

COMP 3361 Natural Language Processing

Tutorial #2: Transformer and Its Implementation

Agenda

- Transformer and Its Implementation
- FAQ

Reading Materials

- **The Illustrated Transformer**

- <https://jalammar.github.io/illustrated-transformer/>

- **The Annotated Transformer**

- <https://nlp.seas.harvard.edu/2018/04/03/attention.html>

- **Attention Is All You Need**

- <https://arxiv.org/abs/1706.03762>

- **NanoGPT**

- <https://github.com/karpathy/nanoGPT>

- https://colab.research.google.com/drive/1JMLa53HDuA-i7ZBmqV7ZnA3c_fvtXnx-

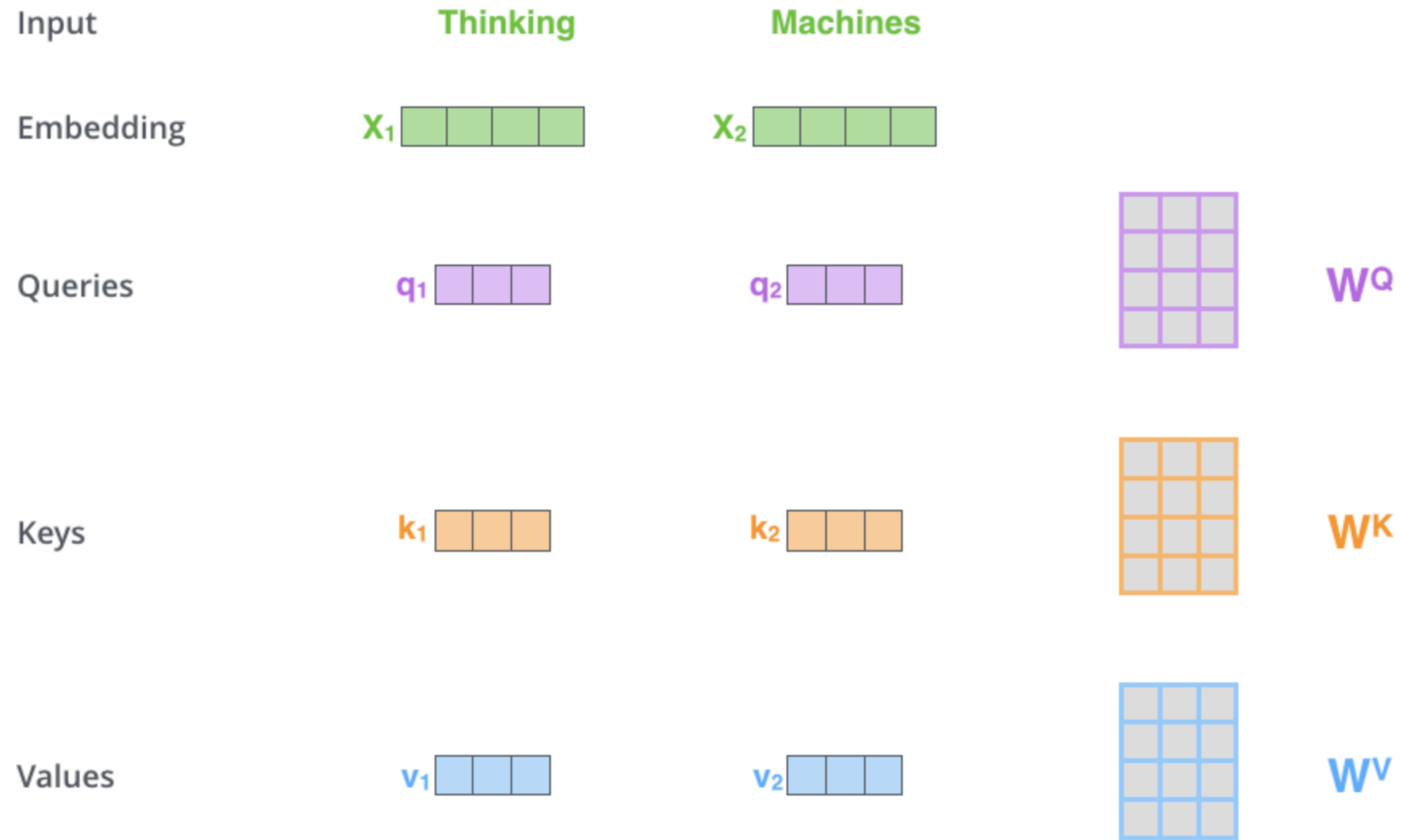
Multi-Head Self-Attention

Multi-Head Self-Attention

- Let $E = [\text{sent len, embedding dim}]$ be the input sentence. This will be passed through three different linear layers to produce three mats:
 - Query $Q = EW^Q$: each token “chooses” what to attend to
 - Keys $K = EW^K$: these control what each token looks like as a “target”
 - Values $V = EW^V$: these vectors get summed up to form the output

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

Self-Attention



Self-Attention



sent len x sent len
(attn for each word to each other)

$$\text{softmax} \left(\frac{Q \times K^T}{\sqrt{d_k}} \right) V$$

Z

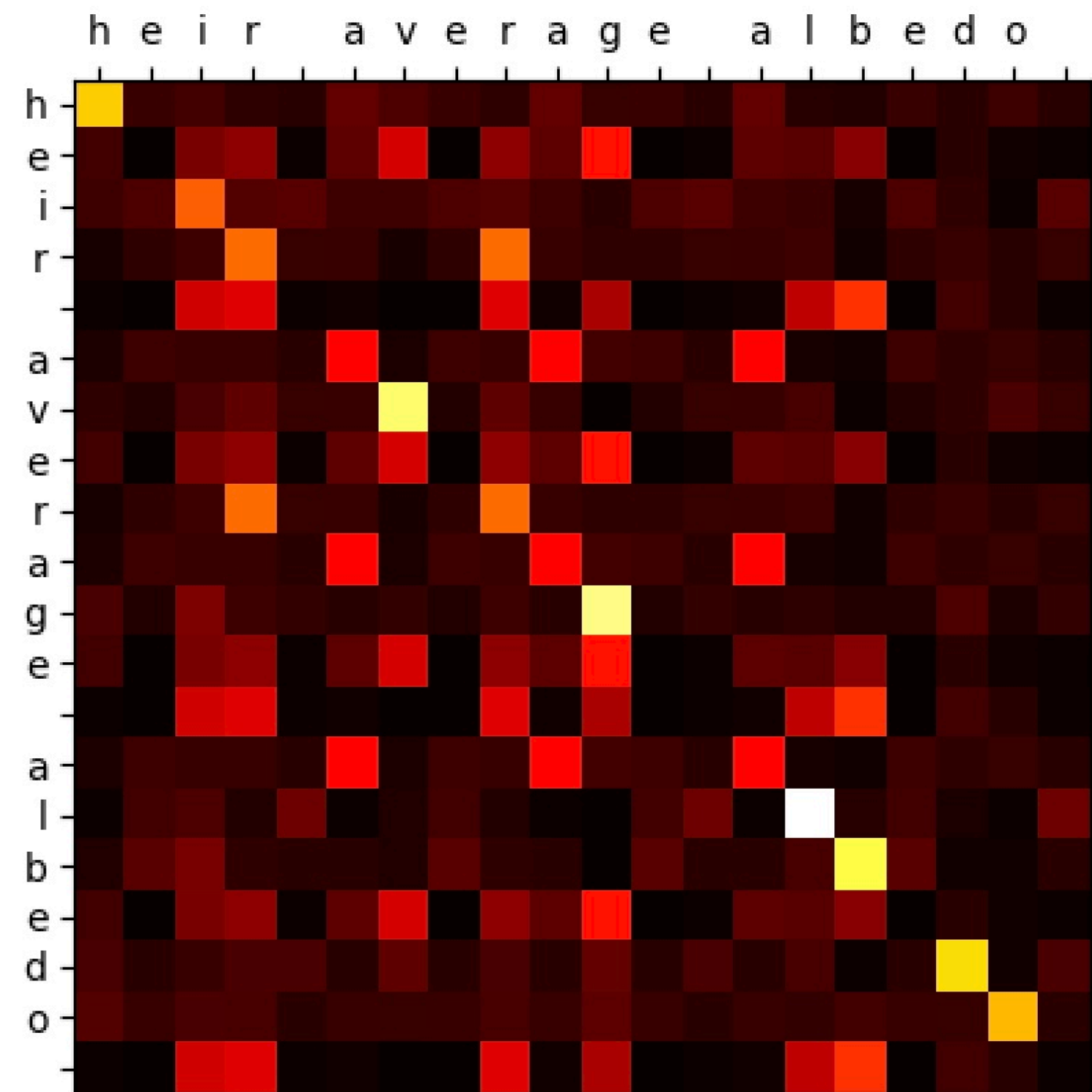
sent len x hidden dim
Z is a weighted combination of V rows

Self-Attention

```
def attention(query, key, value, mask=None, dropout=None):
    "Compute 'Scaled Dot Product Attention'"
    d_k = query.size(-1)
    scores = torch.matmul(query, key.transpose(-2, -1)) / math.sqrt(d_k)
    if mask is not None:
        scores = scores.masked_fill(mask == 0, -1e9)
    p_attn = scores.softmax(dim=-1)
    if dropout is not None:
        p_attn = dropout(p_attn)
    return torch.matmul(p_attn, value), p_attn
```

