

CS11-711 Advanced NLP

Transformers

Sean Welleck

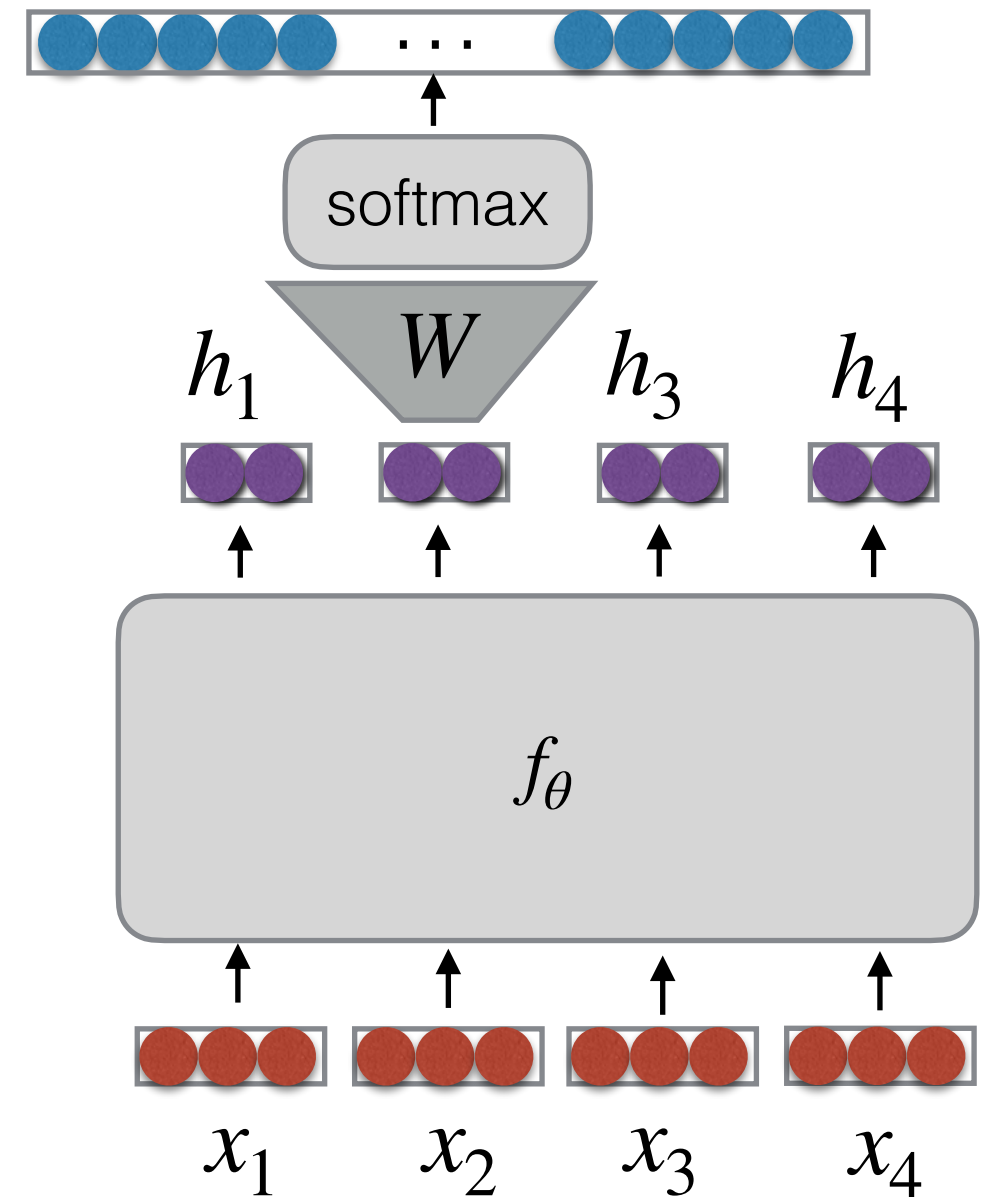
**Carnegie
Mellon
University**



<https://cmu-l3.github.io/anlp-spring2026/>
<https://github.com/cmu-l3/anlp-spring2026-code>

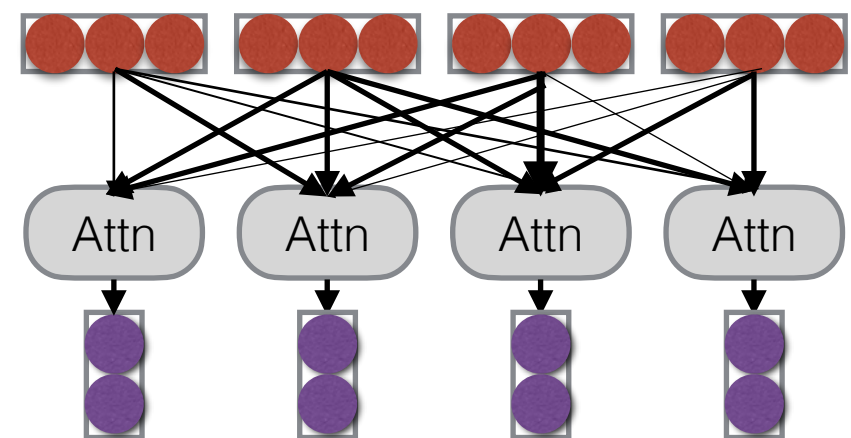
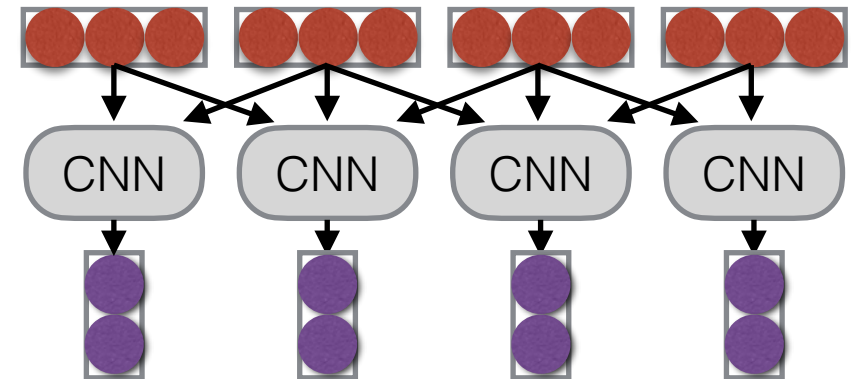
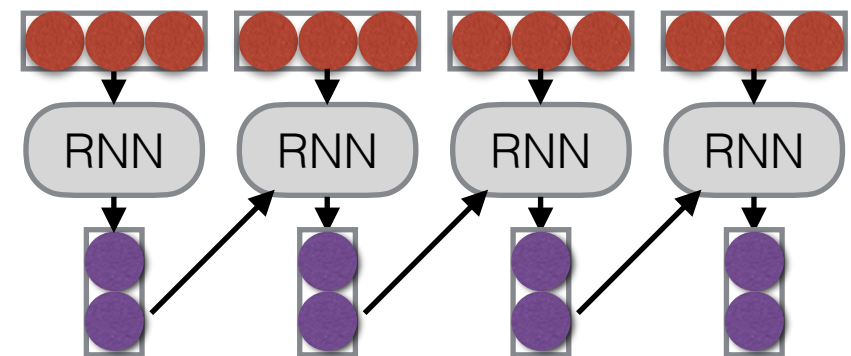
Recap: sequence model

- $f_{\theta}(x_1, \dots, x_{|x|}) \rightarrow h_1, \dots, h_{|x|}$
 - $h_t \in \mathbb{R}^d$: hidden state
- Language modeling:
 - $p_{\theta}(\cdot | x_{<t}) = \text{softmax}(Wh_t^{\top})$



Three types of sequence models

- **Recurrence:** Condition representations on an encoding of the history
- **Convolution:** Condition representations on local context
- **Attention:** Condition representations on a weighted average of all tokens



Today's lecture

- **Transformer**: a sequence model based on attention
- Roadmap:
 - Attention
 - Transformer architecture
 - Improved transformer architecture

Attention

Basic Idea

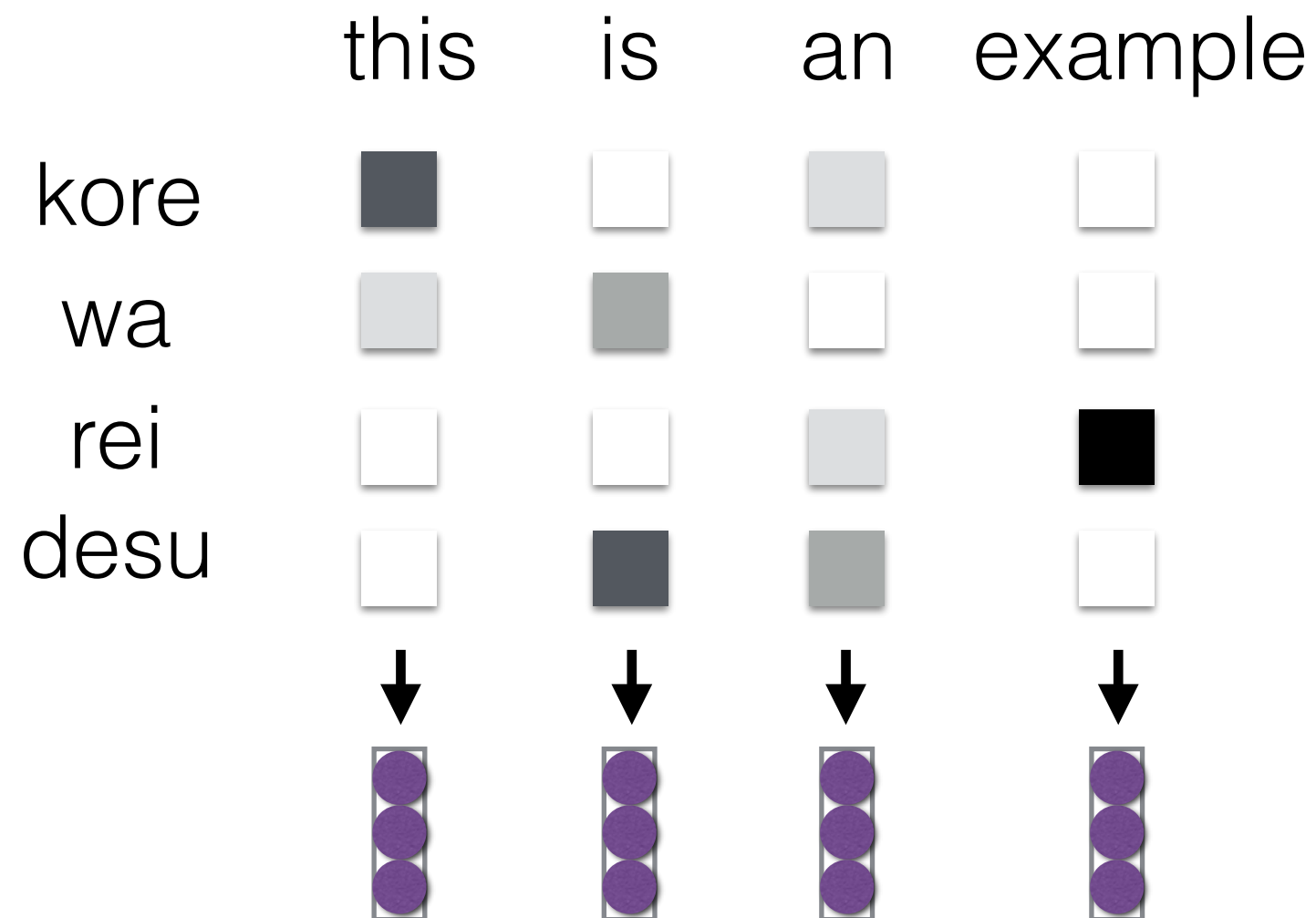
(Bahdanau et al. 2015)

- Encode each token in the sequence into a vector
- When decoding, perform a linear combination of these vectors, weighted by “attention weights”

Cross Attention

(Bahdanau et al. 2015)

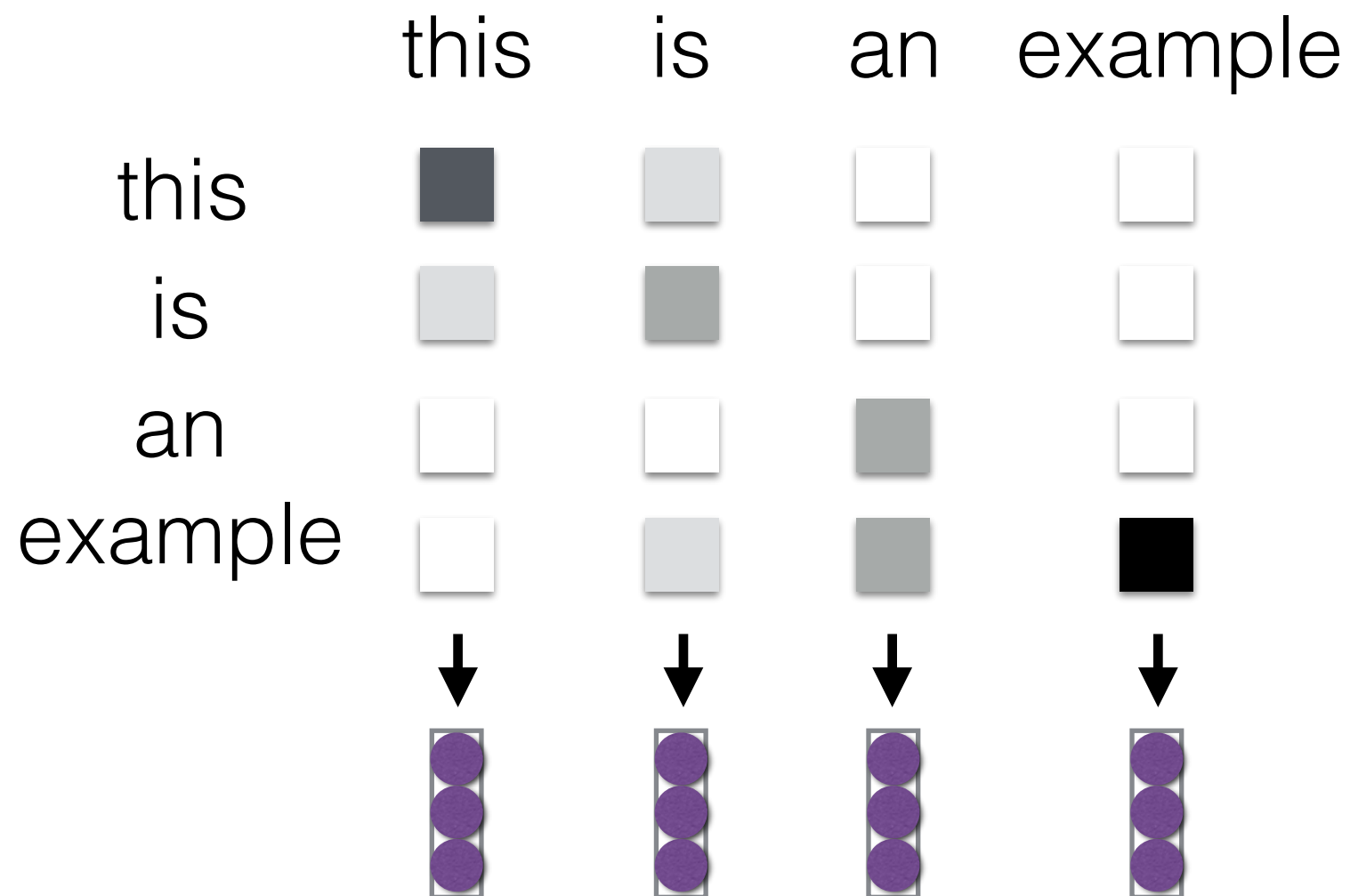
- Each element in a sequence attends to elements of another sequence



