CS11-711 Advanced NLP

Attention and Transformers

Sean Welleck

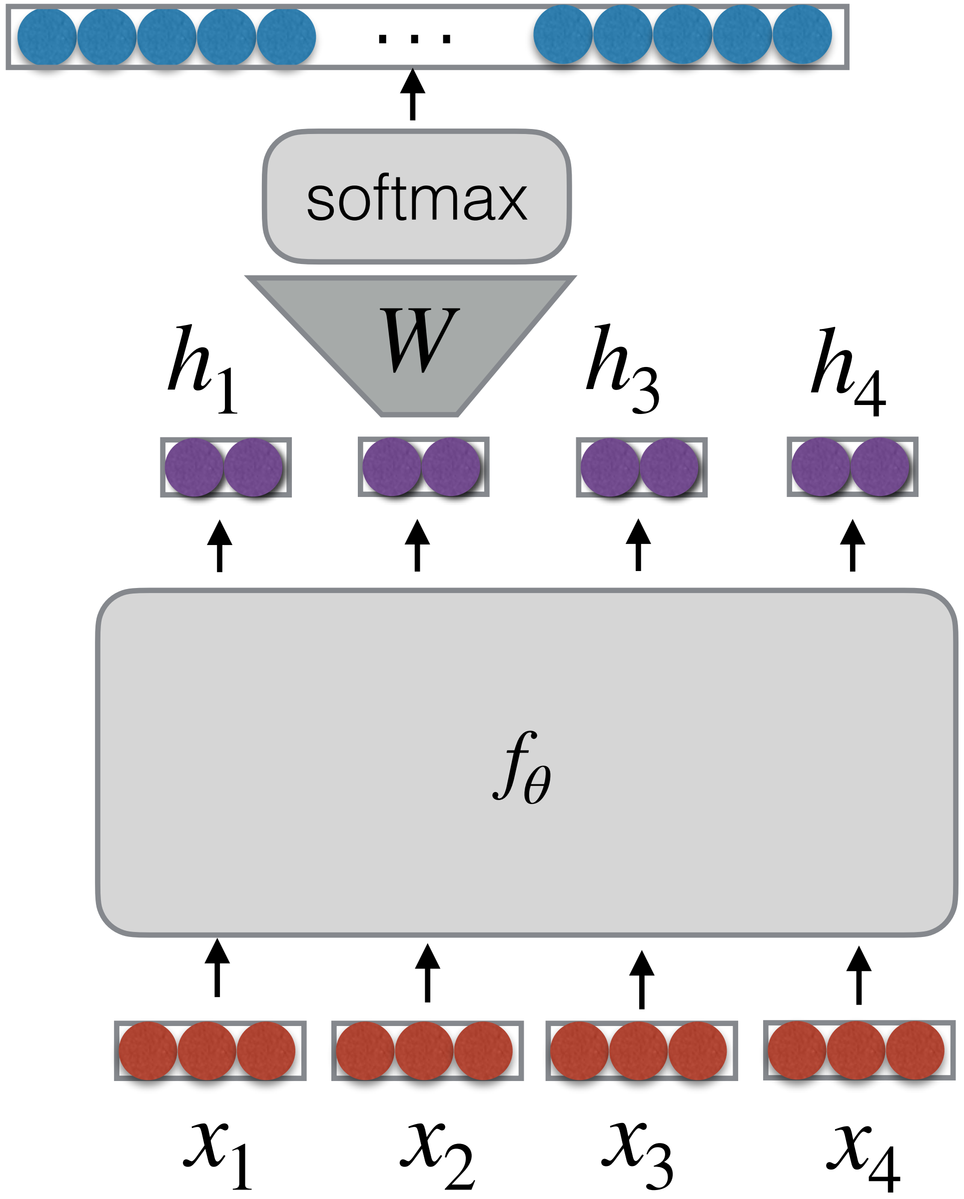
Carnegie Mellon University
Language Technologies Institute

https://cmu-l3.github.io/anlp-spring2025/

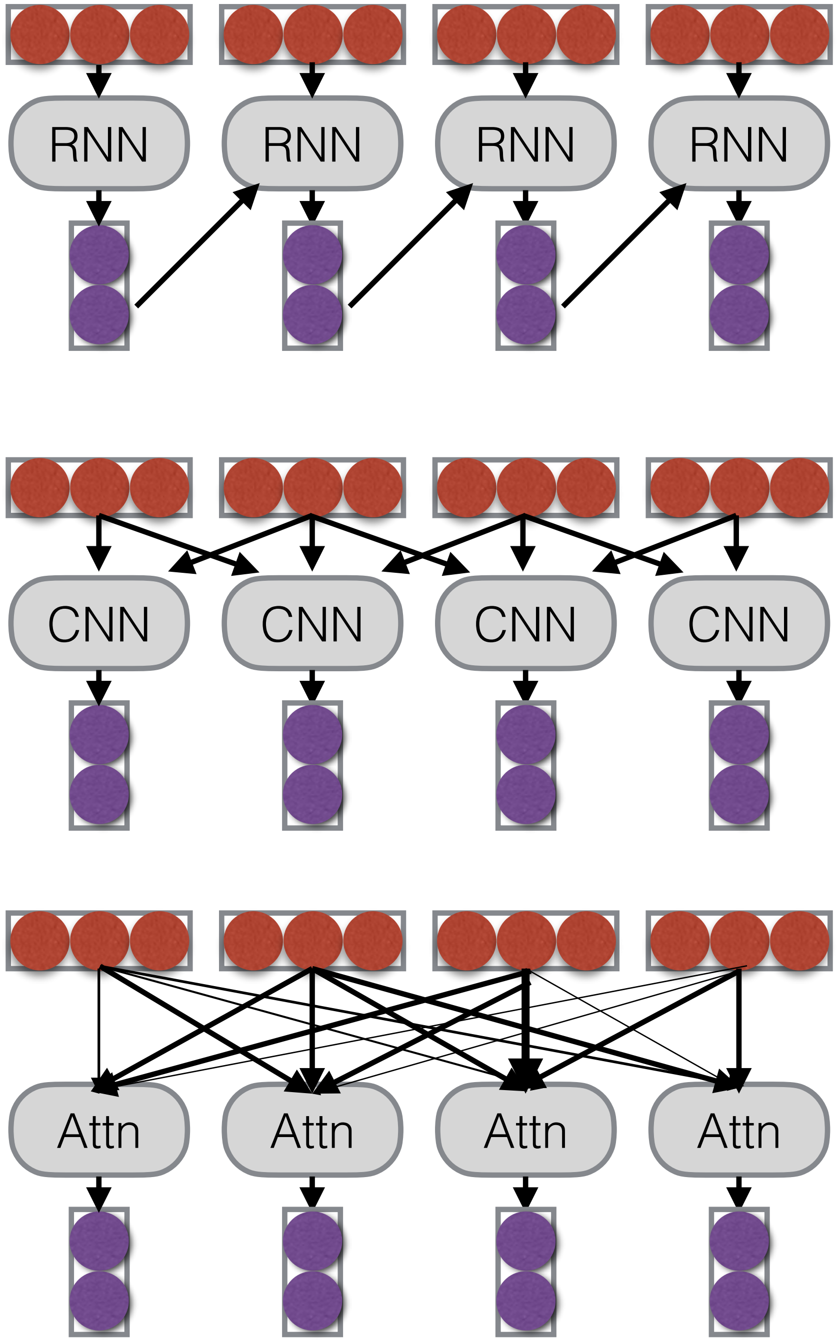
Many slides from Graham Neubig from Fall 2024

Recap: sequence model

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



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

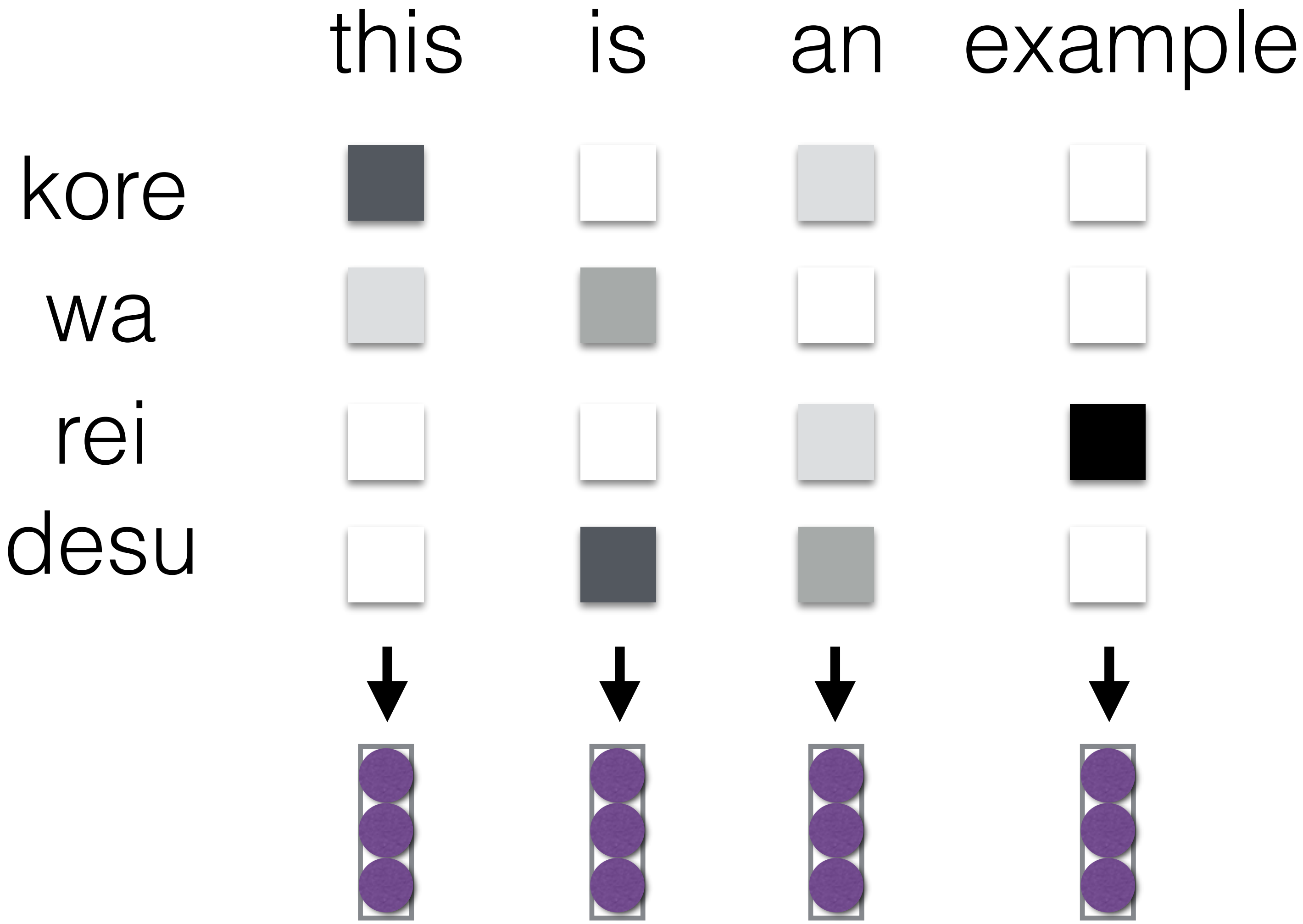
(Bahdanau et al. 2015)

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

Cross Attention

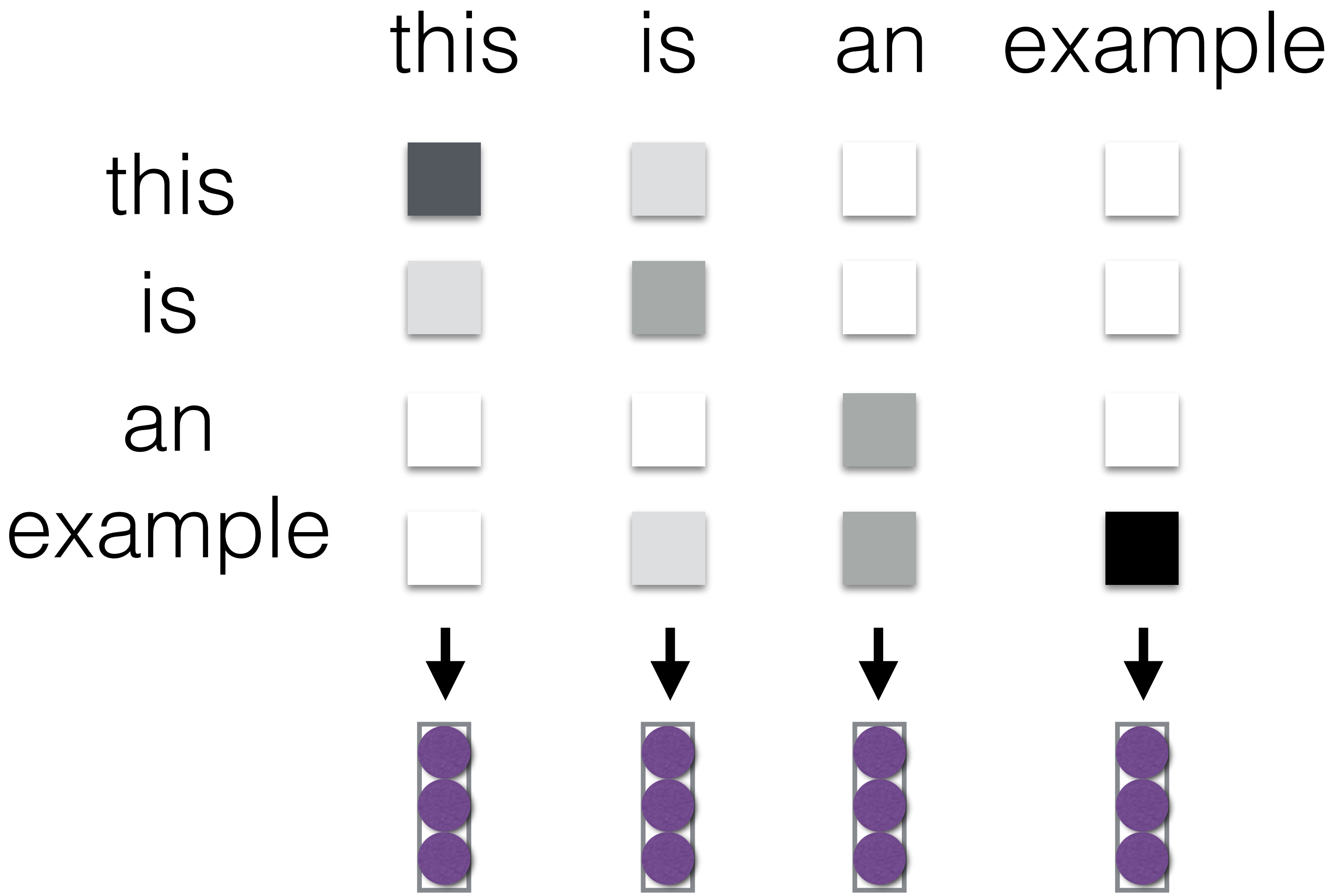
(Bahdanau et al. 2015)

