COS 584

Advanced Natural Language Processing

P10: Transformers

Spring 2021

NIPS 2017

Attention Is All You Need

Ashish Vaswani* Google Brain avaswani@google.com

Noam Shazeer* Google Brain noam@google.com

Niki Parmar* Google Research nikip@google.com

Jakob Uszkoreit* Google Research usz@google.com

Llion Jones* Google Research llion@google.com

Aidan N. Gomez* † University of Toronto aidan@cs.toronto.edu

Łukasz Kaiser* Google Brain lukaszkaiser@google.com

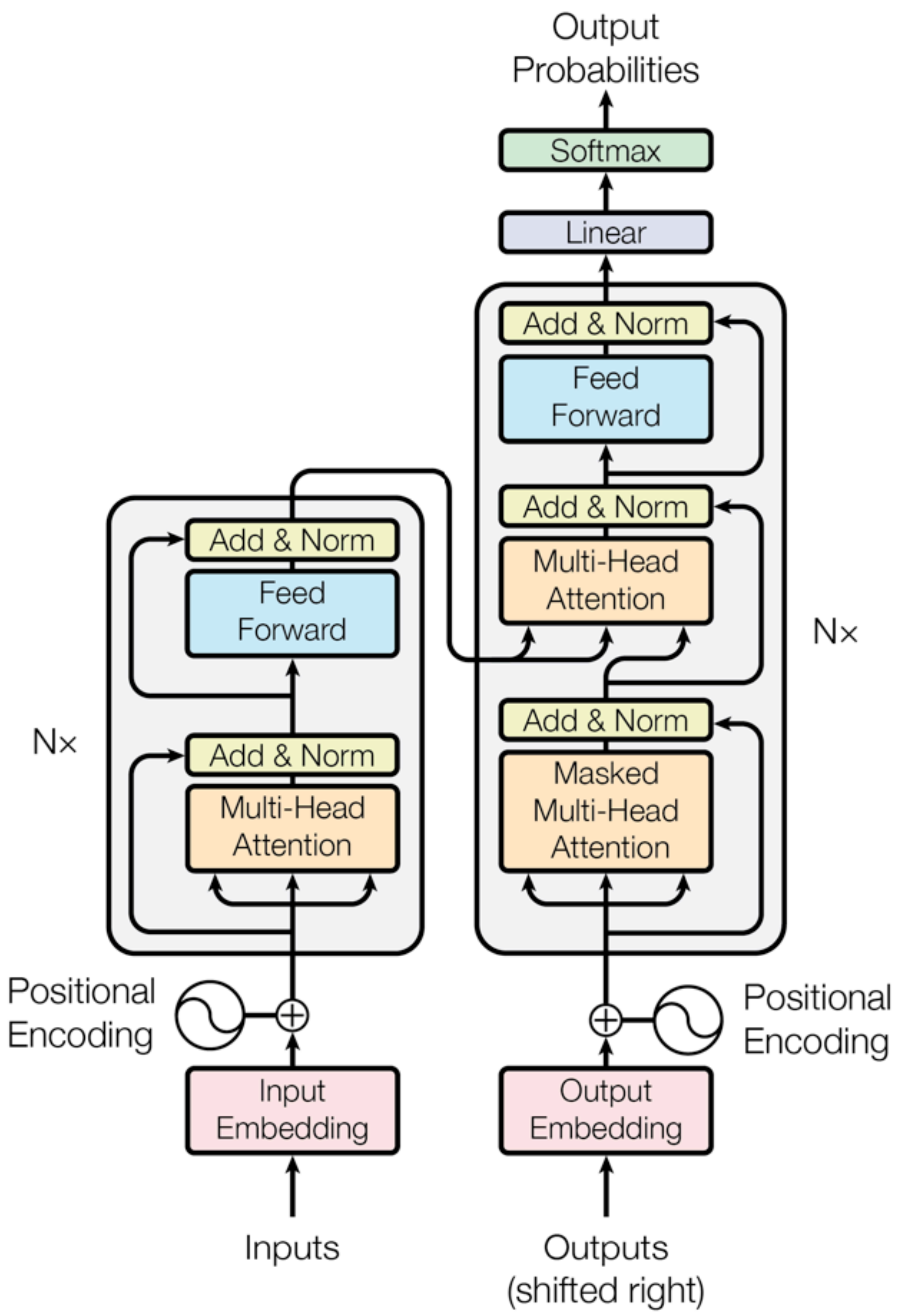
Illia Polosukhin* ‡ illia.polosukhin@gmail.com