Self Attention

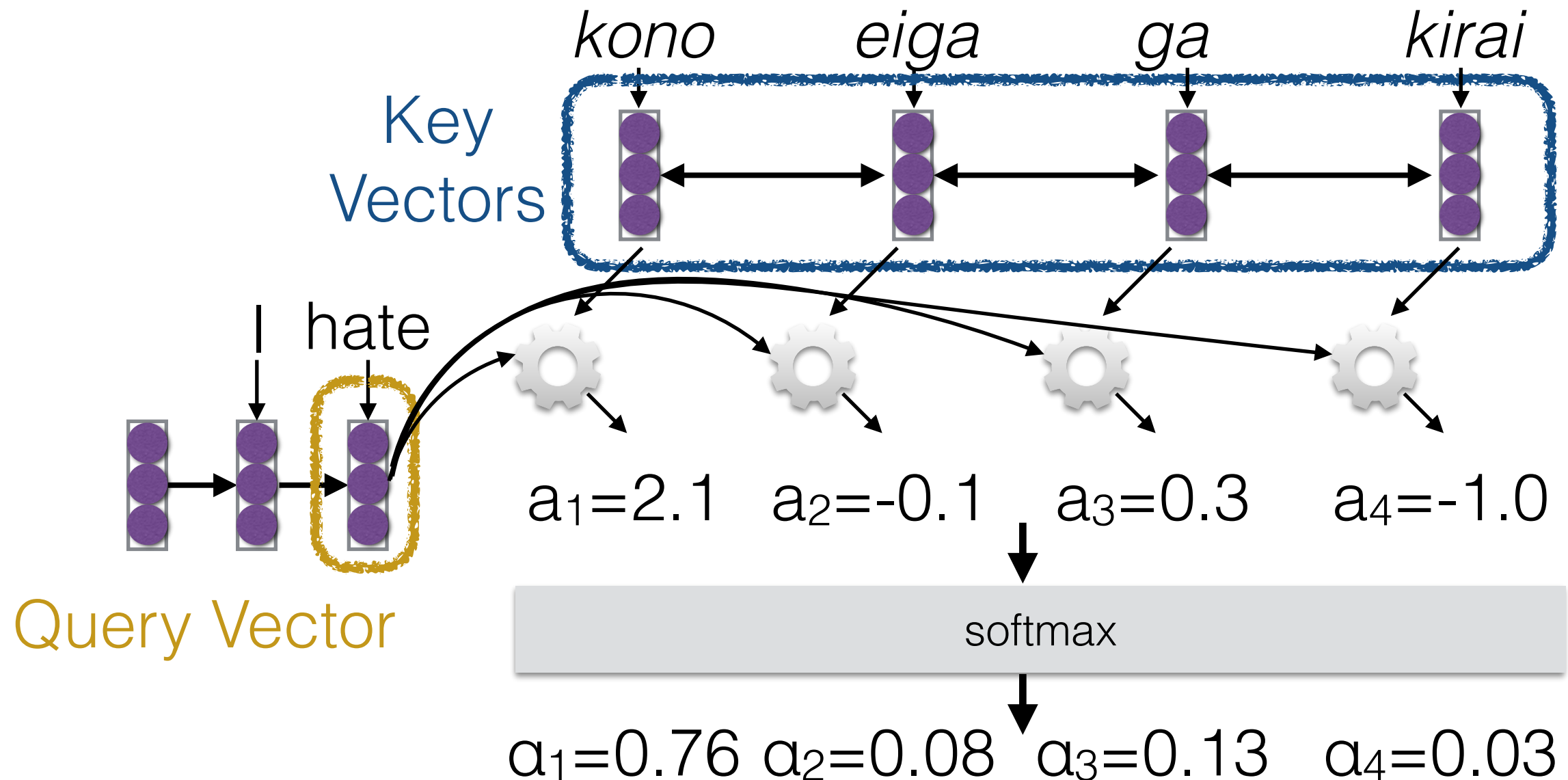
(Cheng et al. 2016, Vaswani et al. 2017)

- Each element in the sequence attends to elements of that sequence



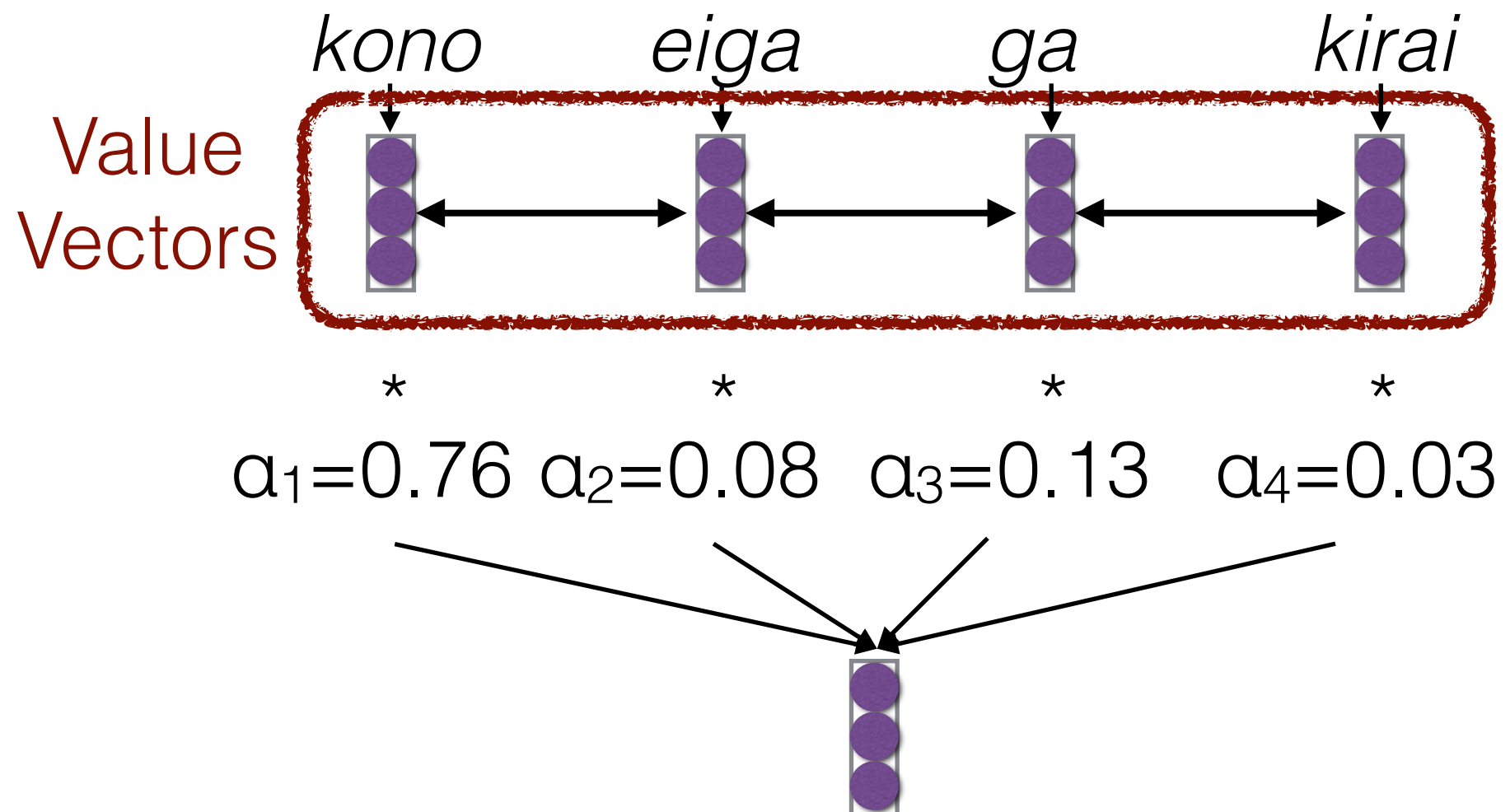
Calculating Attention (1)

- Use “query” vector (decoder state) and “key” vectors (all encoder states)
- For each query-key pair, calculate weight
- Normalize to add to one using softmax



Calculating Attention (2)

- Combine together value vectors (usually encoder states, like key vectors) by taking the weighted sum



- Use this in any part of the model you like

Query-key-value framework

- Keys $k_1, \dots, k_N$

- $k_i = W_K h_i$

- Values $v_1, \dots, v_N$

- $v_i = W_V h_i$

- Query q_t

- $q_t = W_Q h_t$

- $c_t = \sum_{i=1}^N \alpha_{t,i} v_i$ where

- $$\alpha_{t,i} = \frac{\exp(a(q_t, k_i))}{\sum_{j=1}^N \exp(a(q_t, k_j))}$$

- $a(q, k)$ is a weighting/compatibility function, e.g. $a(q, k) = q^\top k$

Cross-attention example

- Keys: based on encoder states h_i
- Values: based on encoder states h_i
- Query: based on decoder state $\tilde{h}_t$

Self-attention example

- Keys: based on decoder states h_i
- Values: based on decoder states h_i
- Query: based on decoder state h_t

A Graphical Example

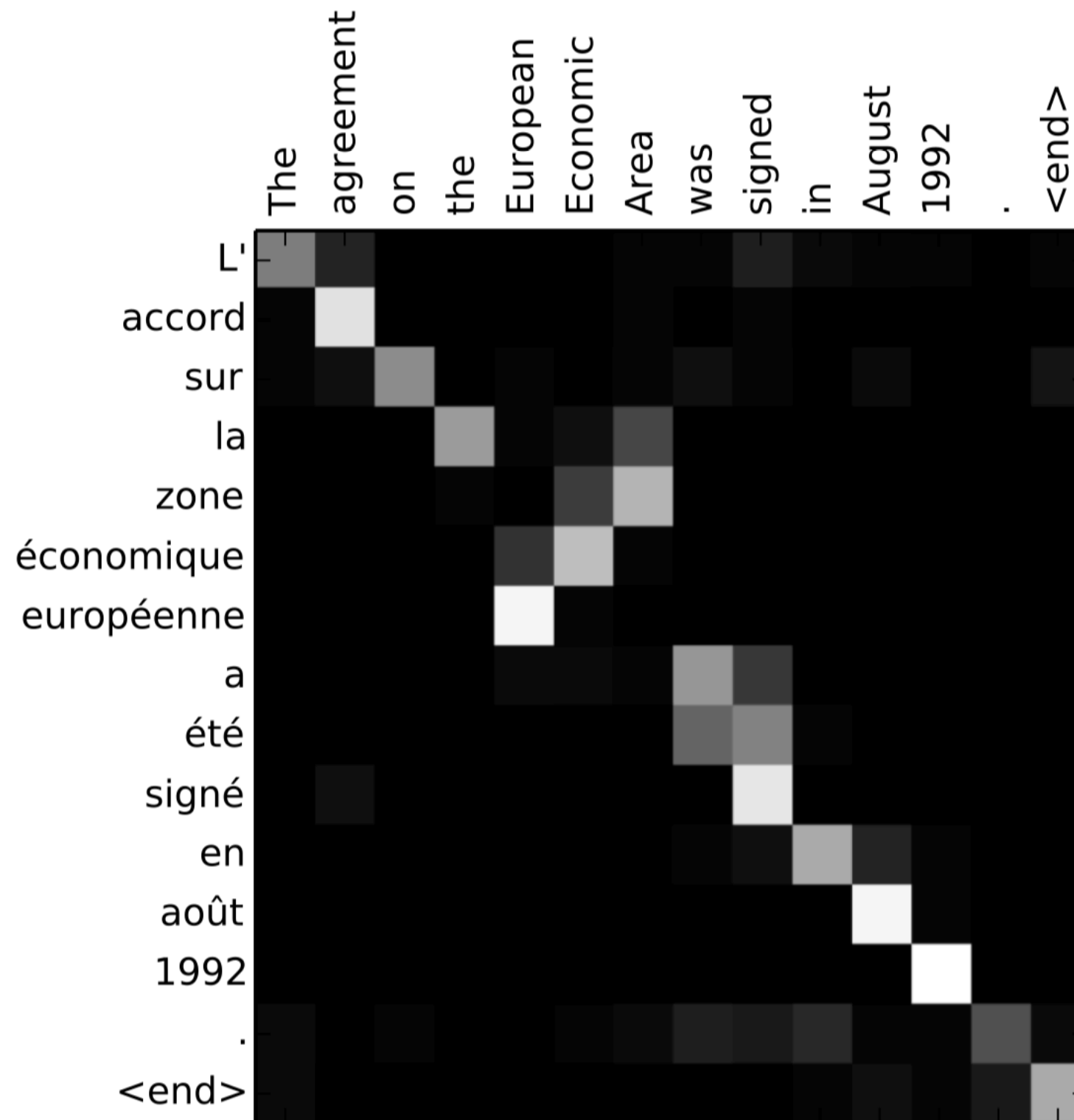


Image from Bahdanau et al. (2015)

Attention Score Functions

- **Dot Product** (Luong et al. 2015)

$$a(\mathbf{q}, \mathbf{k}) = \mathbf{q}^\top \mathbf{k}$$

- **Scaled Dot Product** (Vaswani et al. 2017)

- *Problem:* scale of dot product increases as dimensions get larger
- *Fix:* scale by size of the vector

$$a(\mathbf{q}, \mathbf{k}) = \frac{\mathbf{q}^\top \mathbf{k}}{\sqrt{|\mathbf{k}|}}$$

Today's lecture

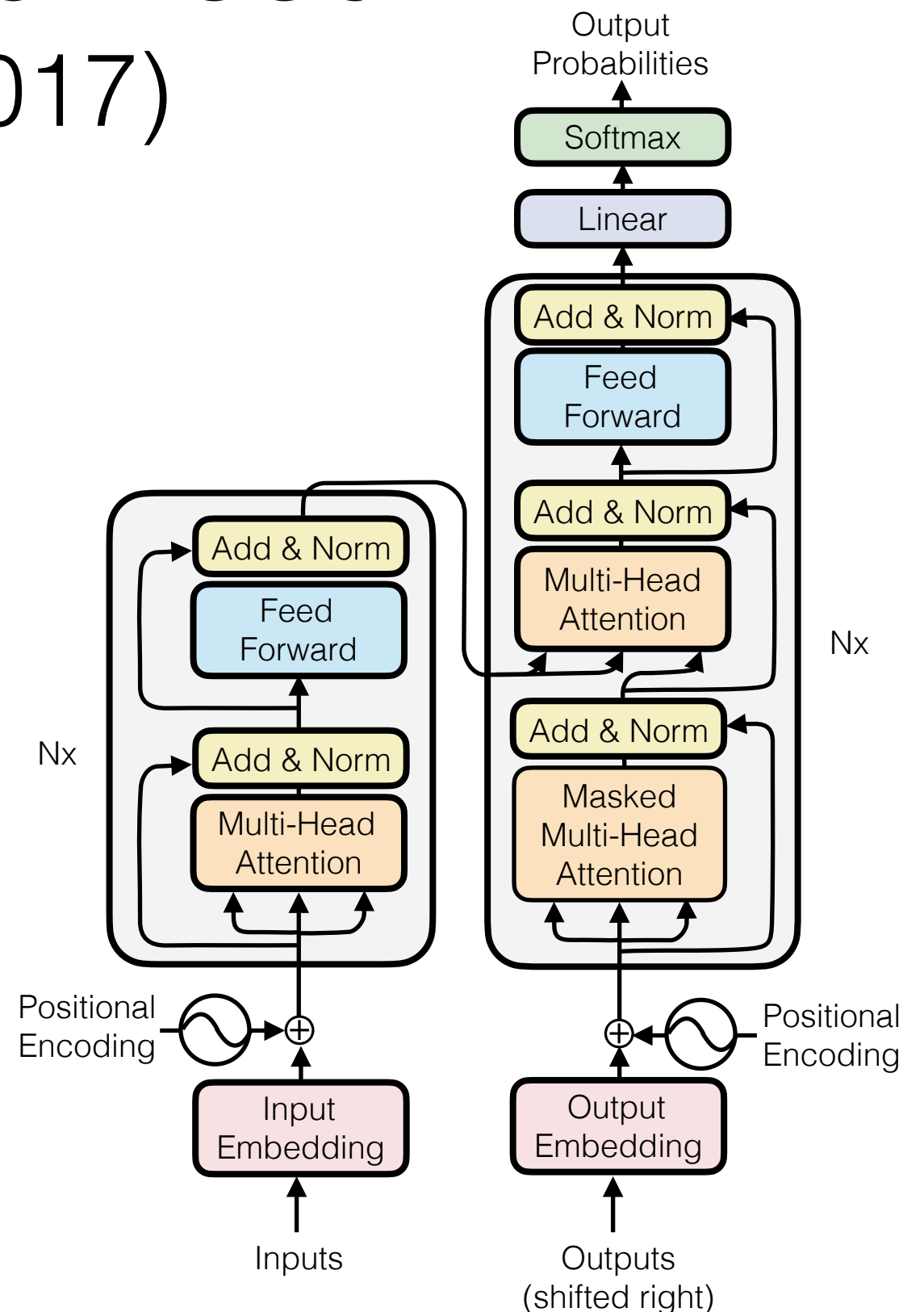
- Roadmap:
 - Attention
 - **Transformer architecture**
 - Improved transformer architecture

Transformers

“Attention is All You Need”

