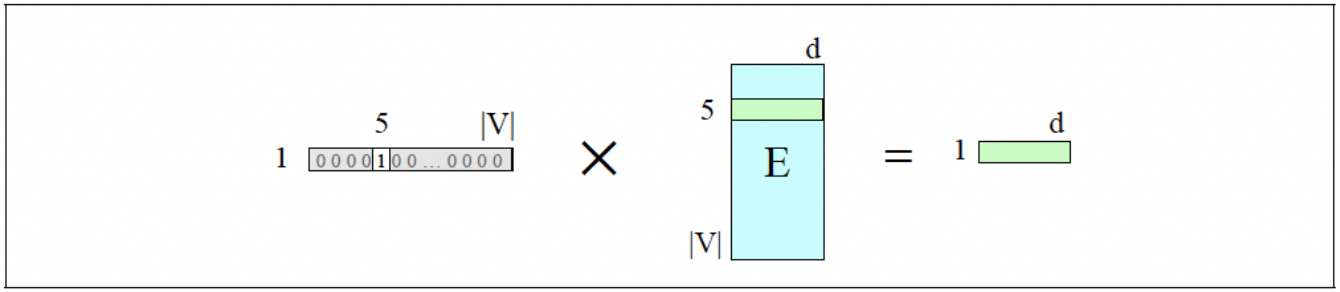


Figure 10.10 Selecting the embedding vector for word V_5 by multiplying the embedding matrix **E** with a one-hot vector with a 1 in index 5.

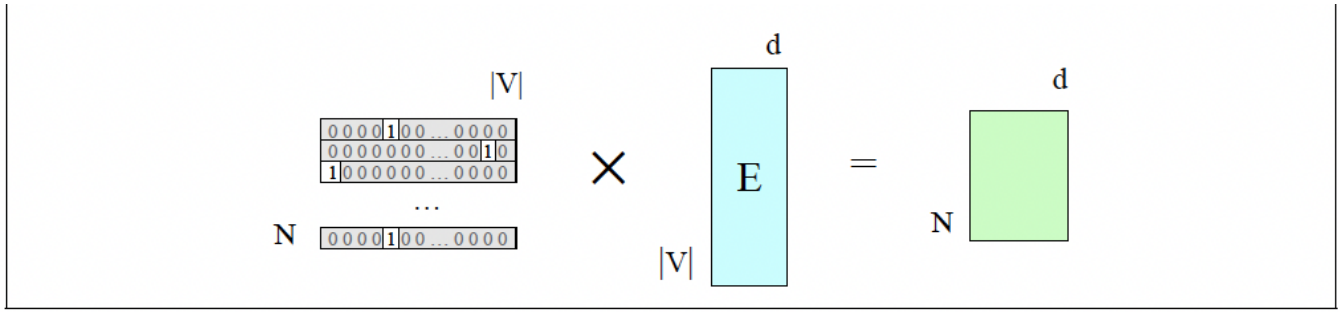
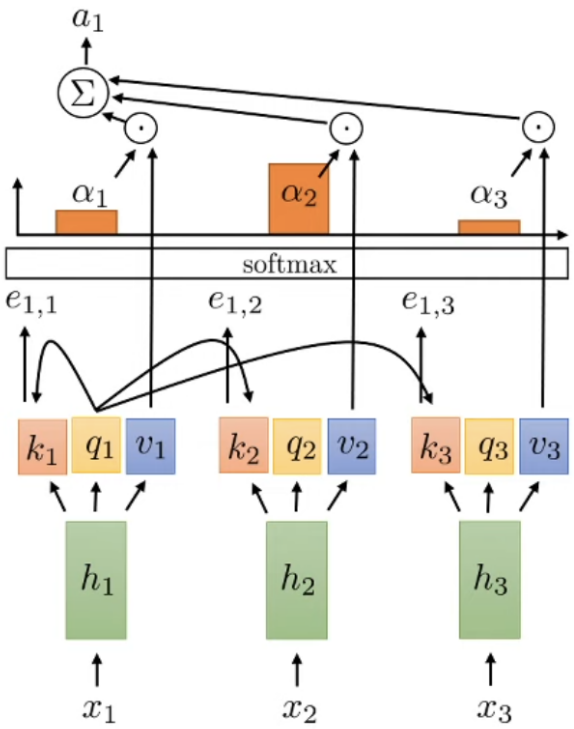


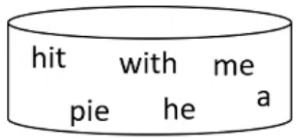
Figure 10.11 Selecting the embedding matrix for the input sequence of token ids W by multiplying a one-hot matrix corresponding to W by the embedding matrix **E**.

Positional encoding: what is the order?



what we see:

he hit me with a pie



what naïve self-attention sees:

a pie hit me with he

a hit with me he pie

he pie me with a hit

most alternative orderings are nonsense, but some change the meaning

in general the position of words in a sentence carries information!

Idea: add some information to the representation at the beginning that indicates where it is in the sequence!

$$h_t = f(x_t, t)$$

some function

Slide due to RAIL CS182: Lecture 12:Part 2: Transformers
https://www.youtube.com/watch?v=4AzsiCMw_-s

How to incorporate positional encoding?

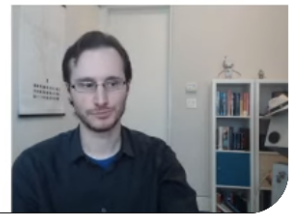
At each step, we have x_t and p_t

Simple choice: just concatenate them $\bar{x}_t = \begin{bmatrix} x_t \\ p_t \end{bmatrix}$

More often: just add after **embedding** the input

input to self-attention is $\text{emb}(x_t) + p_t$

some learned function (e.g., some fully connected layers with linear layers + nonlinearities)



Slide due to RAIL CS182: Lecture 12:Part 2: Transformers
https://www.youtube.com/watch?v=4AzsiCMw_-s