The Annotated Transformer

Alexander M. Rush
srush@seas.harvard.edu
Harvard University

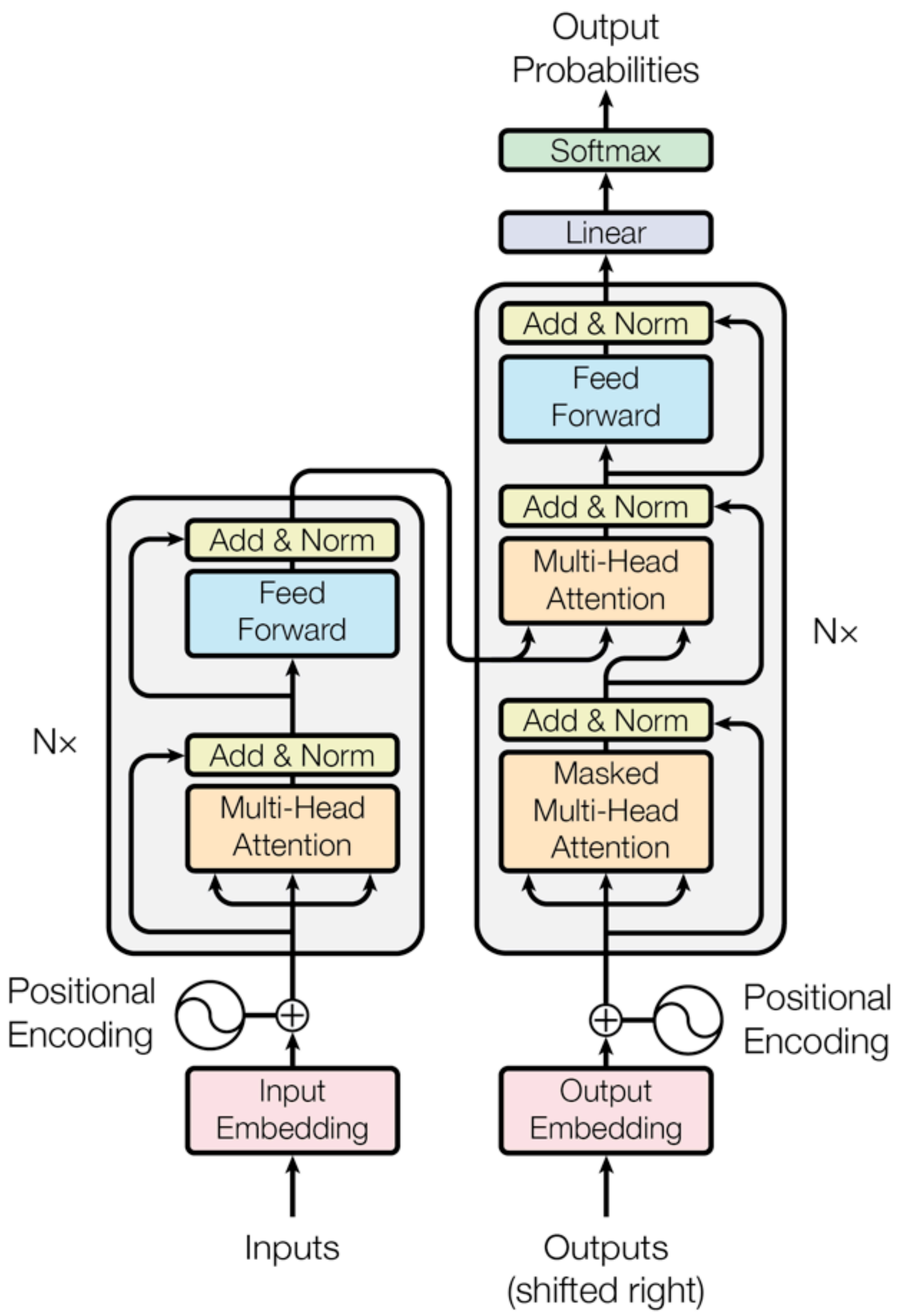
http://nlp.seas.harvard.edu/2018/04/03/attention.html

Transformer encoder-decoder



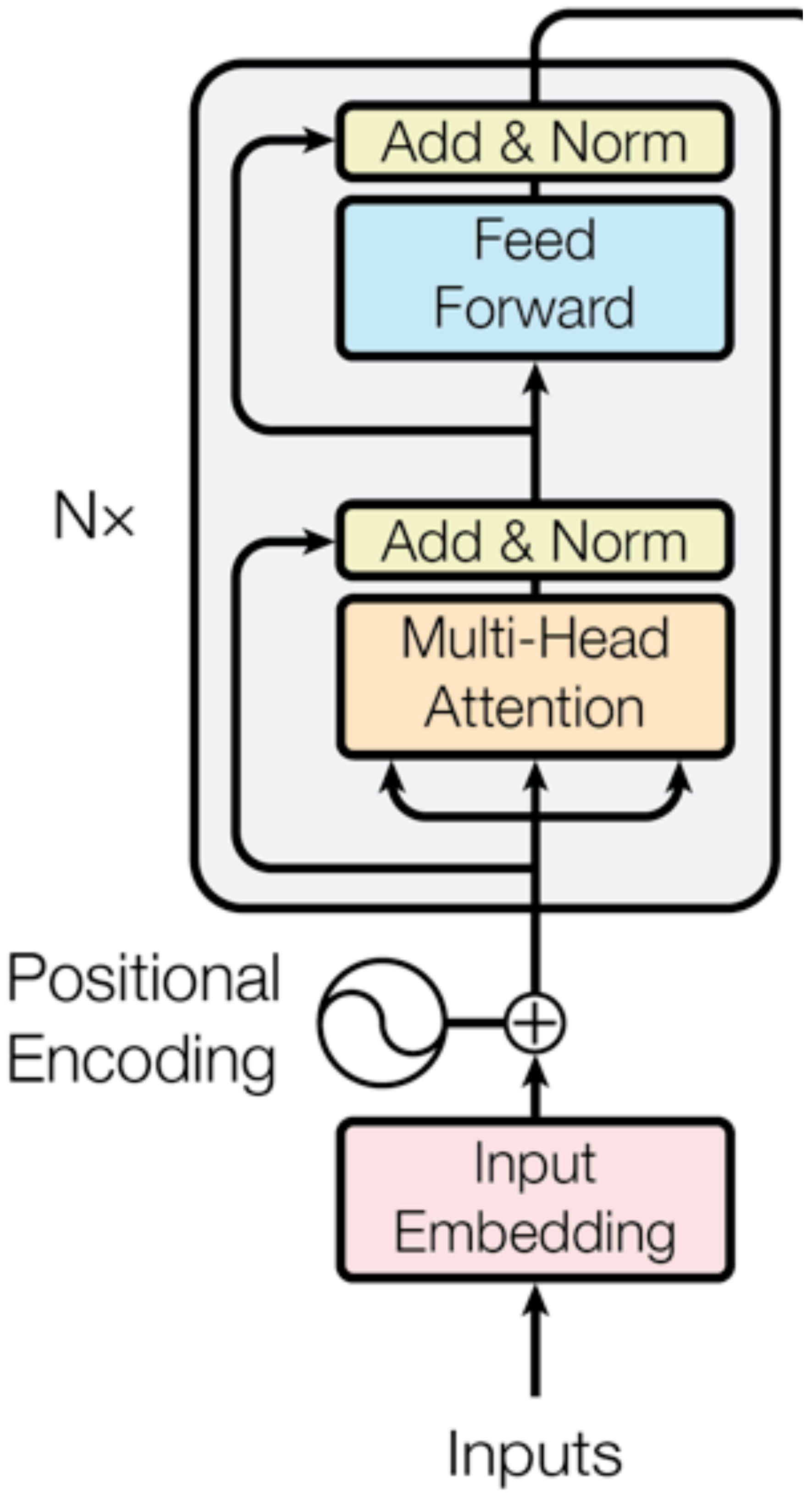
- Both encoder and decoder consist of N layers
- Each encoder layer has two sub-layers
  - Multi-head **self-**attention
  - FeedForward
- Each decoder layer has three sub-layers
  - Masked multi-head **self-**attention
  - Multi-head **cross-**attention
  - FeedForward
- Decoder: generate output probabilities for predicting next word