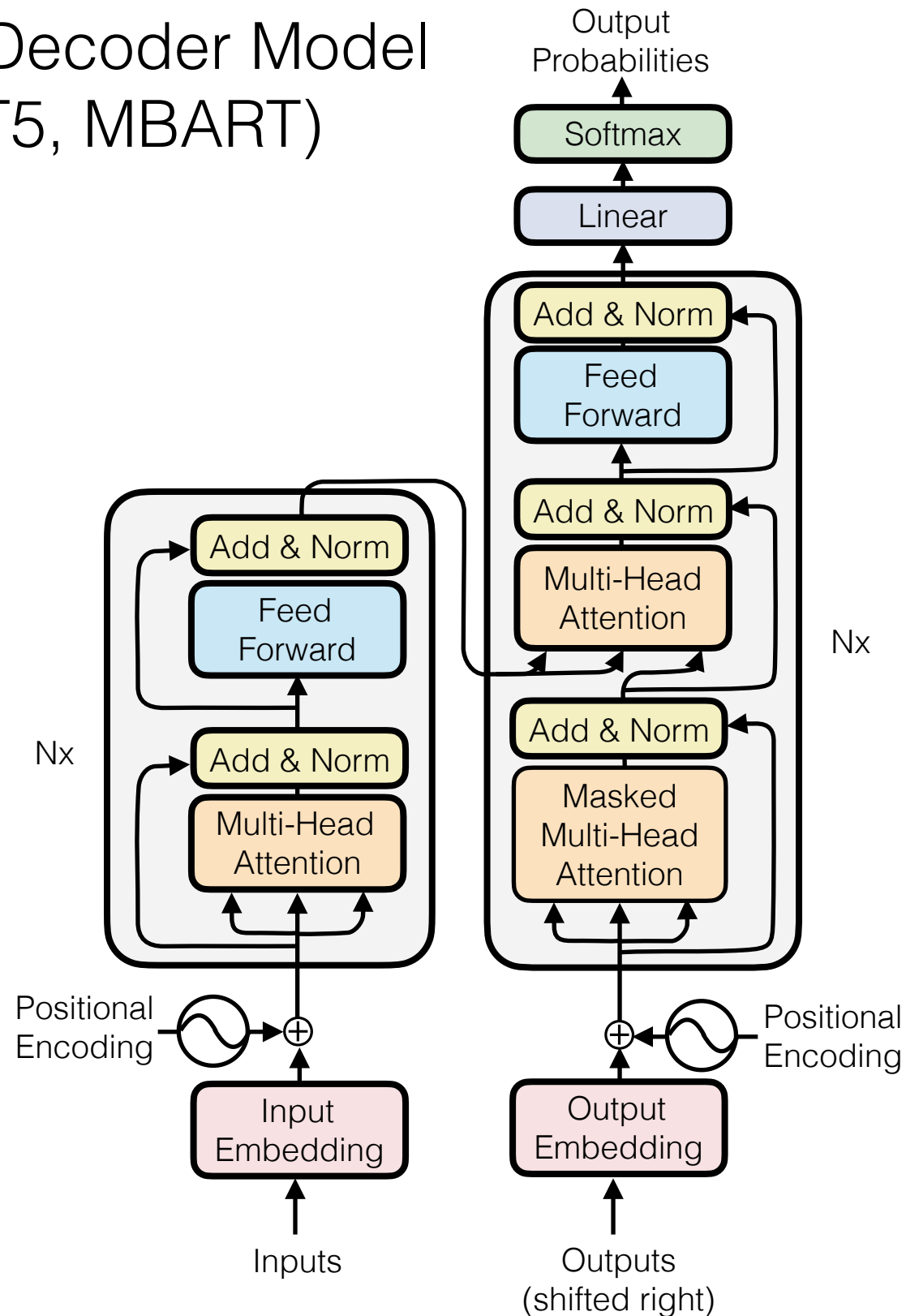
(Vaswani et al. 2017)

- A sequence-to-sequence architecture based entirely on attention
- Strong results on machine translation
- Fast: leverages parallelism from matrix multiplications

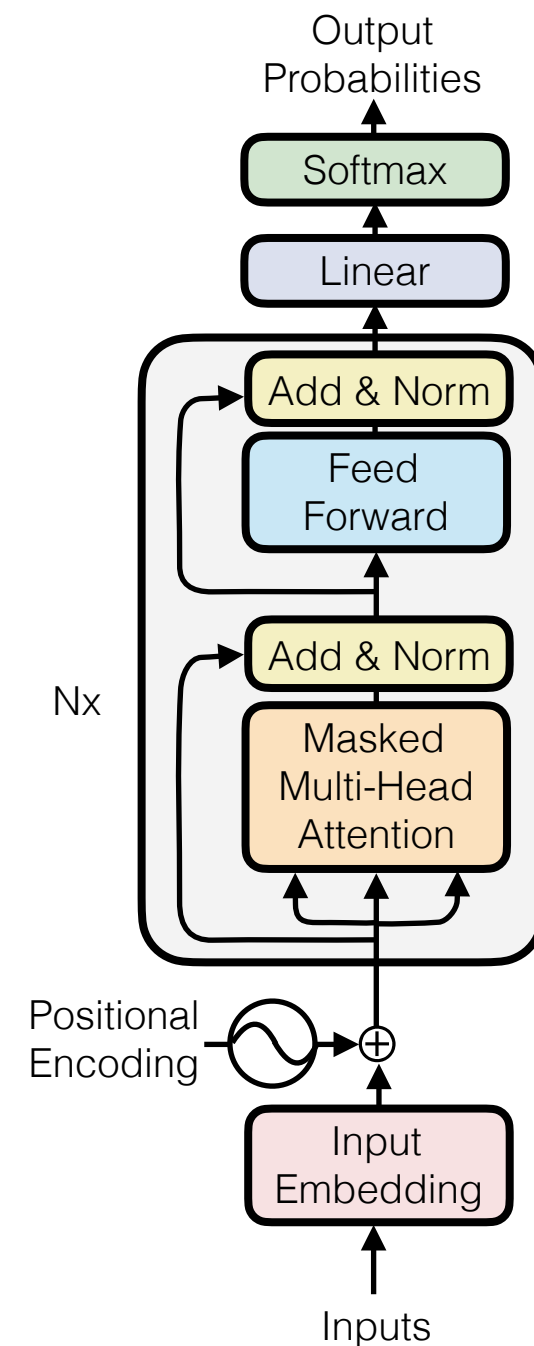


Two Types of Transformers

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

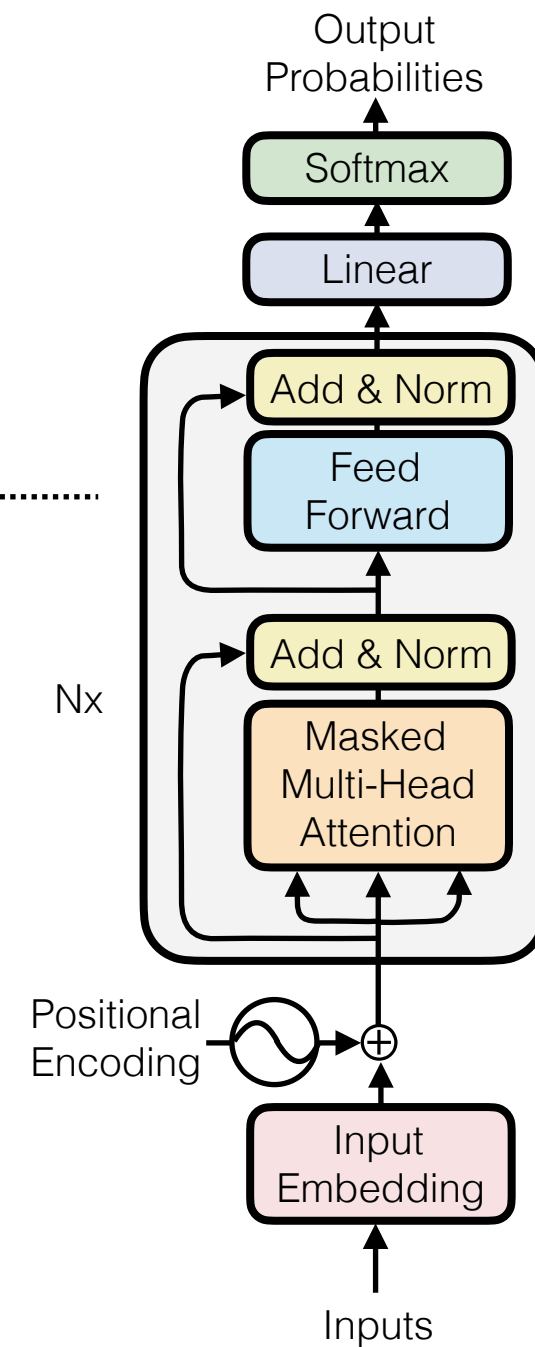


Decoder Only Model
(e.g. GPT, LLaMa)



Basic idea

- Stack “transformer layers”
- 5 key concepts in the layer design and how we embed inputs



Core Transformer Concepts

- Positional encodings
- Scaled dot product self-attention
- Multi-headed attention
- Residual + layer normalization
- Feed-forward layer

(Review)

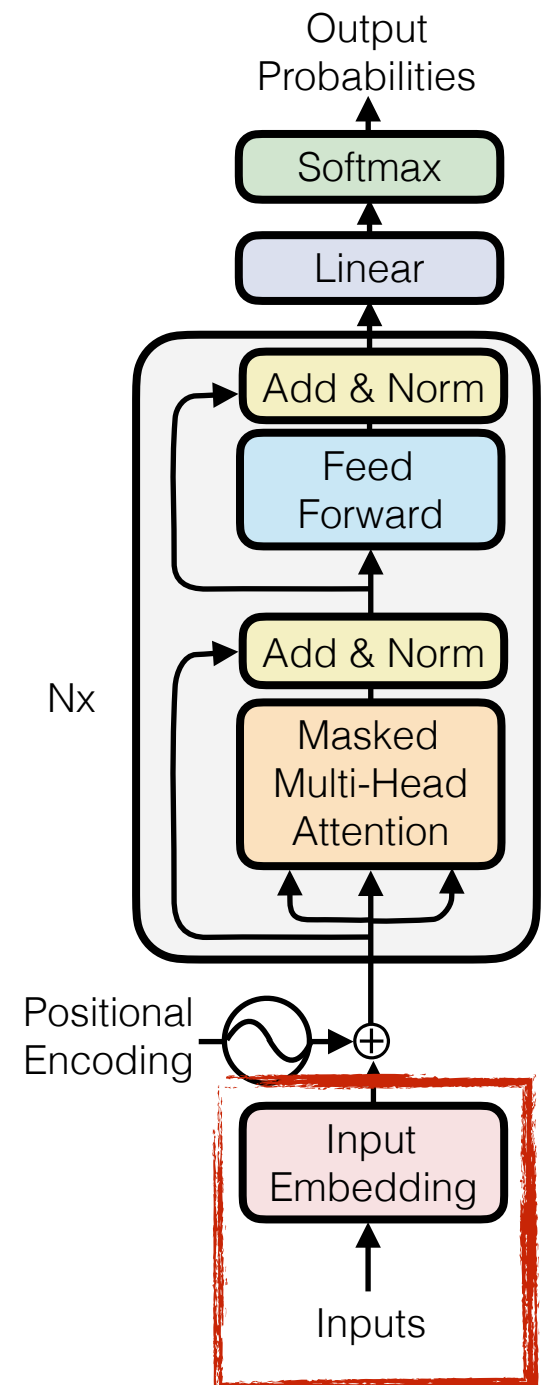
Inputs and Embeddings

- **Inputs:** Generally split using subwords

the books were improved

the book _s were **improv _ed**

- **Input Embedding:** Looked up, like in previously discussed models



Positional Encoding

- The transformer model is purely attentional
 - Permutation equivariant: $f(\pi \circ (x_1, \dots, x_T)) = \pi \circ f(x_1, \dots, x_T)$
- We need a way to identify the position of each token

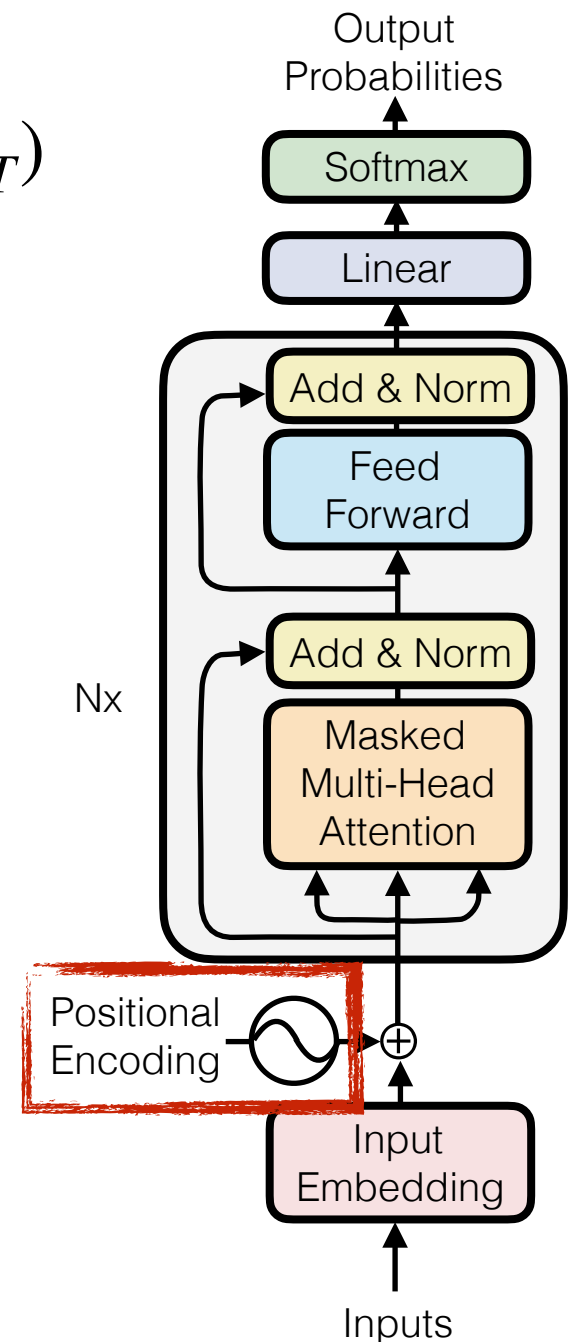
A big dog and a very big cat

A big cat and a very big dog

- Positional encodings add an embedding based on the word position

$W_{\text{big}} + W_{\text{pos}2}$

$W_{\text{big}} + W_{\text{pos}7}$

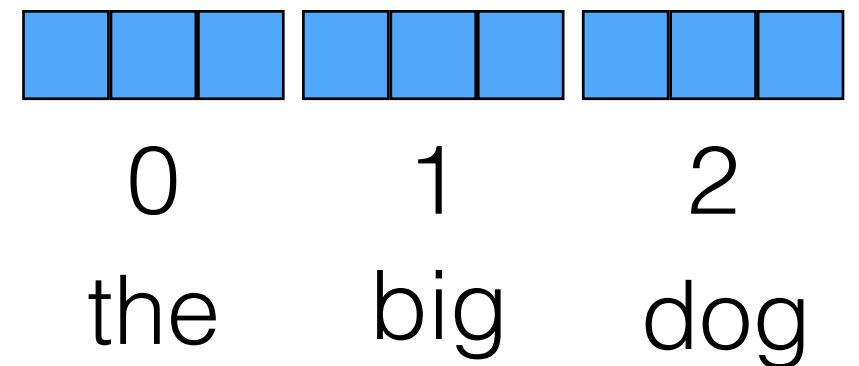


Example: Learned Positional Encoding (Shaw+ 2018)

- Just create a learnable embedding

- $W_{position} \in \mathbb{R}^{T_{max} \times d}$

- Each position $t \in \{1, \dots, T_{max}\}$
has a learned vector representation.