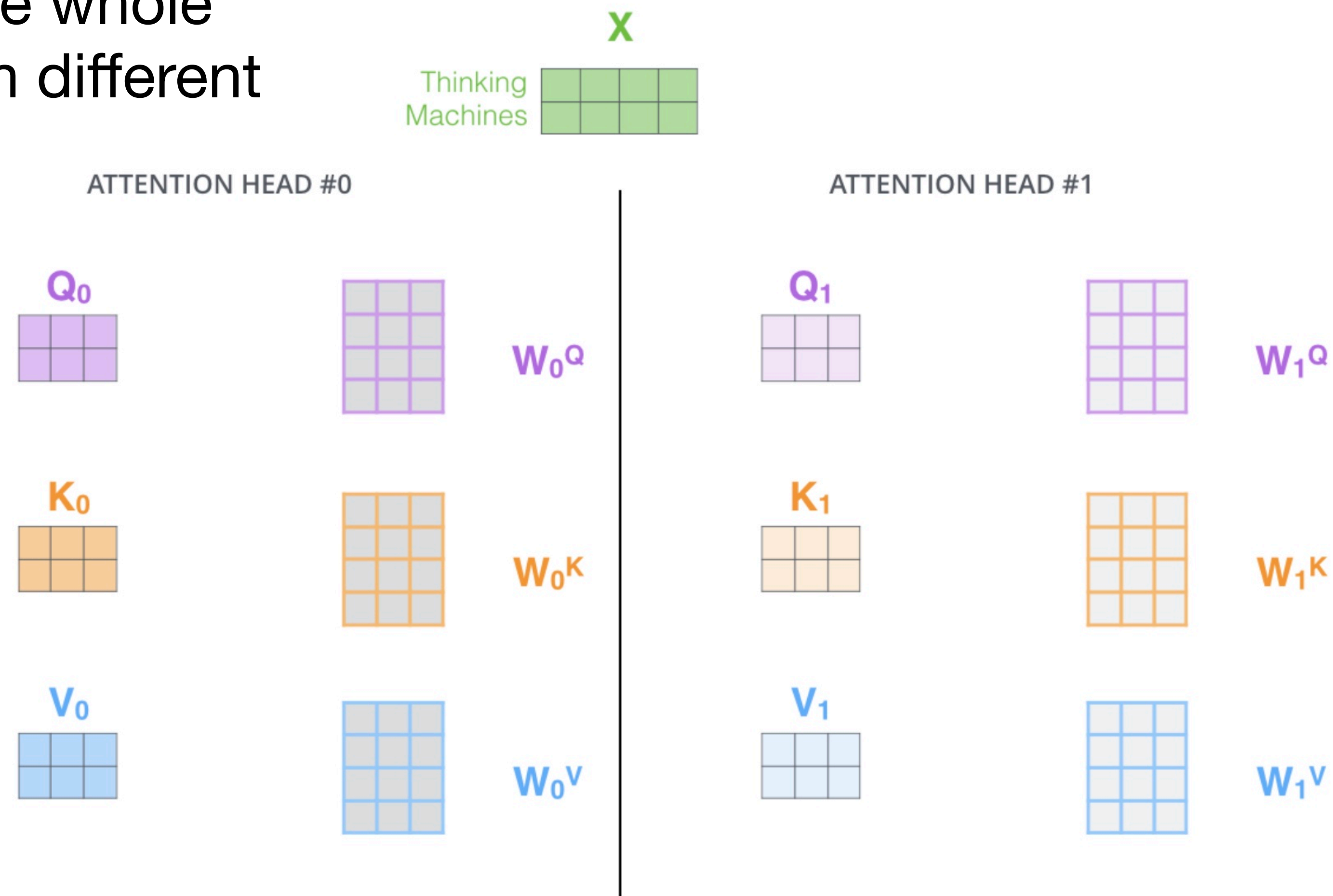

Attention Maps

- Example visualization of attention matrix A (from assignment)
- Each row: distribution over what that token attends to. E.g., the first “v” attends very heavily to itself (bright yellow box)
- **Your task on the HW: assess if the attentions make sense**



Multi-Head Self-Attention

Just duplicate the whole computation with different weights:



Multi-Head Self-Attention

1) This is our input sentence*

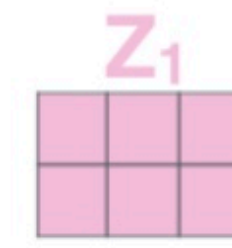
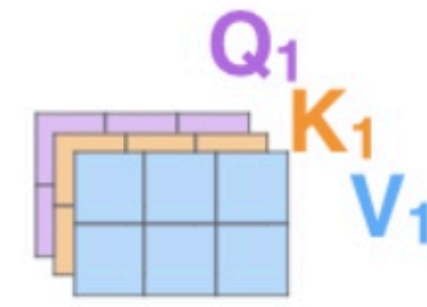
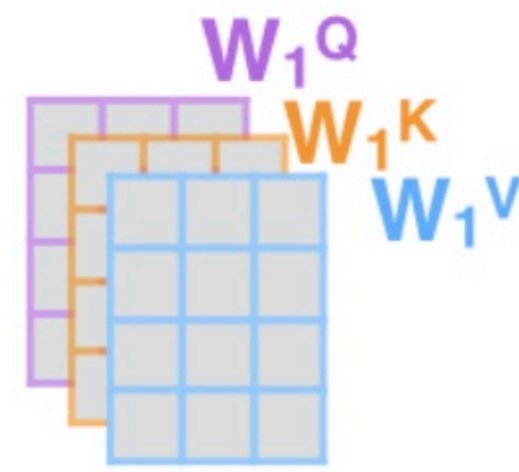
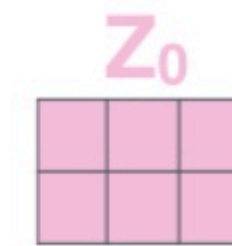
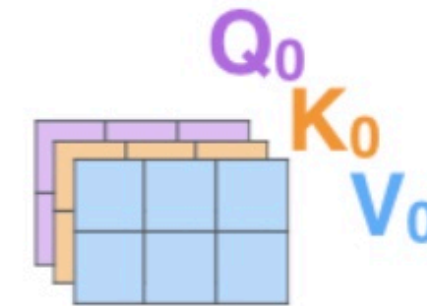
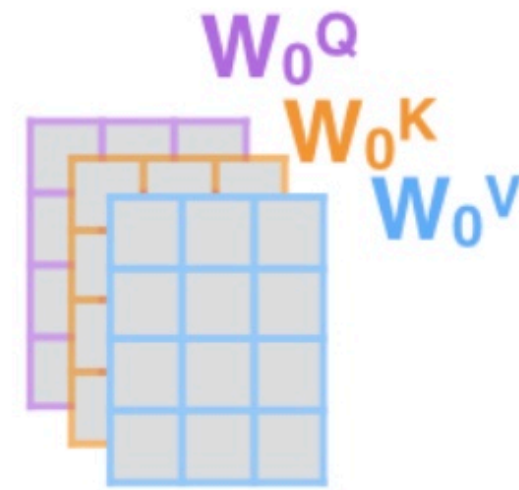
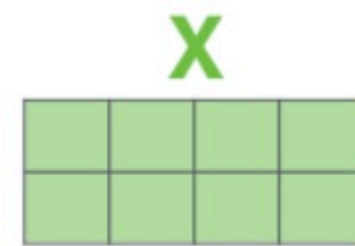
2) We embed each word*

3) Split into 8 heads. We multiply X or R with weight matrices

4) Calculate attention using the resulting $Q/K/V$ matrices

5) Concatenate the resulting Z matrices, then multiply with weight matrix W^O to produce the output of the layer

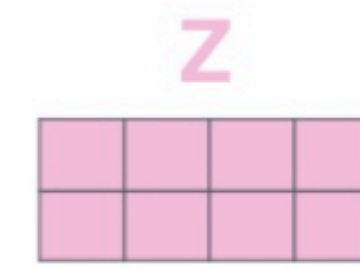
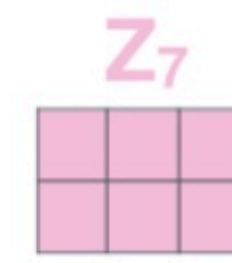
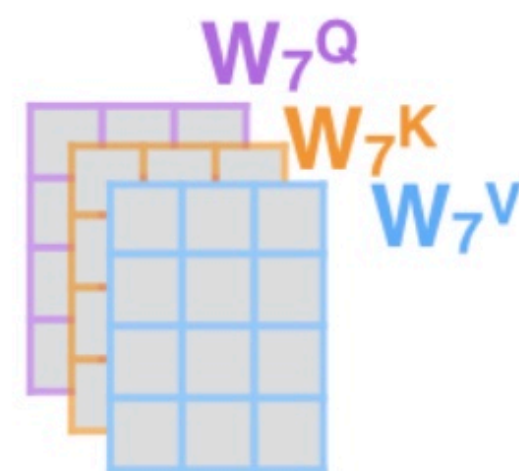
Thinking Machines



...

...

...