Transformer encoder-decoder




```python
class EncoderDecoder(nn.Module):
    """
    A standard Encoder-Decoder architecture.
    Base for this and many other models.
    """
    def __init__(self, encoder, decoder, src_embed,
                 tgt_embed, generator):
        super(EncoderDecoder, self).__init__()
        self.encoder = encoder
        self.decoder = decoder
        self.src_embed = src_embed
        self.tgt_embed = tgt_embed
        self.generator = generator

    def forward(self, src, tgt, src_mask, tgt_mask):
        "Take in and process masked src and target sequences."
        return self.decode(self.encode(src, src_mask),
                           src_mask,
                           tgt, tgt_mask)

    def encode(self, src, src_mask):
        return self.encoder(self.src_embed(src), src_mask)

    def decode(self, memory, src_mask, tgt, tgt_mask):
        return self.decoder(self.tgt_embed(tgt), memory,
                            src_mask, tgt_mask)
```

Transformer encoder



Layer Normalization (Ba et al., 2016)

```
def forward(self, x):
    mean = x.mean(-1, keepdim=True)
    std = x.std(-1, keepdim=True)
    return (self.a_2 * (x - mean) /
            (std + self.eps) + self.b_2)
```

In the paper:

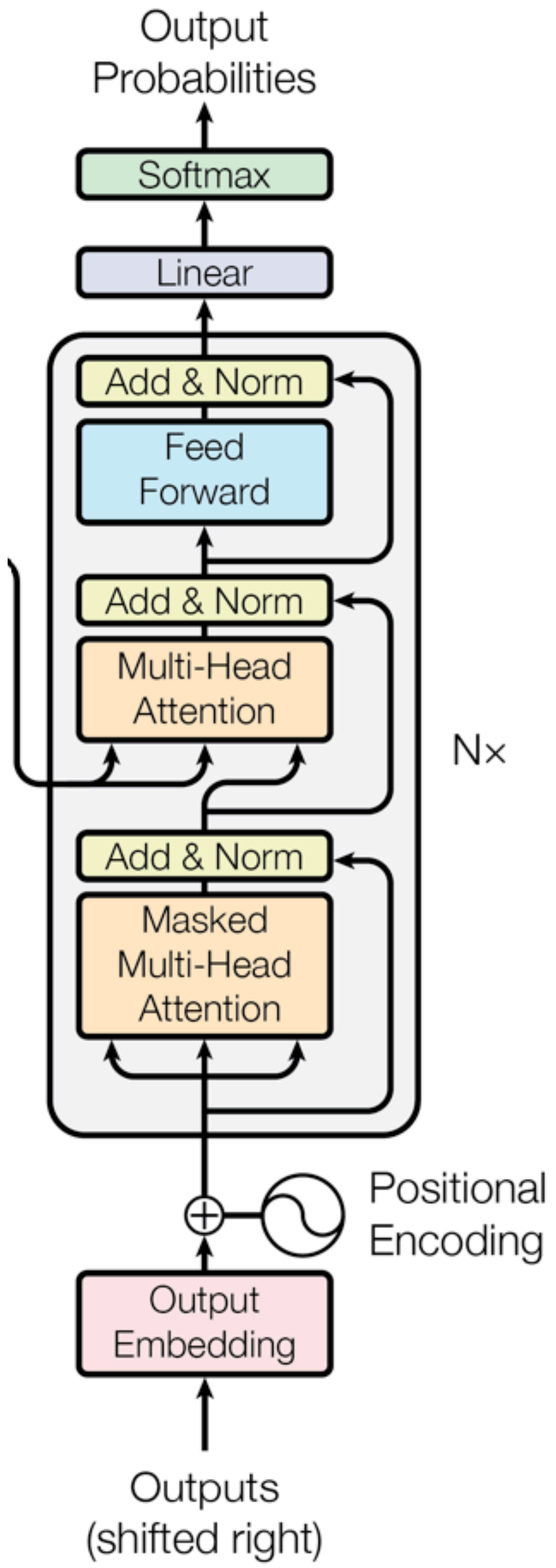
$$\mathrm{LayerNorm}(x + \mathrm{Sublayer}(x))$$

```
def forward(self, x, sublayer):
    "Apply residual connection to sublayer fn."
    return x + self.dropout(sublayer(self.norm(x)))
```

```
class EncoderLayer(nn.Module):
    "Encoder calls self-attn and feed forward."
    def __init__(self, size, self_attn,
                 feed_forward, dropout):
        super(EncoderLayer, self).__init__()
        self.self_attn = self_attn
        self.feed_forward = feed_forward
        sublayer = SublayerConnection(size, dropout)
        self.sublayer = clones(sublayer, 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)
```

We will come back to this!