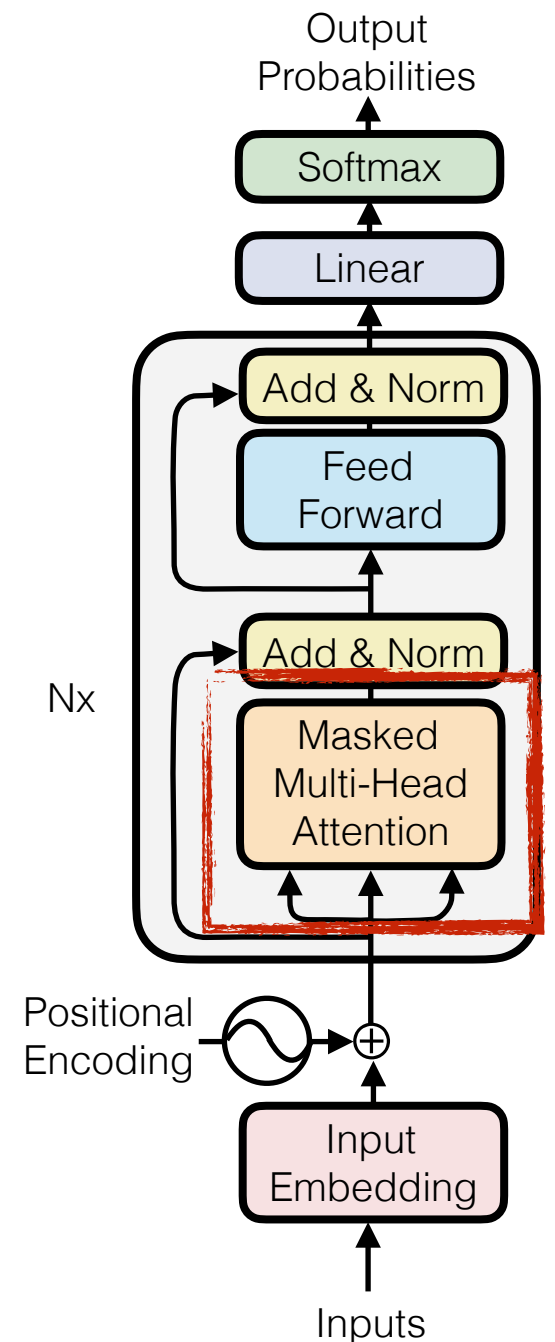


- **Advantages:** flexibility
- **Disadvantages:** cannot extrapolate to longer sequences

S: other positional encoding techniques

Core Transformer Concepts

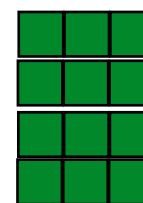
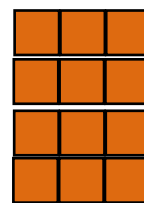
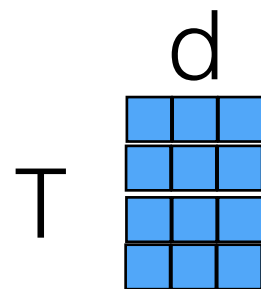
- Positional encodings
- **Scaled dot product self-attention**
- Multi-headed attention
- Residual + layer normalization
- Feed-forward layer



Scaled dot product attention

- As we saw on the previous slide: $a(\mathbf{q}, \mathbf{k}) = \frac{\mathbf{q}^\top \mathbf{k}}{\sqrt{|\mathbf{k}|}}$
- Full version, efficient matrix version:

$$Q \in \mathbb{R}^{T \times d} \quad K \in \mathbb{R}^{T \times d} \quad V \in \mathbb{R}^{T \times d}$$



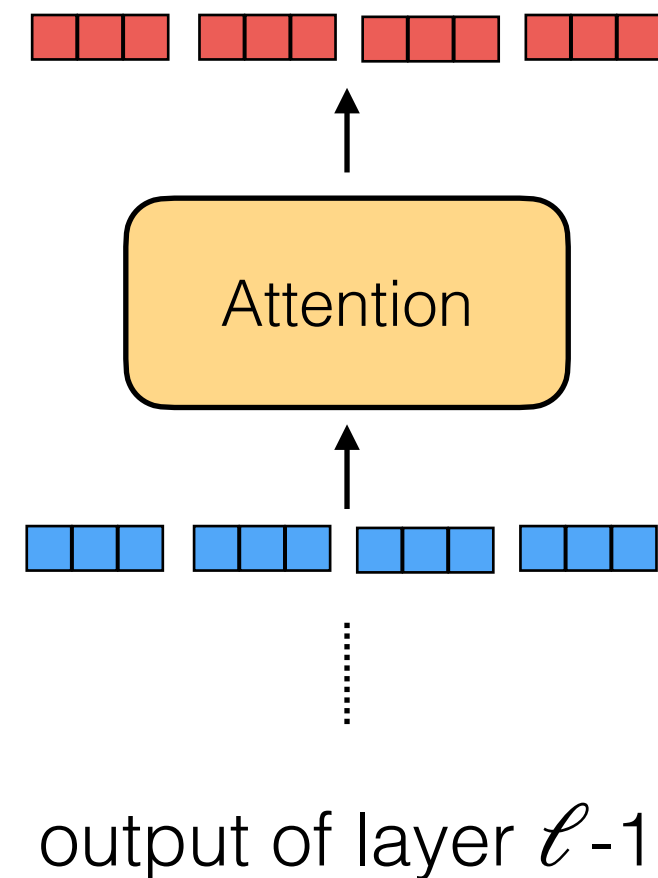
$$\text{Attention}(Q, K, V) = \text{softmax} \left(\frac{QK^\top}{\sqrt{d_k}} \right) V$$

S: dimensions of (i) softmax output, (ii) attention output

Scaled dot product self-attention

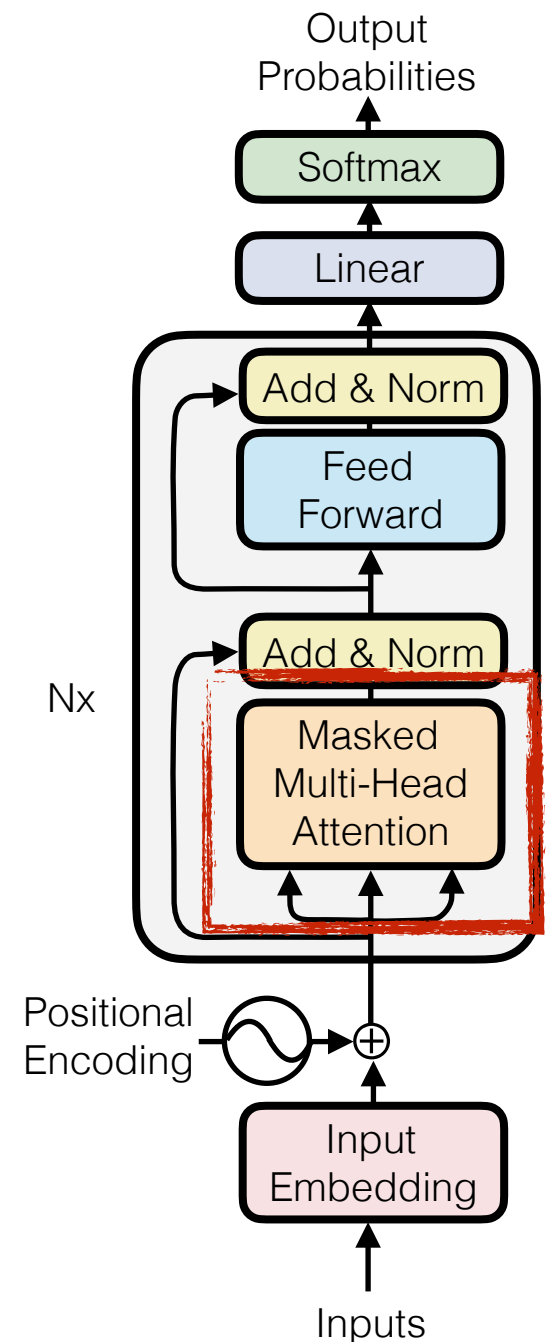
- Apply attention to the output of the previous layer:
 - $Q = H^{(\ell-1)}W_Q$
 - $K = H^{(\ell-1)}W_K$
 - $V = H^{(\ell-1)}W_V$
- Where $H^{(\ell-1)} \in \mathbb{R}^{T \times d}$ is the output of the previous transformer layer
- W_Q, W_K, W_V are learned weights

$$\text{Attention}(Q, K, V) \rightarrow \tilde{H}^\ell$$



Core Transformer Concepts

- Positional encodings
- Scaled dot product self-attention
- **Multi-headed attention**
- Residual + layer normalization
- Feed-forward layer



Intuition for Multi-heads

- **Intuition:** Information from different parts of the sentence can be useful to disambiguate in different ways

I **run** a small business

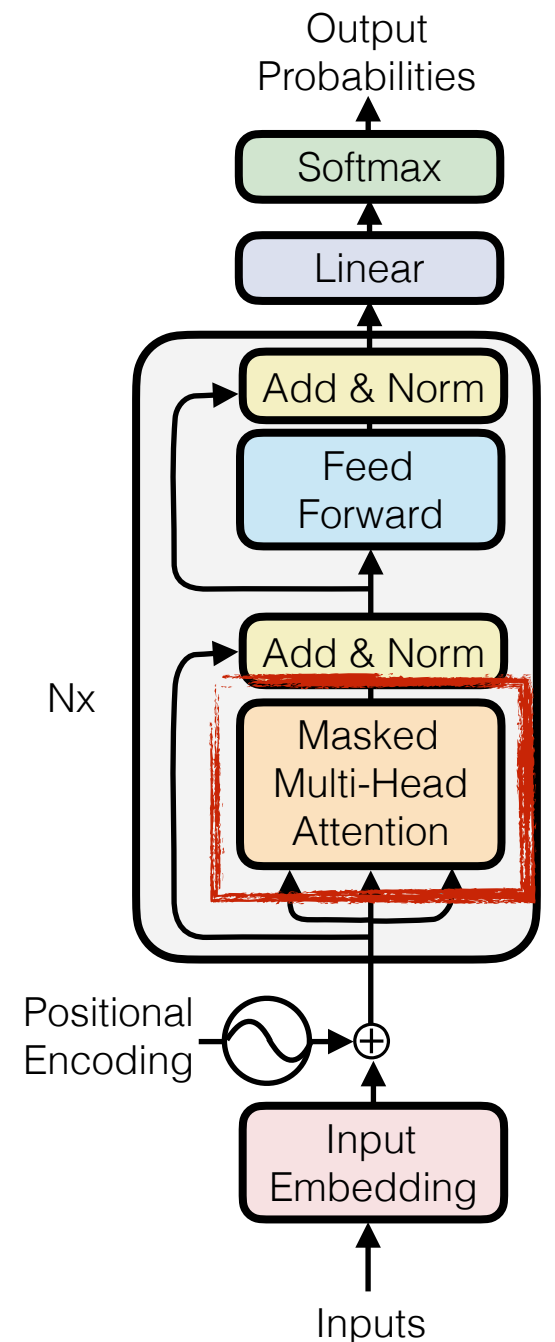
I **run** a mile in 10 minutes

The robber made a **run** for it

The stocking had a **run**

syntax
(nearby context)

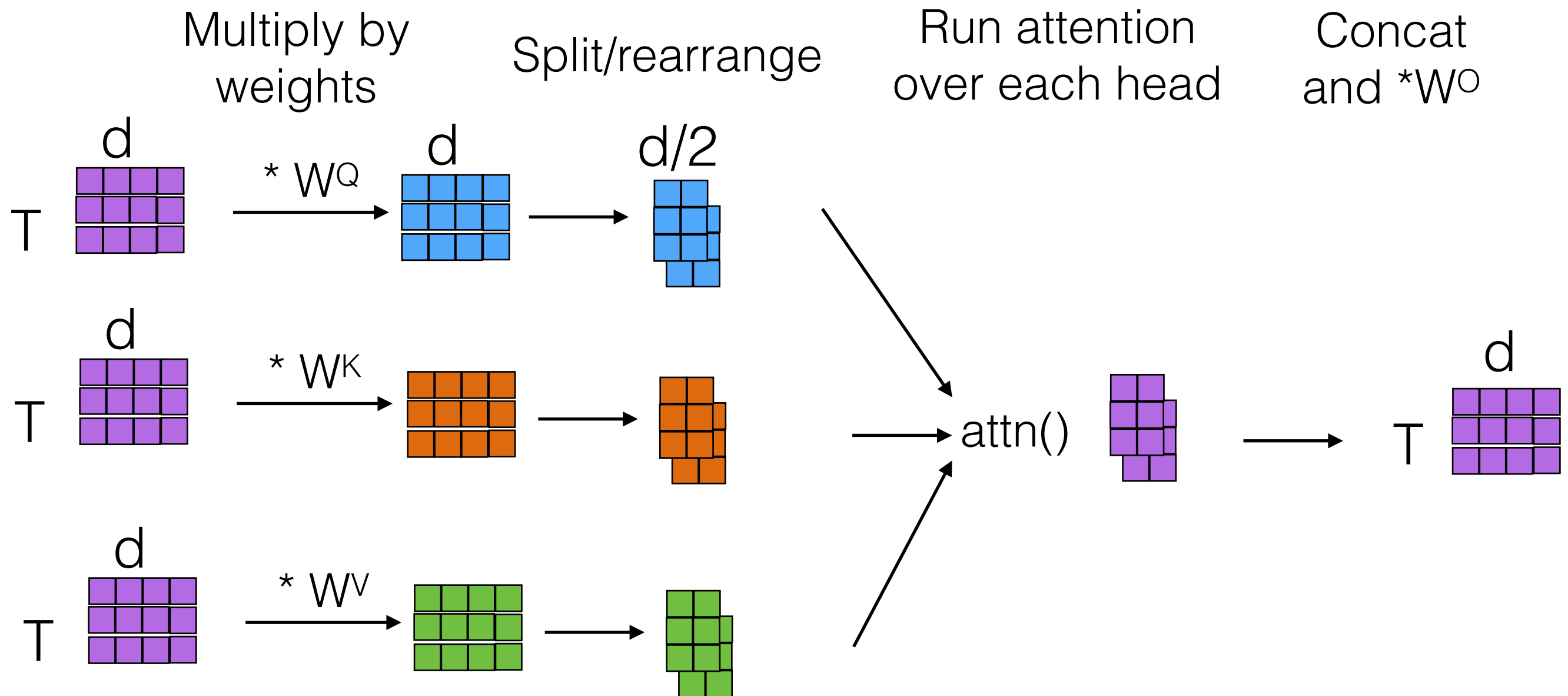
semantics
(farther context)



Multi-head Attention Concept

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

where $\text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$



Typically $d_k = d_v = d/\text{numheads}$

Code Example

```
def forward(self, x):
    B, T, C = x.size() # batch size, sequence length, embedding dimensionality (n_embd)

    # calculate query, key, values for all heads in batch and move head forward to be the batch dim
    q, k, v = self.c_attn(x).split(self.n_embd, dim=2)
    k = k.view(B, T, self.n_head, C // self.n_head).transpose(1, 2) # (B, nh, T, hs)
    q = q.view(B, T, self.n_head, C // self.n_head).transpose(1, 2) # (B, nh, T, hs)
    v = v.view(B, T, self.n_head, C // self.n_head).transpose(1, 2) # (B, nh, T, hs)

    # causal self-attention; Self-attend: (B, nh, T, hs) x (B, nh, hs, T) -> (B, nh, T, T)
    att = (q @ k.transpose(-2, -1)) * (1.0 / math.sqrt(k.size(-1)))
    att = att.masked_fill(self.bias[:, :, :T, :T] == 0, float('-inf'))
    att = F.softmax(att, dim=-1)
    att = self.attn_dropout(att)
    y = att @ v # (B, nh, T, T) x (B, nh, T, hs) -> (B, nh, T, hs)
    y = y.transpose(1, 2).contiguous().view(B, T, C) # re-assemble all head outputs side by side

    # output projection
    y = self.resid_dropout(self.c_proj(y))
    return y
```

What Happens w/ Multi-heads?

- Example from Vaswani et al.