Simple Modelling of Position

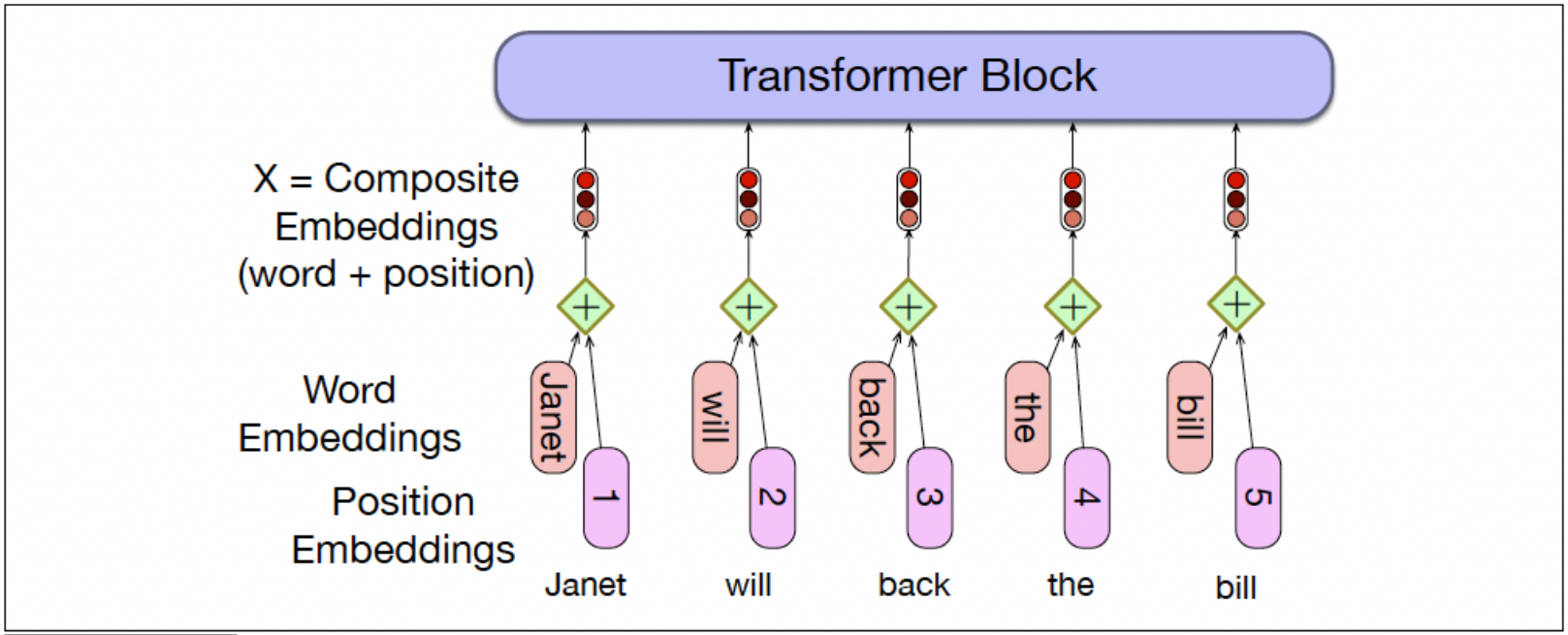


Figure 10.12 A simple way to model position: add an embedding of the absolute position to the token embedding to produce a new embedding of the same dimenionality.

Positional encoding: sin/cos

Naïve positional encoding: just append t to the input $\quad \bar{x}_t = \begin{bmatrix} x_t \\ t \end{bmatrix}$

This is not a great idea, because **absolute** position is less important than **relative** position

I walk my dog every day

every single day I walk my dog

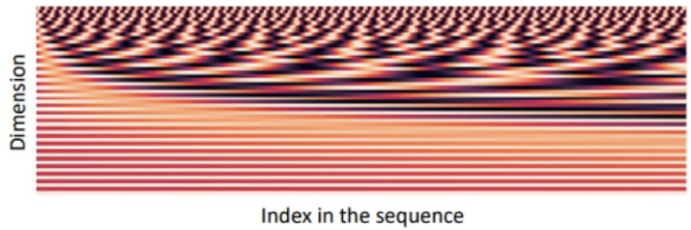
The fact that "my dog" is right after "I walk" is the important part, not its absolute position

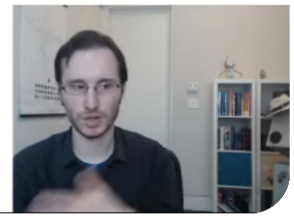
we want to represent **position** in a way that tokens with similar **relative** position have similar **positional encoding**

Idea: what if we use **frequency-based** representations?

$$p_t = \begin{bmatrix} \sin(t/10000^{2*1/d}) \\ \cos(t/10000^{2*1/d}) \\ \sin(t/10000^{2*2/d}) \\ \cos(t/10000^{2*2/d}) \\ \dots \\ \sin(t/10000^{2*\frac{d}{2}/d}) \\ \cos(t/10000^{2*\frac{d}{2}/d}) \end{bmatrix}$$

dimensionality of positional encoding

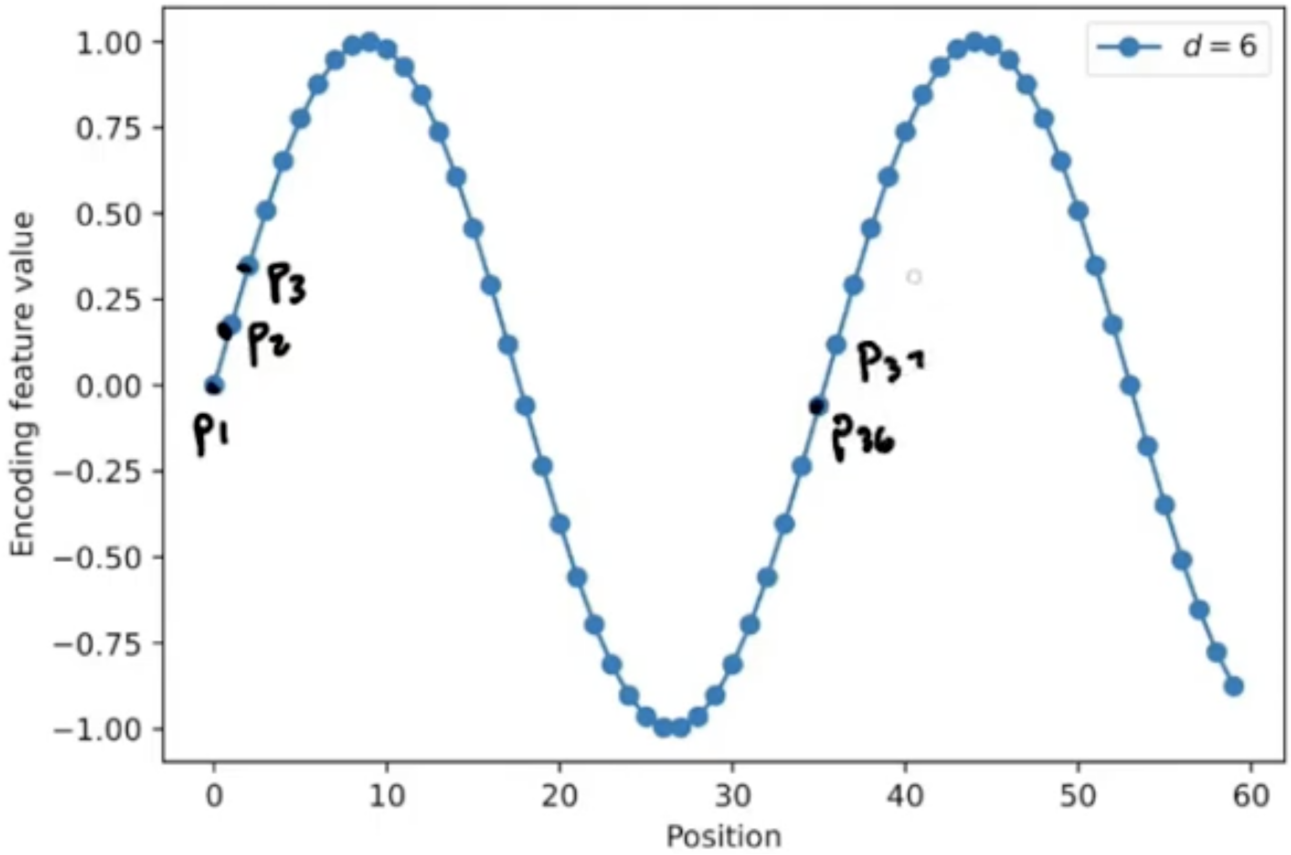




Slide due to RAIL CS182: Lecture 12:Part 2: Transformers
https://www.youtube.com/watch?v=4AzsiCMw_-s