Transformer decoder




```python
class DecoderLayer(nn.Module):
    "Decoder calls self-attn, src-attn, and feed forward."
    def __init__(self, size, self_attn,
                 src_attn, feed_forward, dropout):
        super(DecoderLayer, self).__init__()
        self.self_attn = self_attn
        self.src_attn = src_attn
        self.feed_forward = feed_forward
        sublayer = SublayerConnection(size, dropout)
        self.sublayer = clones(sublayer, 3)
        self.size = size

    def forward(self, x, memory, s_mask, t_mask):
        "Follow Figure 1 (right) for connections."
        m = memory
        x = self.sublayer[0](x, lambda x:
                             self.self_attn(x, x, x, t_mask))
        x = self.sublayer[1](x, lambda x:
                             self.src_attn(x, m, m, s_mask))
        return self.sublayer[2](x, self.feed_forward)
```

self-attention   cross-attention

Attention

Attention(query, key, value, mask)

Encoder

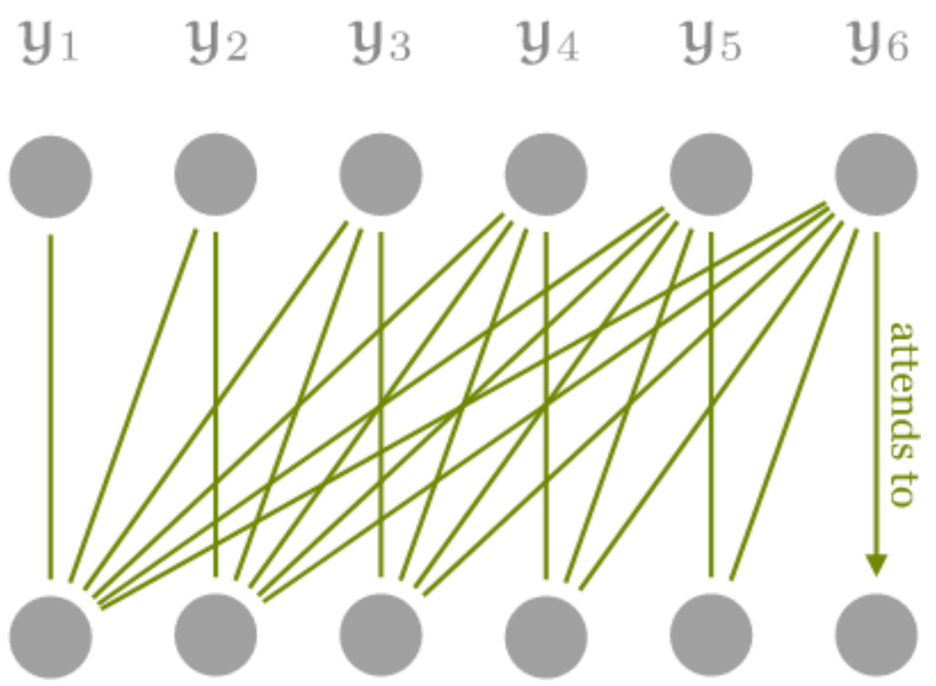
```
x = self.sublayer[0](x, lambda x:
                     self.self_attn(x, x, x, mask))
```

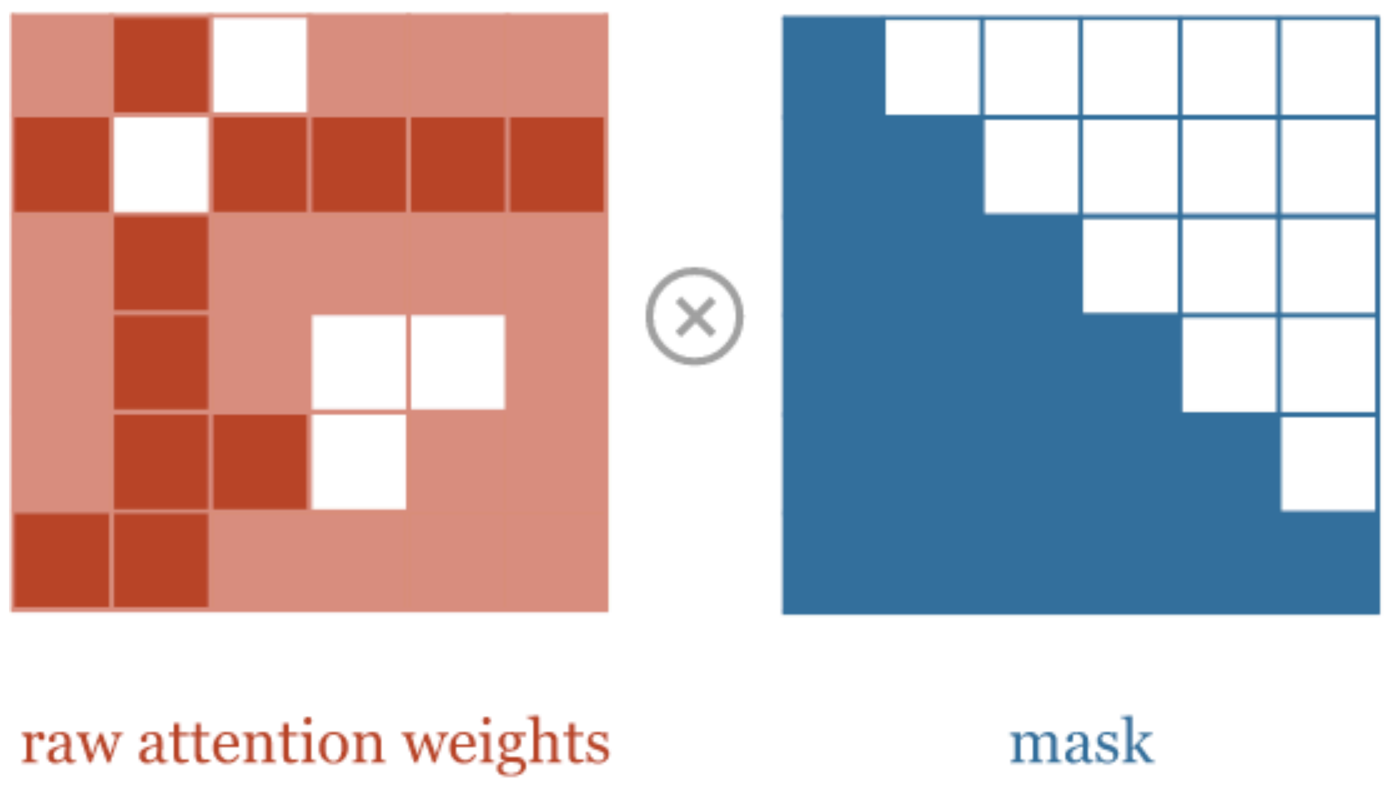
Decoder

```
x = self.sublayer[0](x, lambda x:
                     self.self_attn(x, x, x, t_mask))
x = self.sublayer[1](x, lambda x:
                     self.src_attn(x, m, m, s_mask))
```