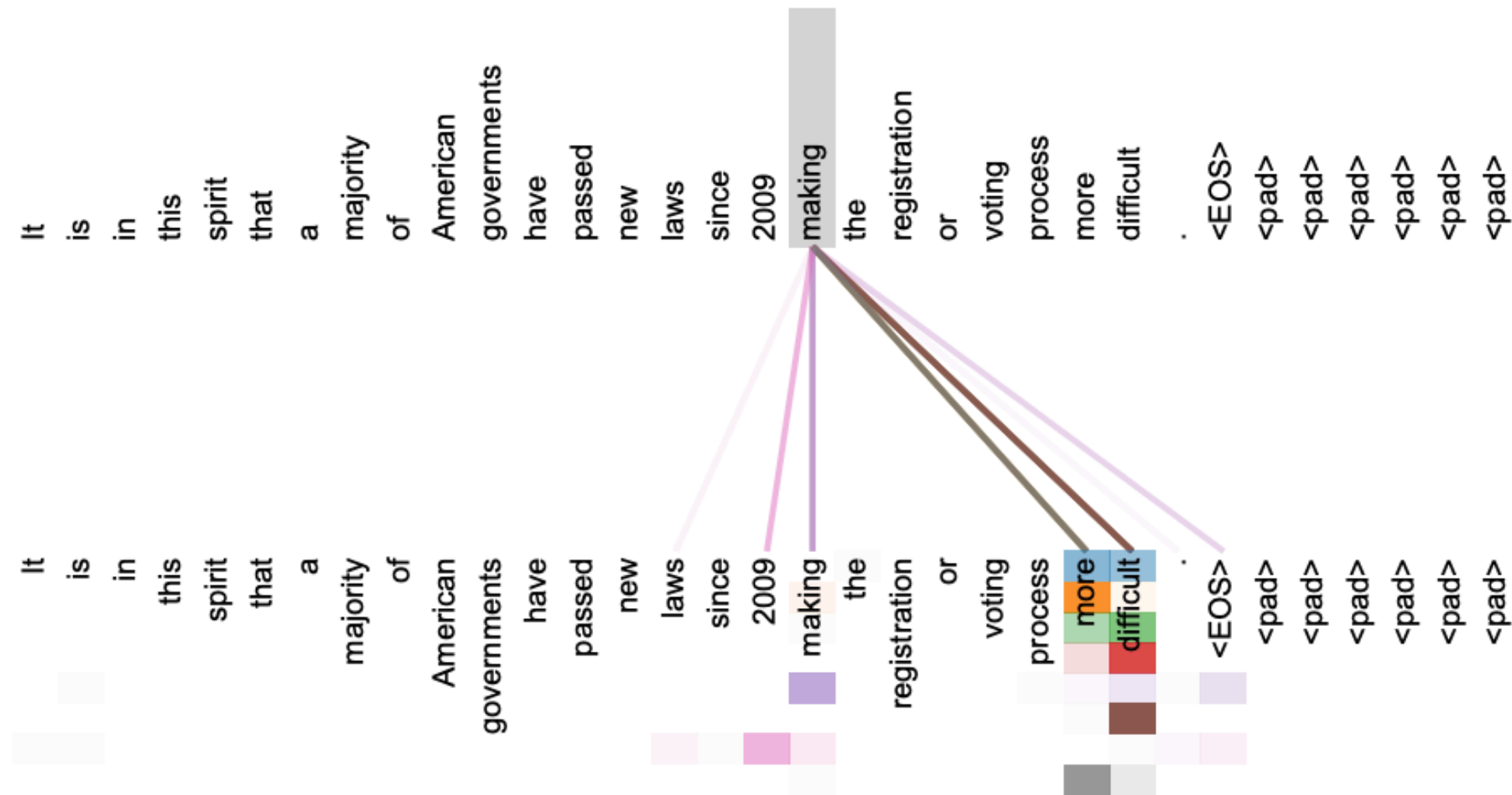


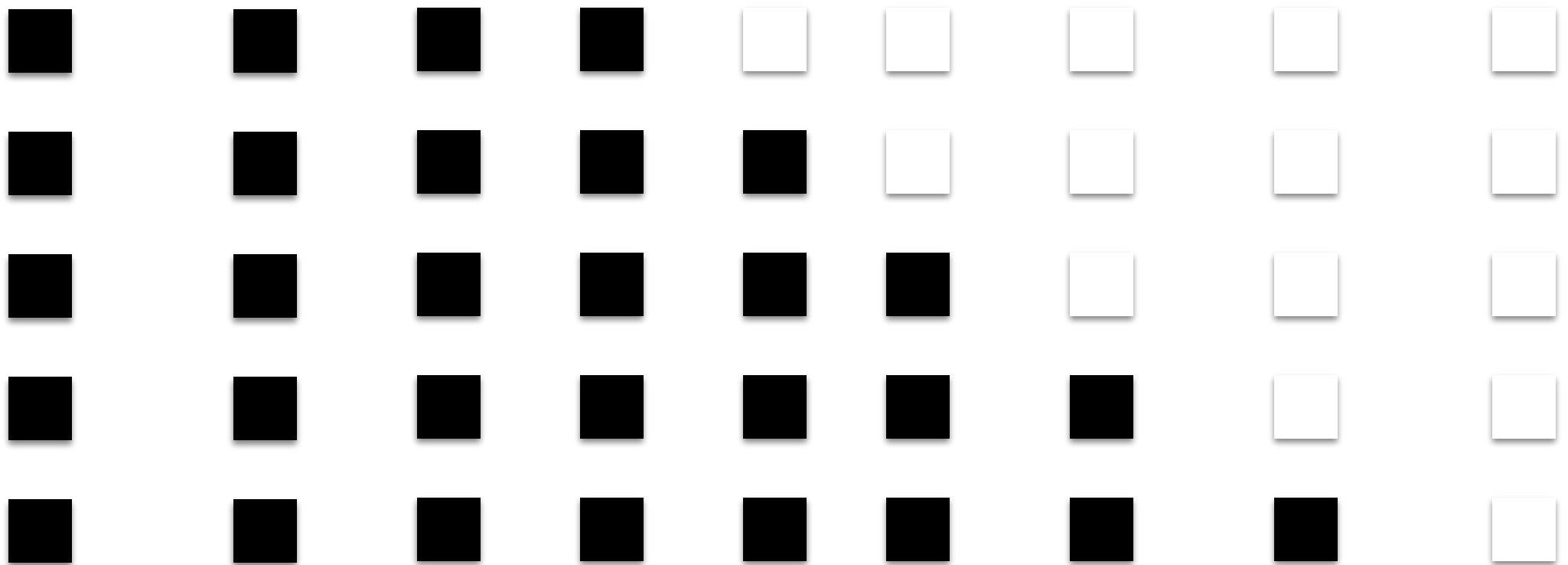
Figure 3: An example of the attention mechanism following long-distance dependencies in the encoder self-attention in layer 5 of 6. Many of the attention heads attend to a distant dependency of the verb 'making', completing the phrase 'making...more difficult'. Attentions here shown only for the word 'making'. Different colors represent different heads. Best viewed in color.

- See also BertVis: <https://github.com/jessevig/bertviz>

Masking for Language Model Training

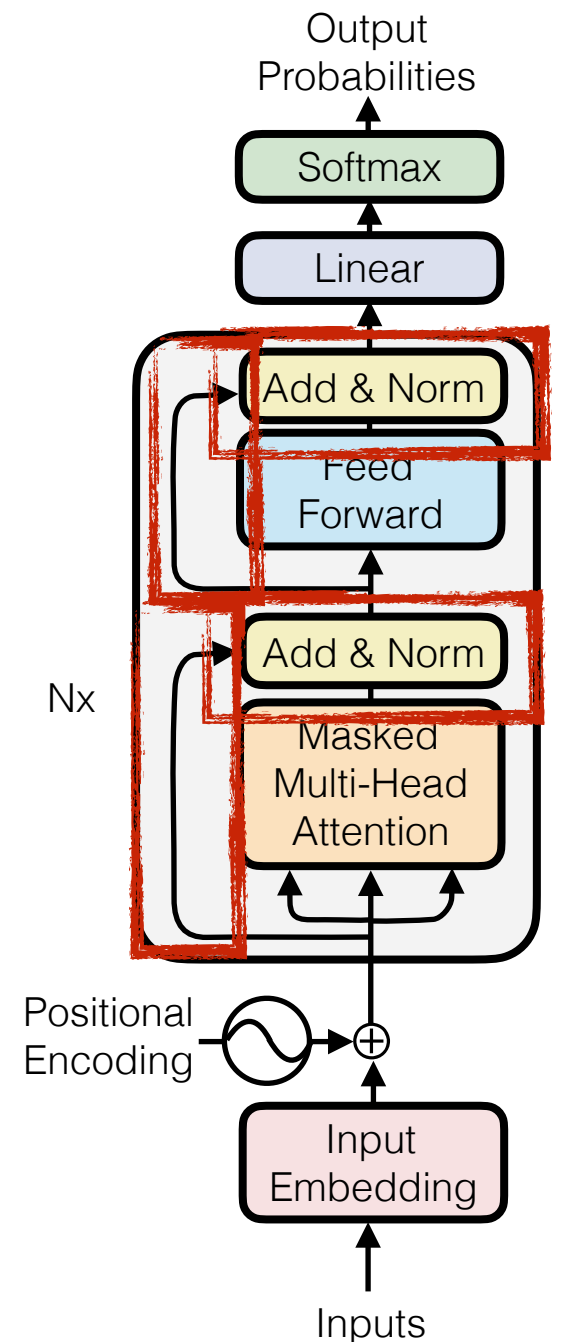
- Mask the attention from future timesteps
 - Prevents the model from cheating when predicting the next token

kono eiga ga kirai | I hate this movie </s>



Core Transformer Concepts

- Positional encodings
- Scaled dot product self-attention
- Multi-headed attention
- **Residual + layer normalization**
- Feed-forward layer

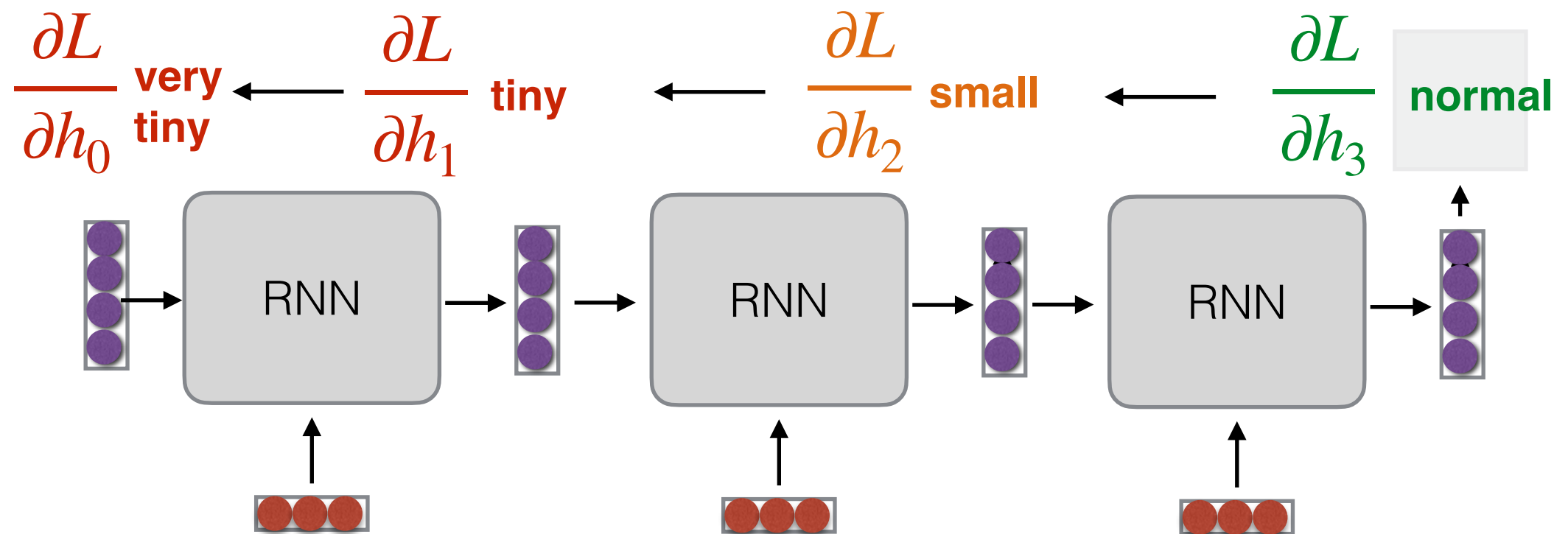


Layer Normalization and Residual Connections

Reminder:

Gradients and Training Instability

- RNNs: backpropagation can make gradients vanish or explode



- The same issue occurs in multi-layer transformers!

Layer Normalization

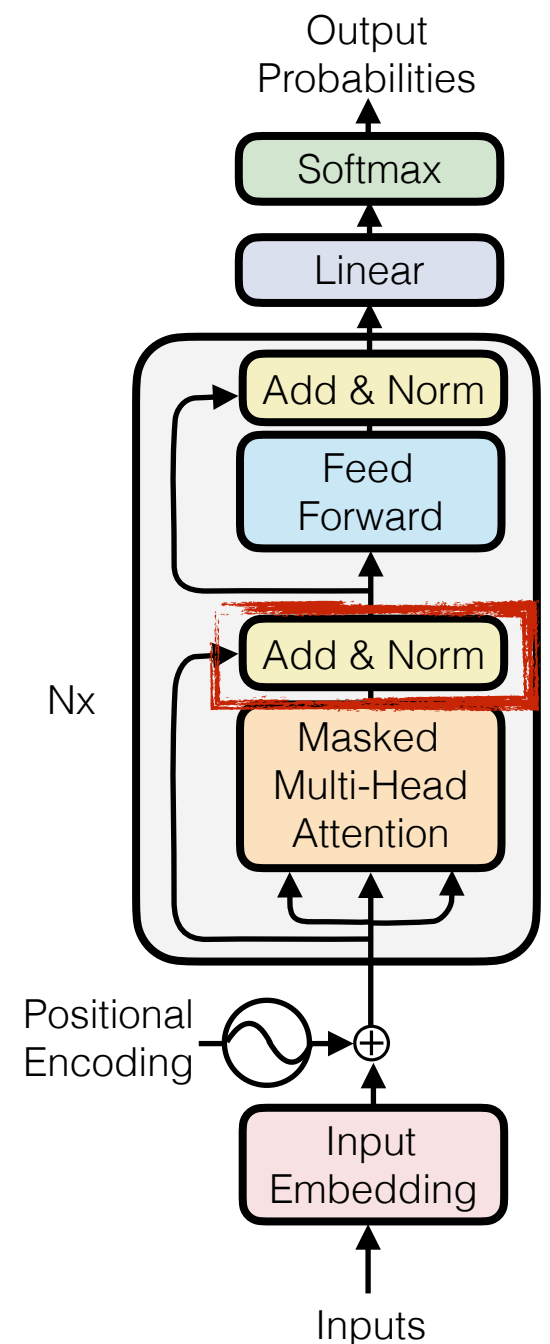
(Ba et al. 2016)

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

$$\text{LayerNorm}(\mathbf{x}; \mathbf{g}, \mathbf{b}) = \frac{\mathbf{g}}{\sigma(\mathbf{x})} \odot (\mathbf{x} - \mu(\mathbf{x})) + \mathbf{b}$$

gain bias
vector stddev vector mean

$$\mu(\mathbf{x}) = \frac{1}{n} \sum_{i=1}^n x_i \quad \sigma(\mathbf{x}) = \sqrt{\frac{1}{n} \sum_{i=1}^n (x_i - \mu)^2}$$

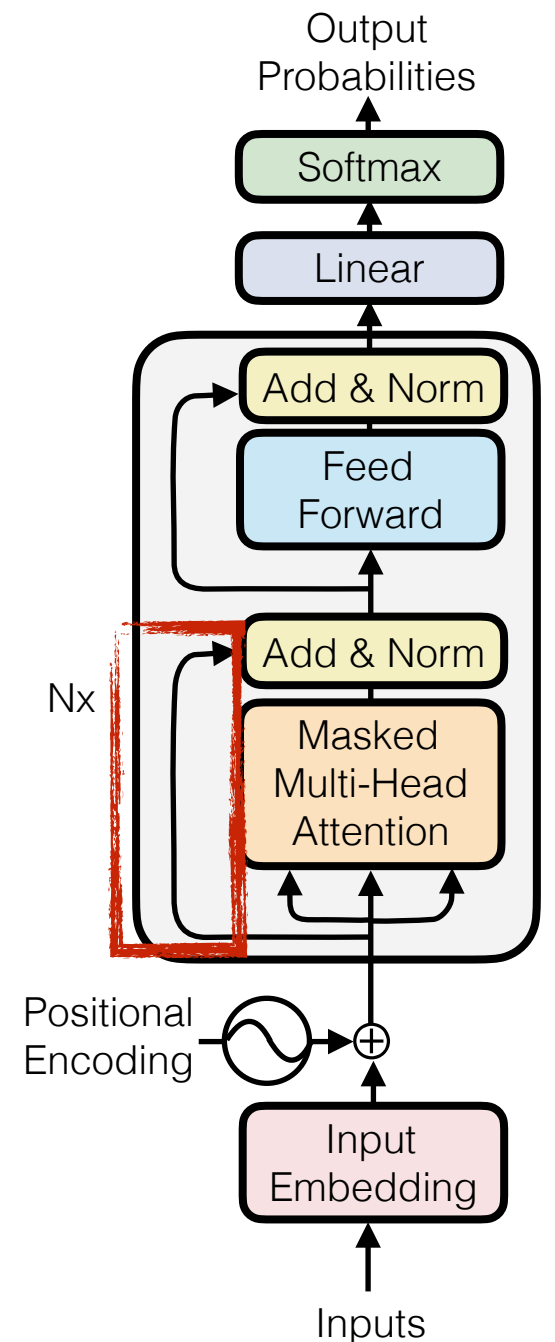


Residual Connections

- Add an additive connection between the input and output

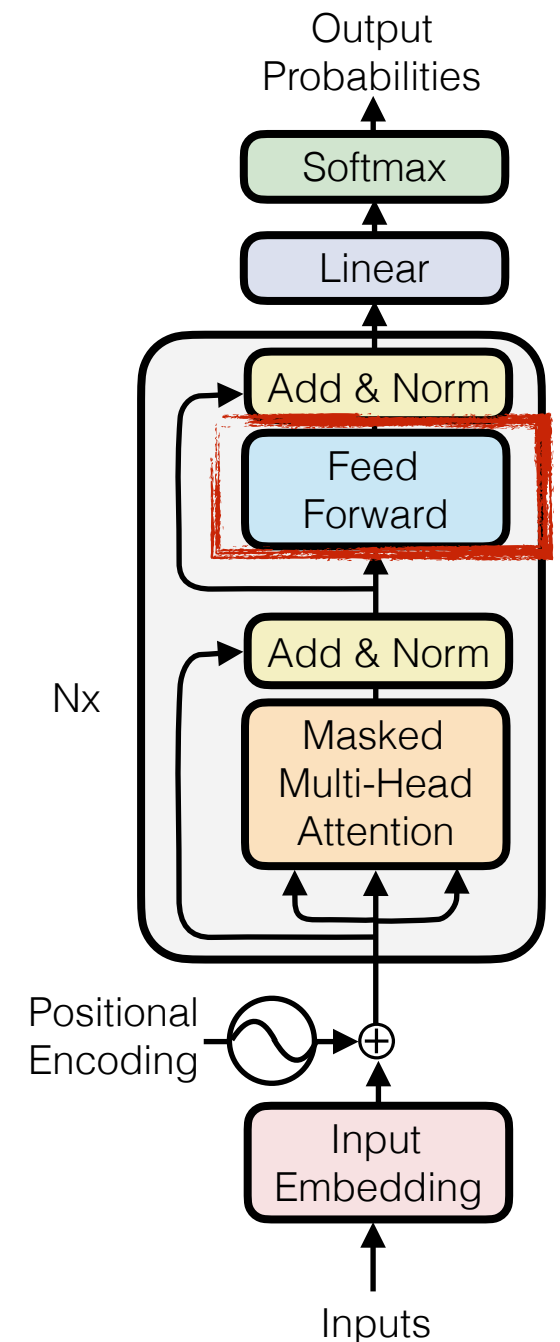
$$\text{Residual}(\mathbf{x}, f) = f(\mathbf{x}) + \mathbf{x}$$