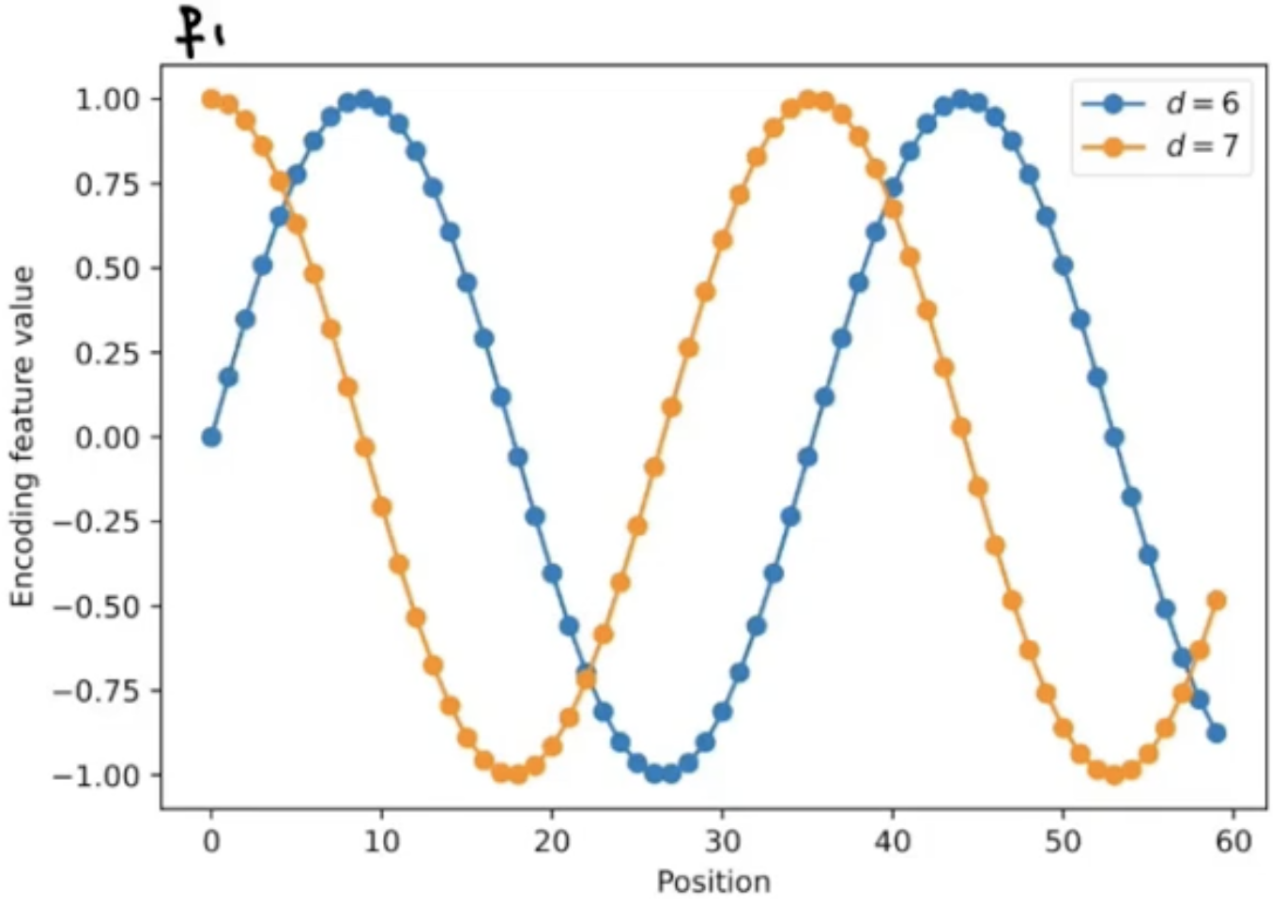
Represent position using sinusoids

Let's use a single sinusoid as our p_t:



Slide due to `https://www.youtube.com/watch?v=5V9gZcAd6cE&t=4s`

Let's add a cosine to obtain $\mathbf{p}_t$:



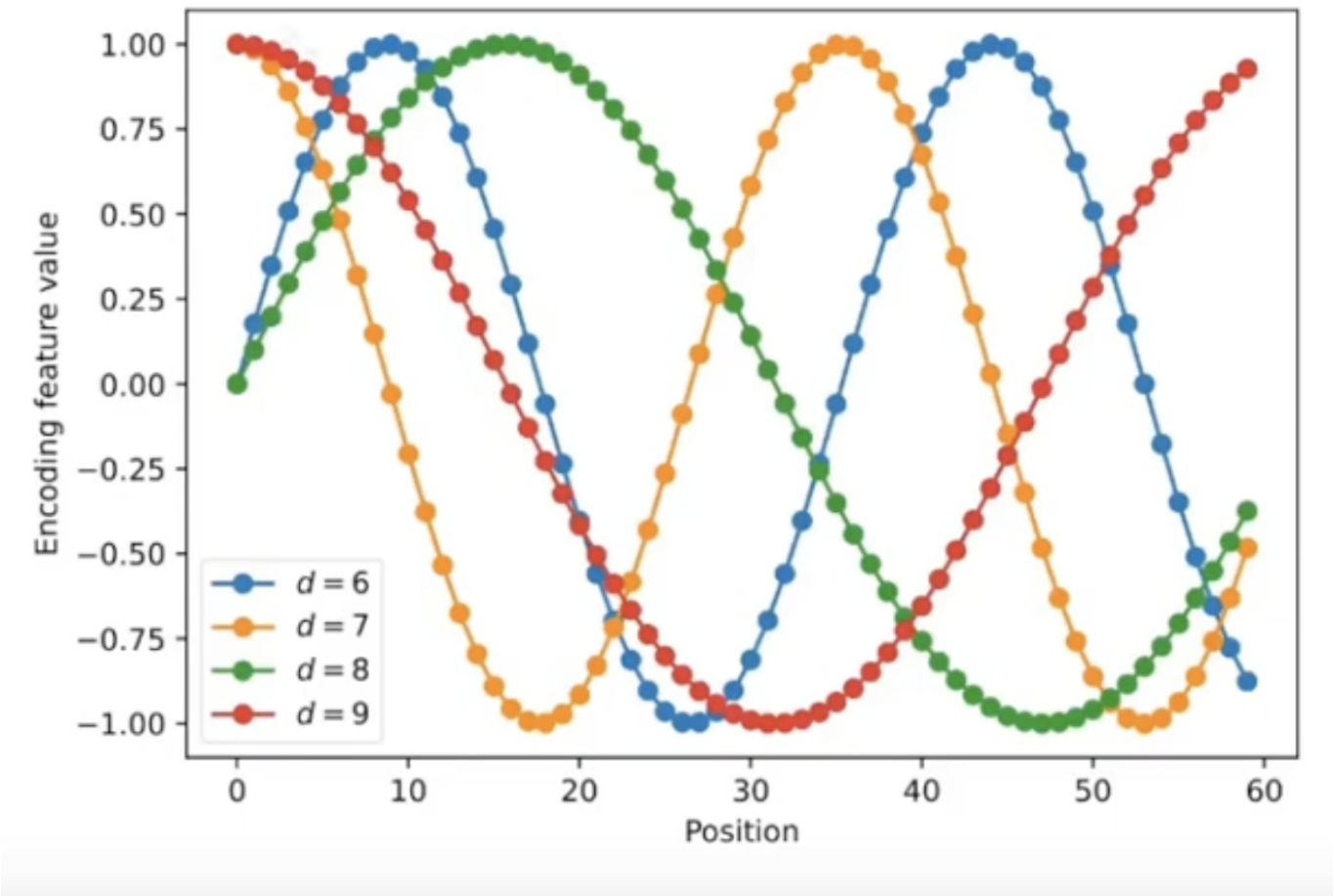
Slide due to `https://www.youtube.com/watch?v=5V9gZcAd6cE&t=4s`