* In all encoders other than #0, we don't need embedding. We start directly with the output of the encoder right below this one



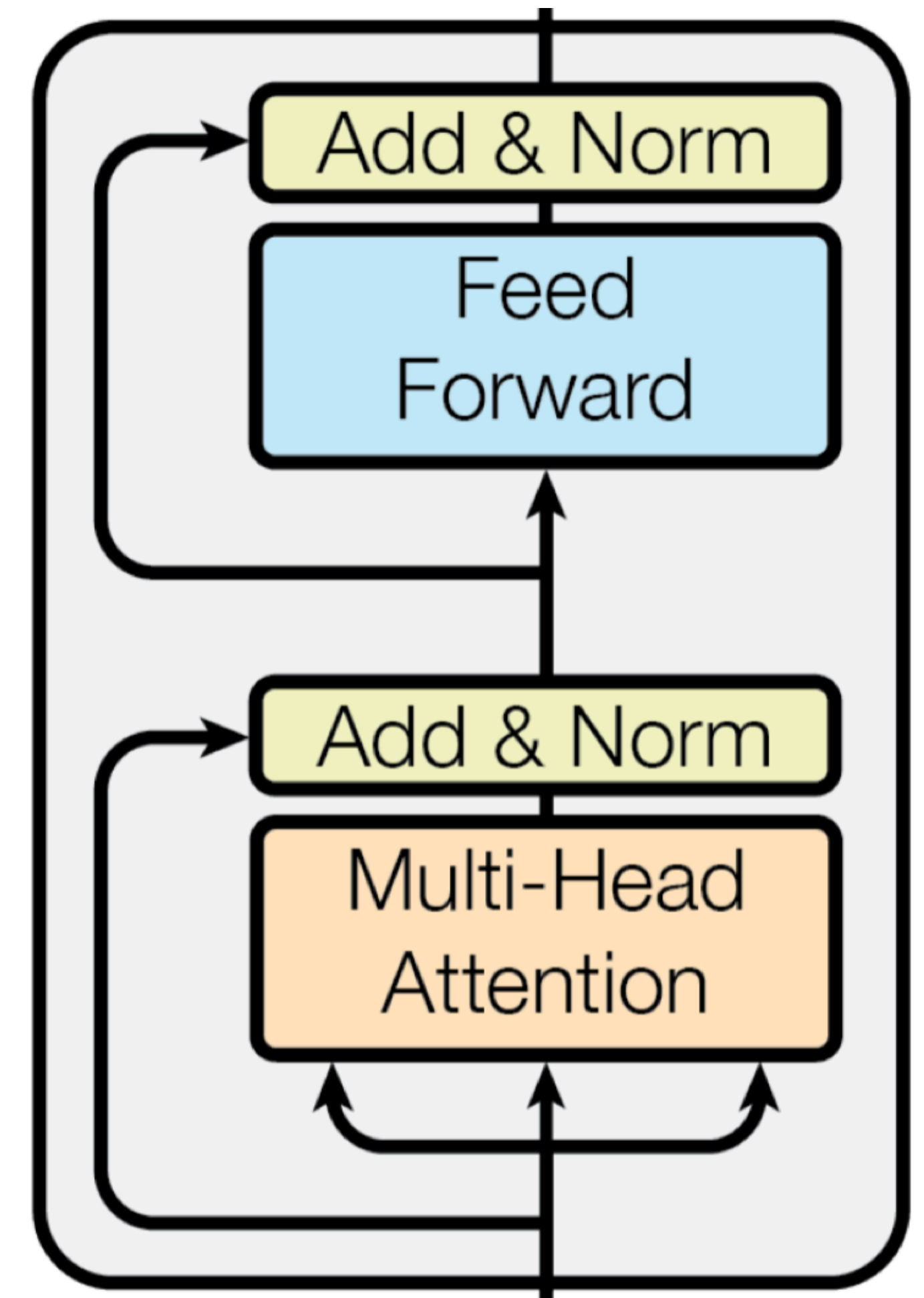
Transformers

Architecture

- Alternate multi-head self-attention with feedforward layers that operate over each word individually

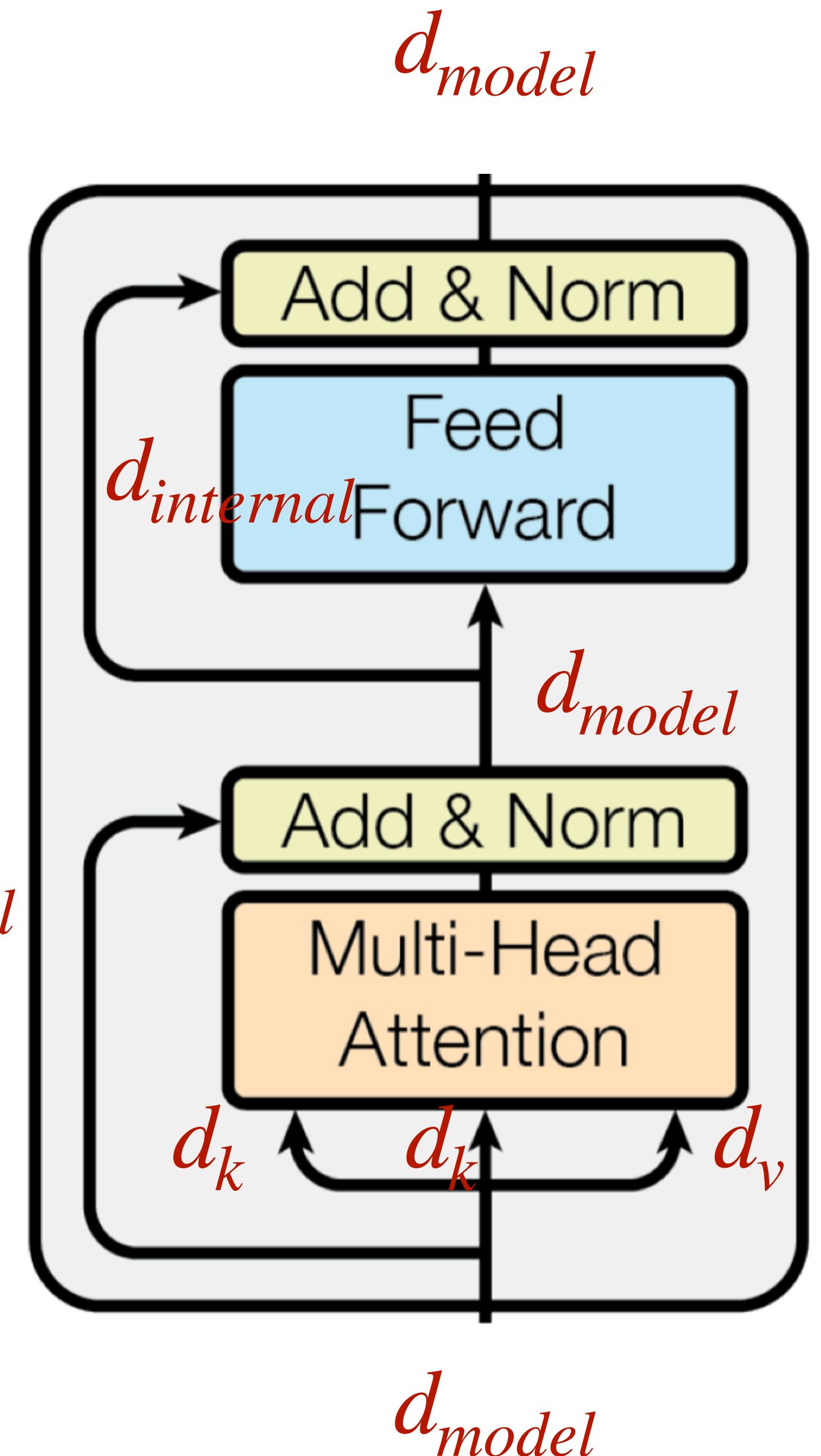
$$\text{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2$$

- These feedforward layers are where most of the parameters are
- Residual connections in the model: input of a layer is added to its output
- Layer normalization: controls the scale of different layers in very deep networks (**not needed in A2**)



Dimensions

- Vectors: d_{model}
- Queries/Keys: d_k , always smaller than d_{model}
- Values: separate dimension d_v , output is multiplied by W^O which is $d_v \times d_{model}$ so we can get back to d_{model} before the residual
- FFN can explode the dimension with W_1 and collapse it back with W_2
- $$\text{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2$$