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



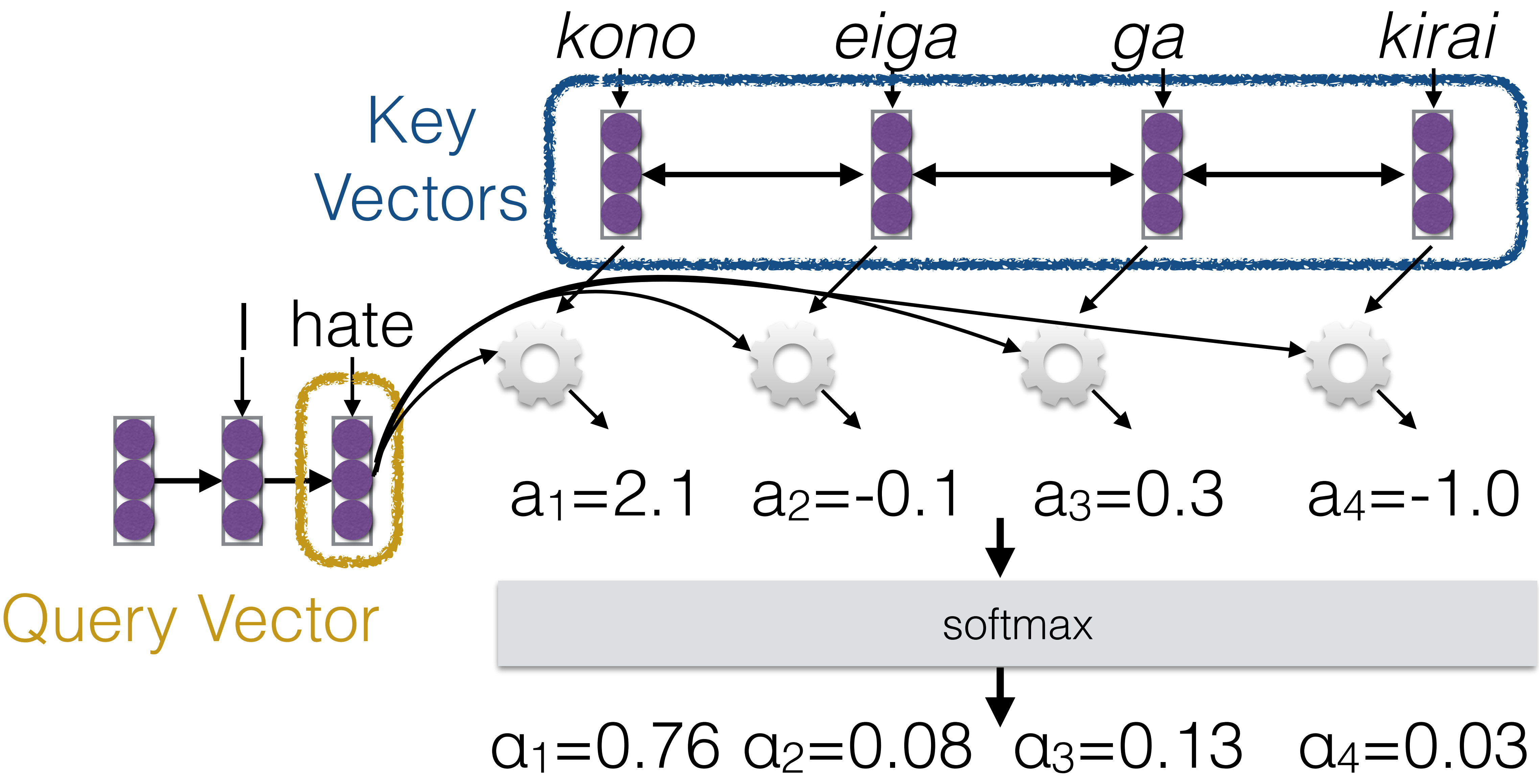
Self Attention
(Cheng et al. 2016, Vaswani et al. 2017)

- 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



kono *eiga* *ga* *kirai*

Key Vectors

I hate

Query Vector

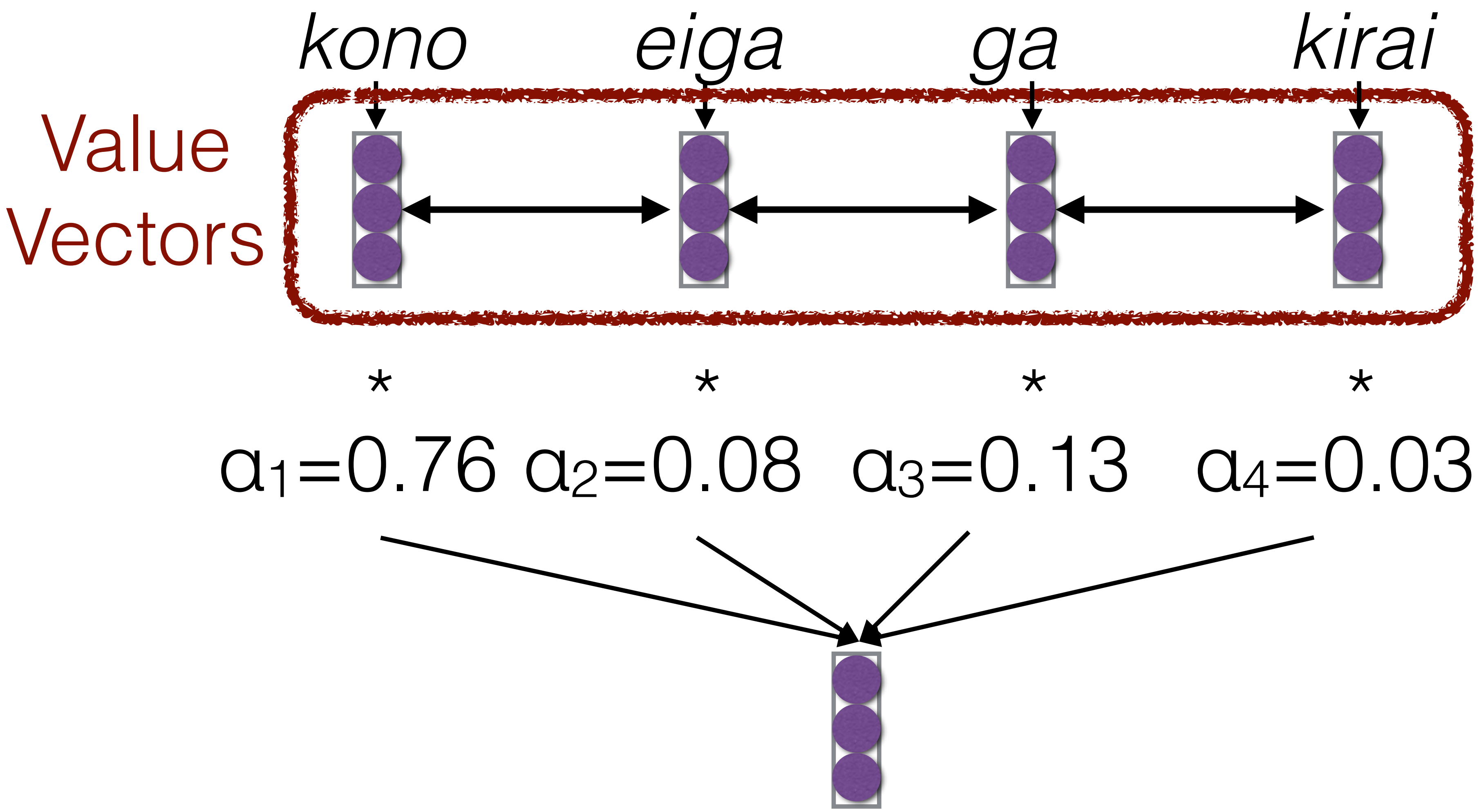
a_1=2.1 a_2=-0.1 a_3=0.3 a_4=-1.0

softmax

α_1=0.76 α_2=0.08 α_3=0.13 α_4=0.03

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

A Graphical Example

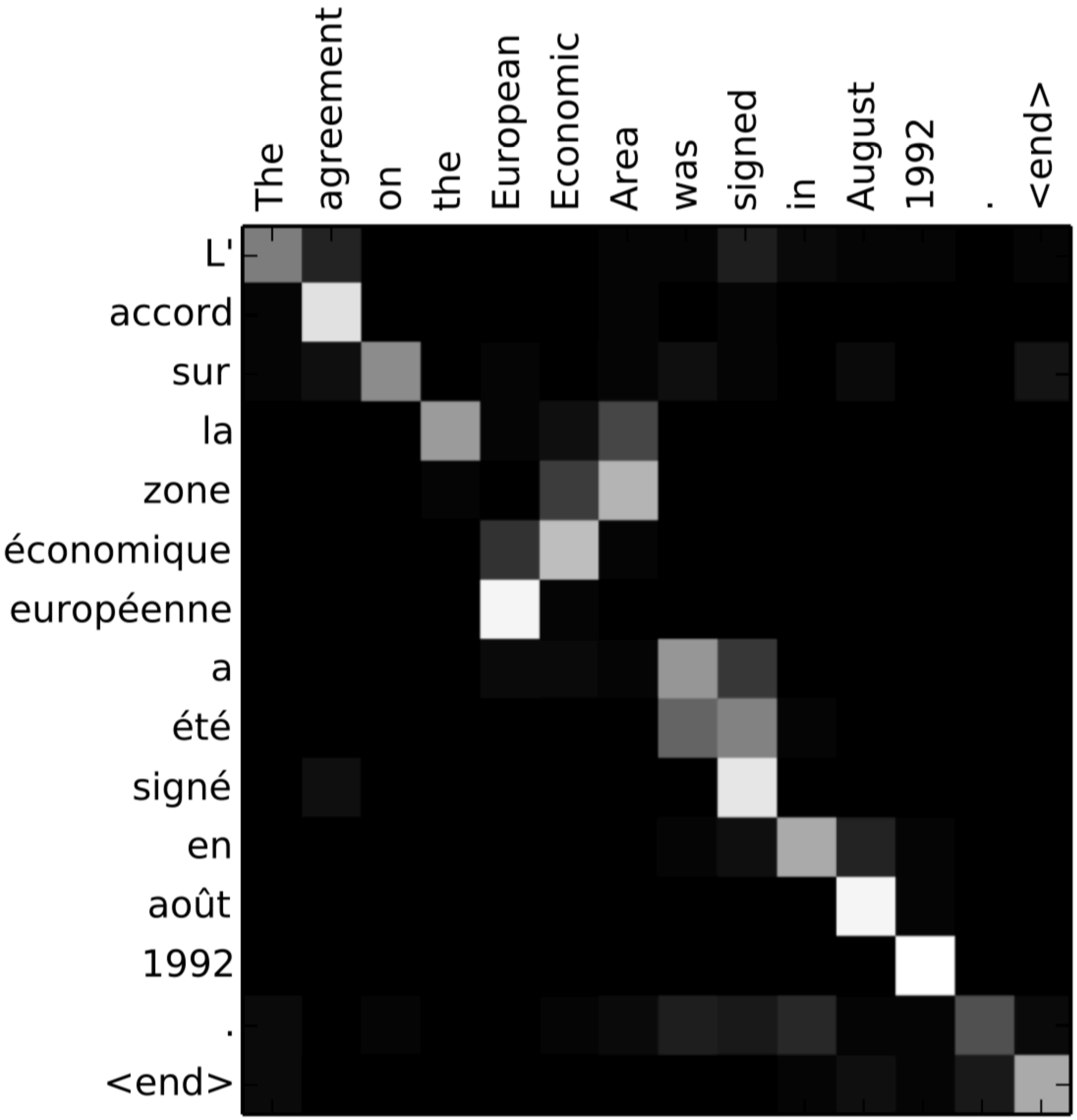


Image from Bahdanau et al. (2015)

Attention Score Functions (1)

- ***q*** is the query and ***k*** is the key
- **Nonlinear** (Bahdanau et al. 2015)

$$a(\boldsymbol{q}, \boldsymbol{k}) = \boldsymbol{w}_2^{\intercal} \tanh(W_1[\boldsymbol{q}; \boldsymbol{k}])$$

- **Bilinear** (Luong et al. 2015)

$$a(\boldsymbol{q}, \boldsymbol{k}) = \boldsymbol{q}^{\intercal} W \boldsymbol{k}$$

Attention Score Functions (2)

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

$$a(\boldsymbol{q}, \boldsymbol{k}) = \boldsymbol{q}^{\intercal}\boldsymbol{k}$$

- **Scaled Dot Product** (Vaswani et al. 2017)
  - *Problem:* scale of dot product increases as dimensions get larger
  - *Fix:* scale by size of the vector

$$a(\boldsymbol{q}, \boldsymbol{k}) = \frac{\boldsymbol{q}^{\intercal}\boldsymbol{k}}{\sqrt{|\boldsymbol{k}|}}$$

Today's lecture

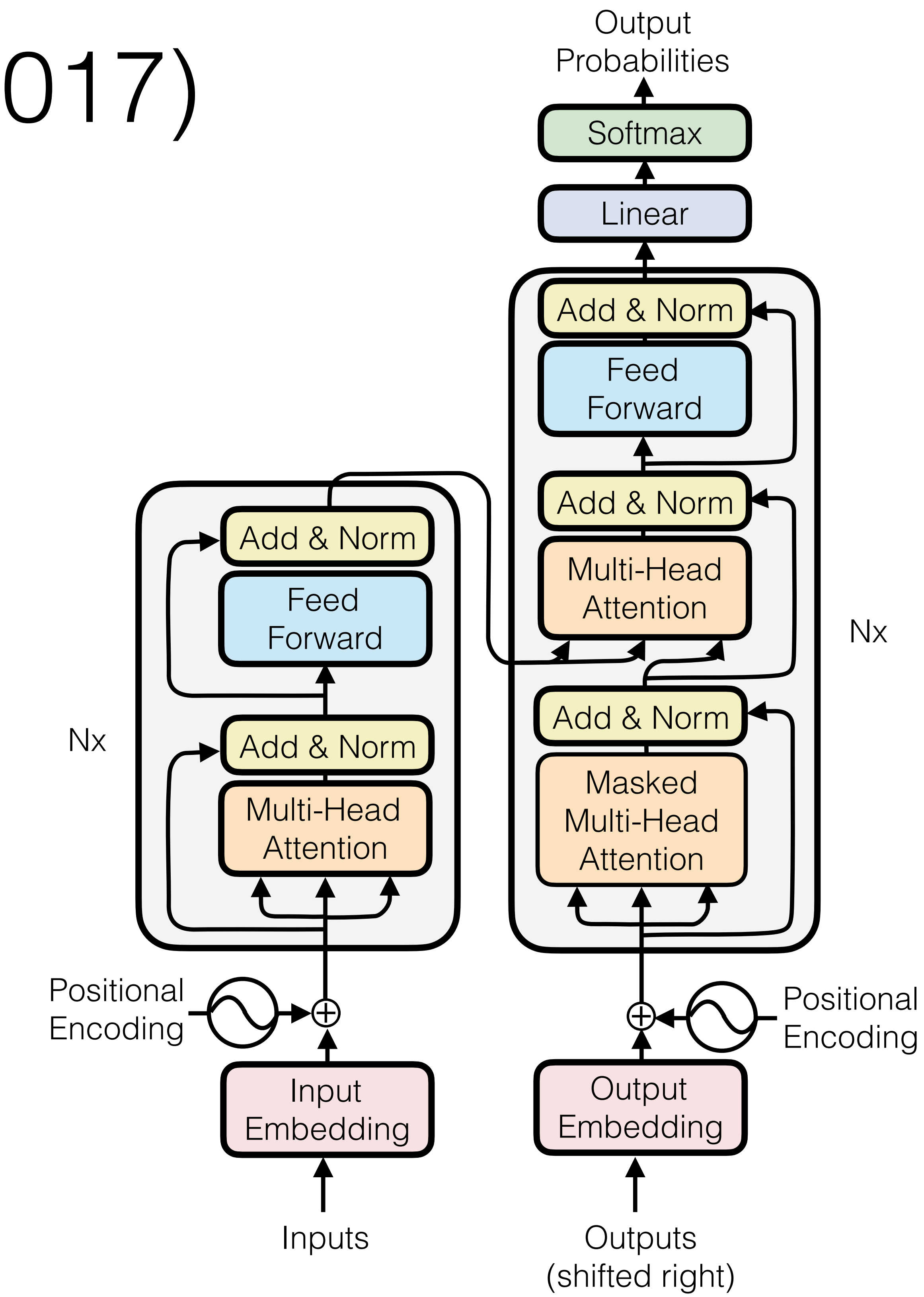
- Roadmap:
    - Attention
    - **Transformer architecture**
    - Improved transformer architecture