Formally (Vaswani et al., 2017):

$$\mathbf{p}_t = \begin{bmatrix} \sin\left(\frac{t}{\lambda_1}\right) \\ \cos\left(\frac{t}{\lambda_1}\right) \\ \sin\left(\frac{t}{\lambda_2}\right) \\ \cos\left(\frac{t}{\lambda_2}\right) \\ \vdots \\ \sin\left(\frac{t}{\lambda_{D/2}}\right) \\ \cos\left(\frac{t}{\lambda_{D/2}}\right) \end{bmatrix}$$

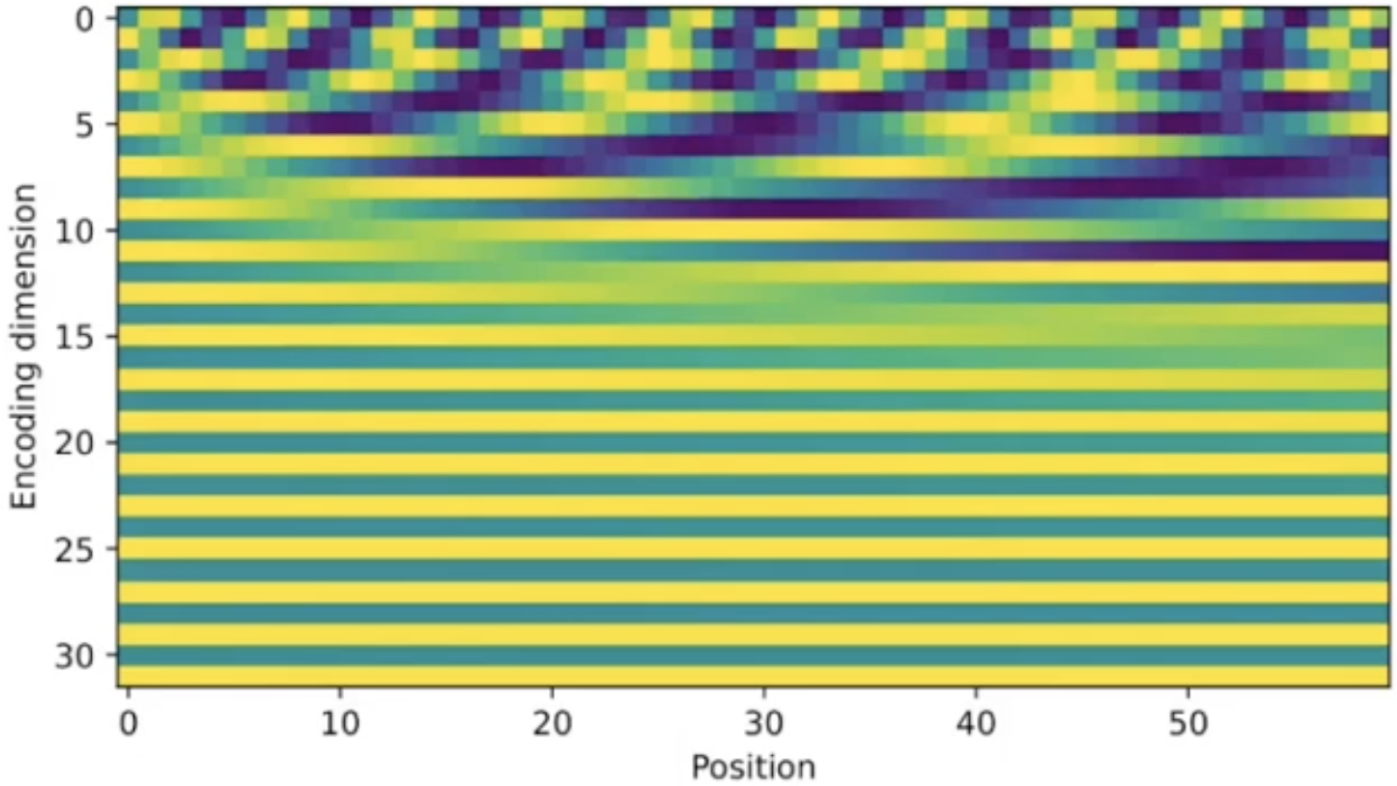
where

$$\lambda_m = 10\,000^{2m/D}$$



Slide due to `https://www.youtube.com/watch?v=5V9gZcAd6cE&t=4s`

If we stack all these into $\mathbf{P} \in \mathbb{R}^{D \times T}$:



Slide due to `https://www.youtube.com/watch?v=5V9gZcAd6cE&t=4s`

Recap of Language Modeling of Earlier Chapters

- Simple n-gram models (Chapter 3) with limited context window of n-1 prior words, depending on the choice of n-grams (typically bigrams or trigrams)
- Feedforward language models (Chapter 7), using word embeddings and fixed-size, sliding context windows as input
- RNN language models (Chapter 9), using recurrent (gated) connections and extended contexts

Language Modeling with Transformers

- Transformer-based language models can use extended context sizes: up to 2048 or evn 4096 tokens for very large models.
- Language modeling heads takes the ouput of the final transformer layer from the last token N and uses it to predict the following word at position N+1.