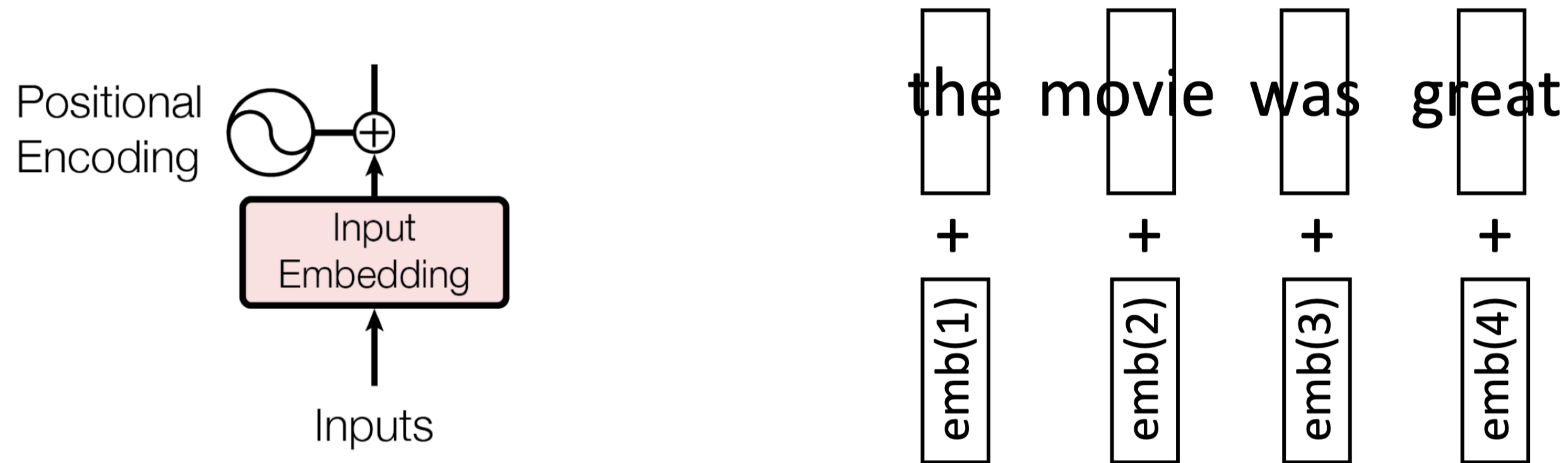
FFN Layer

```
class PositionwiseFeedForward(nn.Module):
    "Implements FFN equation."

    def __init__(self, d_model, d_ff, dropout=0.1):
        super(PositionwiseFeedForward, self).__init__()
        self.w_1 = nn.Linear(d_model, d_ff)
        self.w_2 = nn.Linear(d_ff, d_model)
        self.dropout = nn.Dropout(dropout)

    def forward(self, x):
        return self.w_2(self.dropout(self.w_1(x).relu()))
```

Transformers: Position Sensitivity

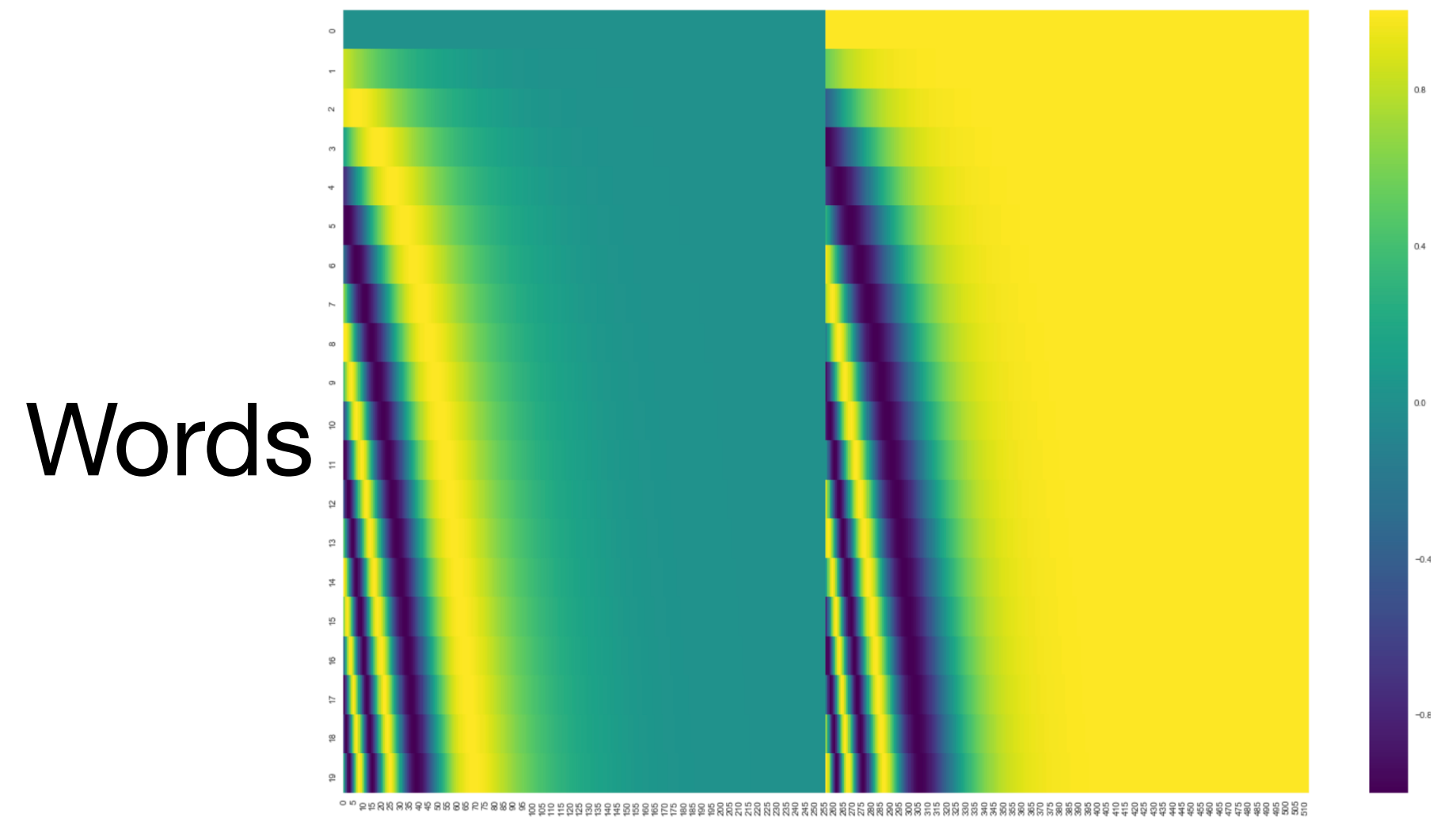


- Transformers have no notion of position by default
- Encode each sequence position as an integer, add it to the word embedding vector

Position Encoding

$$PE_{(pos,2i)} = \sin(pos/10000^{2i/d_{\text{model}}})$$

$$PE_{(pos,2i+1)} = \cos(pos/10000^{2i/d_{\text{model}}})$$



- Where **pos** is the position and **i** is the dimension.
- That is, each dimension of the positional encoding corresponds to a sinusoid. The wavelengths form a geometric progression from 2π to $10000 \cdot 2\pi$

Position Encoding

```
class PositionalEncoding(nn.Module):
    "Implement the PE function."

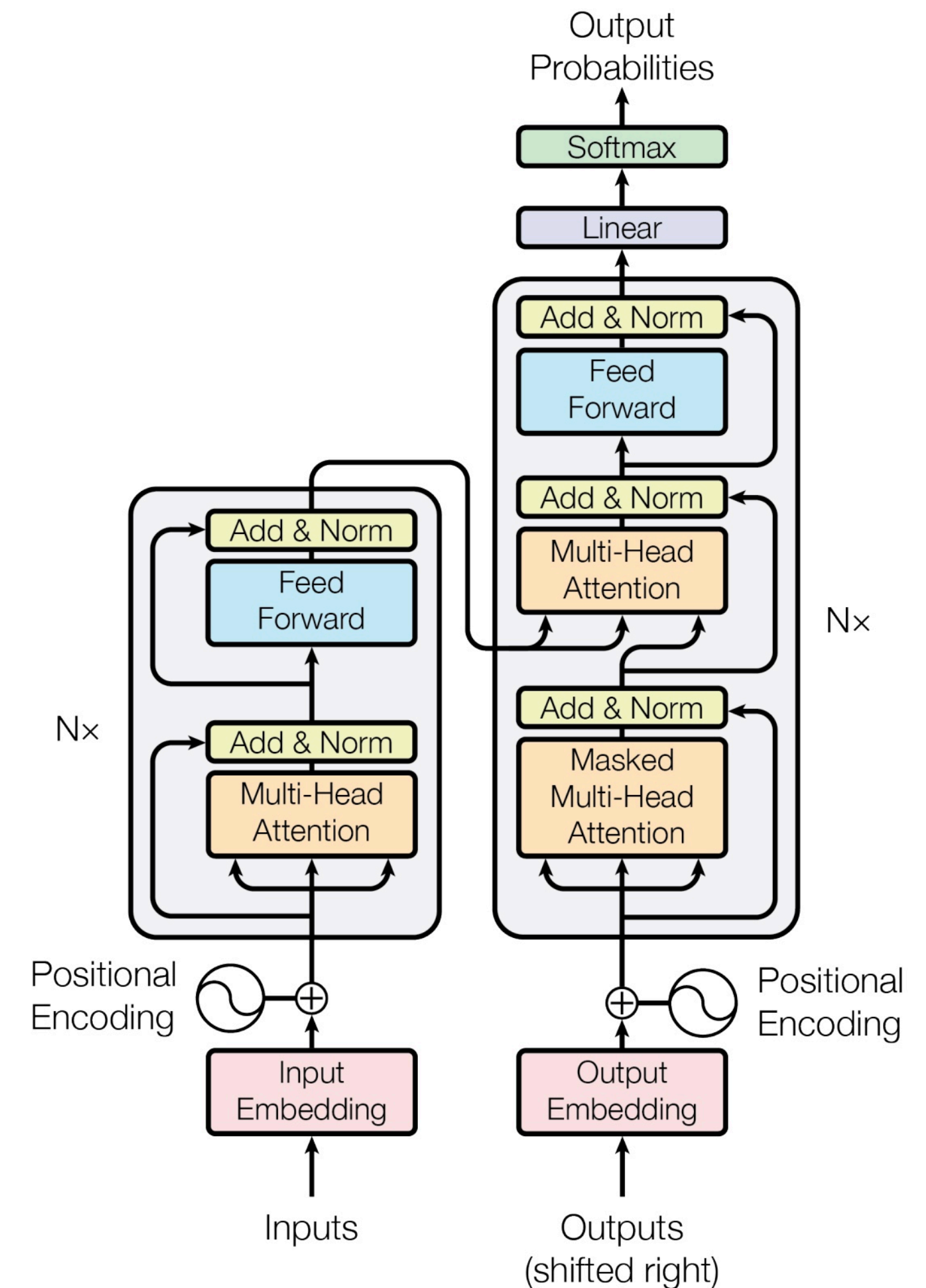
    def __init__(self, d_model, dropout, max_len=5000):
        super(PositionalEncoding, self).__init__()
        self.dropout = nn.Dropout(p=dropout)

        # Compute the positional encodings once in log space.
        pe = torch.zeros(max_len, d_model)
        position = torch.arange(0, max_len).unsqueeze(1)
        div_term = torch.exp(
            torch.arange(0, d_model, 2) * -(math.log(10000.0) / d_model)
        )
        pe[:, 0::2] = torch.sin(position * div_term)
        pe[:, 1::2] = torch.cos(position * div_term)
        pe = pe.unsqueeze(0)
        self.register_buffer("pe", pe)

    def forward(self, x):
        x = x + self.pe[:, : x.size(1)].requires_grad_(False)
        return self.dropout(x)
```