Transformers

"Attention is All You Need"

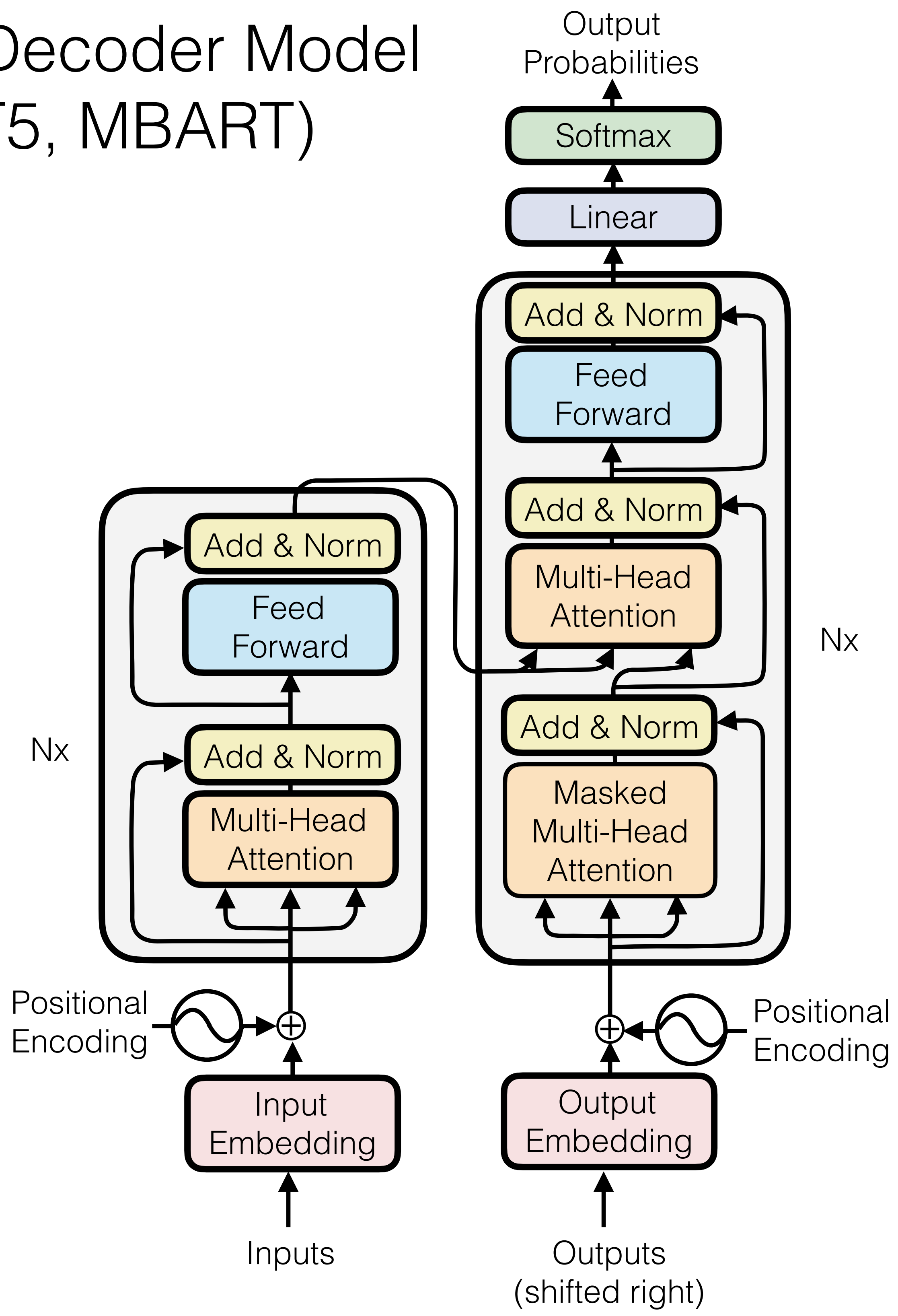
(Vaswani et al. 2017)

- A sequence-to-sequence model based entirely on attention
- Strong results on machine translation
- Fast: only matrix multiplications

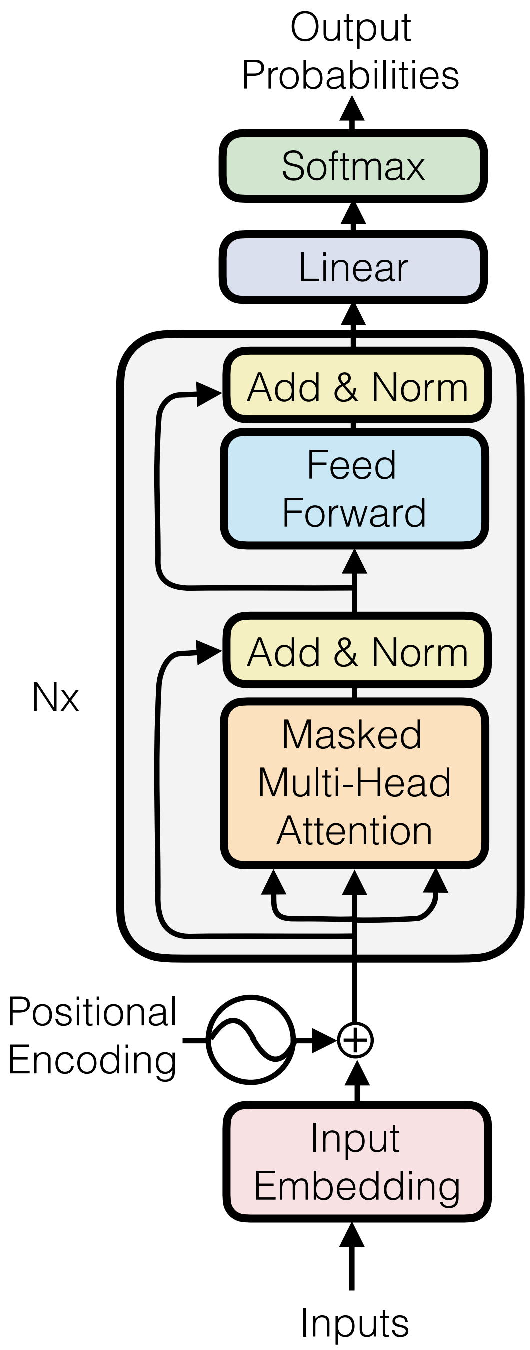


Two Types of Transformers

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



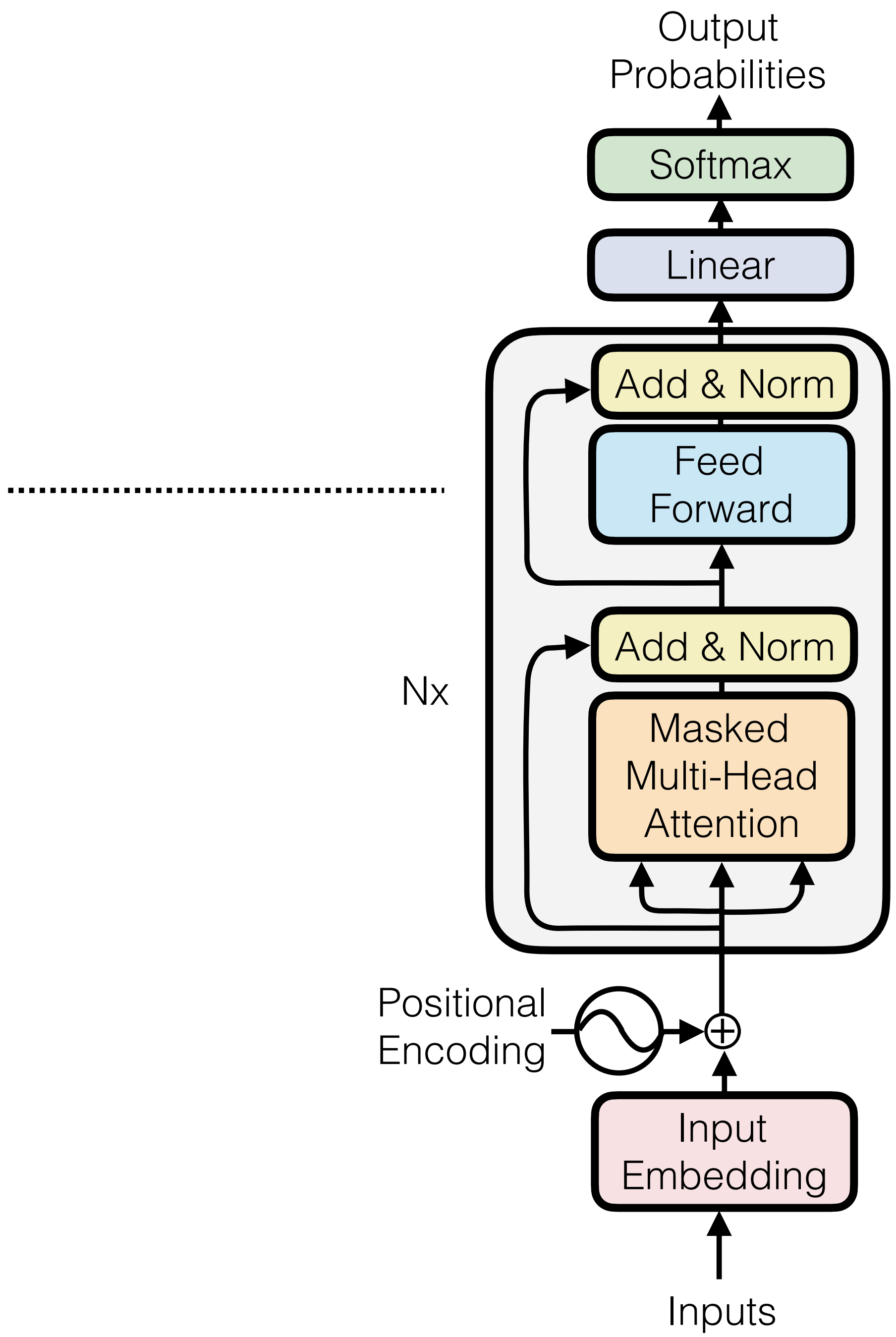
Decoder Only Model
(e.g. GPT, LLaMa)



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

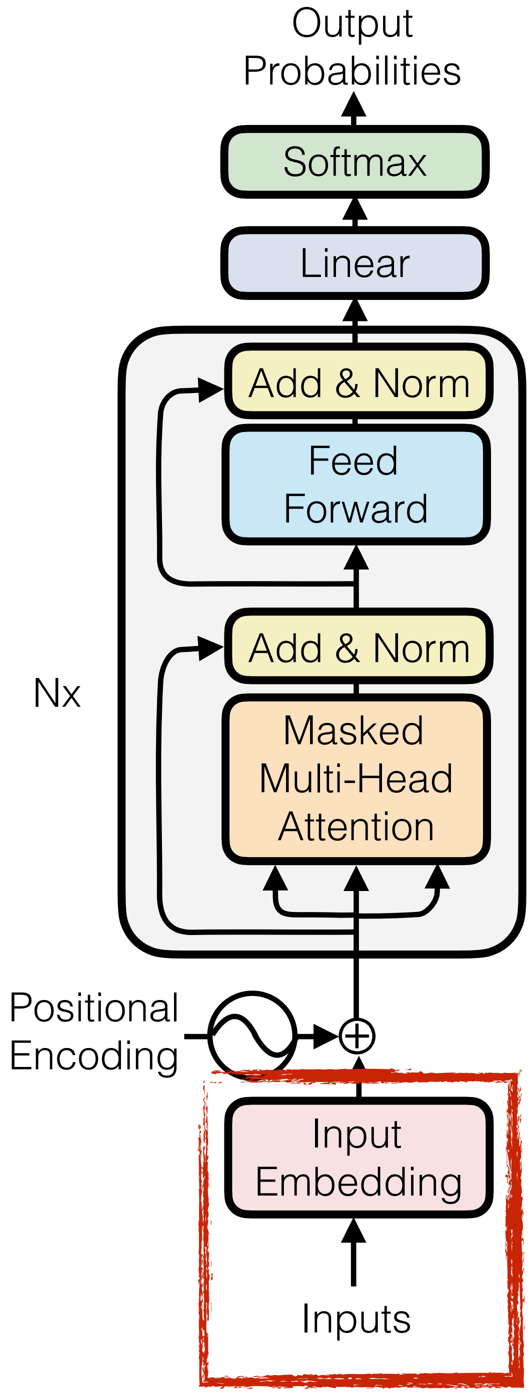
(Review)
Inputs and Embeddings

- **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

Positional Encoding

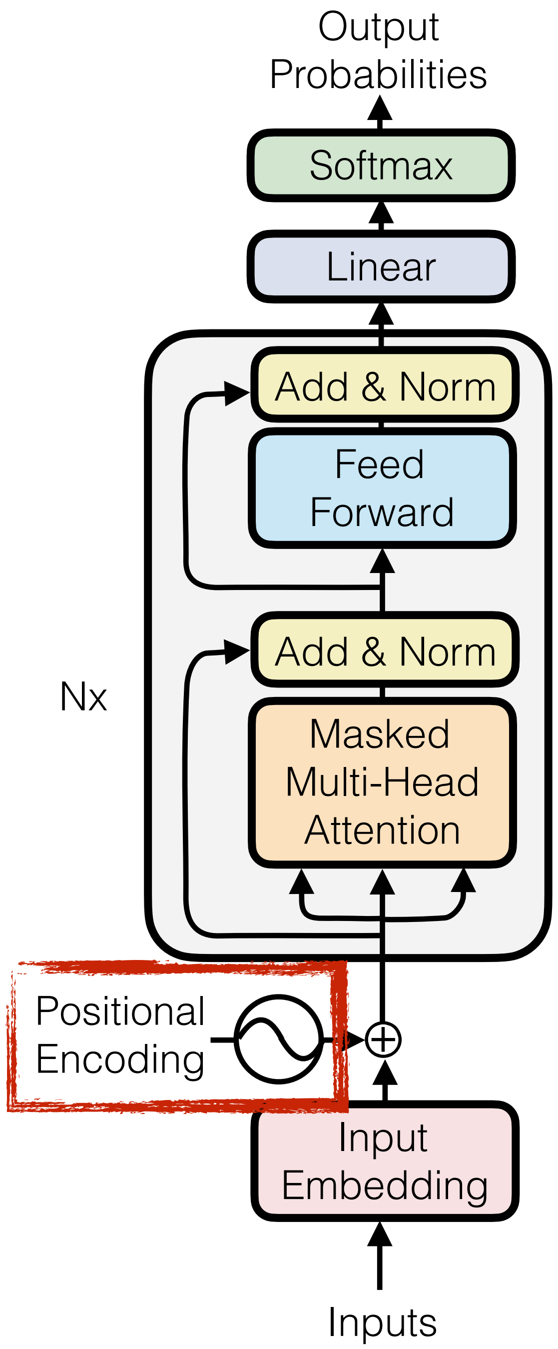
- The transformer model is *purely* attentional
- We need a way to identify the position of each token