The Language Modeling Head

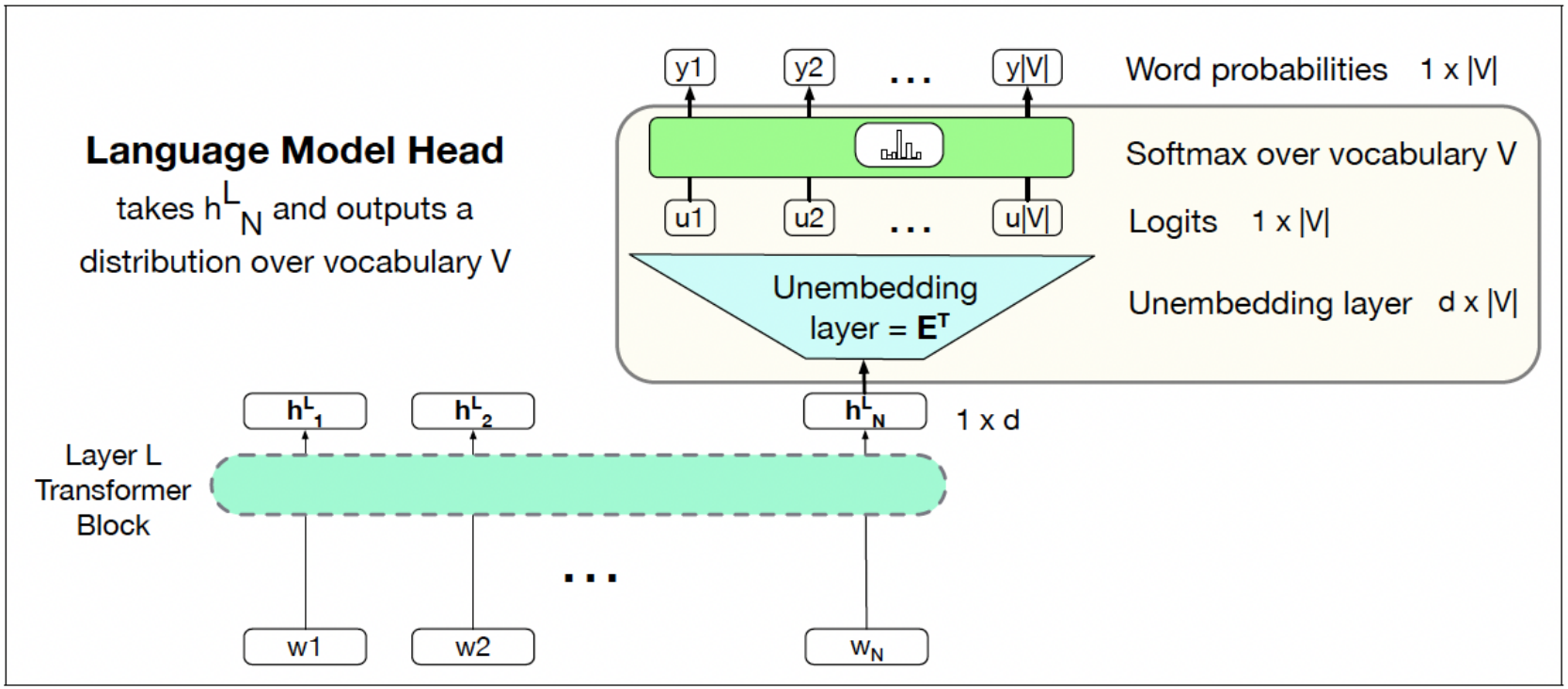


Figure 10.13 The language modeling head: the circuit at the top of a transformer that maps from the output embedding for token N from the last transformer layer ($\mathbf{h}_N^L$) to a probability distribution over words in the vocabulary V.

Modules of the Language Modeling Head

- Input : h_N^L (for token N of the last transformer layer L
- Unembedding layer with shape [d x | V |]
- Logit (or score) vecto**r u** with shape [1 x | V |]
- Softmax calculation over the vocabulary | V |
- Output: word probability distribution, with shape [1 x | V |]

$$\mathbf{u} = \mathbf{h_N^L E^T} \tag{25a}$$

$$\mathbf{y} = \mathit{softmax}(\mathbf{u}) \tag{25b}$$

Unembedding as Weight Tying

- The initial linear layer of the language modeling head that outputs the logits is typically tied to the embedding matrix **E** as the transpose of **E**.

- At the input stage of the transformer, the embedding matrix **E** of shape [|V| x d] is used to map from a one-hot vector over the vocabulary of shape [1 x |V|] to an embedding (of shape [1 x d])

- In the language model head, the transpose of the embedding matrix (of shape [d x |V|]) is used to map back from an embedding (of shape [1 x d]) to a vector over the vocabulary (shape [1 x |V|]).

Unembedding as Weight Tying

- The transpose $\mathbf{E}^T$ is often called the *unembedding layer* because it is performing this reverse mapping, compared to the mapping at the input stage.
- In the learning process $\mathbf{E}$ will be optimized to be good at doing both of these mappings.
- The unembedding layer can also be used as a tool, called *the logit lens*, for the interpretability of the internals of the transformer

Transformer Decoder only Model

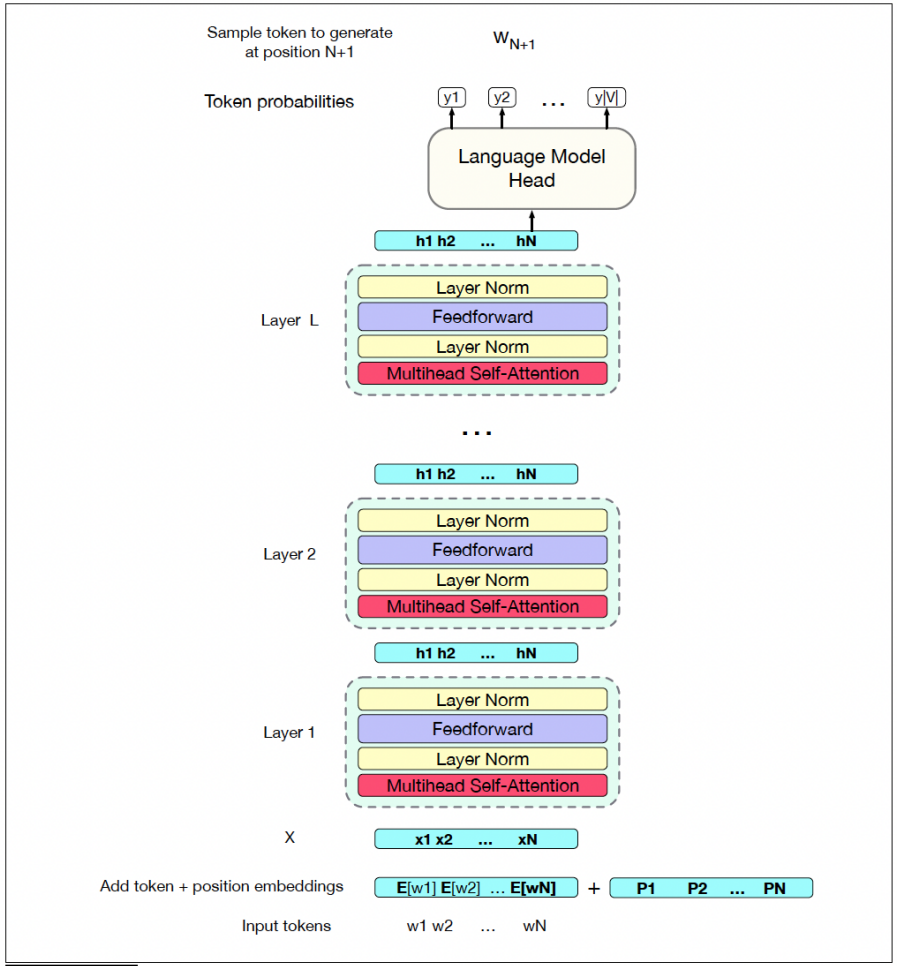


Figure 10.14 A final transformer decoder-only model, stacking post-norm transformer blocks and mapping from a set of input tokens w_1 to w_N to a predicted next word w_{N+1}.