source
hidden states

Masked self-attention






```
def subsequent_mask(size):
    "Mask out subsequent positions."
    attn_shape = (1, size, size)
    subsequent_mask = np.triu(np.ones(attn_shape), k=1)
    return torch.from_numpy(
        subsequent_mask.astype('uint8')) == 0
```

Attention

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

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

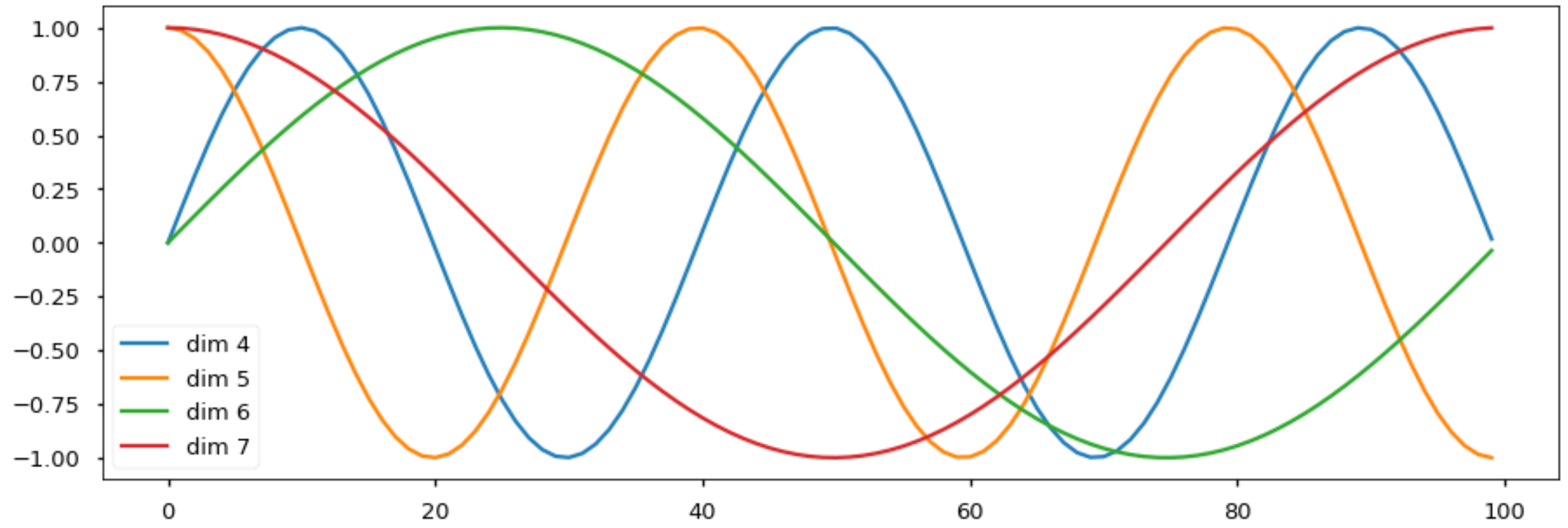
Set masked positions
to -1e9 before softmax

```python
class MultiHeadedAttention(nn.Module):
    def __init__(self, h, d_model, dropout=0.1):
        "Take in model size and number of heads."
        super(MultiHeadedAttention, self).__init__()
        assert d_model % h == 0
        # We assume d_v always equals d_k
        self.d_k = d_model // h
        self.h = h
        self.linears = clones(nn.Linear(d_model, d_model), 4)
        self.attn = None
        self.dropout = nn.Dropout(p=dropout)

    def forward(self, query, key, value, mask=None):
        "Implements Figure 2"
        if mask is not None:
            # Same mask applied to all h heads.
            mask = mask.unsqueeze(1)
        nb = query.size(0)

        # 1) Do all the linear projections in batch from d_model
        query, key, value = [
            l(x).view(nb, -1, self.h, self.d_k).transpose(1, 2)
            for l, x in zip(self.linears, (query, key, value))]

        # 2) Apply attention on all the projected vectors in bat
        x, self.attn = attention(query, key, value, mask=mask,
                                 dropout=self.dropout)

        # 3) "Concat" using a view and apply a final linear.
        x = x.transpose(1, 2).contiguous().view(
            nb, -1, self.h * self.d_k)
        return self.linears[-1](x)
```