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

$W_{big} + W_{pos2}$ $W_{big} + W_{pos7}$



Sinusoidal Encoding

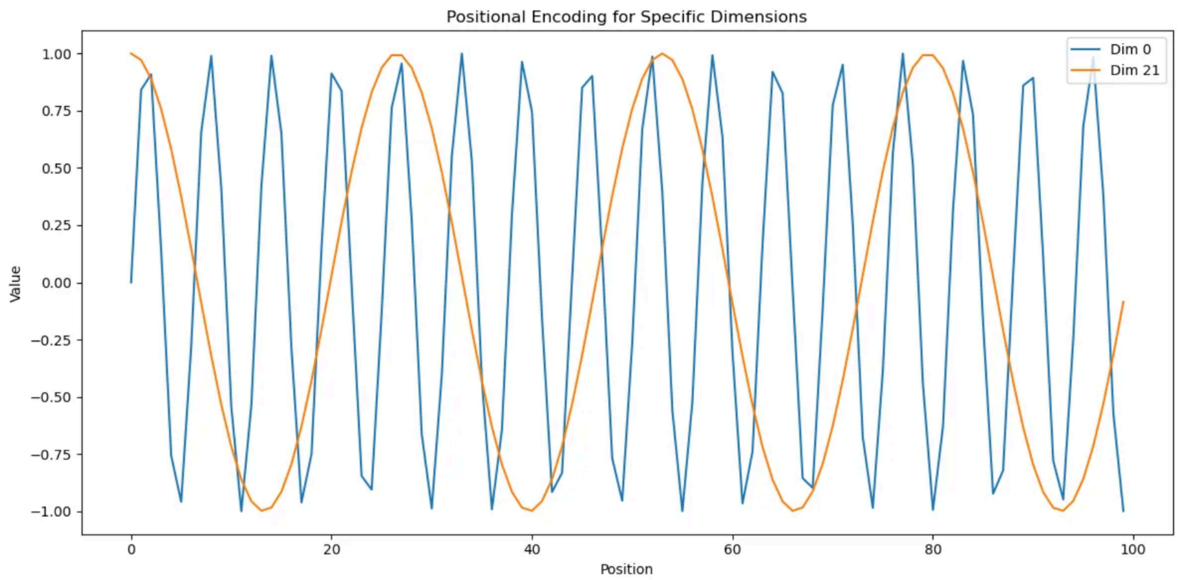
(Vaswani+ 2017, Kazemnejad 2019)

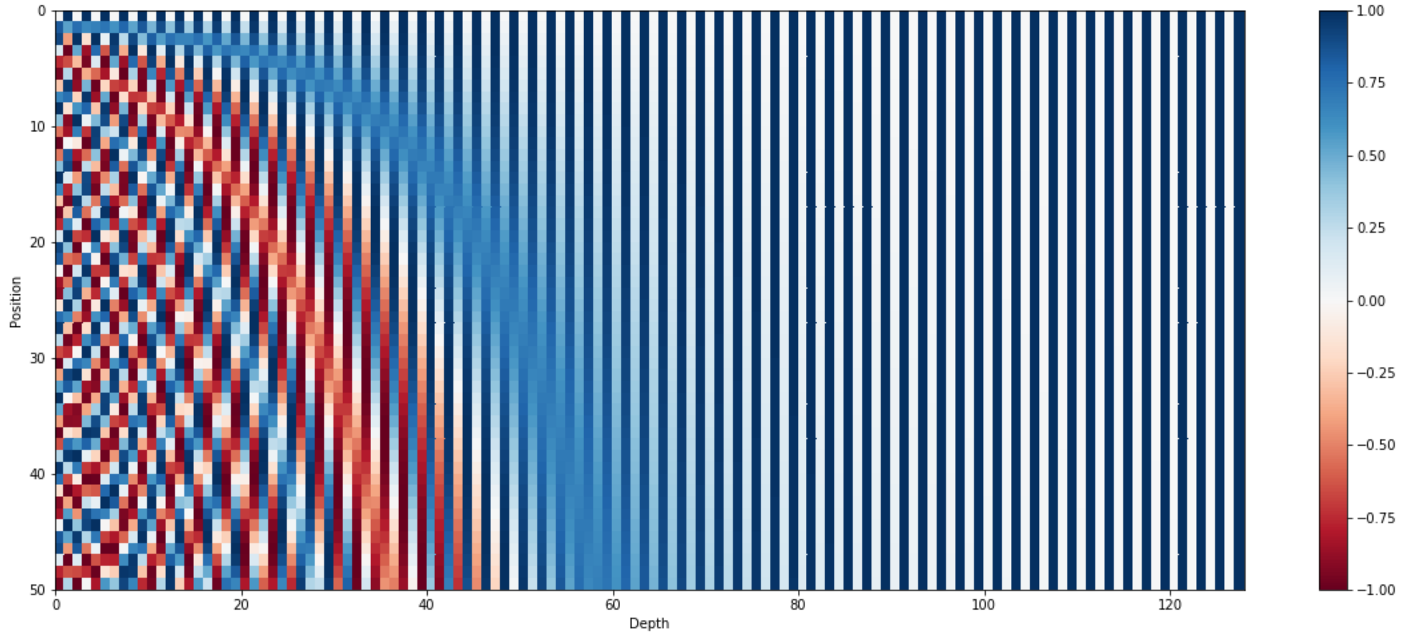
- Calculate each dimension with a sinusoidal function

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

where

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

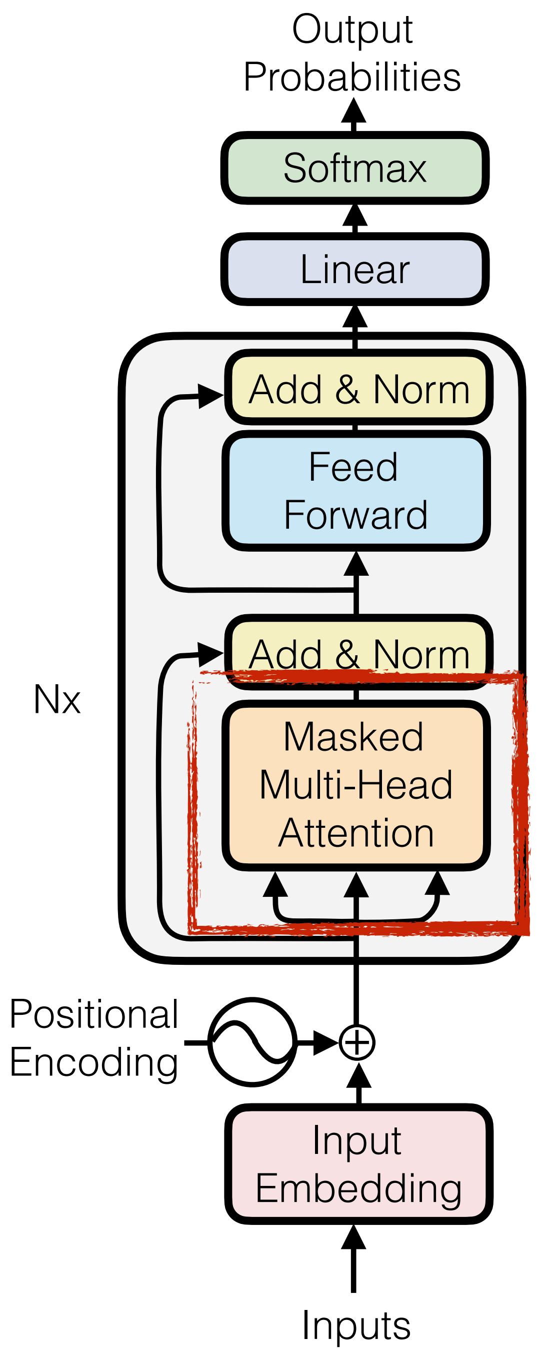




- Motivation: may be easy to learn relative positions, since PE_{pos+k} is a linear function of PE_{pos}

Core Transformer Concepts

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

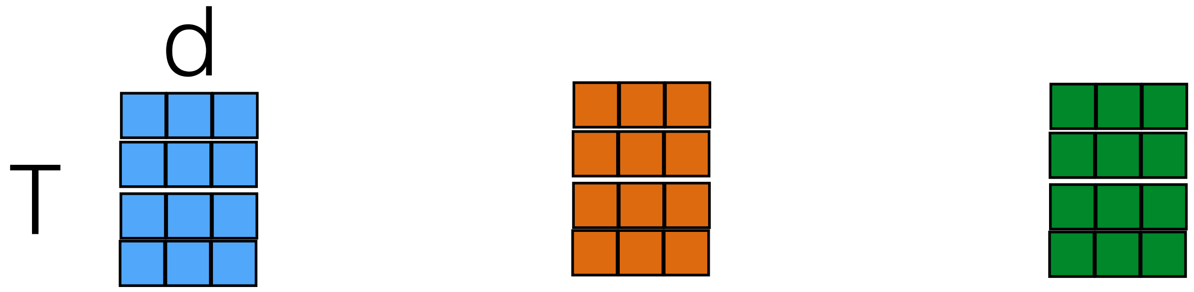


Scaled Dot-Product Self-Attention

Scaled dot product attention

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

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

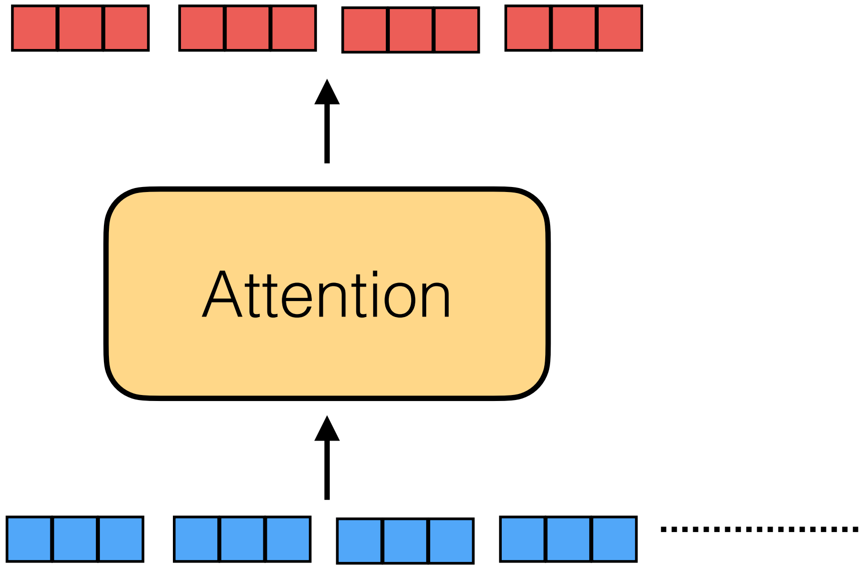


$$\mathrm{Attention}(Q, K, V) = \mathrm{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right)V$$

Scaled dot product self-attention

- Apply attention to the output of the previous layer:

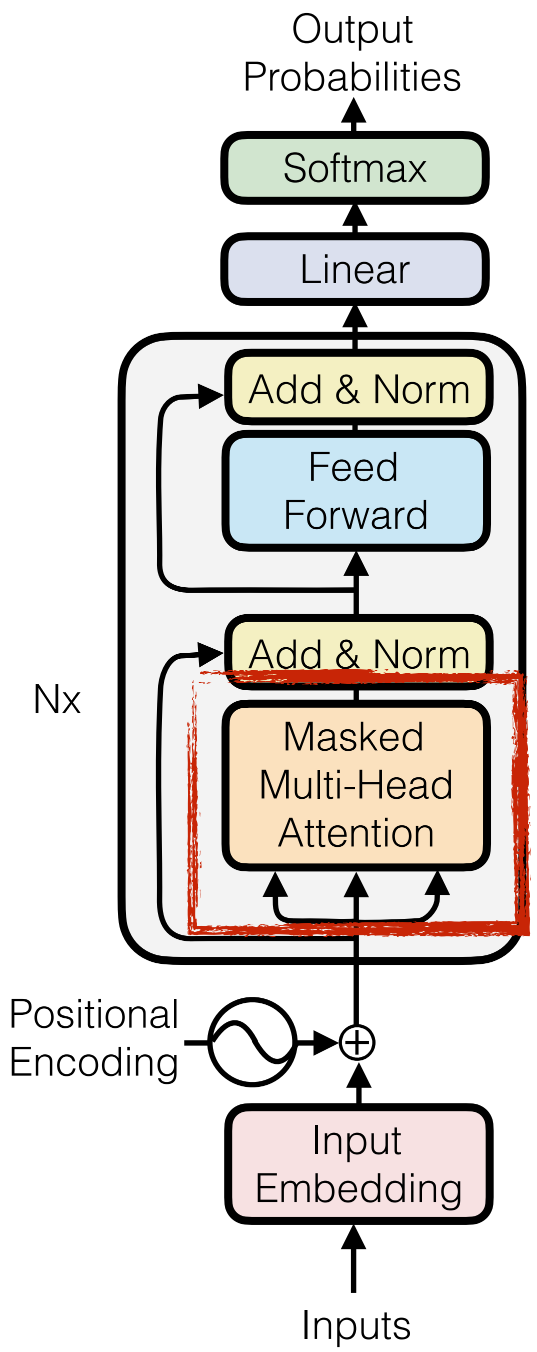
$$\text{Attention}(H^{\ell-1}, H^{\ell-1}, H^{\ell-1}) \rightarrow \tilde{H}^{\ell} \qquad H \in \mathbb{R}^{T \times d}$$



ℓ'th layer:
output of ℓ-1'th layer

Core Transformer Concepts

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



Multi-head Attention

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

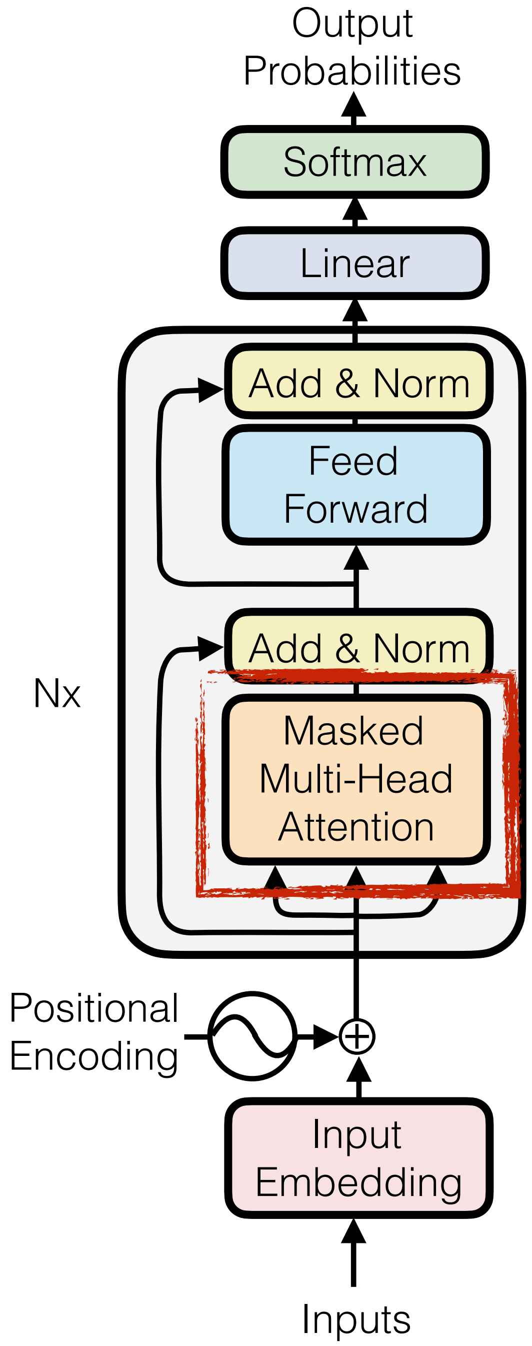
I **run** a mile in 10 minutes

The robber made a **run** for it

The stocking had a **run**

syntax
(nearby context)

semantics
(farther context)