Autoregressive Text Completion with Transformer-based Large Language Models

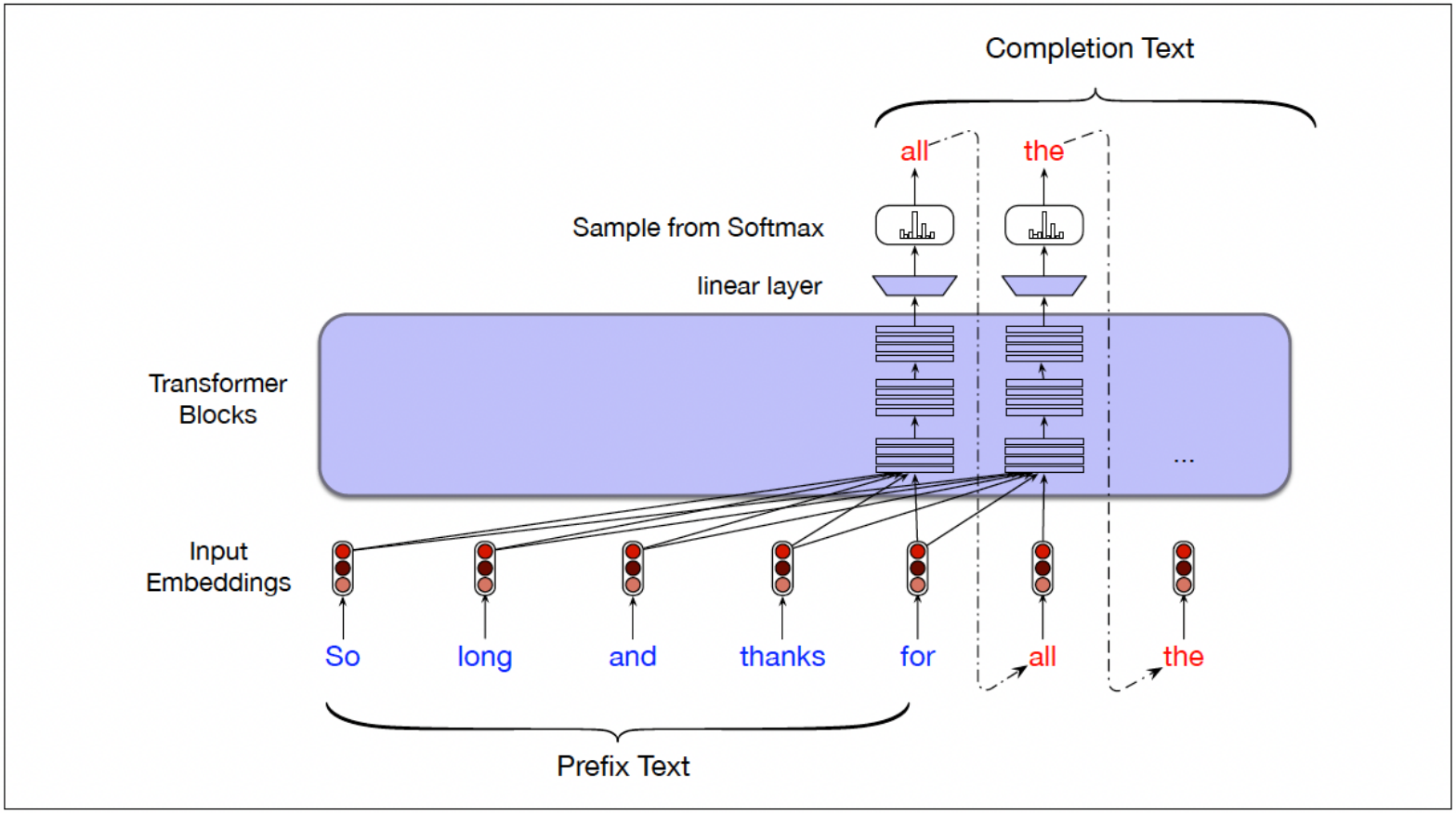
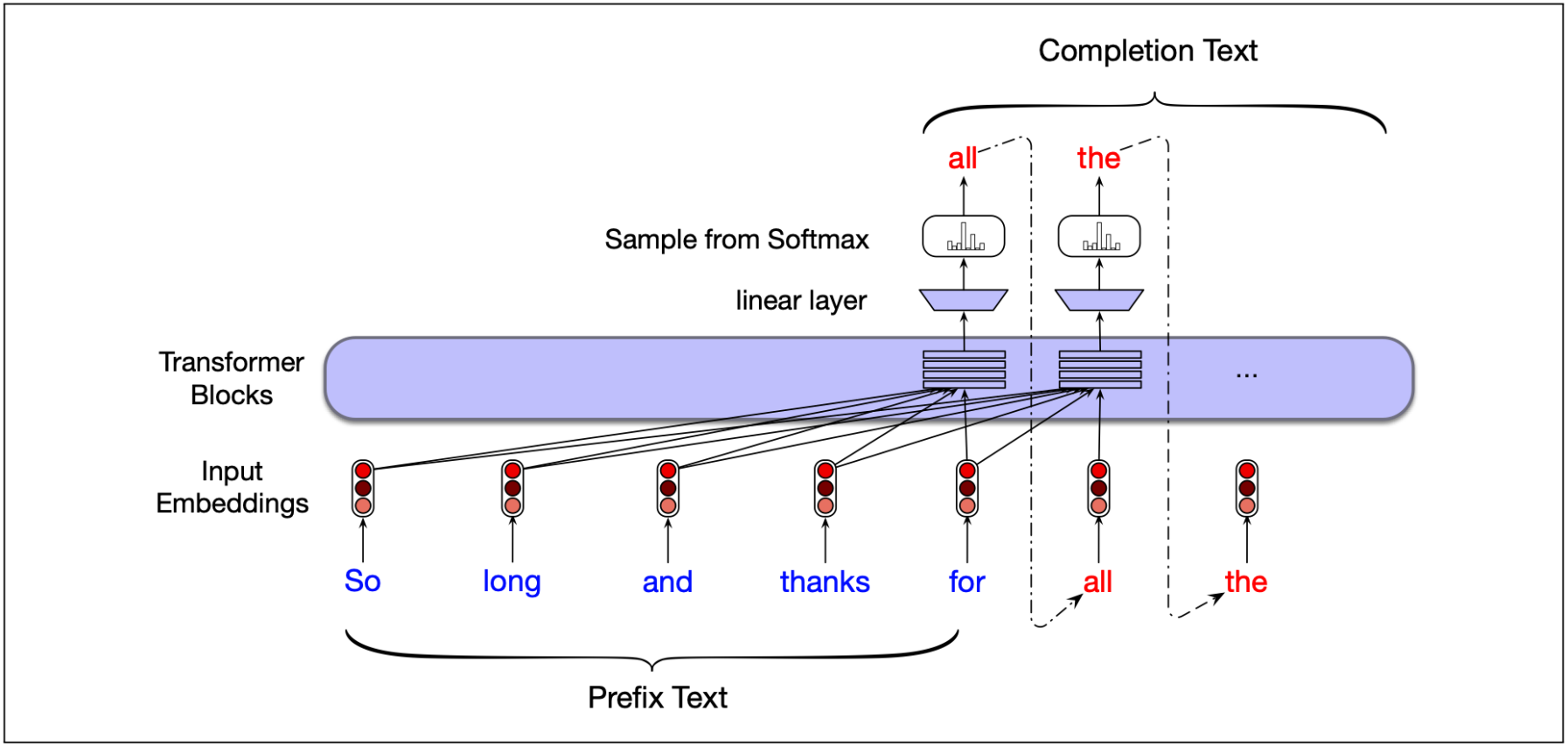


Figure 10.15 Autoregressive text completion with transformer-based large language models.

Contextual Generation and Summarization



Contextual Generation and Summarization

Original Article

The only thing crazier than a guy in snowbound Massachusetts boxing up the powdery white stuff and offering it for sale online? People are actually buying it. For $89, self-styled entrepreneur Kyle Waring will ship you 6 pounds of Boston-area snow in an insulated Styrofoam box – enough for 10 to 15 snowballs, he says. But not if you live in New England or surrounding states. “We will not ship snow to any states in the northeast!” says Waring’s website, ShipSnowYo.com. “We’re in the business of expunging snow!”