Transformers: Complete Model

- Original Transformer paper presents an **encoder-decoder** model
- In this assignment we don't need to think about both of these parts.
- Turn the encoder into a decoder-only model through use of a triangular causal attention mask (only allow attend on to previous tokens)



```

class Encoder(nn.Module):
    "Core encoder is a stack of N layers"

    def __init__(self, layer, N):
        super(Encoder, self).__init__()
        self.layers = clones(layer, N)
        self.norm = LayerNorm(layer.size)

    def forward(self, x, mask):
        "Pass the input (and mask) through each layer in turn."
        for layer in self.layers:
            x = layer(x, mask)
        return self.norm(x)

```

```

class EncoderLayer(nn.Module):
    "Encoder is made up of self-attn and feed forward (defined below)"

    def __init__(self, size, self_attn, feed_forward, dropout):
        super(EncoderLayer, self).__init__()
        self.self_attn = self_attn
        self.feed_forward = feed_forward
        self.sublayer = clones(SublayerConnection(size, dropout), 2)
        self.size = size

    def forward(self, x, mask):
        "Follow Figure 1 (left) for connections."
        x = self.sublayer[0](x, lambda x: self.self_attn(x, x, x, mask))
        return self.sublayer[1](x, self.feed_forward)

```

```

class SublayerConnection(nn.Module):
    """
    A residual connection followed by a layer norm.
    Note for code simplicity the norm is first as opposed to last.
    """

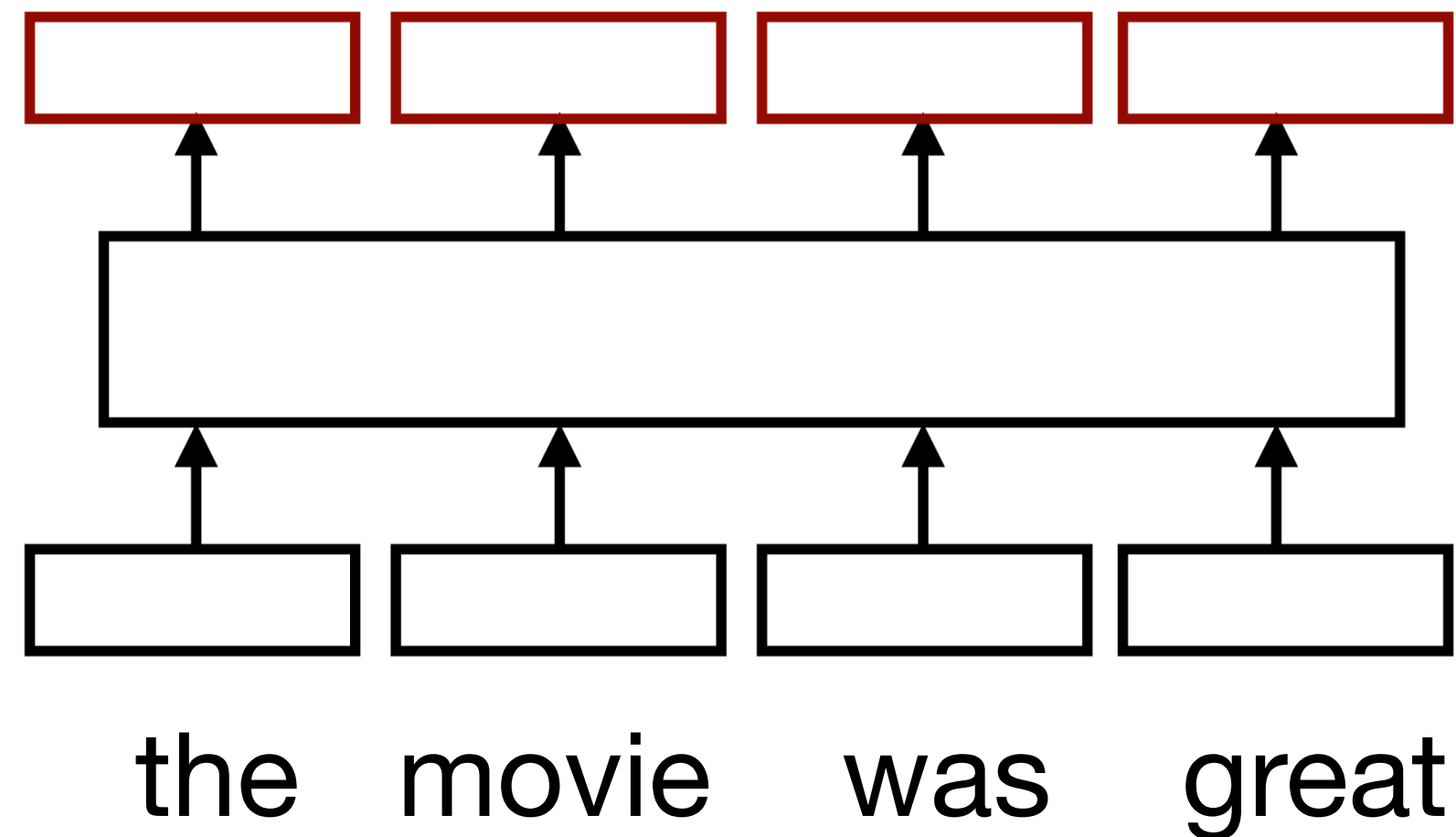
    def __init__(self, size, dropout):
        super(SublayerConnection, self).__init__()
        self.norm = LayerNorm(size)
        self.dropout = nn.Dropout(dropout)

    def forward(self, x, sublayer):
        "Apply residual connection to any sublayer with the same size."
        return x + self.dropout(sublayer(self.norm(x)))