Multi-head Attention Concept

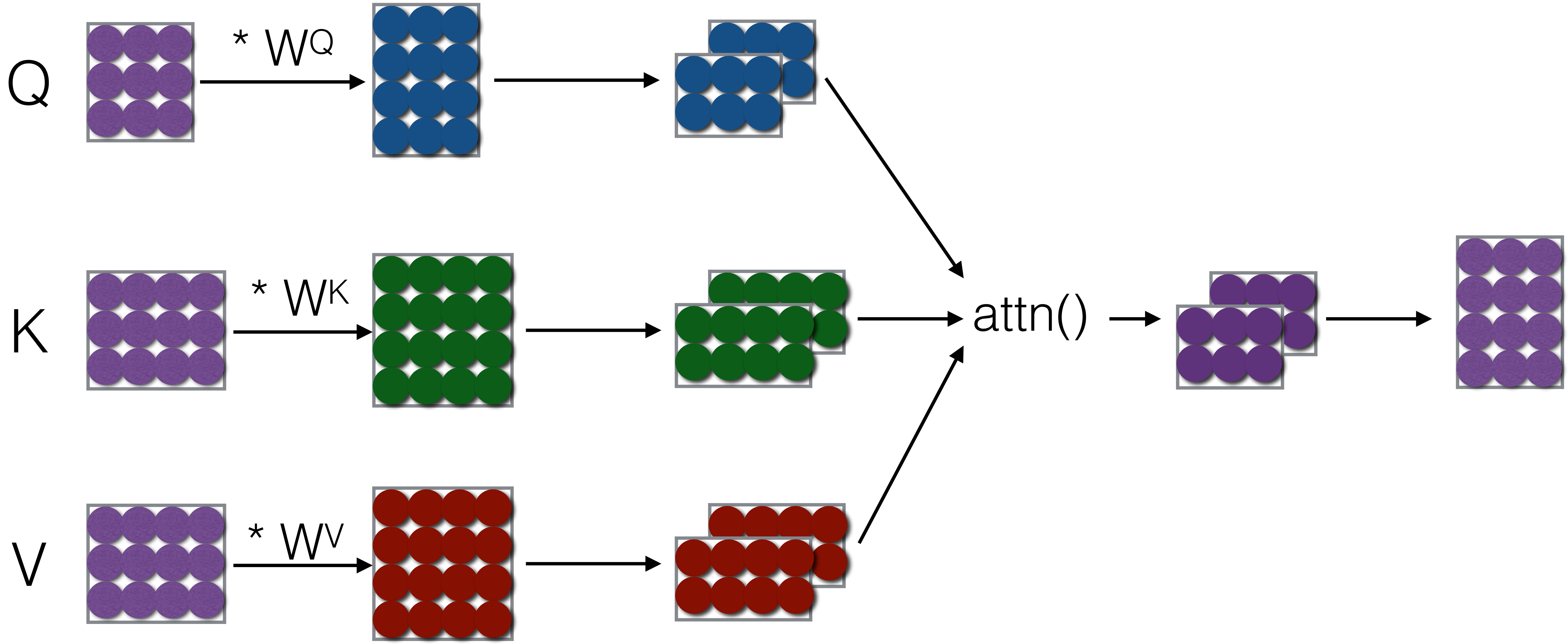
$$\begin{aligned}\mathrm{MultiHead}(Q, K, V) &= \mathrm{Concat}(\mathrm{head}_1, ..., \mathrm{head}_h)W^O \\ \text{where } \mathrm{head}_i &= \mathrm{Attention}(QW_i^Q, KW_i^K, VW_i^V)\end{aligned}$$