- Prevents vanishing gradients and allows f to learn the *difference* from the input



Core Transformer Concepts

- Positional encodings
- Scaled dot product self-attention
- Multi-headed attention
- Residual + layer normalization
- **Feed-forward layer**

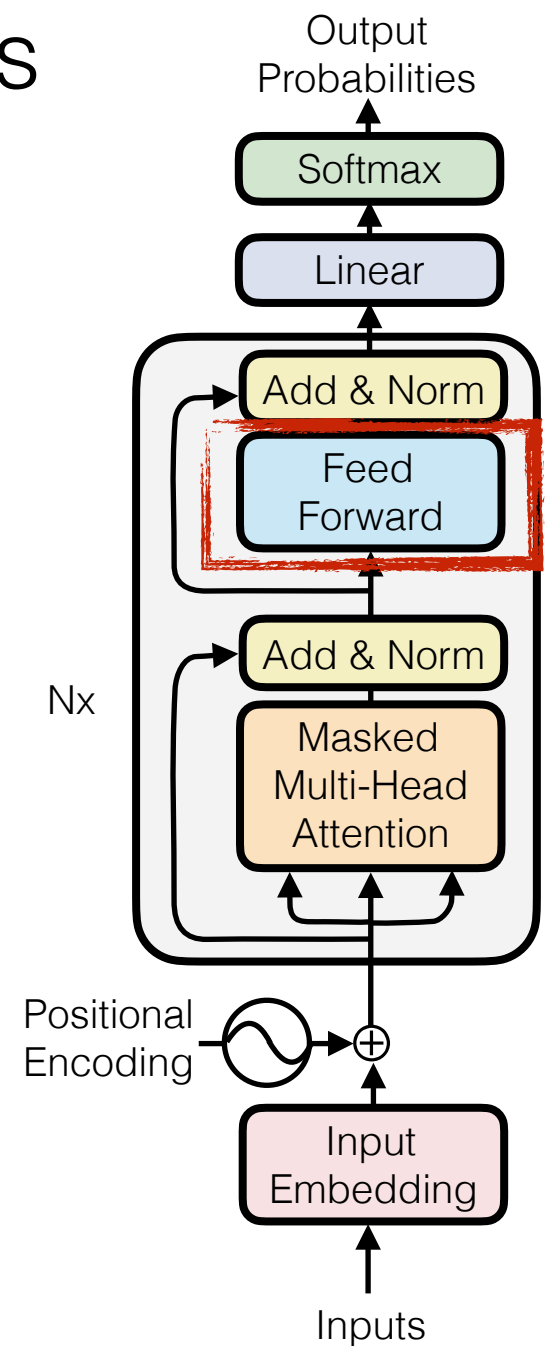
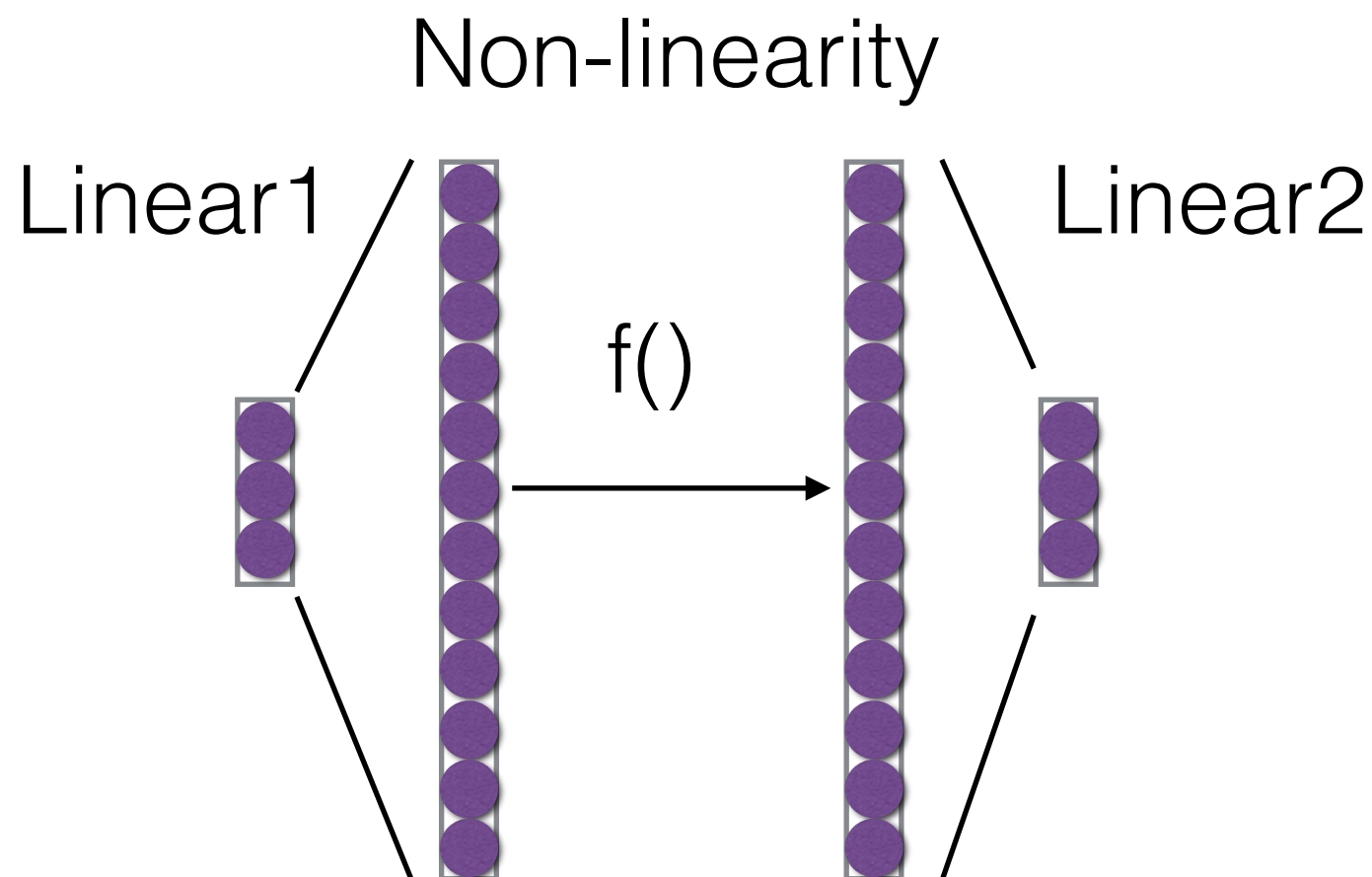


Feed Forward Layers

Feed Forward Layers

- Extract features from the attended outputs

$$\text{FFN}(x; W_1, \mathbf{b}_1, W_2, \mathbf{b}_2) = f(\mathbf{x}W_1 + \mathbf{b}_1)W_2 + \mathbf{b}_2$$



In code

[https://github.com/cmu-l3/anlp-fall2025-code/blob/main/
05_transformers/transformer.ipynb](https://github.com/cmu-l3/anlp-fall2025-code/blob/main/05_transformers/transformer.ipynb)

In code

```
import torch.nn as nn

class Block(nn.Module):
    def __init__(self, d_model, nhead, dim_ff=64, max_len=128):
        super(Block, self).__init__()
        self.attn = nn.MultiheadAttention(d_model, nhead, dropout=0.0, batch_first=True)
        self.ff1 = nn.Linear(d_model, dim_ff)
        self.ff2 = nn.Linear(dim_ff, d_model)
        self.ln1 = nn.LayerNorm(d_model)
        self.ln2 = nn.LayerNorm(d_model)
        self.act = nn.ReLU()
        self.register_buffer('mask', torch.triu(torch.ones(max_len, max_len), diagonal=1).bool())

    def forward(self, x):
        B, T, D = x.size()

        # Self-attention block
        residual = x
        x = self.ln1(x) # Pre-normalization
        x = self.attn(x, x, x, is_causal=True, attn_mask=self.mask[:T,:T])[0]
        x = residual + x

        # Feed-forward block
        residual = x
        x = self.ln2(x)
        x = self.ff2(self.act(self.ff1(x)))
        x = residual + x
        return x
```

In code

```
class TransformerLM(nn.Module):
    def __init__(self, vocab_size, d_model, nhead, num_layers, dim_ff, max_len=128):
        super(TransformerLM, self).__init__()
        self.embedding = nn.Embedding(vocab_size, d_model)
        self.pos_encoder = nn.Embedding(max_len, d_model)
        self.blocks = nn.ModuleList([
            Block(d_model, nhead, dim_ff) for _ in range(num_layers)
        ])
        self.fc = nn.Linear(d_model, vocab_size)
        self.d_model = d_model

    def forward(self, x):
        pos = torch.arange(x.size(1), device=x.device).unsqueeze(0)
        x = self.embedding(x) + self.pos_encoder(pos)
        for block in self.blocks:
            x = block(x)
        logits = self.fc(x)
        return logits
```

Today's lecture

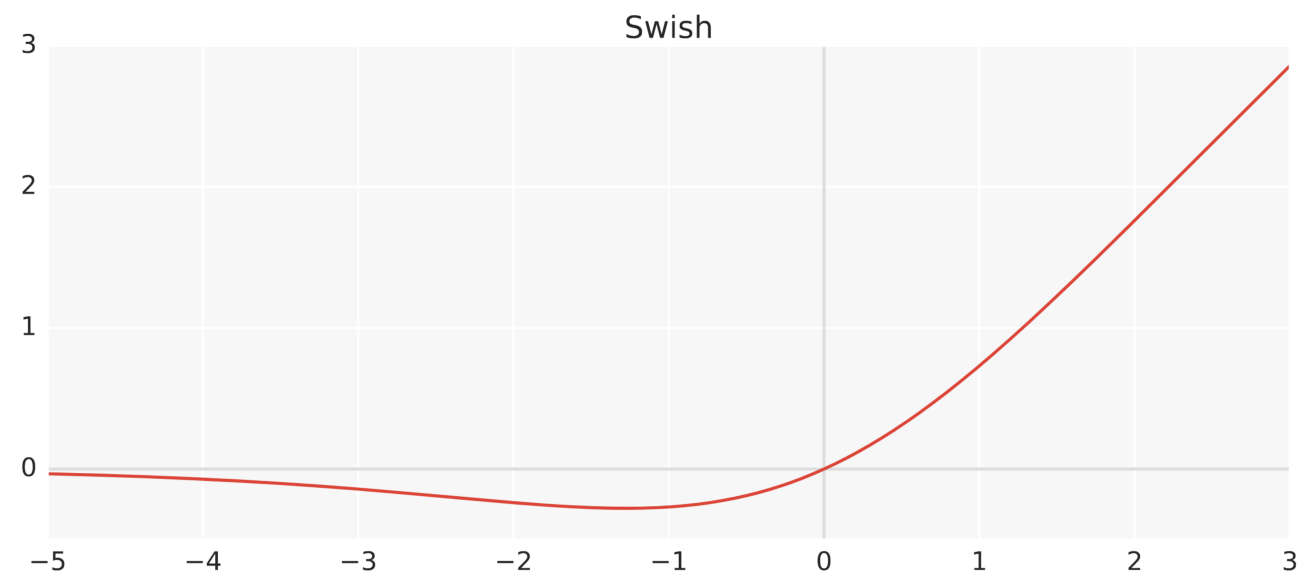
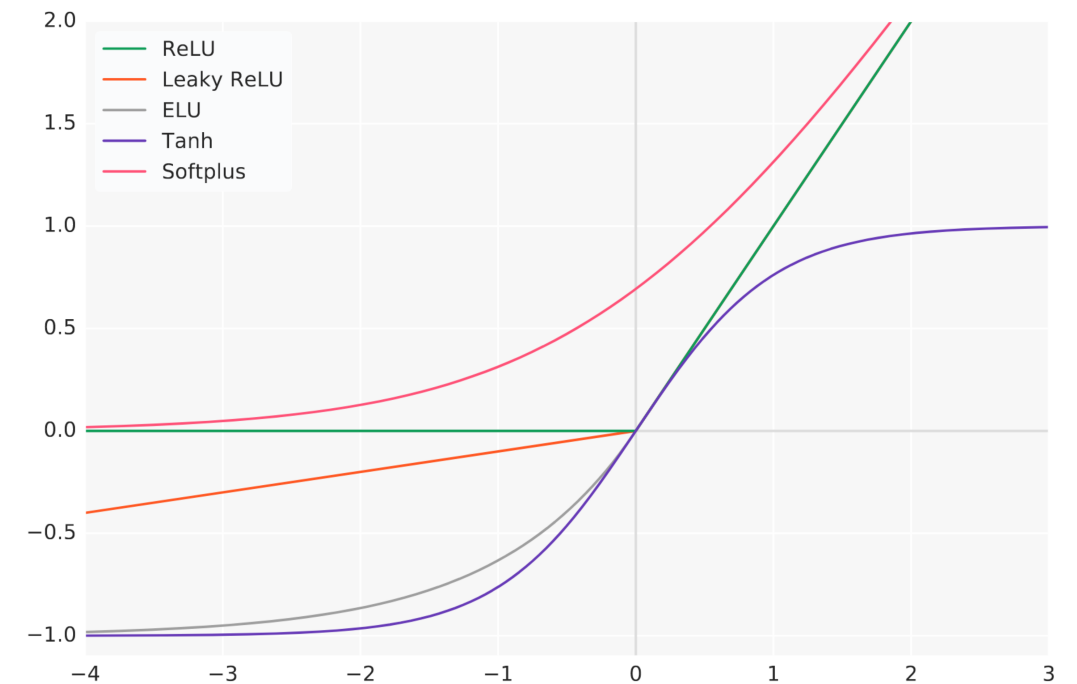
- Roadmap:
 - Attention
 - Transformer architecture
 - **Improved transformer architecture**

Transformer improvements

SiLU/Swish Activation

[Hendricks & Gimpel 2016, Ramachandran et al 2017]

- Sigmoid: $\sigma(x) = \frac{1}{1 + \exp(-x)}$
- ReLU: $f(x) = \max(0, x)$
- SiLU/Swish: $f(x) = x\sigma(x)$
 - Unbounded above
 - Bounded below
 - Non-monotonic
 - Smooth



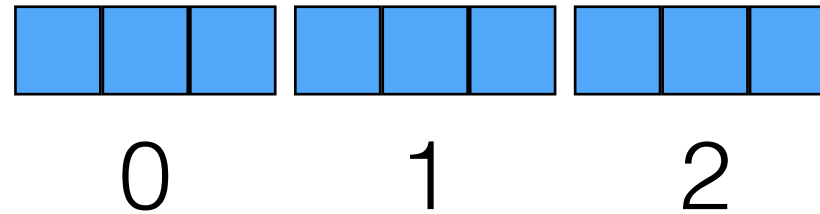
SwiGLU Feed-Forward Layer [Shazeer 2020]