Contextual Generation and Summarization

Continued

His website and social media accounts claim to have filled more than 133 orders for snow – more than 30 on Tuesday alone, his busiest day yet. With more than 45 total inches, Boston has set a record this winter for the snowiest month in its history. Most residents see the huge piles of snow choking their yards and sidewalks as a nuisance, but Waring saw an opportunity.

Contextual Generation and Summarization

Continued

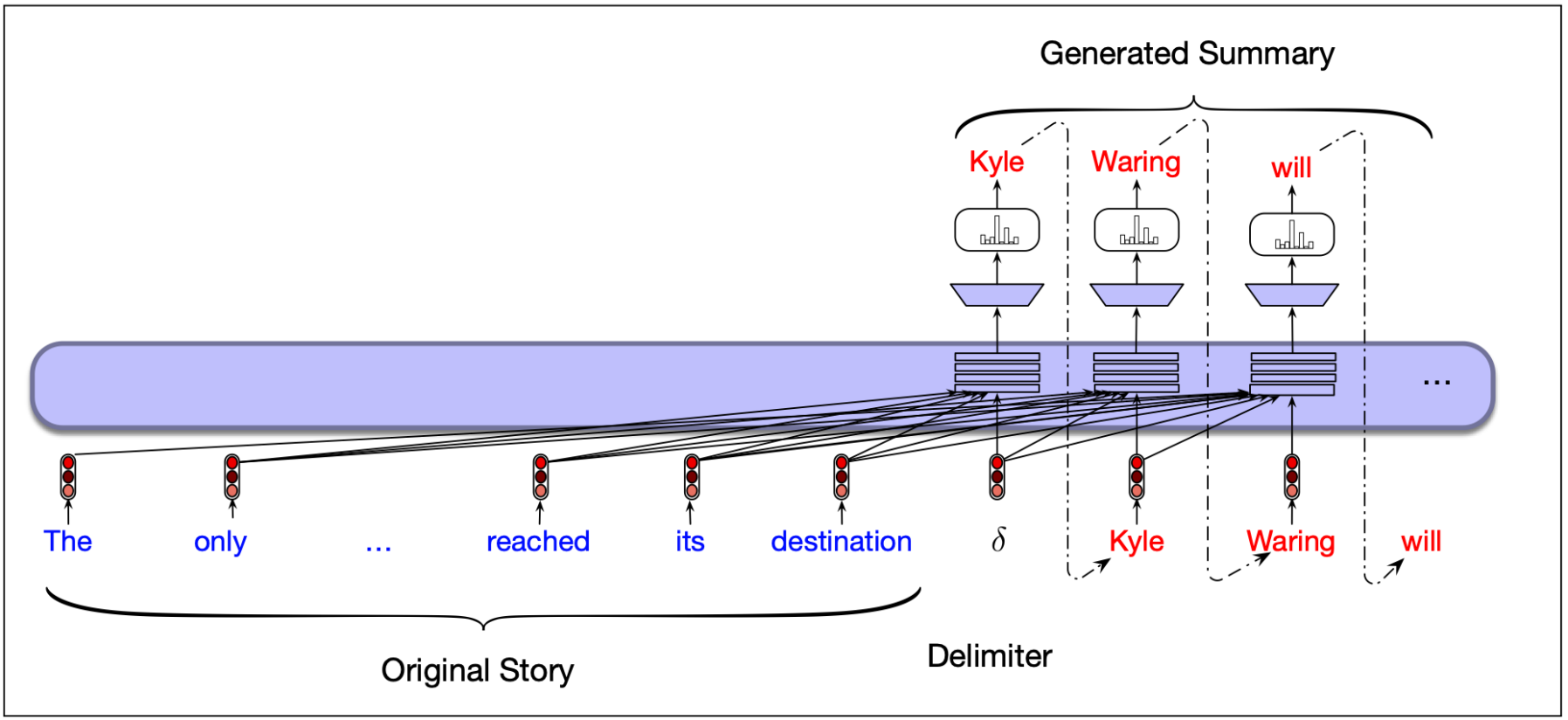
According to Boston.com, it all started a few weeks ago, when Waring and his wife were shoveling deep snow from their yard in Manchester-by-the-Sea, a coastal suburb north of Boston. He joked about shipping the stuff to friends and family in warmer states, and an idea was born. His business slogan: "Our nightmare is your dream!" At first, ShipSnowYo sold snow packed into empty 16.9-ounce water bottles for $19.99, but the snow usually melted before it reached its destination...

Contextual Generation and Summarization

Summary

Kyle Waring will ship you 6 pounds of Boston-area snow in an insulated Styrofoam box – enough for 10 to 15 snowballs, he says. But not if you live in New England or surrounding states.

Contextual Generation and Summarization



Large Language Models: Generation by Random Sampling

i $\leftarrow$ 1
$w_i \sim p(w)$
while w_i != EOS
    i $\leftarrow$ i + 1
    $w_i \sim p(w_i \mid w_{<i})$

Large Language Models: Generation by Top Sampling

1. Choose Choose in advance a number of words k
2. For each word in the vocabulary V, use the language model to compute the likelihood of this word given the context $p(w_t \mid w_{<t})$
3. Sort the words by their likelihood, and throw away any word that is not one of the top k most probable words.
4. Renormalize the scores of the k words to be a legitimate probability distribution.

Large Language Models: Generation by Top-p or by Nucleus Sampling

Keep not the top k words, but the top p percent of the probability mass:

Given a distribution $P(w_t \mid w_{<t})$, the top-p vocabulary $V^{(p)}$ is the smallest set of words such that:

$$\sum_{w \in V^{()}} P(w \mid w_{<t}) \geq p \tag{26}$$

Large Language Models: Generation by Temperature Sampling

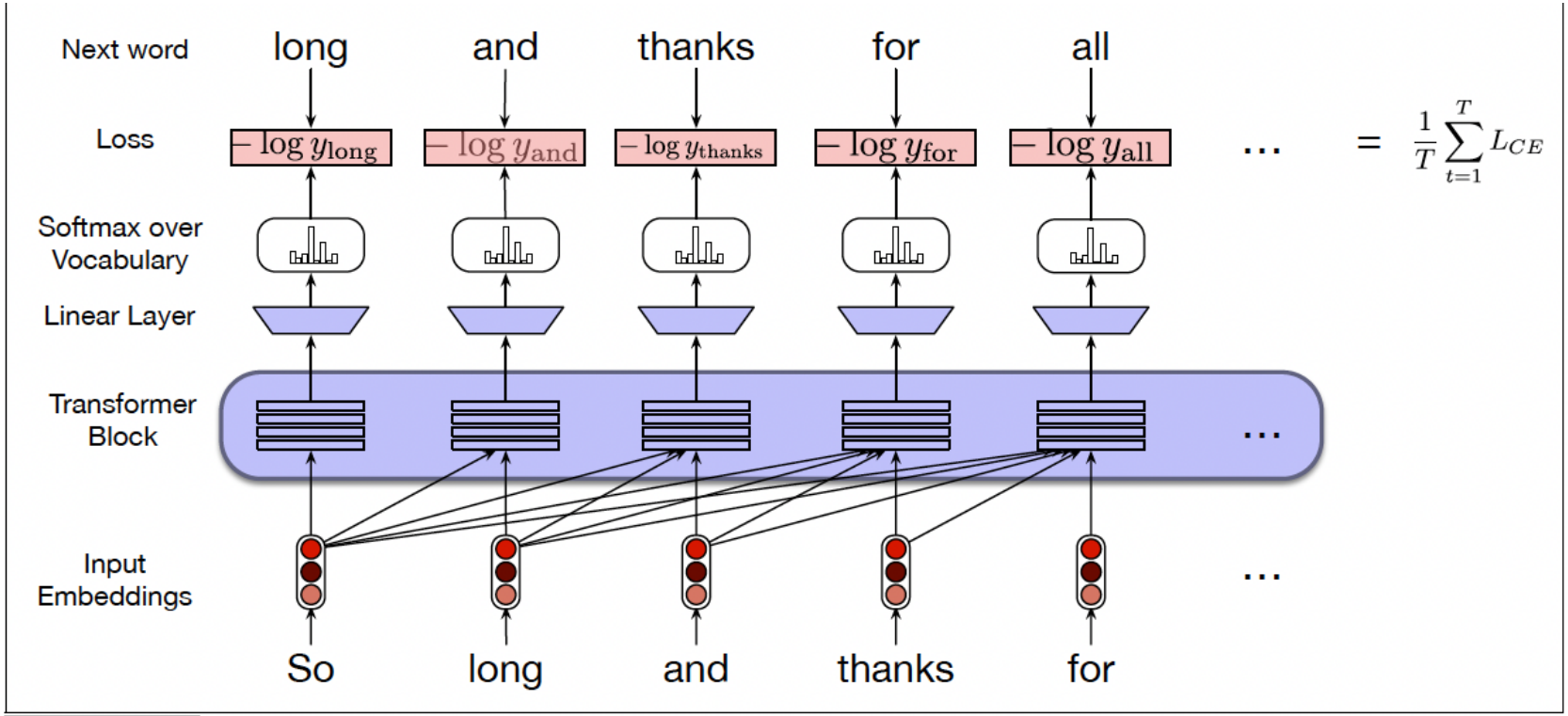
Instead of computing the probability distribution directly from the logit (vector) a in:

$$\mathbf{y} = softmax(u) \tag{27}$$

first divide the logits by τ and compute the probability vector as

$$\mathbf{y} = softmax(u/\tau) \tag{28}$$

Training a Transformer as a Language Model



Training a Transformer as a Language Model

- Training by **self-supervision** on the training corpus with prediction of next word as the task
- **Cross-entropy loss** as the negative log probability assigned to the next word w_{t+1} by the model

$$L_{CE}(\hat{\mathbf{y}}_t, y_t) = -\log \hat{\mathbf{y}}_t[w_{t+1}] \tag{29}$$

- With **teacher forcing:** model is always given the correct history sequence to predict the next word.

Training Corpora for Large Language Model (LLM)

- LLMs are mainly trained on text scraped from the web, augmented by more carefully curated data.
  - Common Crawl (`https://commoncrawl.org`)
  - or cleanup versions such as Collosal Clean Crawled Corpus (C4), a corpus of 156 billion tokens of English, largely taken from patent documents
- GPT3 models are trained mostly on the web (429 billion tokens), some text from books (67 billion tokens), and Wikipedia (3 billion tokens).

Performance of LLMs

Performance of LLMs is mainly determined by 3 factors:

- model size (the number of parameters not counting embeddings), dataset
- size (the amount of training data),
- the amount of computer processing time used for training.

Scaling Laws for LLMs

Loss L as a function of the number of non-embedding parameters N, dataset size D, and the compute budget C.

$$L(N) = \left(\frac{N_c}{N}\right)^{\alpha_N} \tag{30}$$

$$L(D) = \left(\frac{D_c}{D}\right)^{\alpha_D} \tag{31}$$

$$L(C) = \left(\frac{C_c}{C}\right)^{\alpha_C} \tag{32}$$

Empirical Performance for 3 Factors Scaled in tandem

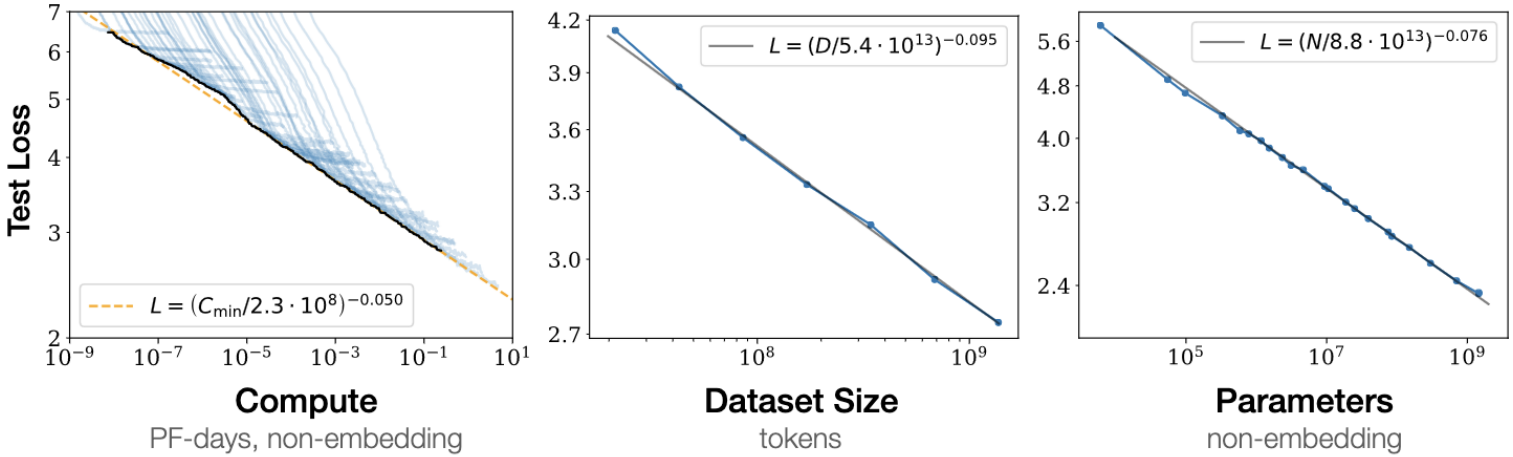


Figure 1 Language modeling performance improves smoothly as we increase the model size, datasetset size, and amount of compute[2] used for training. For optimal performance all three factors must be scaled up in tandem. Empirical performance has a power-law relationship with each individual factor when not bottlenecked by the other two.

Estimating the Number of Parameters for LLMs

The number of (non-embedding) parameters N can be roughly computed as follows (ignoring biases, and with d as the input and output dimensionality of the model, d_{attn} as the self-attention layer size, and d_{ff} the size of the feedforward layer):

$$N \approx 2\,d\,n_{layer}(2d_{attn} + d_{ff}) \approx 12\,n_{layer}\,d^2 \quad (\text{assuming}\, d_{attn} = d_{ff}/4 = d) \tag{33}$$

- Thus, GTP-3 with $n = 96$ layers and dimensionality $d = 12288$, has 12 x 96 x $122882^2 \approx 175$ billion parameters.
- The values of N_c, D_c, C_c, α_N, α_D and α_C depend on the exact transformer architecture, tokenization, and vocabulary size, so rather than all the precise values, scaling laws focus on the relationship with loss.

Potential Harms from (Large) Language Models

- **Hallucination**: LLMs are prone to saying things that are wrong.
- **Cultural Bias**: LLMs reflect the biases inherent in their training data.
- **Toxic Language**: LLMs can generate hate speech output.
- **Privacy and Copyright Issues**: LLMs can leak about their training data.