Multiply by weights

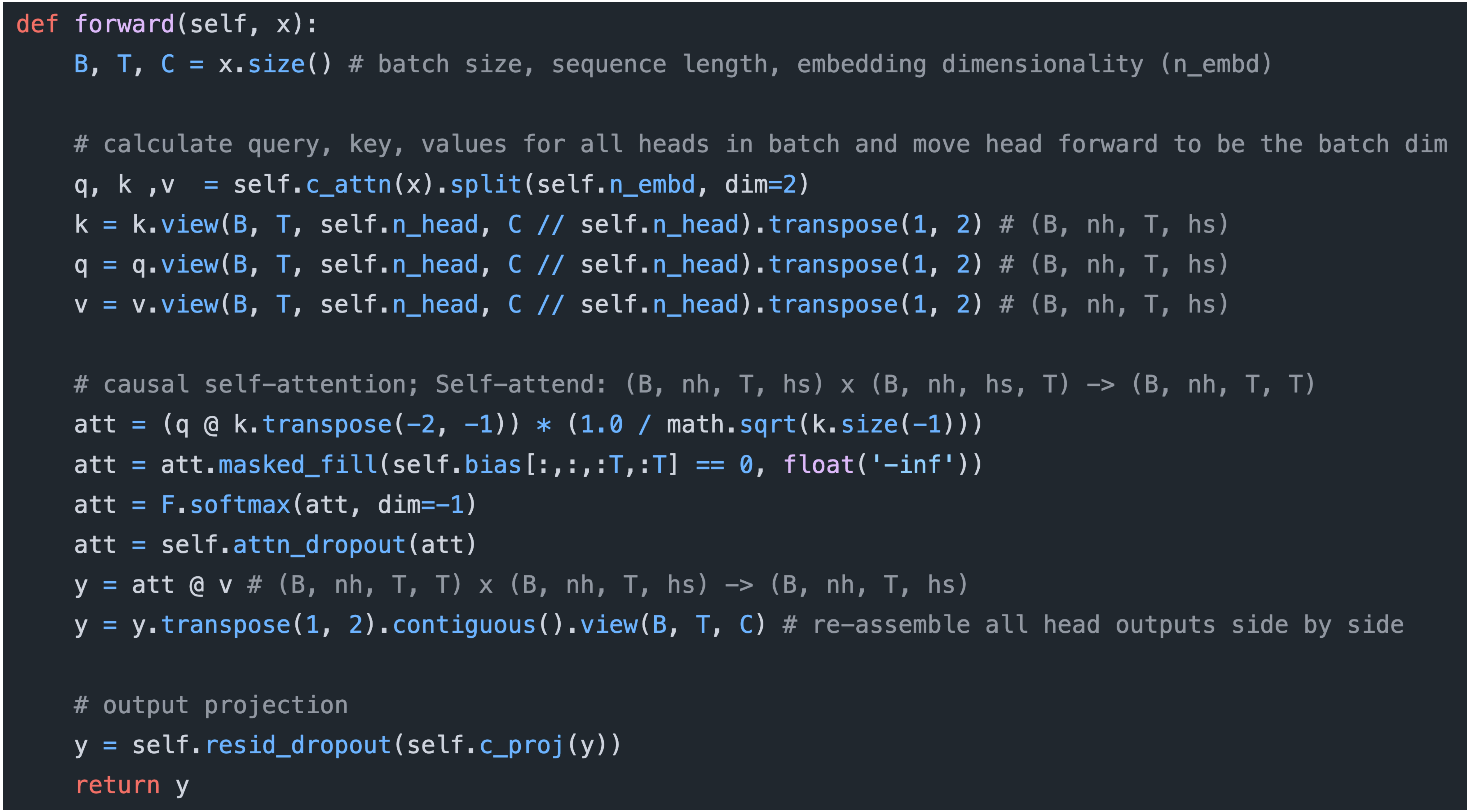
Split/rearrange to *n* attn inputs

Run attn over each head

Concat and $*W^O$



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
```

https://github.com/karpathy/minGPT/blob/master/mingpt/model.py

What Happens w/ Multi-heads?

- Example from Vaswani et al.

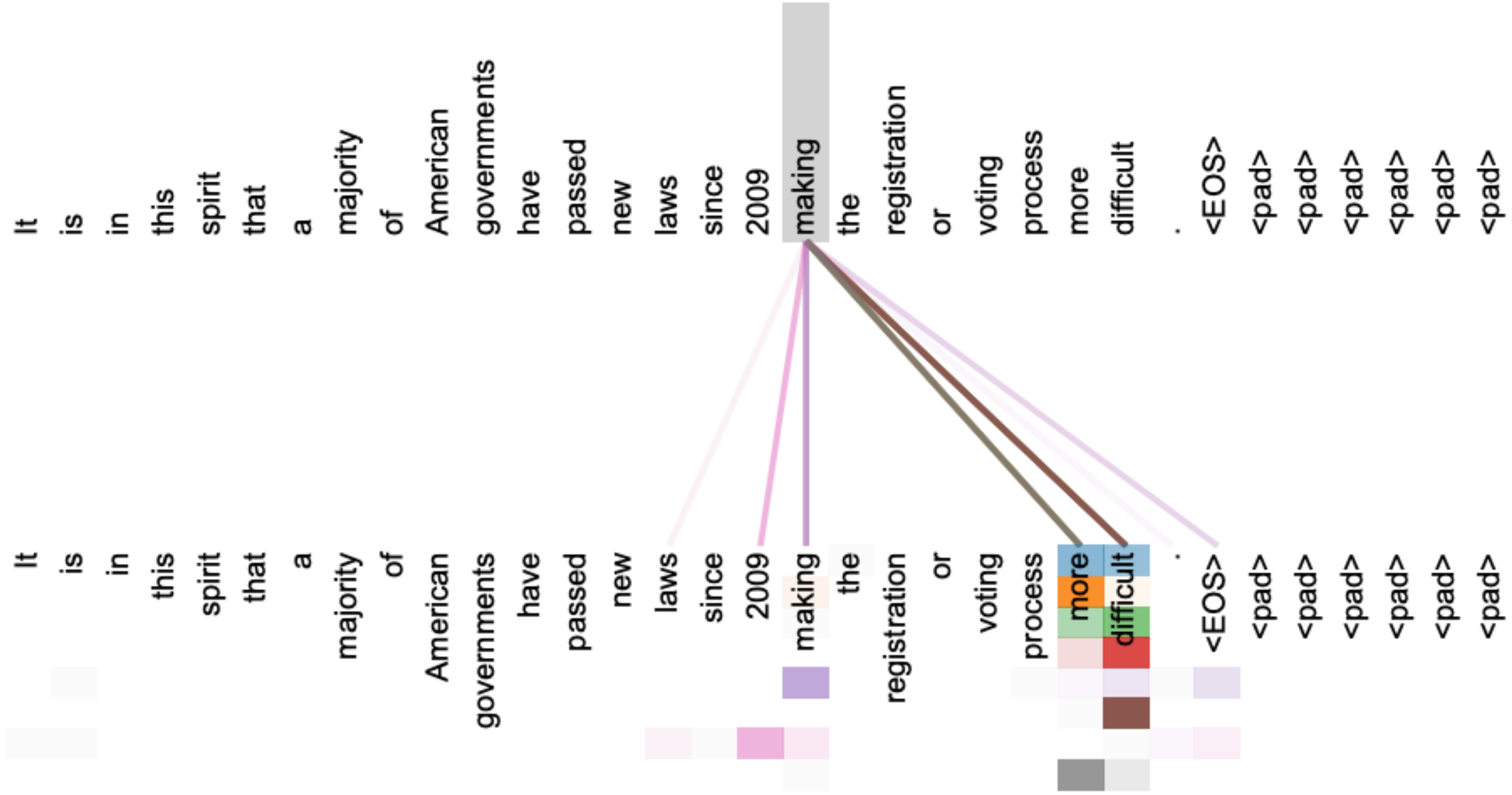
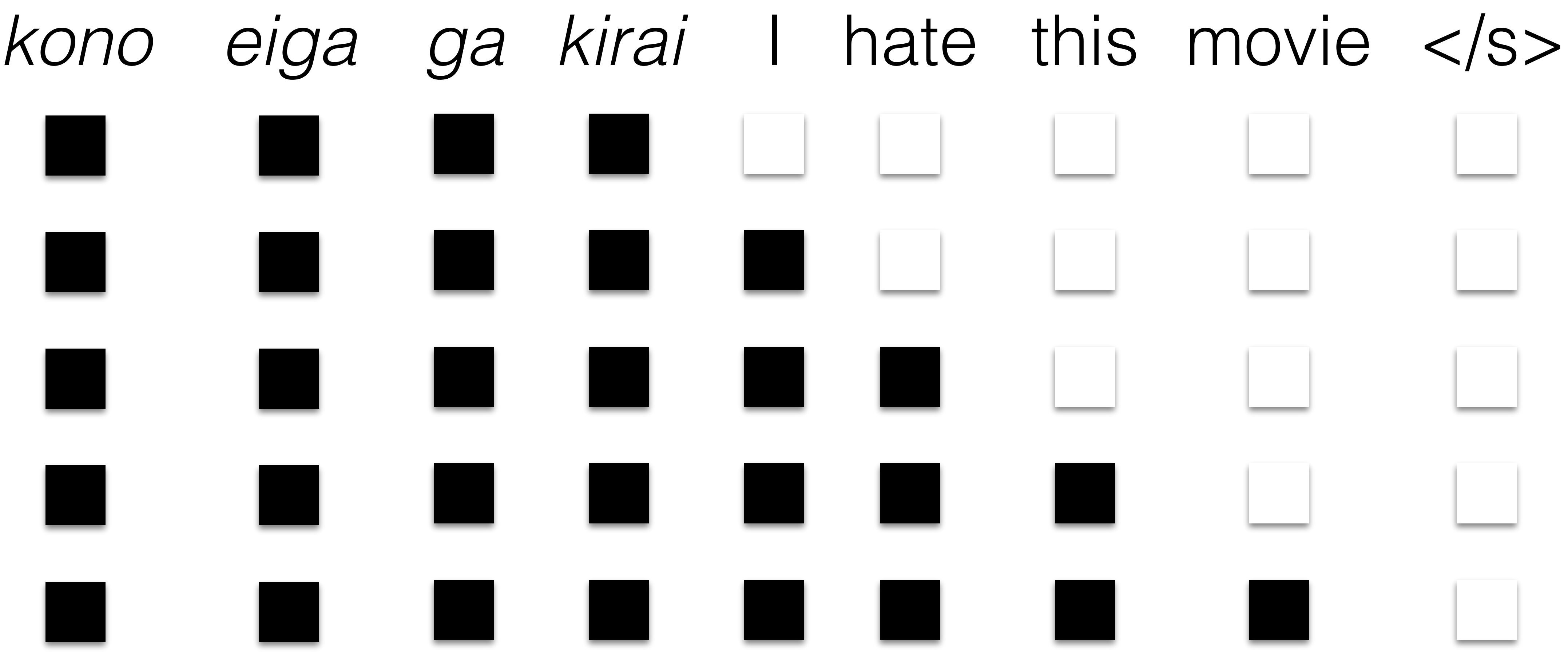


Figure 3: An example of the attention mechanism following long-distance dependencies in the encoder self-attention in layer 5 of 6. Many of the attention heads attend to a distant dependency of the verb 'making', completing the phrase 'making...more difficult'. Attentions here shown only for the word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

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

Masking for Language Model Training

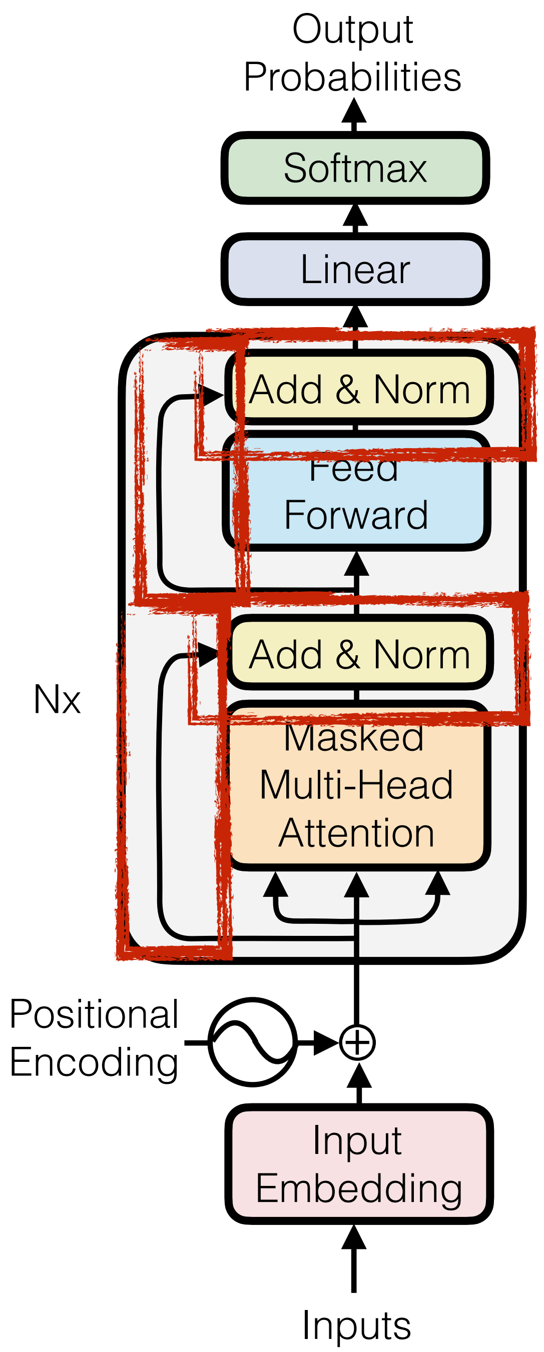
- Mask the attention from future timesteps
    - Prevents the model from cheating when predicting the next token



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

Core Transformer Concepts

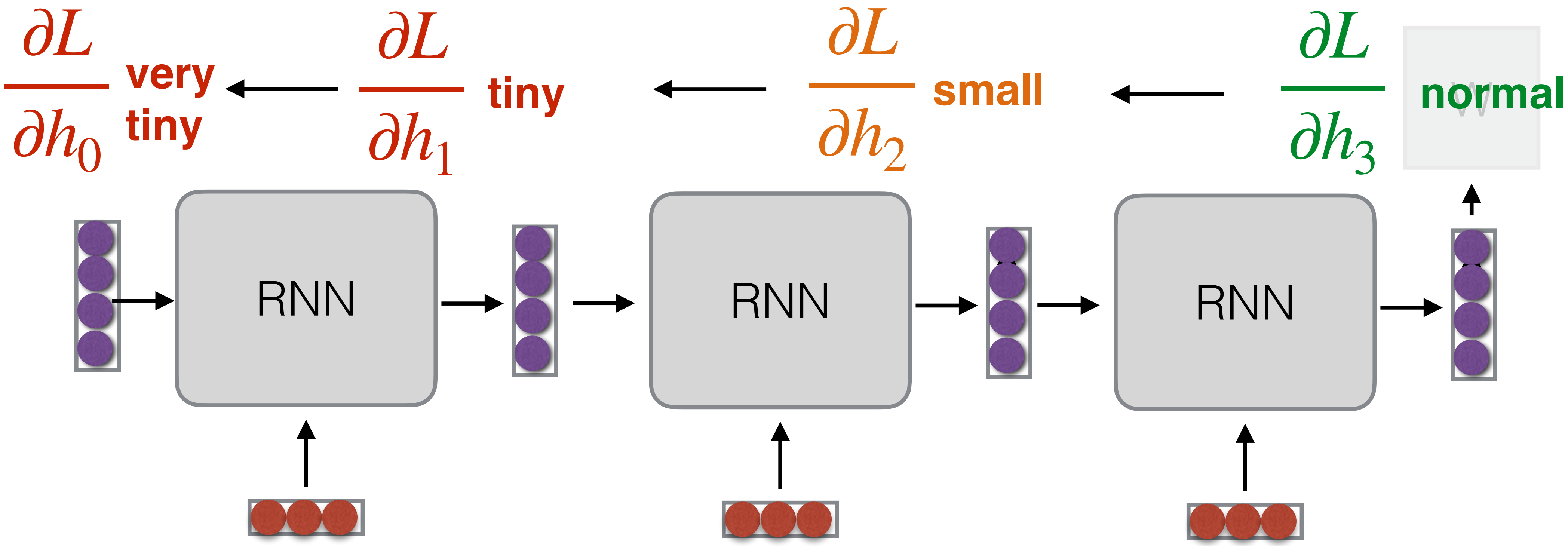
- Positional encodings
- Scaled dot product self-attention
- Multi-headed attention
- **Residual + layer normalization**
- Feed-forward layer



Layer Normalization and Residual Connections

Reminder:
Gradients and Training Instability

- RNNs: backpropagation can make gradients vanish or explode



$\frac{\partial L}{\partial h_0}$ **very tiny** ← $\frac{\partial L}{\partial h_1}$ **tiny** ← $\frac{\partial L}{\partial h_2}$ **small** ← $\frac{\partial L}{\partial h_3}$ **normal**

RNN

RNN

RNN

- 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

gain

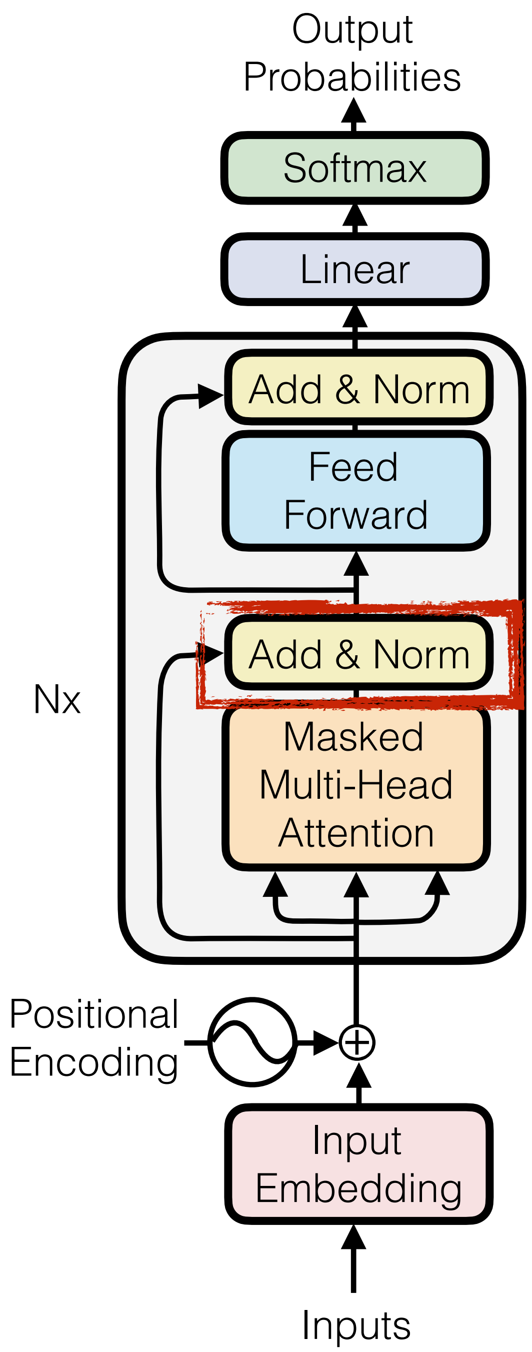
bias

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

vector stddev

vector mean

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

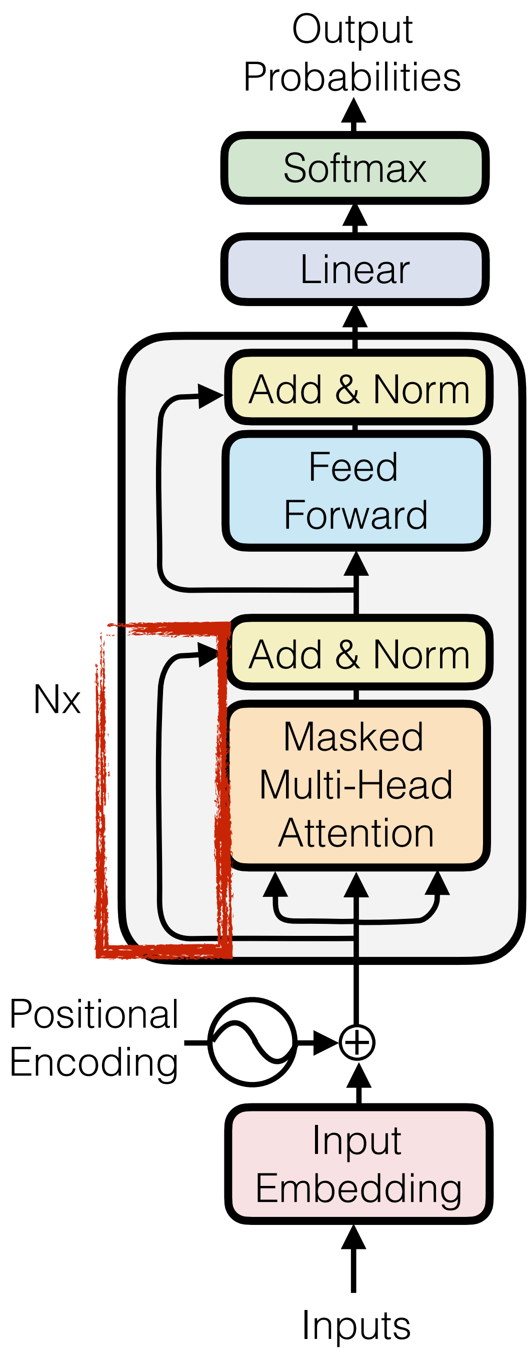


Residual Connections

- Add an additive connection between the input and output

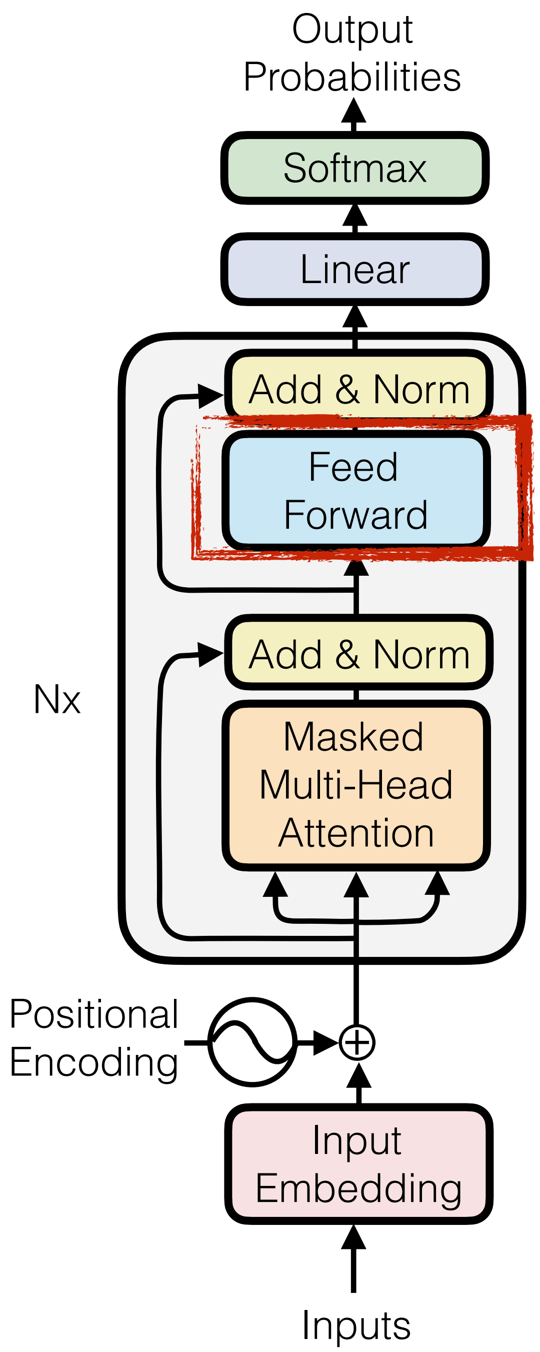
$$\text{Residual}(\mathbf{x}, f) = f(\mathbf{x}) + \mathbf{x}$$

- 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**

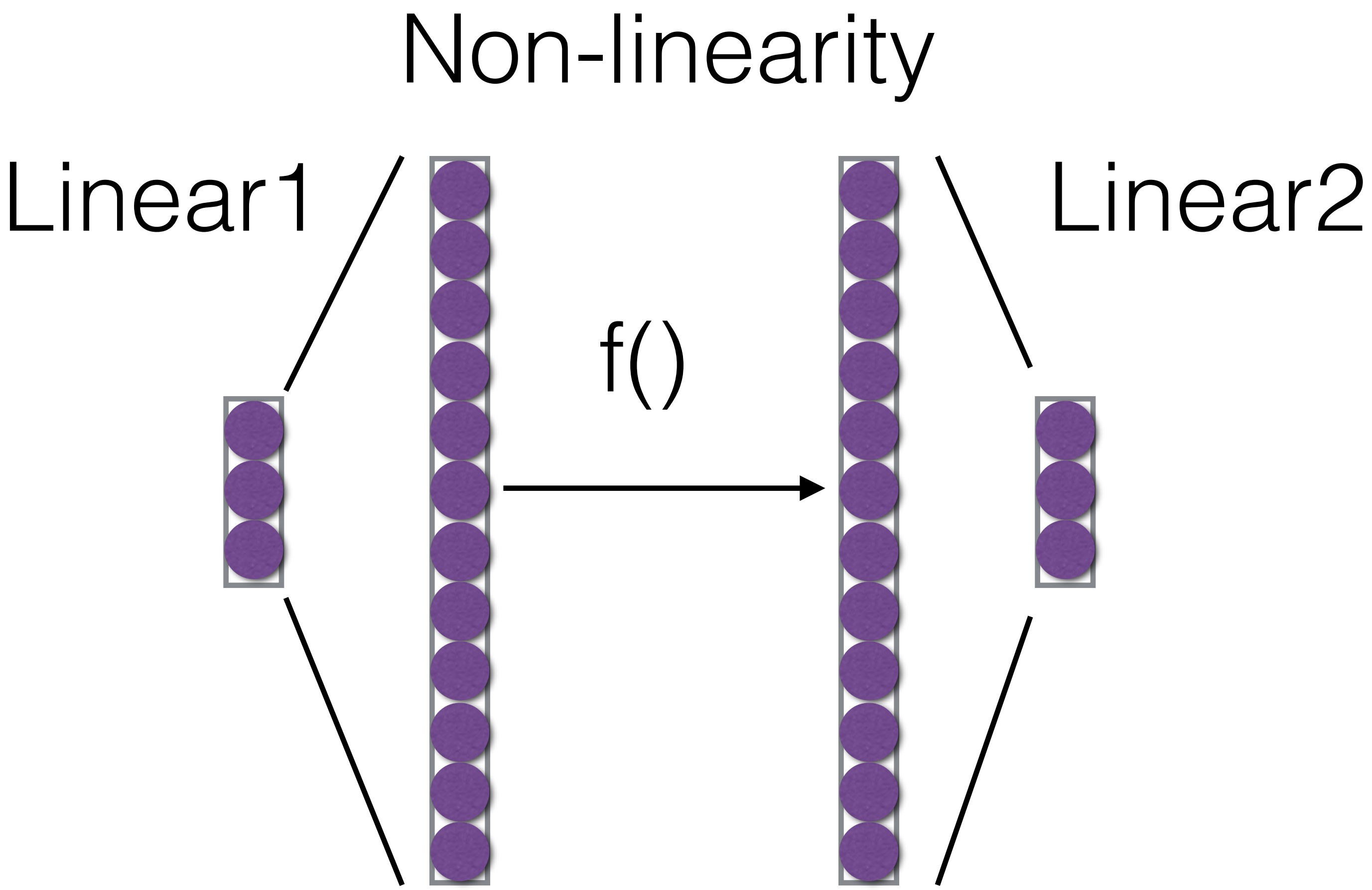


Feed Forward Layers

Feed Forward Layers

- Extract features from the attended outputs

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

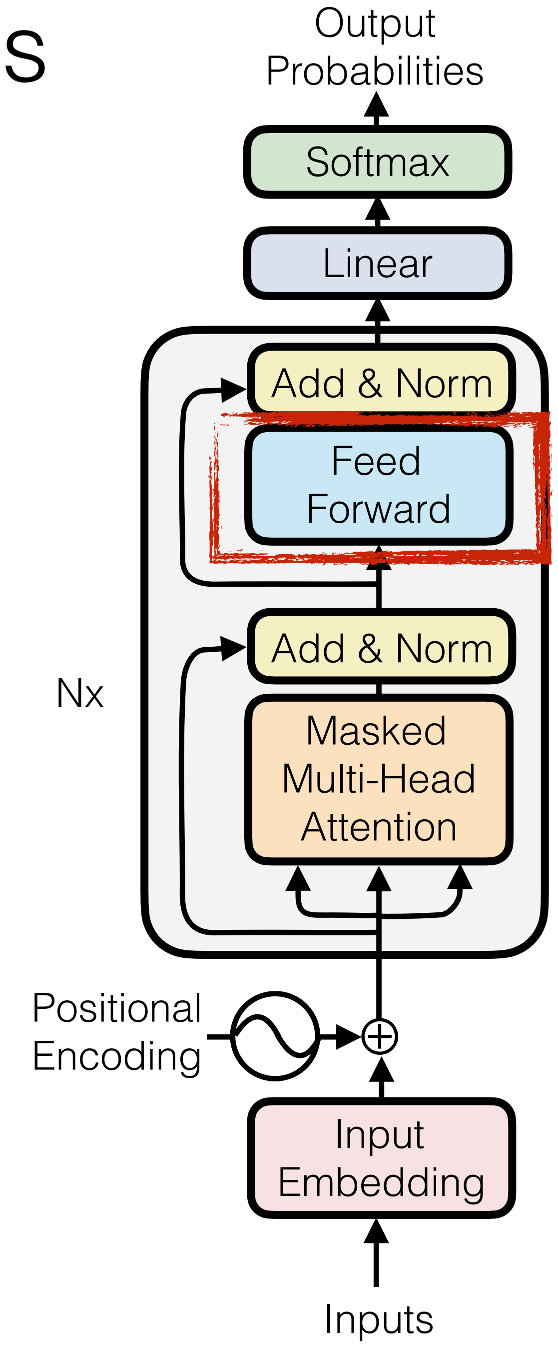


Non-linearity

Linear1

f()

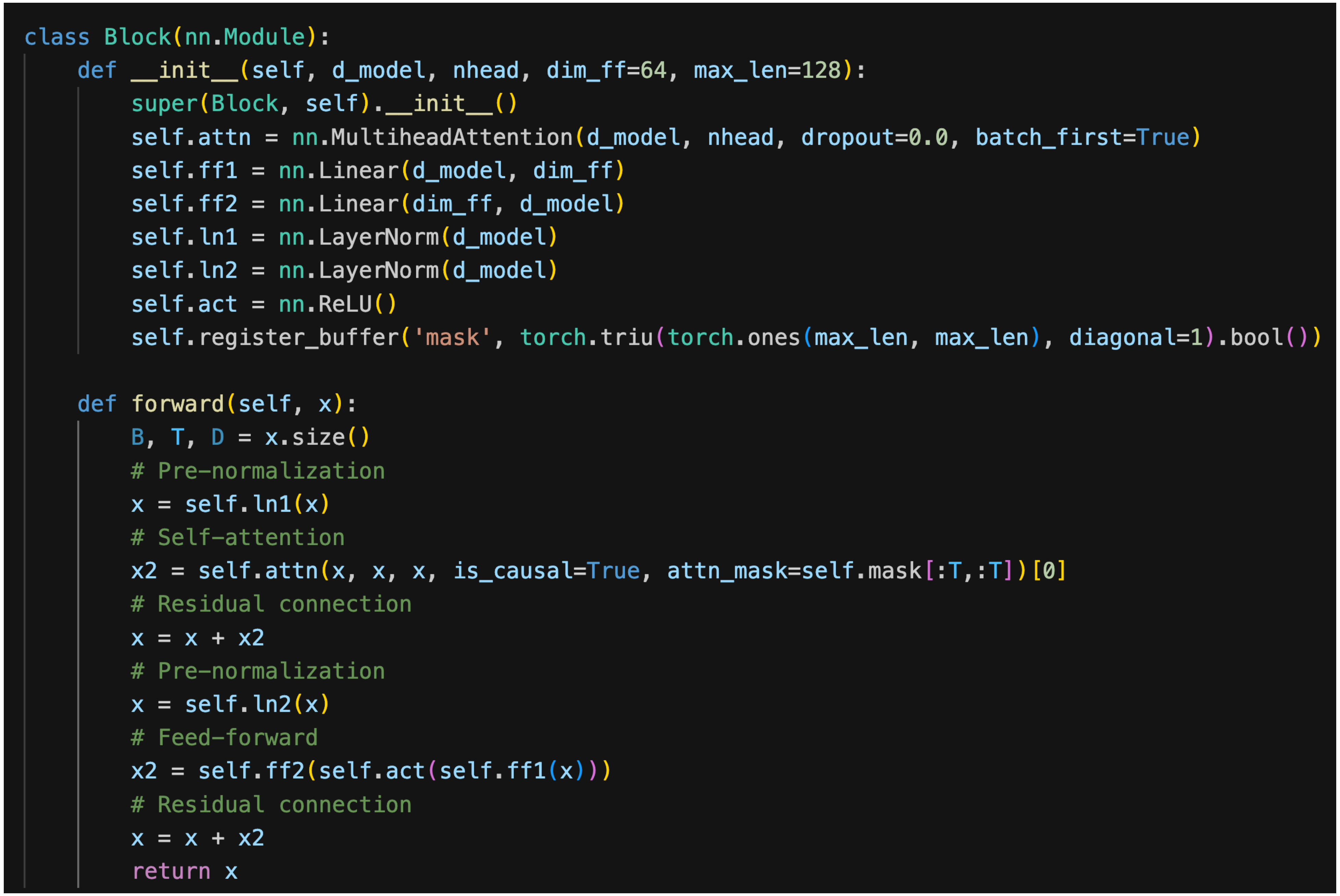
Linear2



In code

https://github.com/cmu-l3/anlp-spring2025-code/blob/main/05_transformers/transformer.ipynb

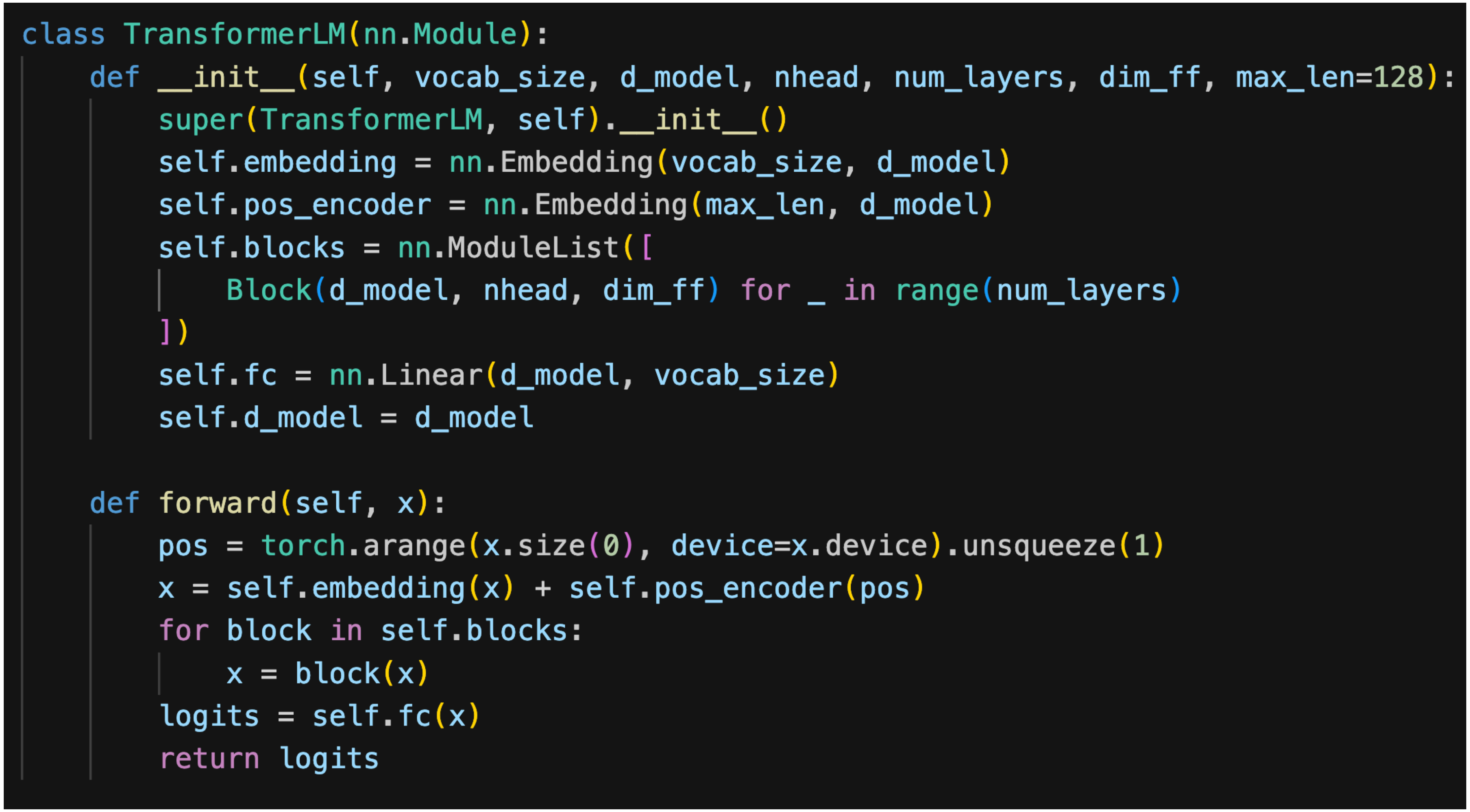
In code