- $\text{FFN}_{\text{relu}} = \max(0, xW_1)W_2$
- $\text{FFN}_{\text{swish}} = \text{swish}(xW_1)W_2$
- $\text{FFN}_{\text{swiglu}} = (\text{swish}(xW_1) \cdot xW_3)W_2$
“gate”

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

Relative Positional Encodings (Shaw+ 2018)

- **Absolute** positional encodings



- **Relative** positional encodings *explicitly* encode relative position

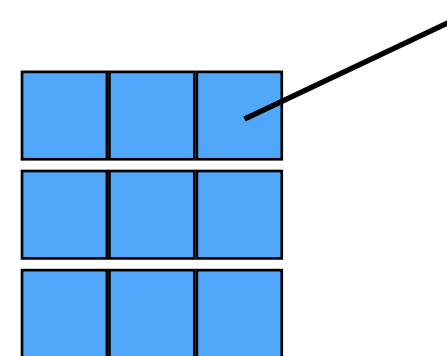
- Example: inside attention layer:

$$e_{ij} = \frac{q_i^\top k_j}{\sqrt{d_h}} + \frac{q_i^\top r_{i-j}}{\sqrt{d_h}}$$

- $\alpha_{ij} = \text{softmax}_j(e_{ij})$

- $R \in \mathbb{R}^{(2K+1) \times d}$, rows r_Δ with $\Delta \in [-K, K]$

“Token 0 and token 2
are $2 - 0 = 2$ tokens apart”



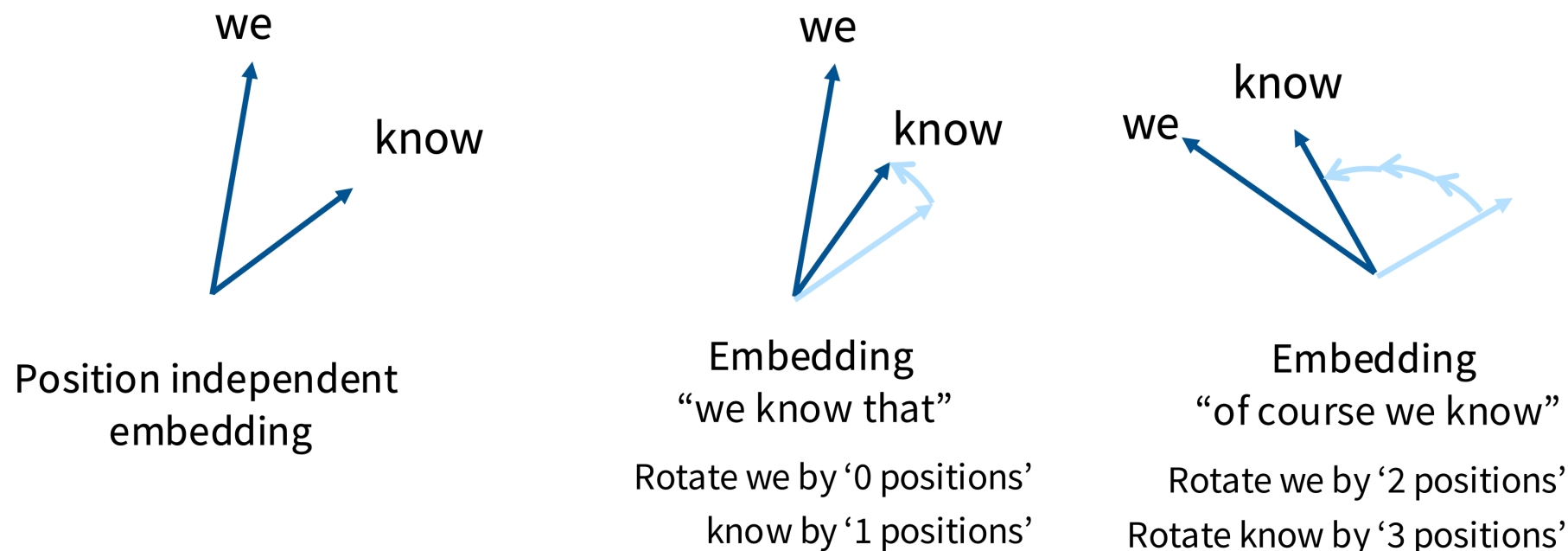
Rotary Positional Encodings (RoPE)

(Su+ 2021)

- Goal: we want the dot product of embeddings to result in a function of relative position

$$\langle f(q, t), f(k, t') \rangle = g(q, k, t' - t)$$

- Idea: leverage nice properties of rotations



Rotary Positional Encodings (RoPE)

- Recall a rotation matrix, e.g. in 2D: $R_\theta = \begin{pmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{pmatrix}$
- We have:
$$(R_{\theta_1} q)^\top (R_{\theta_2} k) = q^\top R_{\theta_1}^\top R_{\theta_2} k$$
$$= q^\top R_{\theta_2 - \theta_1} k$$
- Dot product only depends on the difference between θ_2 and θ_1 !
- RoPE key idea: encode positions t based on rotation matrices $R_{t\theta}$:
 - $R_{t\theta}^\top R_{t'\theta} = R_{(t'-t)\theta}$

Rotary Positional Encodings (RoPE)

- Example:

- $f(q, t) = R_t W_q q$ where R_t is

$$R_t = \begin{pmatrix} \cos t\theta_1 & -\sin t\theta_1 & 0 & 0 & \dots & 0 & 0 \\ \sin t\theta_1 & \cos t\theta_1 & 0 & 0 & \dots & 0 & 0 \\ 0 & 0 & \cos t\theta_2 & -\sin t\theta_2 & \dots & 0 & 0 \\ 0 & 0 & \sin t\theta_2 & \cos t\theta_2 & \dots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \dots & \cos t\theta_{d/2} & -\sin t\theta_{d/2} \\ 0 & 0 & 0 & 0 & \dots & \sin t\theta_{d/2} & \cos t\theta_{d/2} \end{pmatrix}$$

- t : Position in sequence

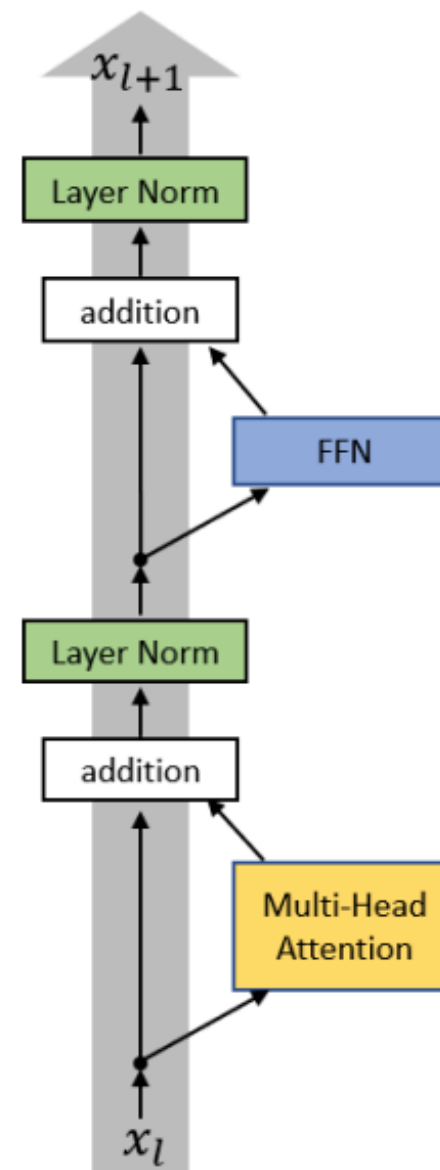
$$\theta_i = 10000^{-2i/d}$$

- Small i : high frequency
- Large i : low frequency

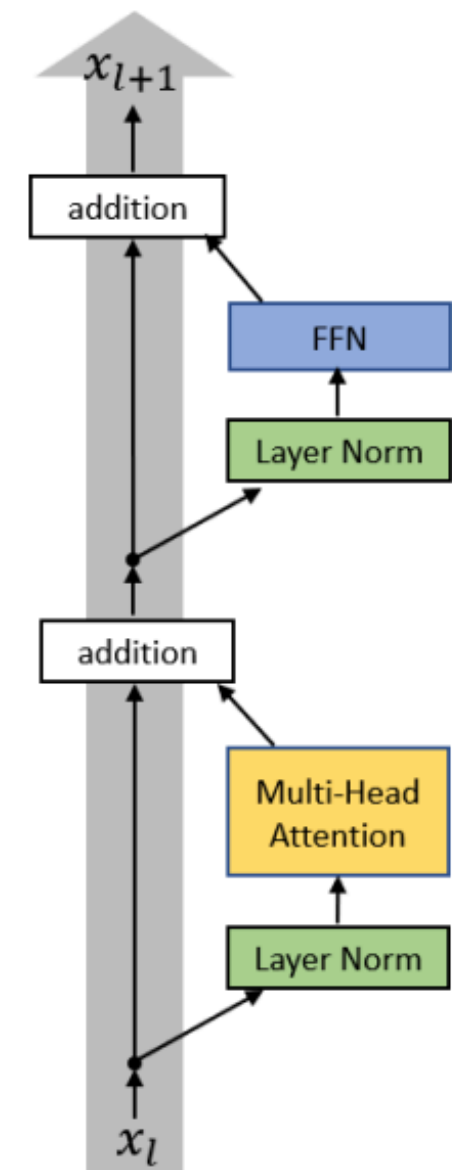
Pre- Layer Norm

(e.g. Xiong et al. 2020)

- Where should LayerNorm be applied? Before or after?
- Pre-layer-norm is better for gradient propagation



post-LayerNorm



pre-LayerNorm

RMSNorm

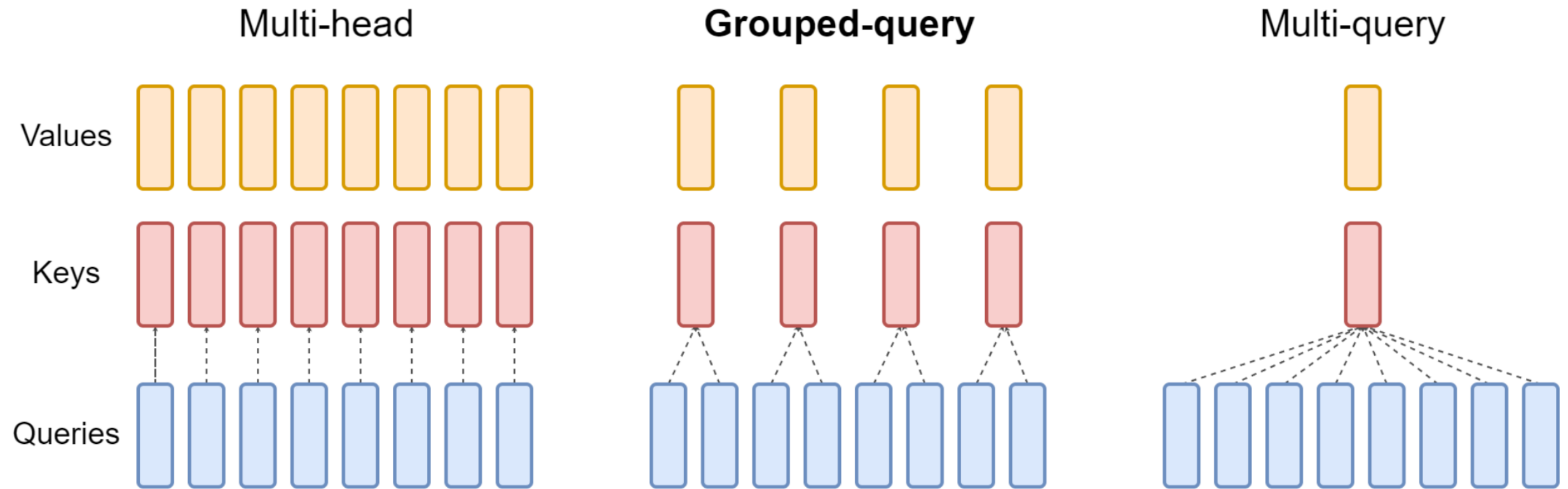
(Zhang and Sennrich 2019)

- Simplifies LayerNorm by removing the mean and bias terms

$$\text{RMS}(\mathbf{x}) = \sqrt{\frac{1}{n} \sum_{i=1}^n x_i^2}$$

$$\text{RMSNorm}(\mathbf{x}) = \frac{\mathbf{x}}{\text{RMS}(\mathbf{x})} \cdot \mathbf{g}$$

Grouped-query attention



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

In code

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

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

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

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

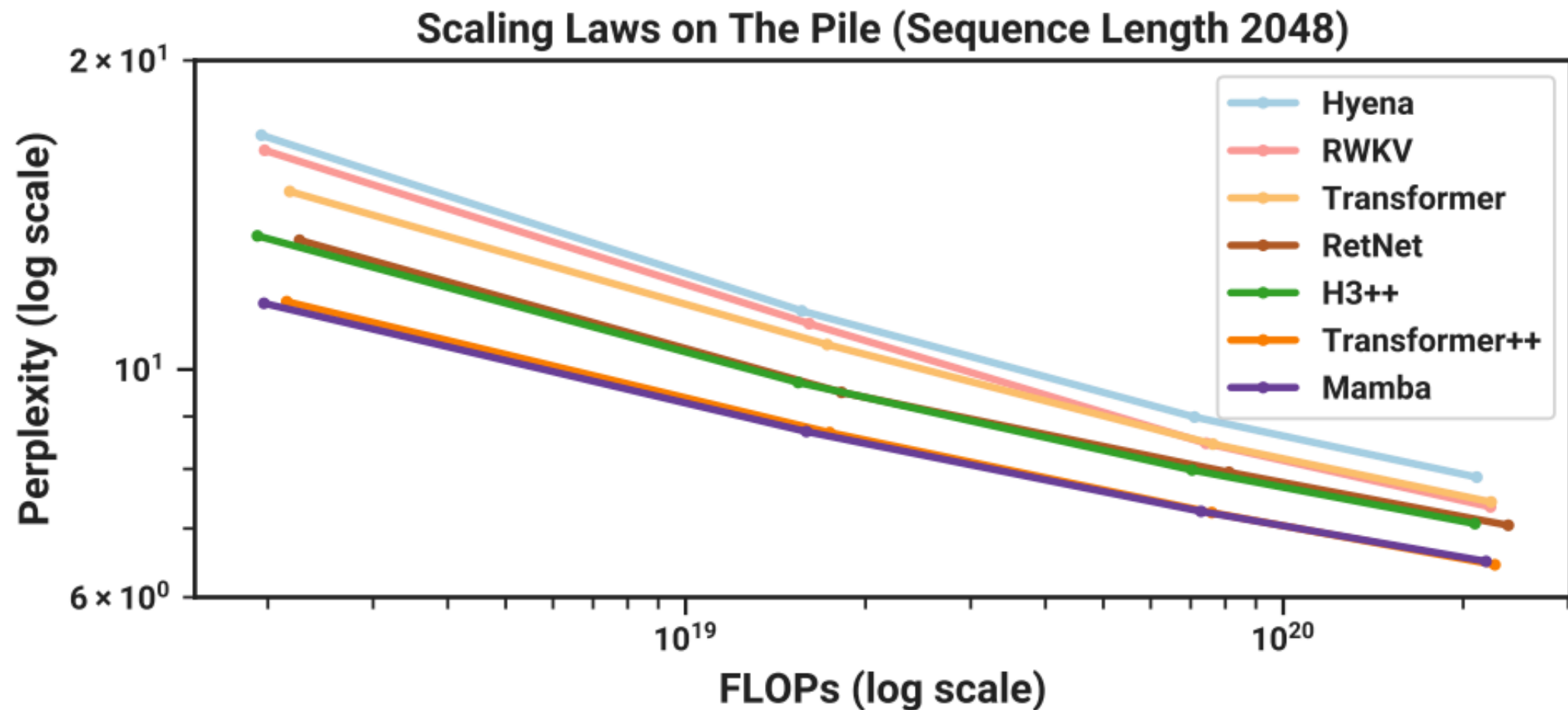
Original Transformer vs. LLama

	Vaswani et al.	LLama	Llama 2
Norm Position	Post	Pre	Pre
Norm Type	LayerNorm	RMSNorm	RMSNorm
FFN/ Activation	ReLU	SwiGLU	SwiGLU
Positional Encoding	Sinusoidal	RoPE	RoPE
Attention	Multi-head	Multi-head	Grouped- query

S: Gemma 2

How Important is It?

- “Transformer” is Vaswani et al., “Transformer++” is (basically) LLaMA2



- Stronger architecture is $\approx 10x$ more efficient!