Linear projection for all the heads,
split them into different slices later

Other interesting things

- Decoder: Input and output embeddings are tied
- Positional encodings



- A dedicated optimizer

$$lrate = d_{\text{model}}^{-0.5} \cdot \min(step\_num^{-0.5}, step\_num \cdot warmup\_steps^{-1.5})$$

- Label smoothing

$$p'(y|x_i) = (1-\varepsilon)p(y|x_i) + \varepsilon u(y|x_i)$$
$$= \begin{cases} 1-\varepsilon+\varepsilon u(y|x_i) & \text{if } y = y_i \\ \varepsilon u(y|x_i) & \text{otherwise} \end{cases}$$

Breakout discussion

- Group 1 (Danqi)
  - Which parts of Transformer implementation (design, optimization, regularization) that you find interesting, surprising or counter-intuitive?
- Group 2 (Kaiyu)
  - Which parts of Transformer implementation (design, optimization, regularization) that you find interesting, surprising or counter-intuitive?
- Group 3 (Mingzhe)
  - How to improve Transformers?
- Group 4 (Zexuan)
  - How to improve Transformers?

Use the remaining time for free-form discussion!!!

How to improve Transformers?

- Re-order the sub-layers…

sfsfsfsfsfsfsfsfsfsfsfsfsfsf

(a) Interleaved Transformer

sssssssfsfsfsfsfsfsfsfffffff

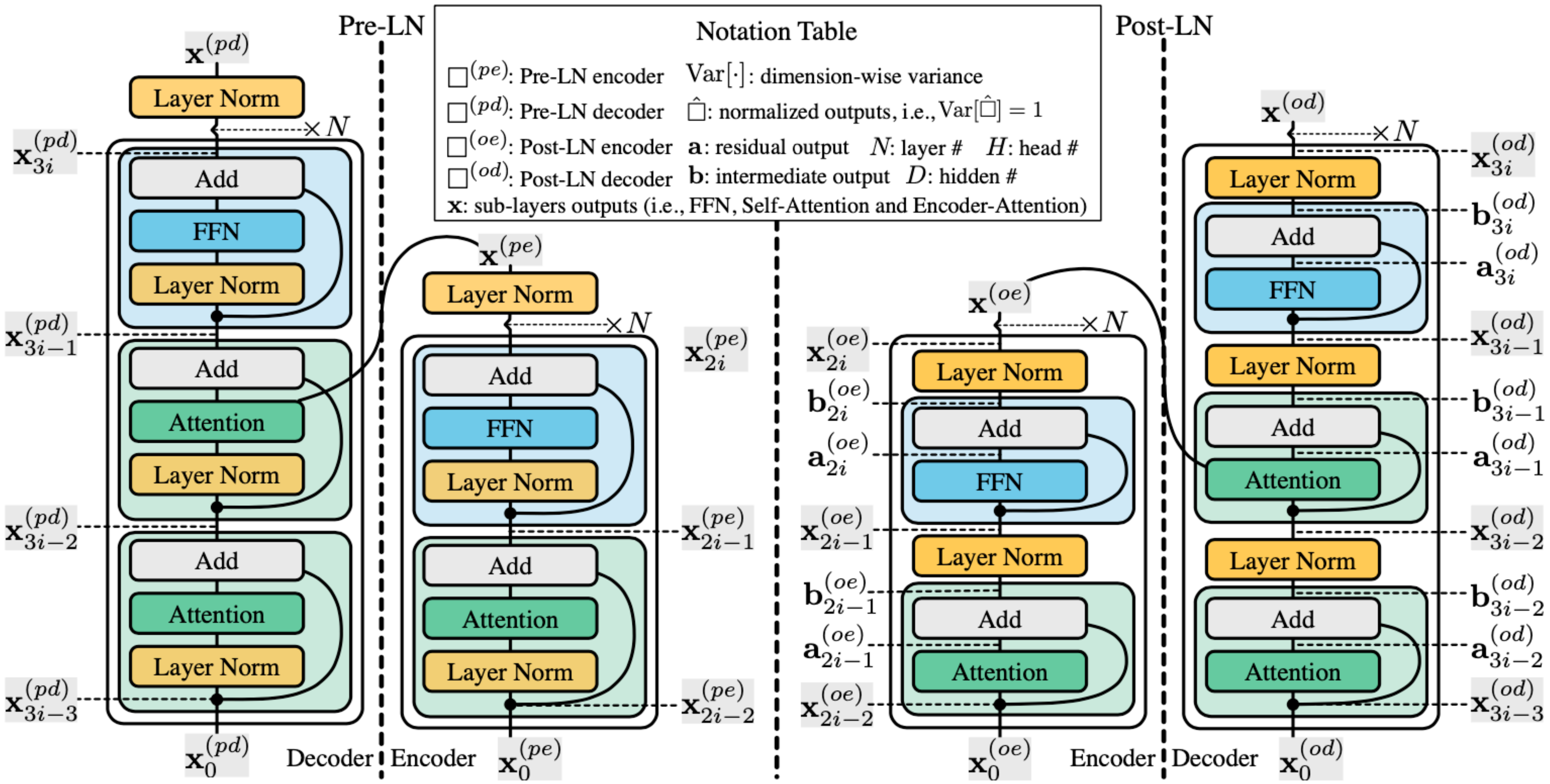
(b) Sandwich Transformer

Model	**PPL**
fsfsfffsffsfsssffsfssfssssffsffs	20.74