```python
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()
        # Pre-normalization
        x = self.ln1(x)
        # Self-attention
        x2 = self.attn(x, x, x, is_causal=True, attn_mask=self.mask[:T,:T])[0]
        # Residual connection
        x = x + x2
        # Pre-normalization
        x = self.ln2(x)
        # Feed-forward
        x2 = self.ff2(self.act(self.ff1(x)))
        # Residual connection
        x = x + x2
        return x
```

In code



```python
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(0), device=x.device).unsqueeze(1)
        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

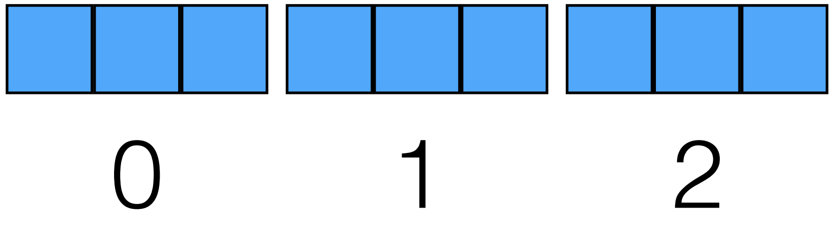
Learned Positional Encoding

(Shaw+ 2018)

- Instead of sinusoidal encodings, just create a learnable embedding

- **Advantages:** flexibility

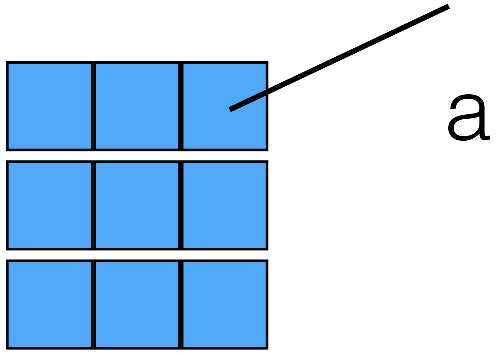
- **Disadvantages:** impossible to extrapolate to longer sequences



Absolute vs. Relative Encodings

(Shaw+ 2018)

- **Absolute** positional encodings add an encoding to the input in *hope* that relative position will be captured
- **Relative** positional encodings *explicitly* encode relative position



"Token 0 and token 2 are 2 - 0 = 2 tokens apart"

Rotary Positional Encodings (RoPE)

(Su+ 2021)

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

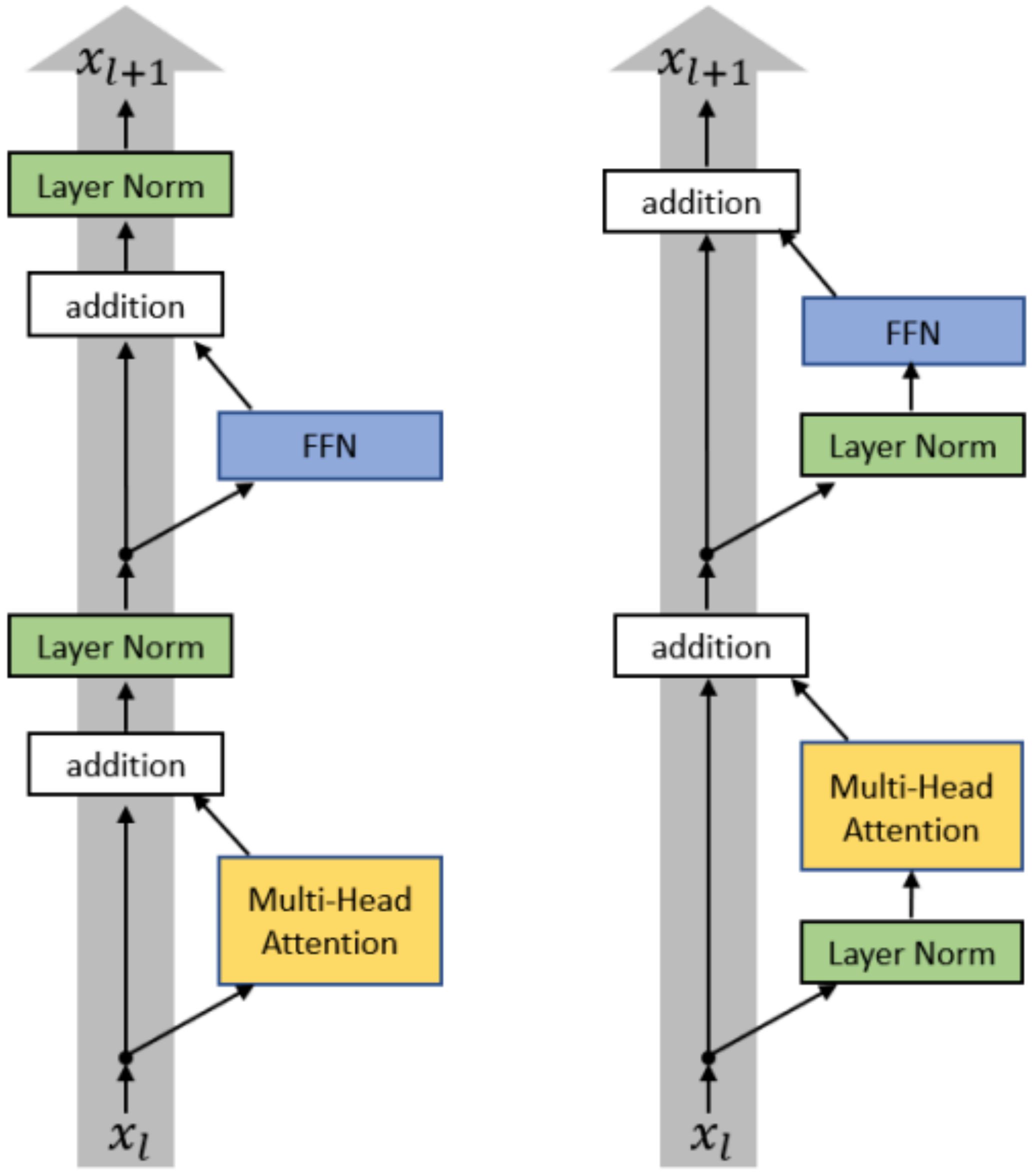
$$f_q(\mathbf{x}_m, m) \cdot f_k(\mathbf{x}_n, n) = g(\mathbf{x}_m, \mathbf{x}_n, m-n)$$