```

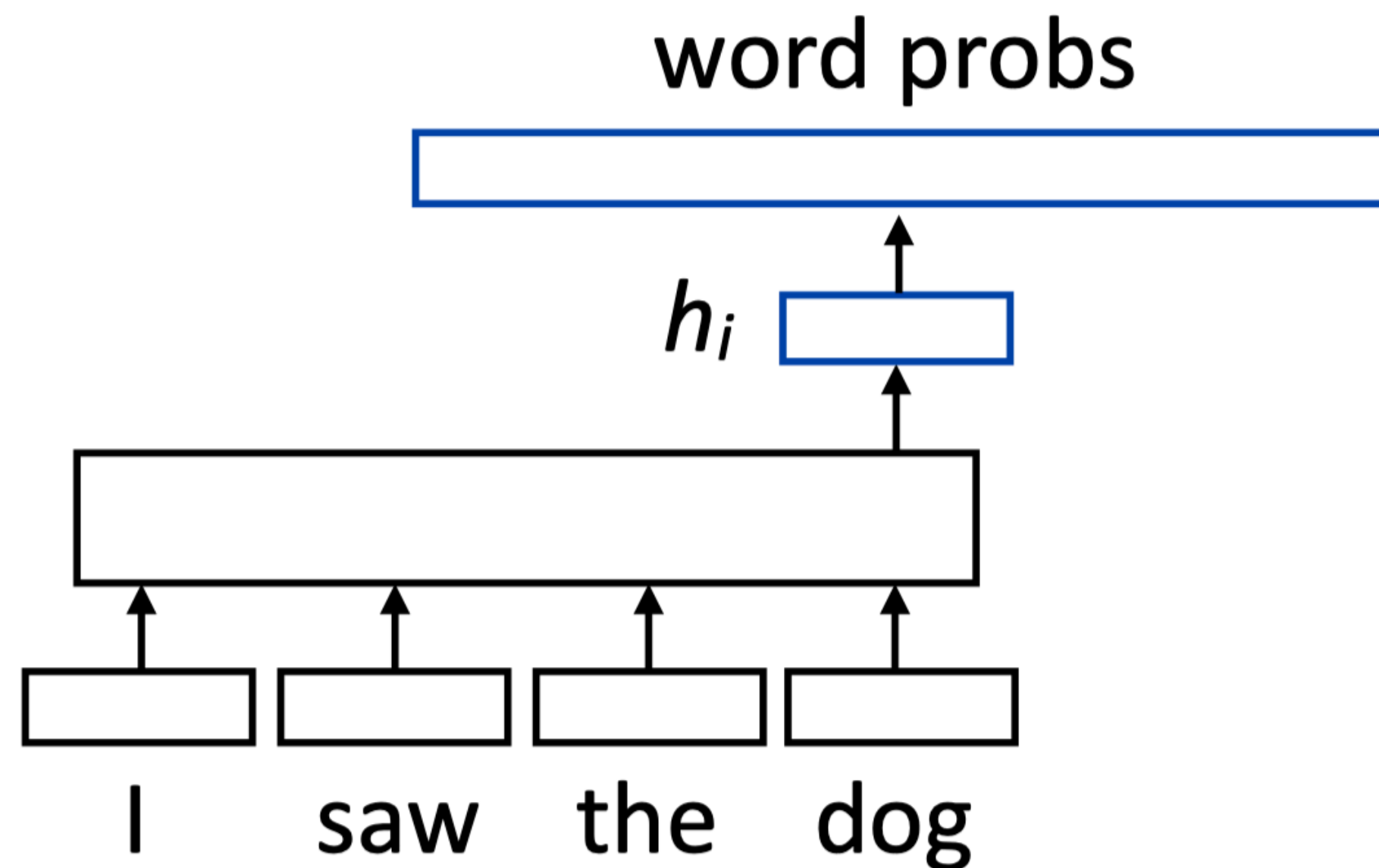
Transformer Language Modeling

What do Transformers produce?



- Encoding of each word — can pass this to another layer to make a prediction (like predicting the next word for language modeling)
- Like RNNs, Transformers can be viewed as a transformation of a sequence of vectors into a sequence of context-dependent vectors

Transformer Language Modeling



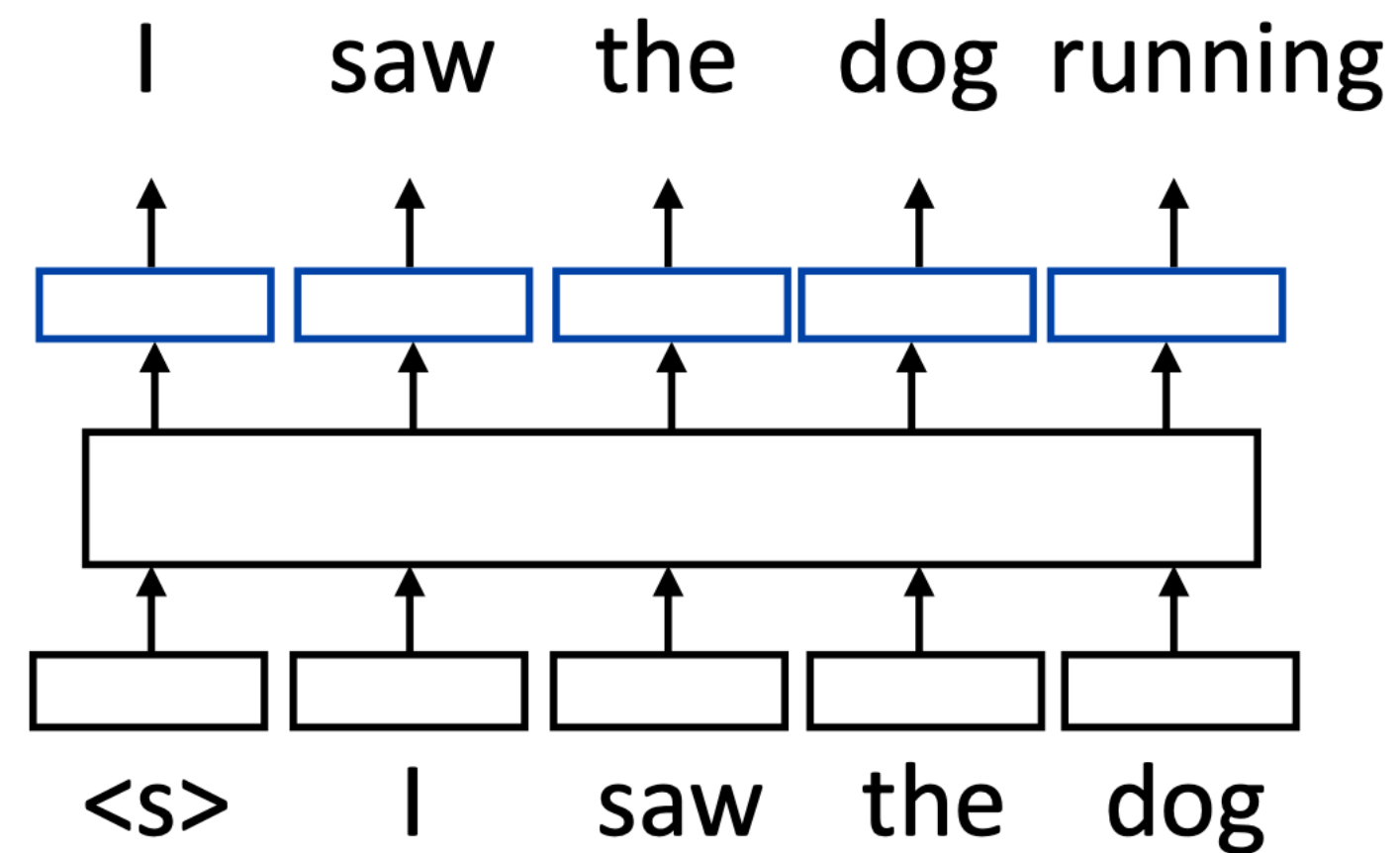
$$P(w|\text{context}) = \frac{\exp(\mathbf{w} \cdot \mathbf{h}_i)}{\sum_{w'} \exp(\mathbf{w}' \cdot \mathbf{h}_i)}$$

equivalent to

$$P(w|\text{context}) = \text{softmax}(W\mathbf{h}_i)$$

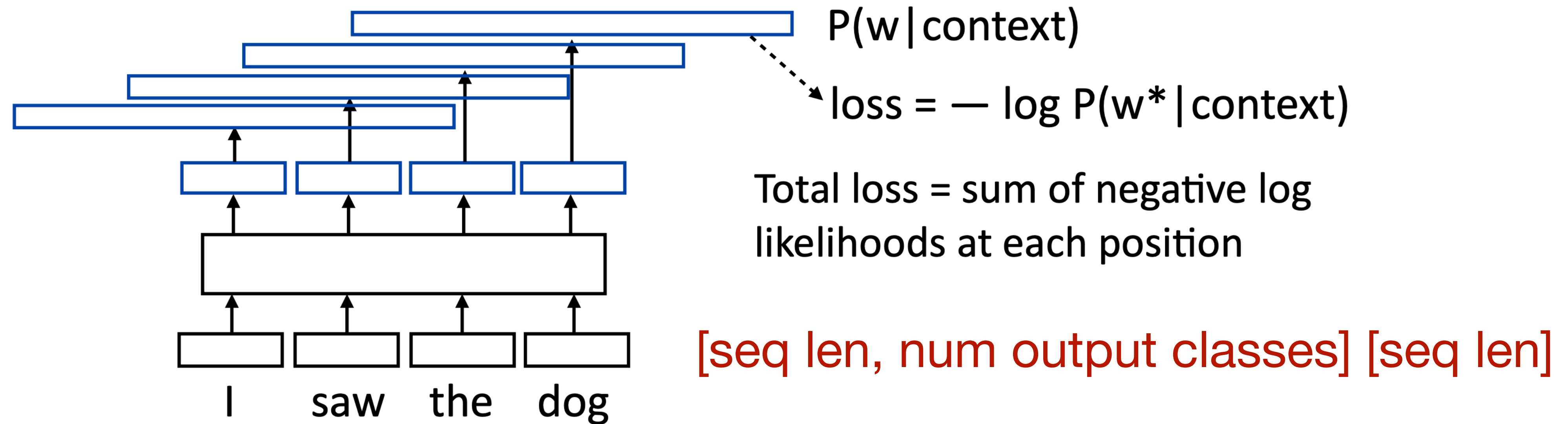
- W is a (vocab size) x (hidden size) matrix; **nn.Linear** in PyTorch (rows are word embeddings)

Training Transformer LMs



- Input is a sequence of words, output is those words shifted by one,
- Allows us to train on predictions across several timesteps simultaneously (similar to batching but this is NOT what we refer to as batching)

Training Transformer LMs



```
loss_fcn = nn.NLLLoss()
```

```
loss += loss_fcn(log_probs, ex.output_tensor)
```