sfssffsffffssssfsfffsfsffsfssssf	20.64
fsffssffssssffsssssffsfssfsfffff	20.33
fsffffffsssfssffsfssffsfsssffsss	20.27
fssffffffsfsssfffssssfffssssffss	19.98
sssfssfsffffssfsfsfsssffsfsfffsf	19.92
fffsfsssfsffsfsffsffsssssffssffs	19.69
fffsffssffsssfssfsssfffffsfsssfs	19.54
sfsfsfsfsfsfsfsfsfsfsfsfsfsfsfsf	**19.13**
fsffssfssfffssssfffsssffffsfssfs	19.08
sfsffssssffssffffsssffsssfsffsff	18.90
sfsfsfsfsfsfsfsfsfsfsfsfsfsfsfsf	**18.83**
sssssssffsffsfsfsffffsfffsfssffs	18.83
sffsfsffsfsssffssfssssssfffffffs	18.77
sssfssffsfssfsffsfffssffsfsffssf	18.68
fffsssssfffsfssssffsfsfsfssffsff	18.64
sfffsssfsfssfsssssfssfffffsfffsf	18.61
ssffssfssssffffffssffsssfsffssff	18.60
fsfsssssfsfsfffffsfffsffssffssss	18.55
sfsfsfsfsfsfsfsfsfsfsfsfsfsfsfsf	**18.54**
sfsfsfsfsfsfsfsfsfsfsfsfsfsfsfsf	**18.49**
fsfsssssfsfffssfsffsfsfsfsffffss	18.38
sfssffsfsfsffsssssfffsssfffsffsf	18.28
sfsfsfsfsfsfsfsfsfsfsfsfsfsfsfsf	**18.25**
sfsfssfsssffsfsfsfsffffssffsfssf	18.19

(Press et al., 2020) Improving Transformer Models by Reordering their Sublayers

How to improve Transformers?

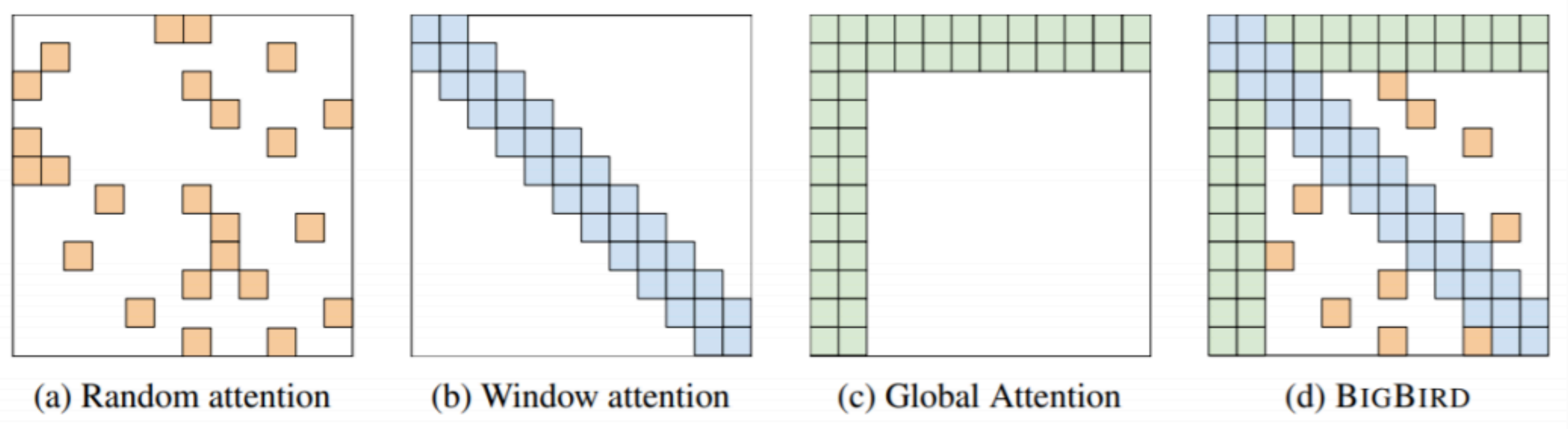
- Pre-LN is more robust than post-LN

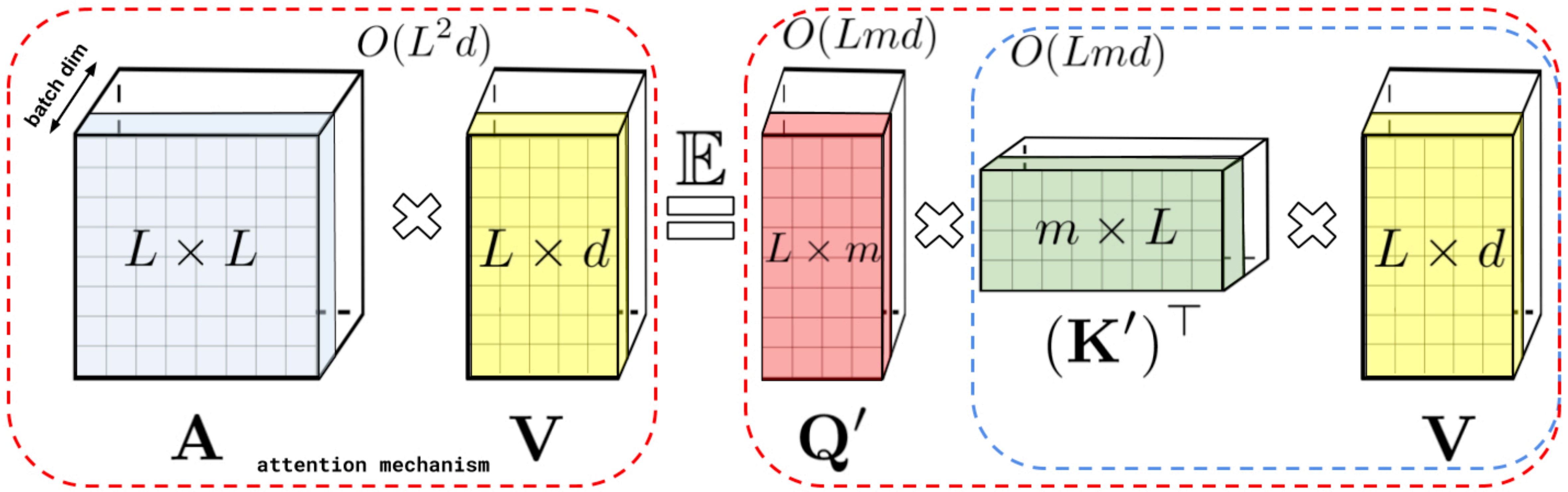


(Liu et al., 2020) Understanding the Difficulty of Training Transformers

How to improve Transformers?