- In summary, RoPE uses trigonometry and imaginary numbers to come up with a function that satisfies this property

$$R^d_{\Theta,m}\mathbf{x} = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ \vdots \\ x_{d-1} \\ x_d \end{pmatrix} \otimes \begin{pmatrix} \cos m\theta_1 \\ \cos m\theta_1 \\ \cos m\theta_2 \\ \cos m\theta_2 \\ \vdots \\ \cos m\theta_{\frac{d}{2}} \\ \cos m\theta_{\frac{d}{2}} \end{pmatrix} + \begin{pmatrix} -x_2 \\ x_1 \\ -x_4 \\ x_3 \\ \vdots \\ -x_d \\ x_{d-1} \end{pmatrix} \otimes \begin{pmatrix} \sin m\theta_1 \\ \sin m\theta_1 \\ \sin m\theta_2 \\ \sin m\theta_2 \\ \vdots \\ \sin m\theta_{\frac{d}{2}} \\ \sin m\theta_{\frac{d}{2}} \end{pmatrix}$$

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

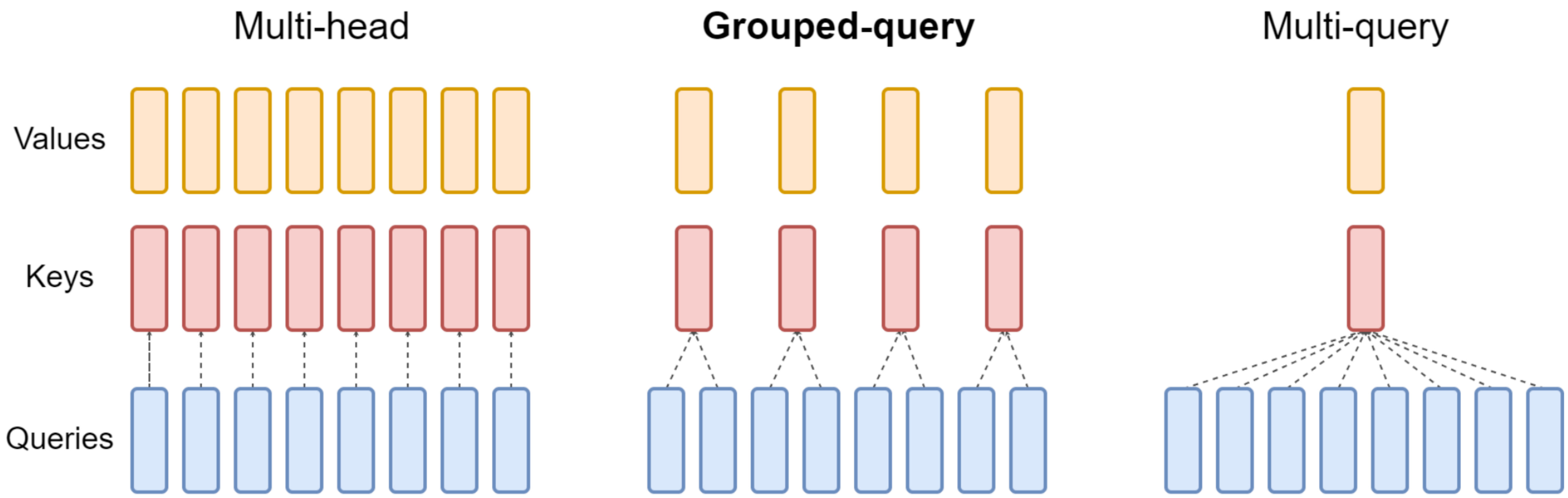
(Zhang and Sennrich 2019)

- Simplifies LayerNorm by removing the mean and bias terms

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

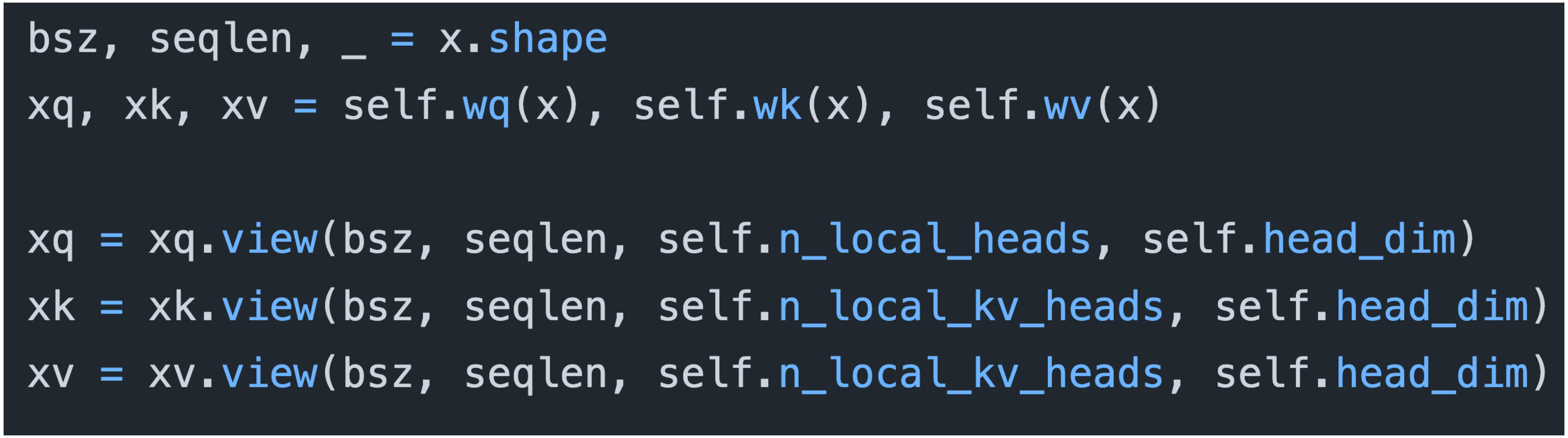
$$\mathrm{RMSNorm}(\mathbf{x}) = \frac{\mathbf{x}}{\mathrm{RMS}(\mathbf{x})} \cdot \mathbf{g}$$

Grouped-query attention



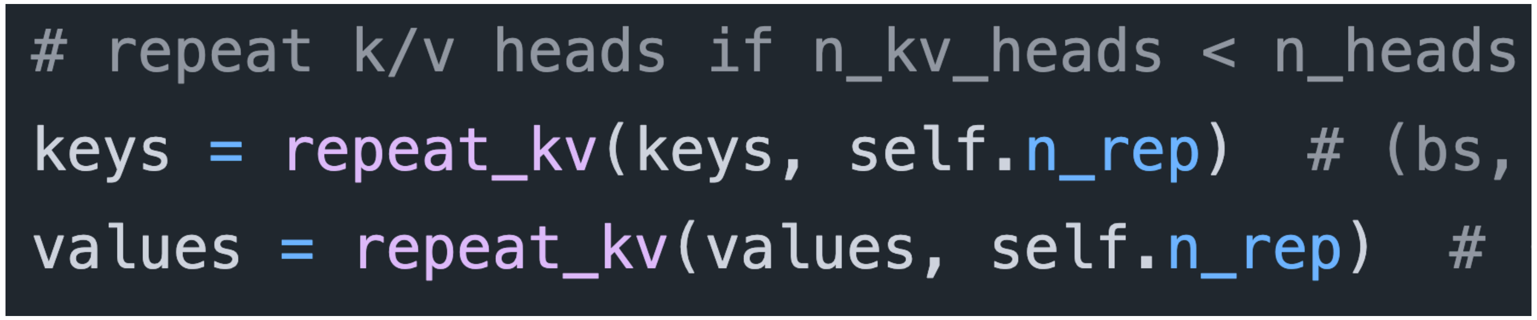
- Shares key and value heads for each *group* of query heads
- Saves on memory, which leads to faster inference

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)  #
```

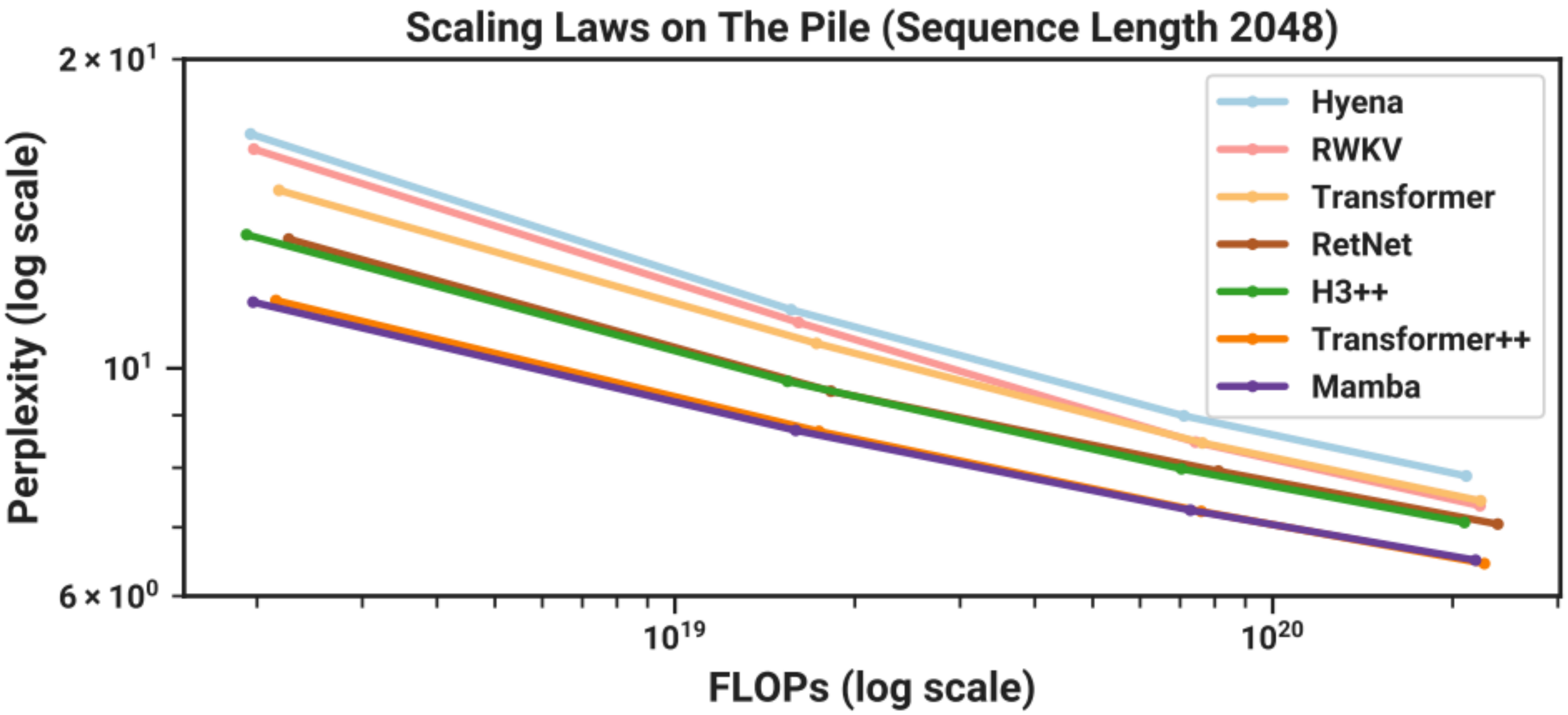
https://github.com/meta-llama/llama/blob/main/llama/model.py

Original Transformer vs. LLama

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

How Important is It?

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



- Stronger architecture is ≈10x more efficient!

Image: Gu and Dao (2023)

Transformer vs RNN

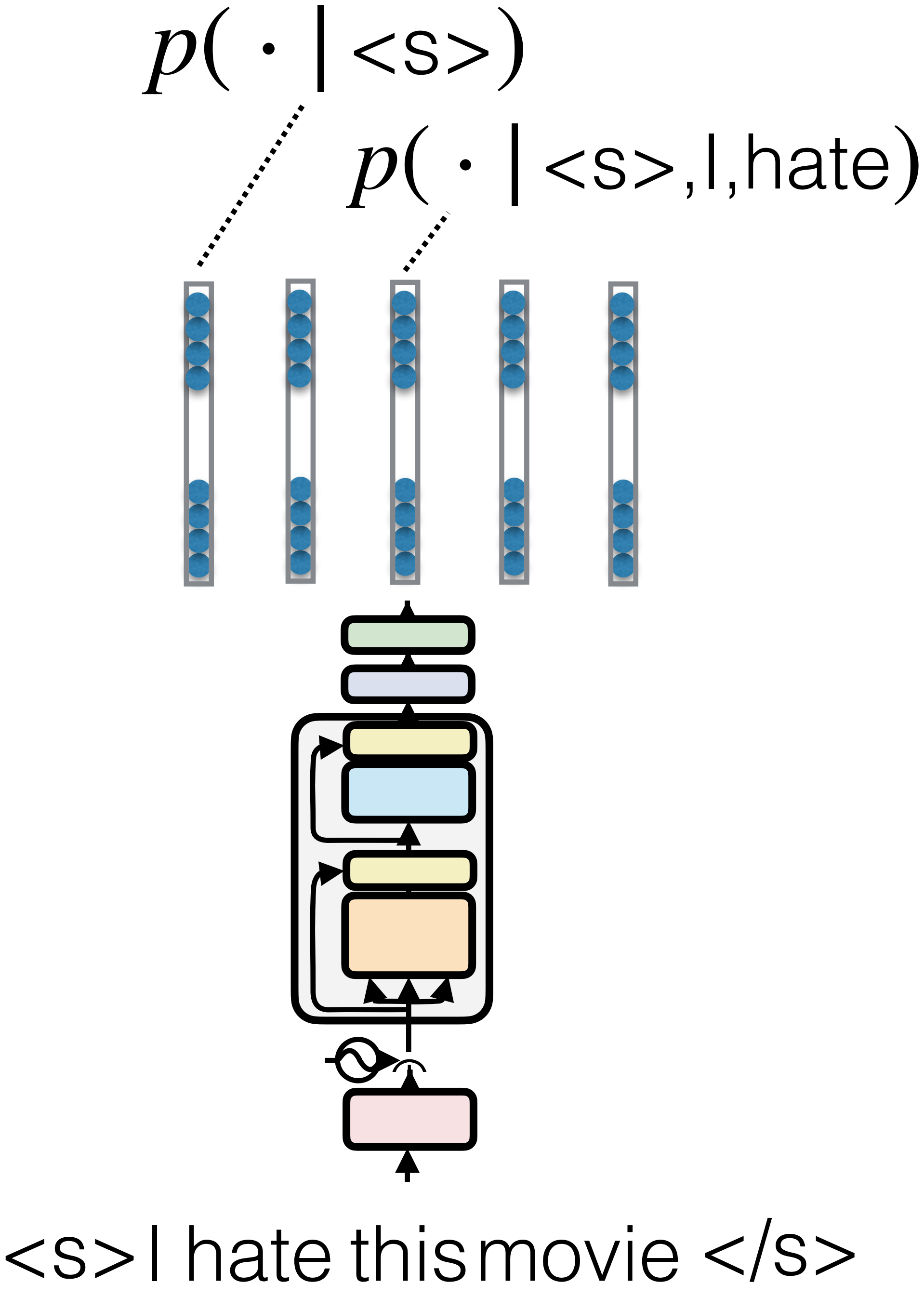
Transformer Training

- 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

- ∴ easy-to-parallelize training



$p(\cdot \mid \text{<s>})$

$p(\cdot \mid \text{<s>}, \text{I}, \text{hate})$

<s> I hate this movie </s>

RNNs vs. Transformers

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

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

Recap

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

Questions?