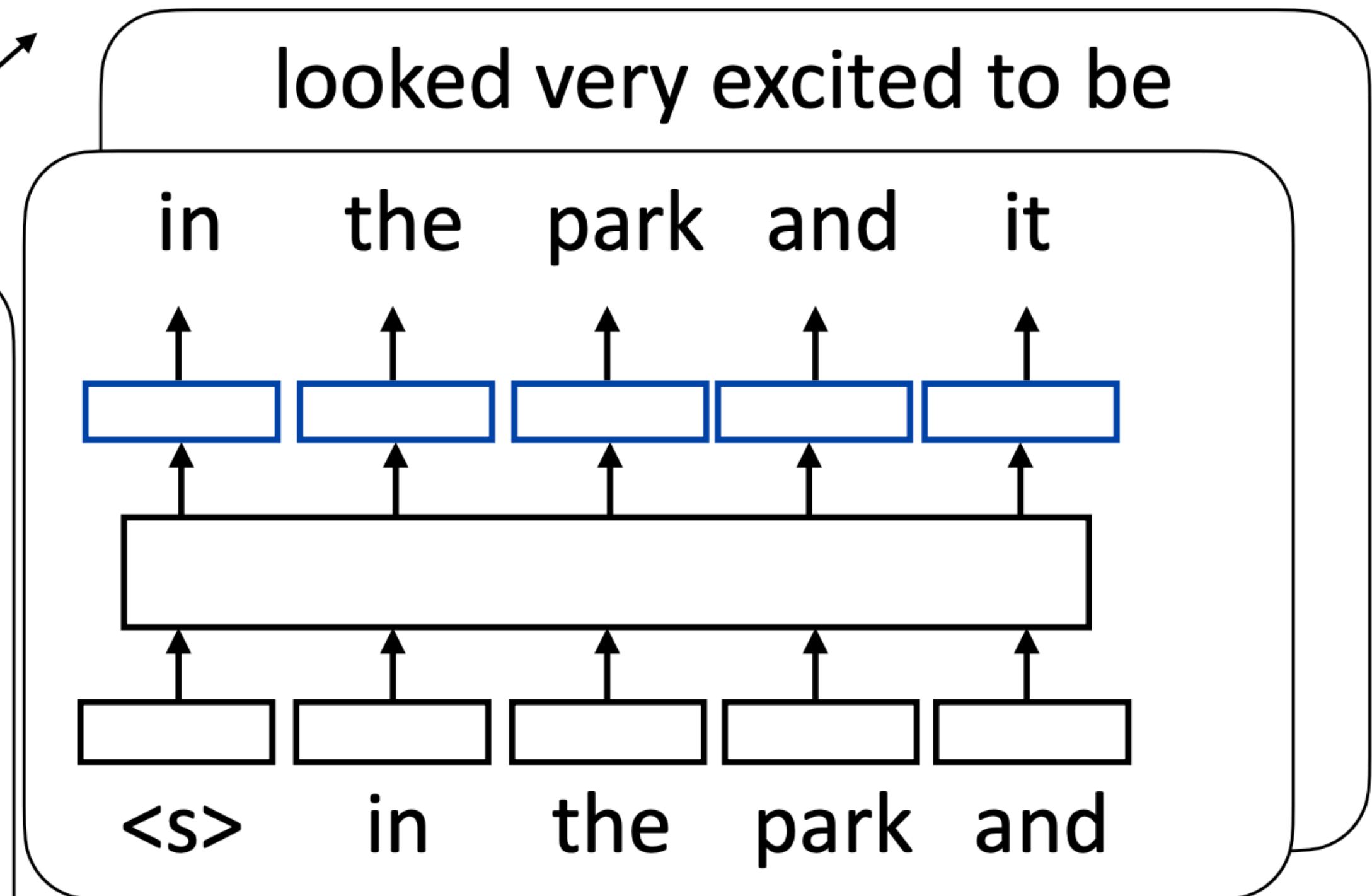
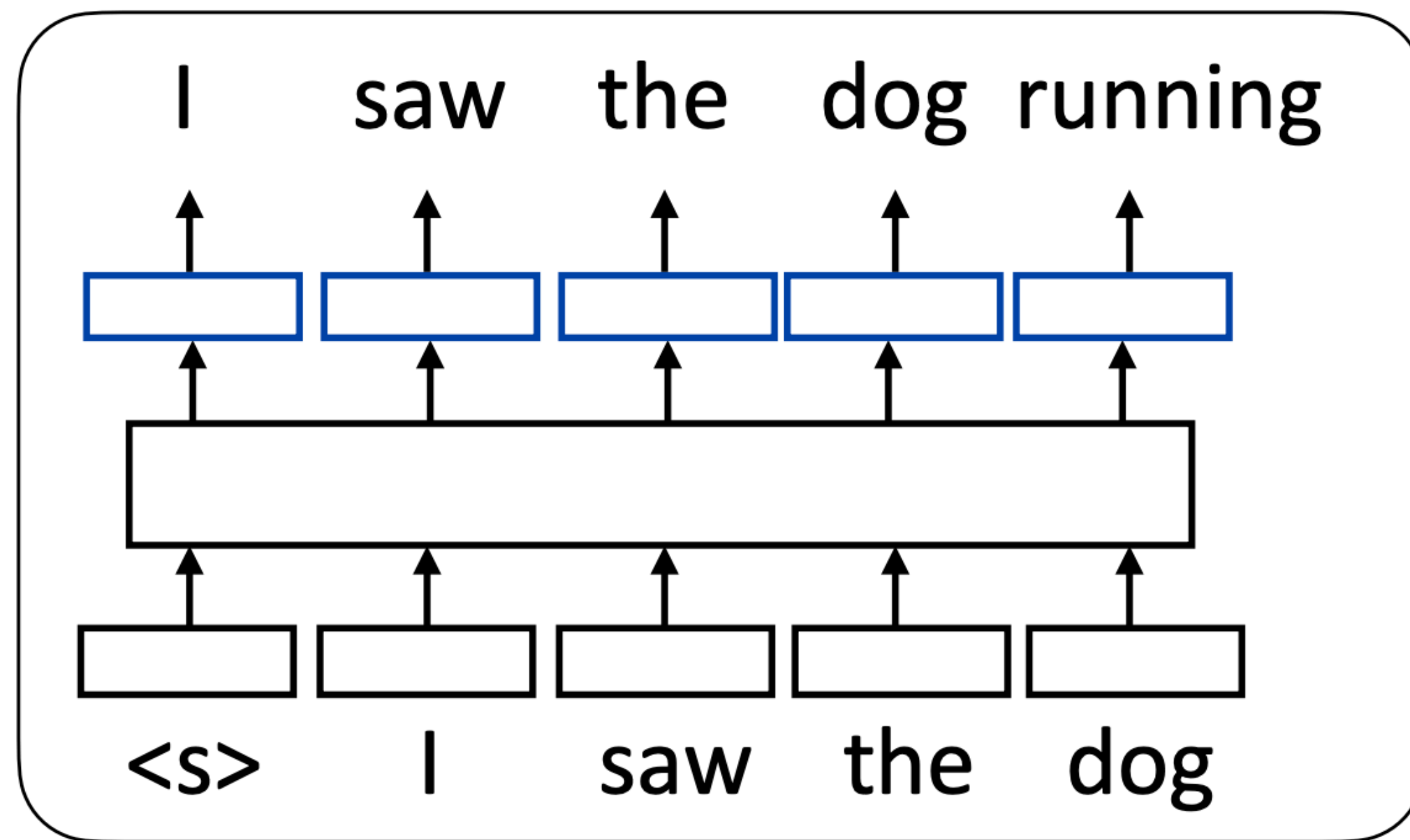
$[\text{seq len}, \text{num classes}]$ $[\text{seq len}]$

- Batching is a little tricky with NLLLoss: need to collate $[\text{batch}, \text{seq len}, \text{num classes}]$ to $[\text{batch} * \text{seq len}, \text{num classes}]$. You do not need to batch