Recap

- **Transformer**: a sequence model based on attention
- We saw:
 - Attention
 - Transformer architecture
 - Improved transformer architecture

Additional topics

- Adam optimizer
- Transformer vs. RNN

Optimizer: Adam

- Most standard optimization option in NLP and beyond
 - Each parameter has an adaptive learning rate
 - Incorporates 2 key ideas: momentum and RMSProp

Optimizer: Adam

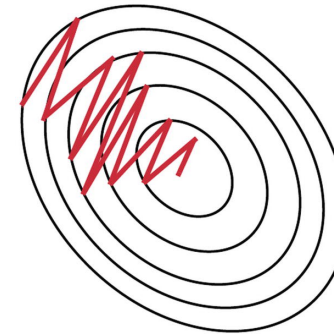
- Momentum

$$\theta_{t+1} = \theta_t - \alpha m_t$$

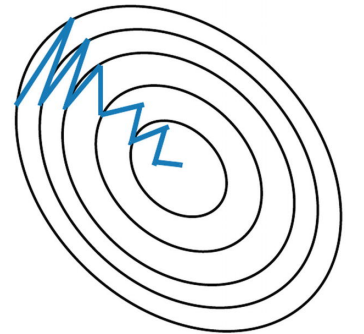
$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) \nabla_{\theta}$$

Intuition: reduces oscillations

- $m_t \in \mathbb{R}^{|\theta|}$, i.e. per-parameter adjustment
- Running estimate of $\mathbb{E}[\nabla_{\theta}]$



Stochastic Gradient
Descent **without**
Momentum



Stochastic Gradient
Descent **with**
Momentum

source

Optimizer: Adam

- RMSProp

$$\theta_{t+1} = \theta_t - \frac{\alpha_t}{\sqrt{v_t + \epsilon}} \nabla_{\theta}$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2)(\nabla_{\theta})^2$$

- v is per-parameter
- Normalizes the update magnitude
 - $(\nabla_{\theta}[i, j])^2$ large: update gets smaller
 - $(\nabla_{\theta}[i, j])^2$ small: update gets larger
- Running estimate of $\mathbb{E}[(\nabla_{\theta})^2]$

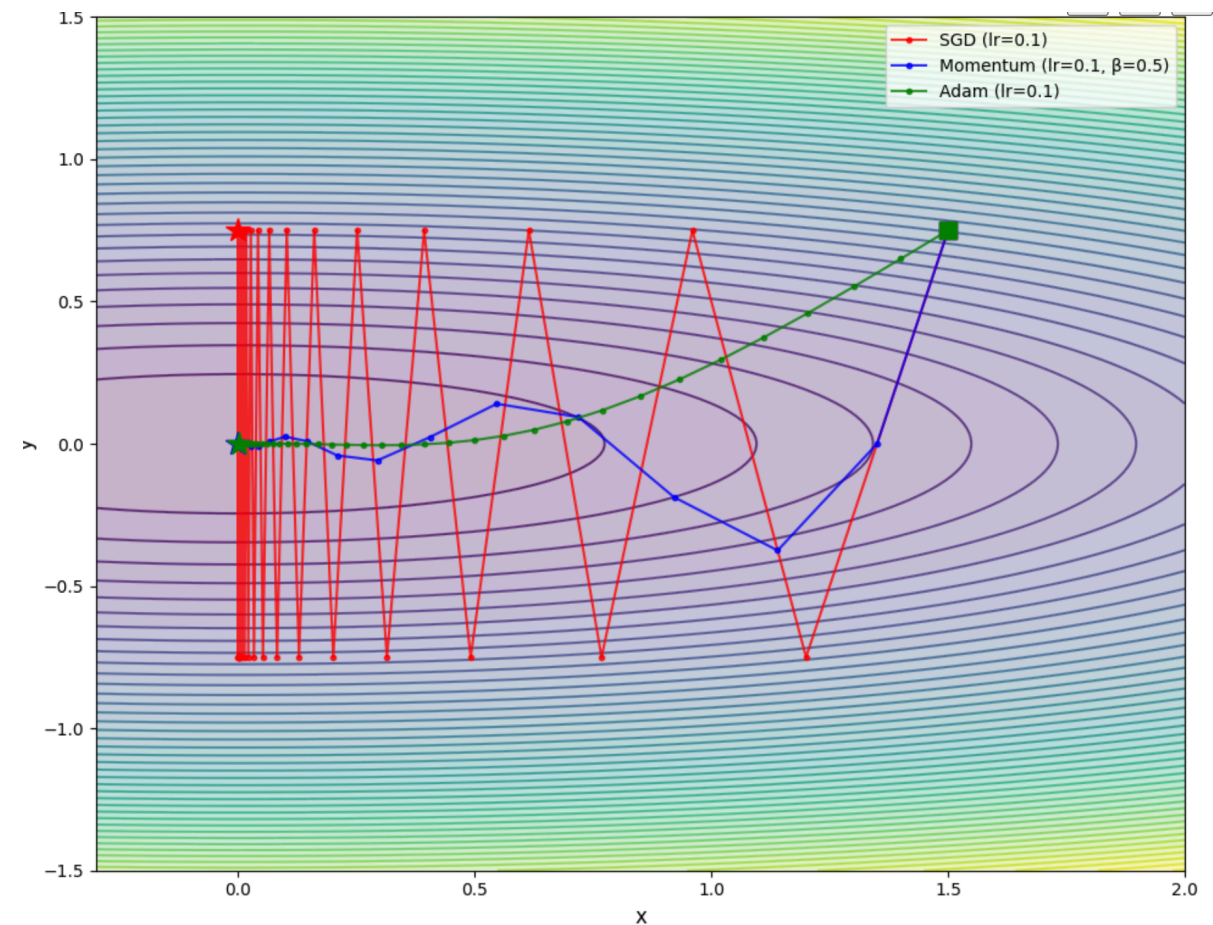
Optimizer: Adam

- Running estimate of $\mathbb{E}[\nabla_{\theta}]$
$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) \nabla_{\theta}$$
- Running estimate of $\mathbb{E}[(\nabla_{\theta})^2]$
$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) (\nabla_{\theta})^2$$
- Correction of early bias

$$\hat{m}_t = \frac{m_t}{1 - (\beta_1)^t} \quad \hat{v}_t = \frac{v_t}{1 - (\beta_2)^t}$$

- Final update

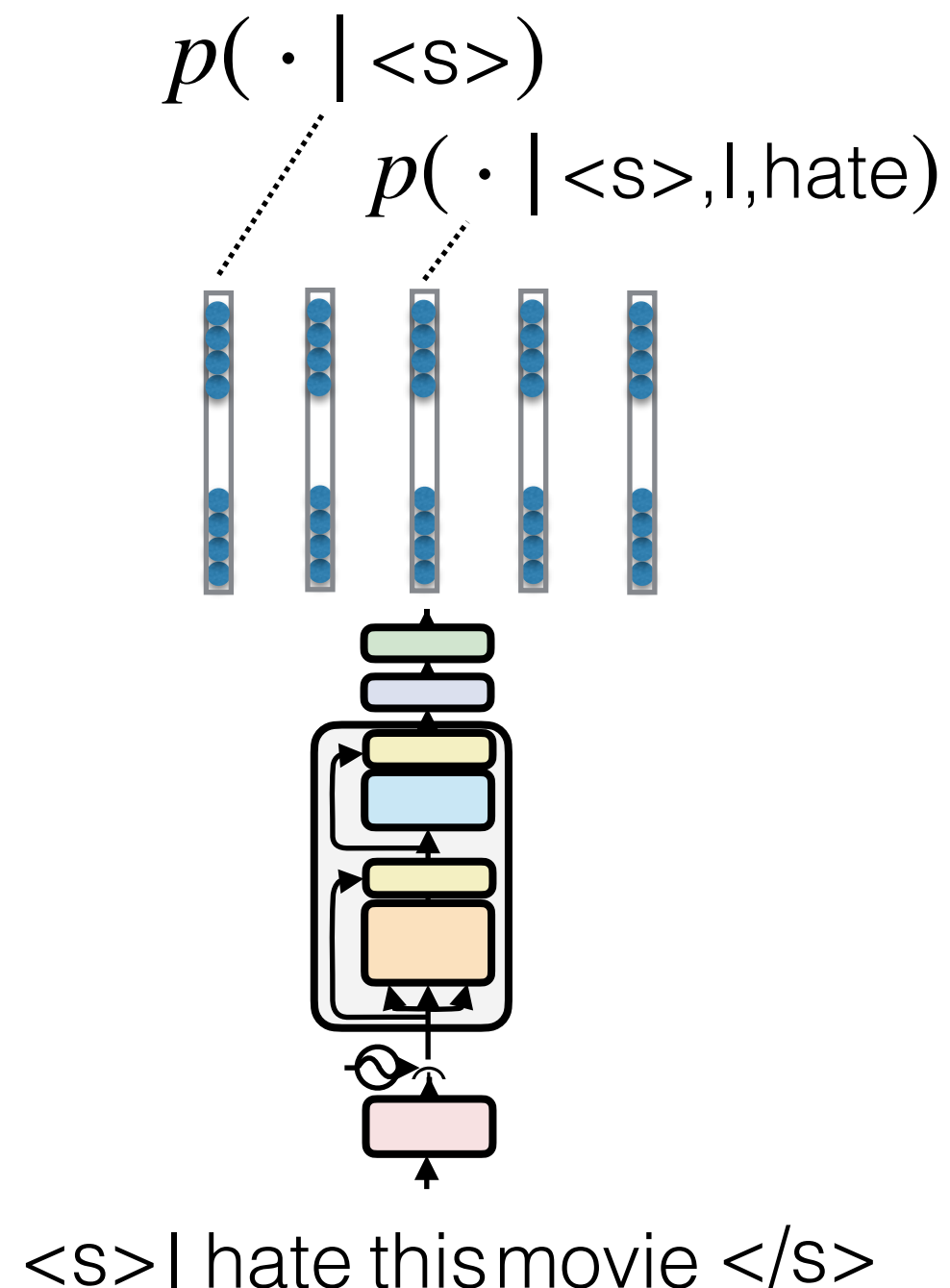
$$\theta_t = \theta_{t-1} - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t$$



Transformer vs RNN

Transformer Training

- We can compute next-token probabilities for *all* positions at once using matrix multiplications
- No sequential hidden state (as in RNNs)
- Modern hardware (e.g. GPU) is optimized for parallel operations like the matrix multiplications in self-attention
- $\therefore$ easy-to-parallelize training



RNNs vs. Transformers

- RNN: $O(Td^2)$
 - At each step $1, \dots, T$, a $O(d^2)$ operation, e.g. Wh
- Transformer attention: $O(T^2d)$
 - E.g., QK^T
 - $Q \in \mathbb{R}^{T \times d}$
 - $K \in \mathbb{R}^{T \times d} \Rightarrow O(T^2d)$

Key difference:
 T (RNNs)
 T^2 (Transformers)

RNNs vs. Transformers

- Transformers: $O(T^2d)$
 - Quadratic in sequence length T
 - Need to store a large $T \times T$ matrix in memory
 - Need to perform $O(T^2d)$ computations
 - Easy to parallelize the training
 - Long-range dependency: handled by attention

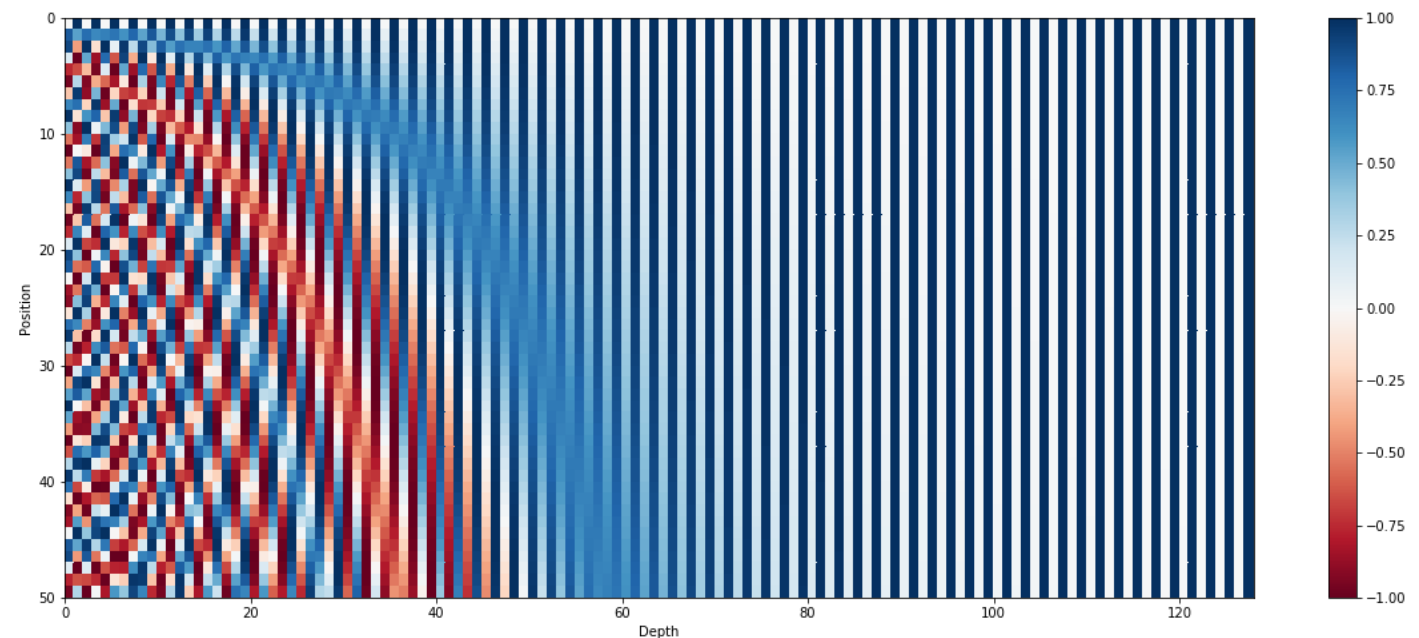
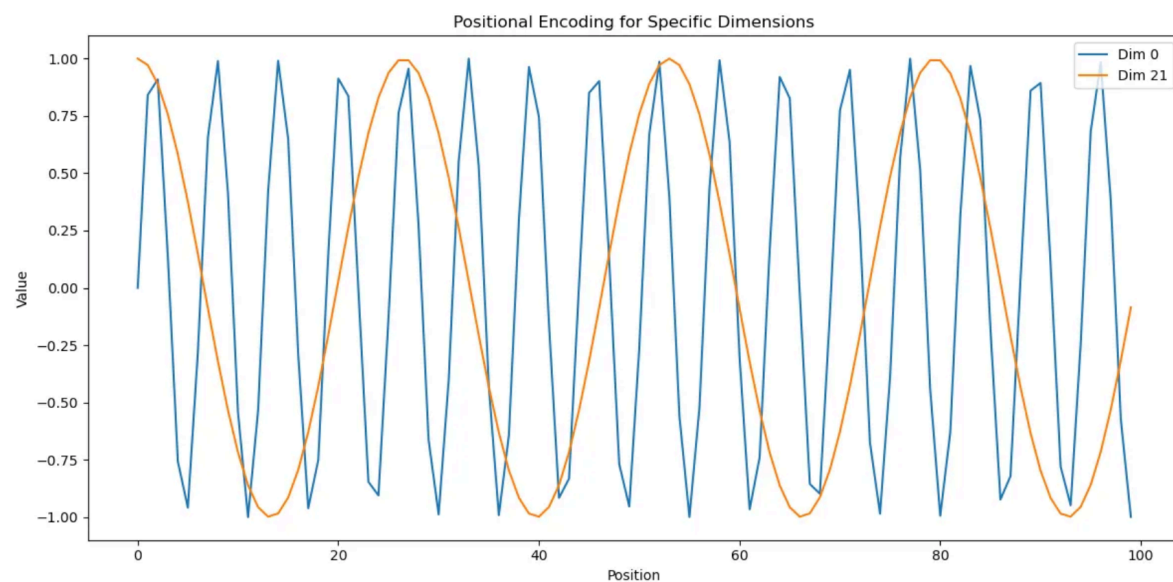
Thank you

Sinusoidal Encoding

(Vaswani+ 2017, Kazemnejad 2019)

- Calculate each dimension with a sinusoidal function

$$p_t^{(i)} = f(t)^{(i)} := \begin{cases} \sin(\omega_k \cdot t), & \text{if } i = 2k \\ \cos(\omega_k \cdot t), & \text{if } i = 2k + 1 \end{cases} \quad \text{where} \quad \omega_k = \frac{1}{10000^{2k/d}}$$



- Motivation: may be easy to learn relative positions, since PE_{pos+k} is a linear function of PE_{pos}