- Scale up to long sequences (and avoid quadratic computation!)



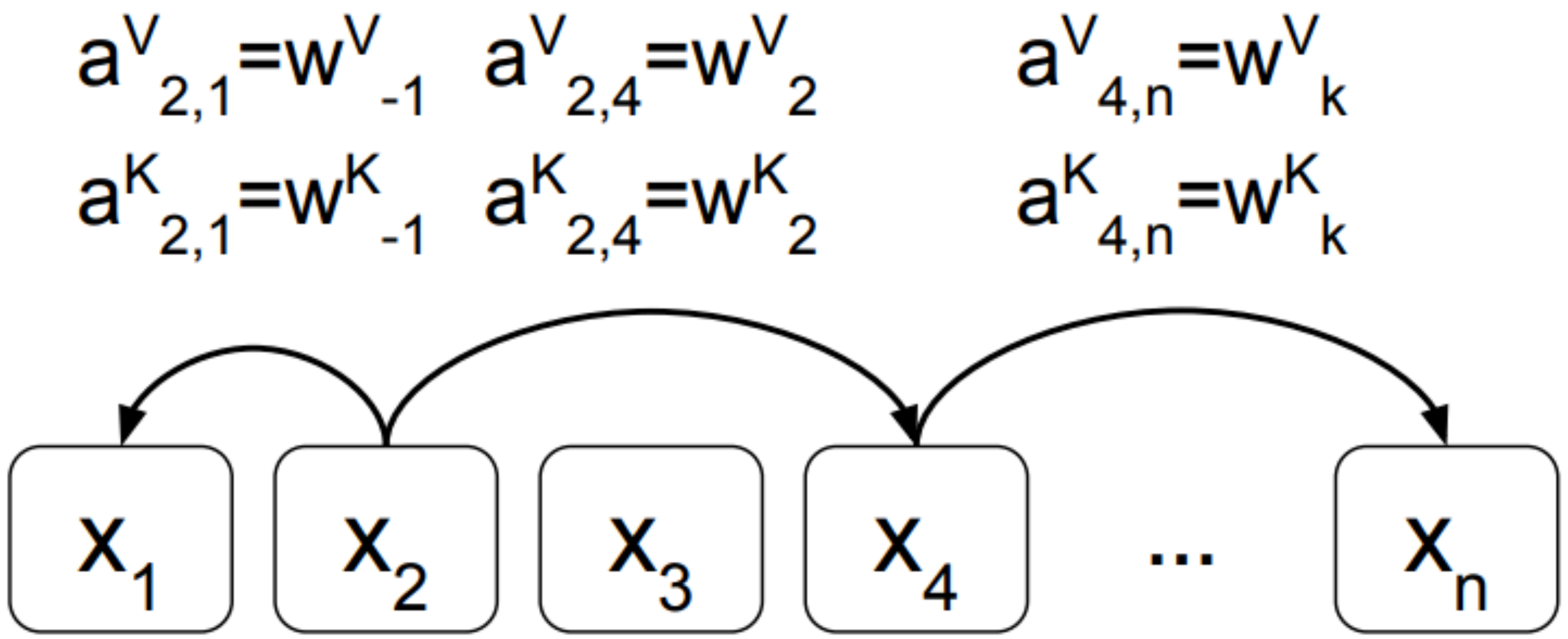


(Zaheer et al., 2020) Big Bird: Transformers for Longer Sequences

(Choromanski et al., 2020) Rethinking Attention with Performers

How to improve Transformers?

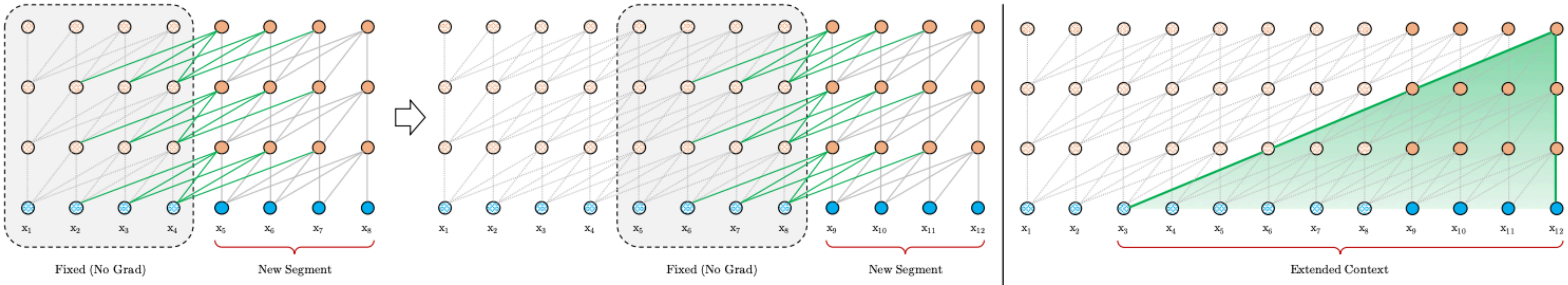
- Relative positional representations



(Shaw et al., 2018) Self-Attention with Relative Position Representations

How to improve Transformers?

- Relative positional encoding + Segment recurrence



(a) Training phase. (b) Evaluation phase.

(Dai et al., 2019): Transformer-XL: Attentive Language Models Beyond a Fixed-Length Context