Batched LM Training

I saw the dog running in the park and it looked very excited to be there

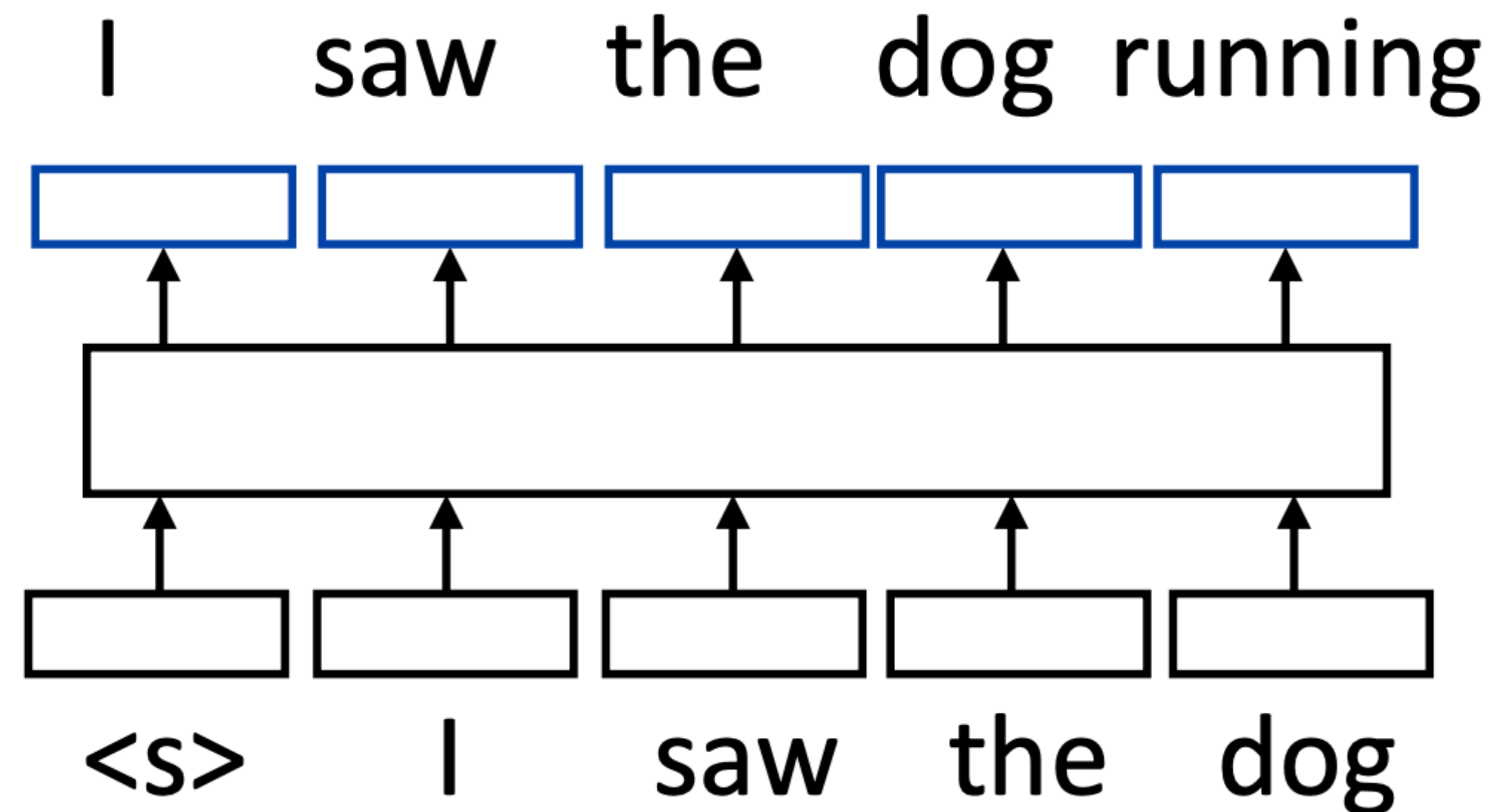
batch dim



- ▶ Multiple sequences **and** multiple timesteps per sequence

A Small Problem with Transformer LMs

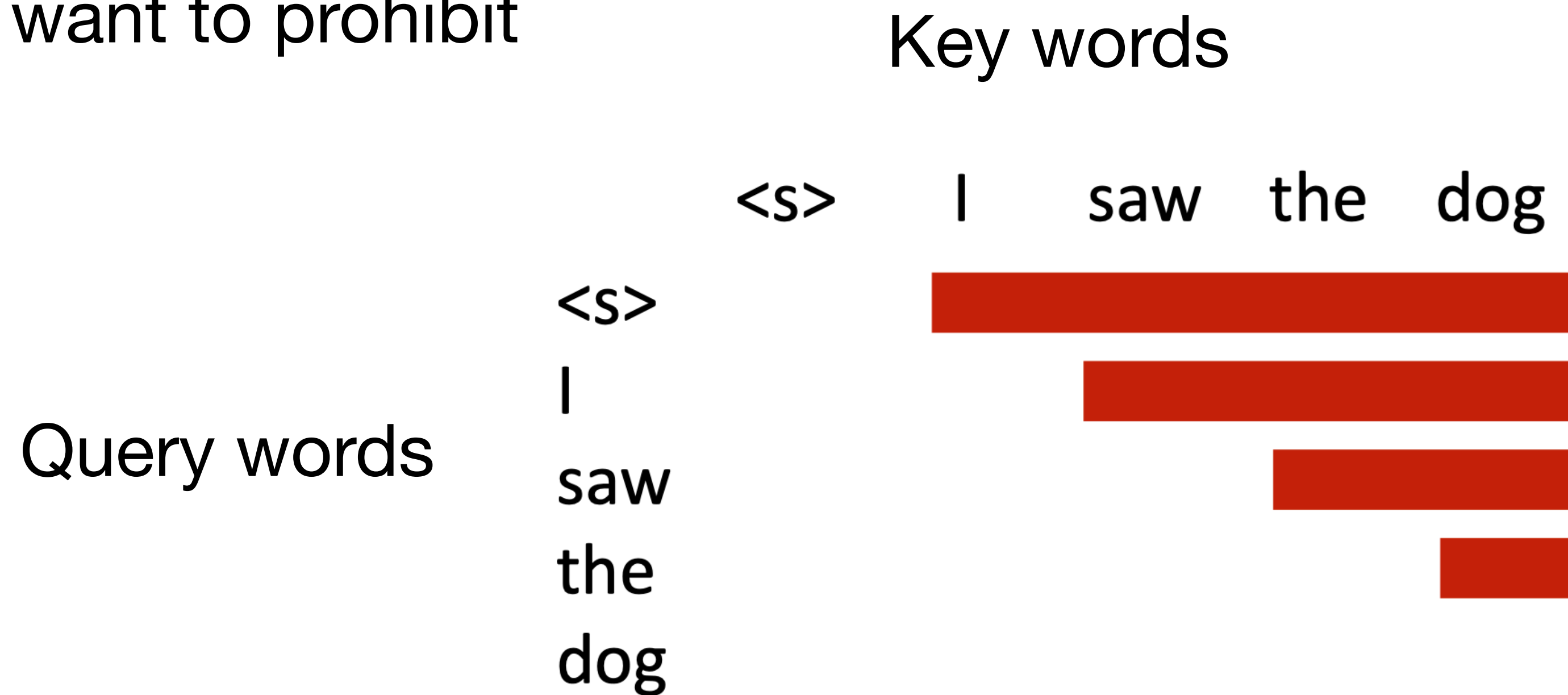
- This Transformer LM as we've described it will easily achieve perfect accuracy. Why?



- With standard self-attention: “I” attends to “saw” and the model is “cheating”. How do we ensure that this doesn't happen?

Attention Masking

- We want to prohibit



- We want to mask out everything in red (an upper triangular matrix)

Implementing in PyTorch

- `nn.TransformerEncoder` can be built out of `nn.TransformerEncoderLayers`, can accept an input and a mask for language modeling:

```
# Inside the module; need to fill in size parameters
```

```
layers = nn.TransformerEncoderLayer(...)
```

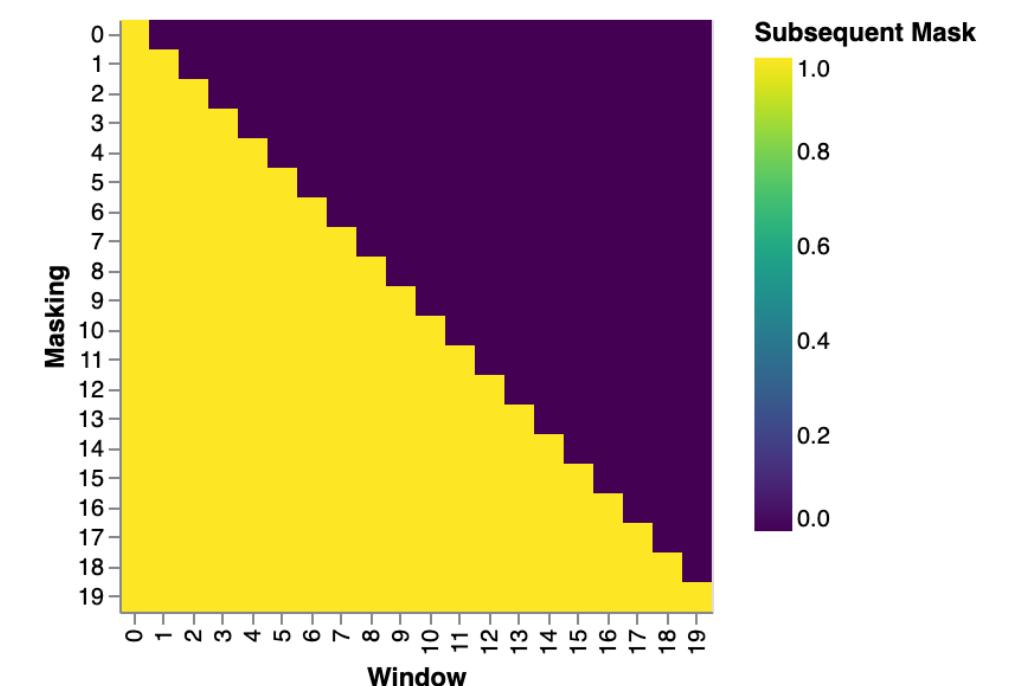
```
transformer_encoder = nn.TransformerEncoder(encoder_layers, num_layers=...)
[...]
```

```
# Inside forward(): puts negative infinities in the red part
```

```
mask = torch.triu(torch.ones(len, len) * float('-inf'), diagonal=1)
```

```
output = transformer_encoder(input, mask=mask)
```

- **You will not use these for Part 1, only for Part 2**



Assignment 2

Assignment 2

- Part 1: Building a “Transformer” Encoder
- Part 2: Transformer for Language Modeling

FAQ: Part 1

- Q: How to turn the text into embedding?
 - A: use `nn.Embedding`
- Q: are we allowed to create or update classes and functions?
 - A: Yes, just make sure your `train_lm` is compatible with the original one.
- Q: Time limitations of autograder?
 - A: Yes we collect these data. Even if your code fails to execute, has a long runtime, or does not reach that number, we still carefully evaluate all solutions and accumulate grades for each correct step.
- Q: what is `d_internal`? Shouldn't it be equal to `d_model`?
 - A: Please refer to previous the Architecture slides.

FAQ: Part 2

- Q: Should we implement encoder-decoder architecture?
 - A: No, please use the `nn.TransformerEncoder` with casual mask.
- Q: Batching
 - A: make sure you create the `nn.TransformerEncoderLayer` with **`batch_first=True`**.