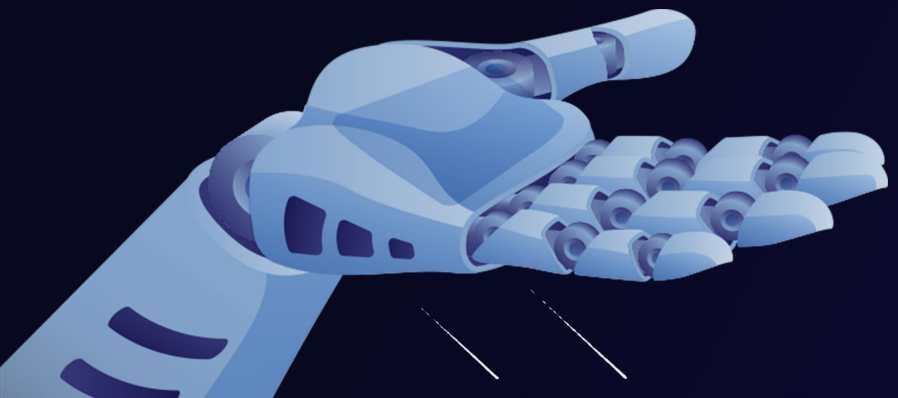
第四章：Transformer详解及翻译模型搭建

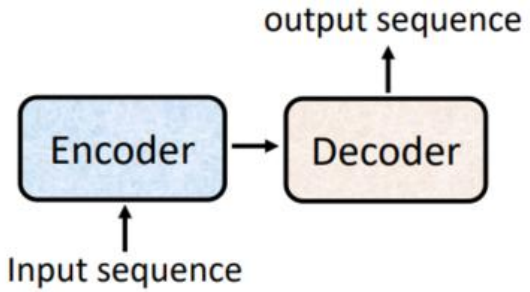


CONTENTS

4.1 Transformer简介：基于自注意力机制的Seq2Seq模型

编码器-解码器

编码器-解码器架构由两部分组成：编码器和解码器。编码器负责将输入序列编码为一个固定长度的向量，而解码器则负责从这个向量中解码出输出序列。这种架构对于处理可变长度的输入和输出序列非常有效。

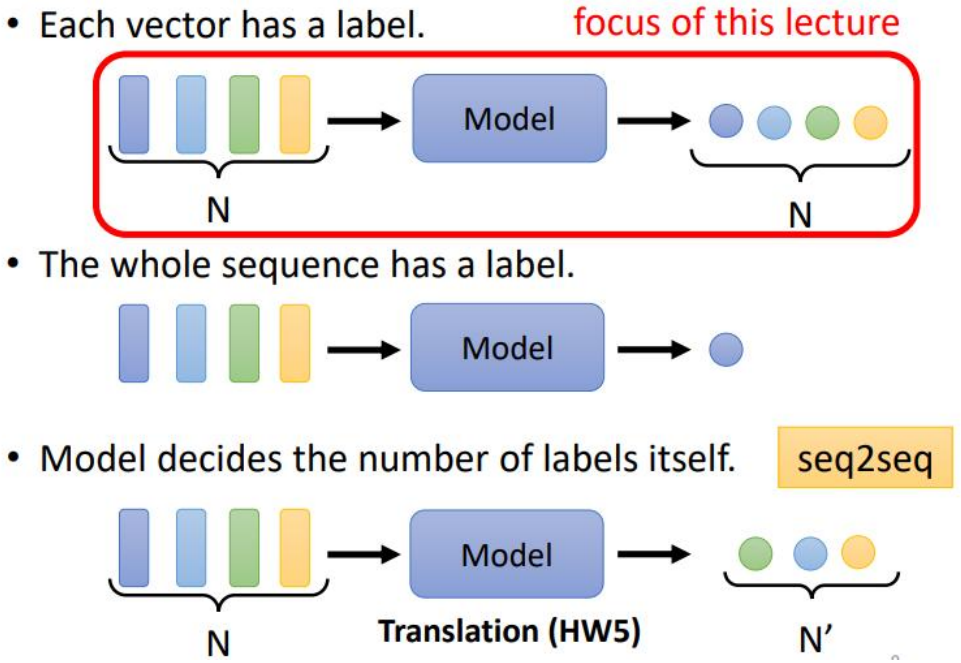


序列传导模型

1）输入N个 词，输出N个词。
例如：词性标注、NER。

2）输入N个词，输出一个词。
例如：句子分类。

3）输入N个词，输出N’个词。
例如：翻译、语音识别、语音合成。



4.1 Transformer简介：基于自注意力机制的Seq2Seq模型

RNN

由于循环神经网络的顺序性质，训练不能并行化。

LSTM、GRU。。。

CNN

可以并行化，但是学习远距离位置之间的依赖关系比较困难

ByteNet、ConvS2S。。。

Transformer

1）训练并行化，且可以学习到远距离依赖。

2）第一个完全依靠**自注意力机制**计算输入和输出表示而不使用序列对齐RNN或卷积的**序列传导模型**。

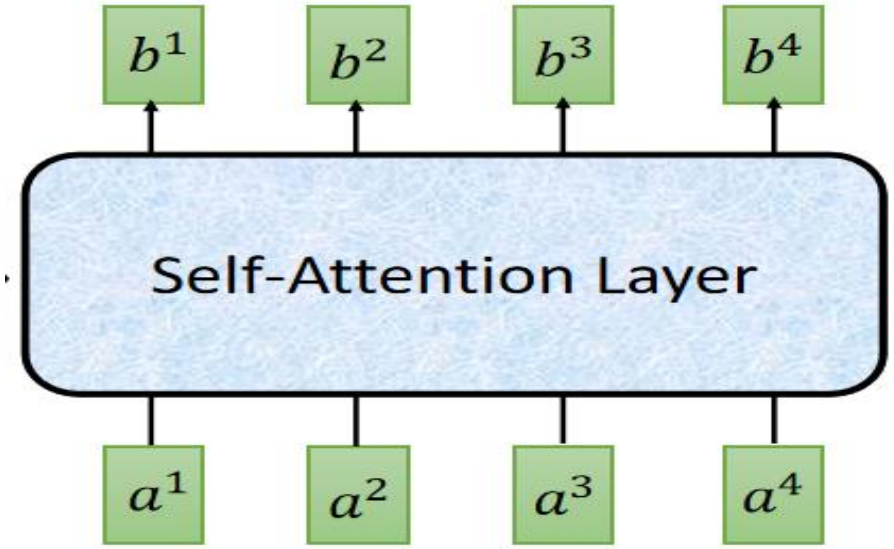
4.2 自注意力机制：充分考虑上下文 【简介】

自注意力机制主要用于处理序列数据，如自然语言处理中的文本。

自注意力机制通过计算每个词与其他词之间的相似度来建立它们之间的关系，并根据这些关系来加权地计算每个词的表示。

优势一：考虑上下文。

优势二：计算速度快。



4.2 自注意力机制：充分考虑上下文 【QKV计算】

1）原始输入x^i为一个个词或token。

2）x^i经过转换，变成新的向量a^i。

3）a^i通过转换，计算得到q^i、k^i和v^i。

q: query (to match others)

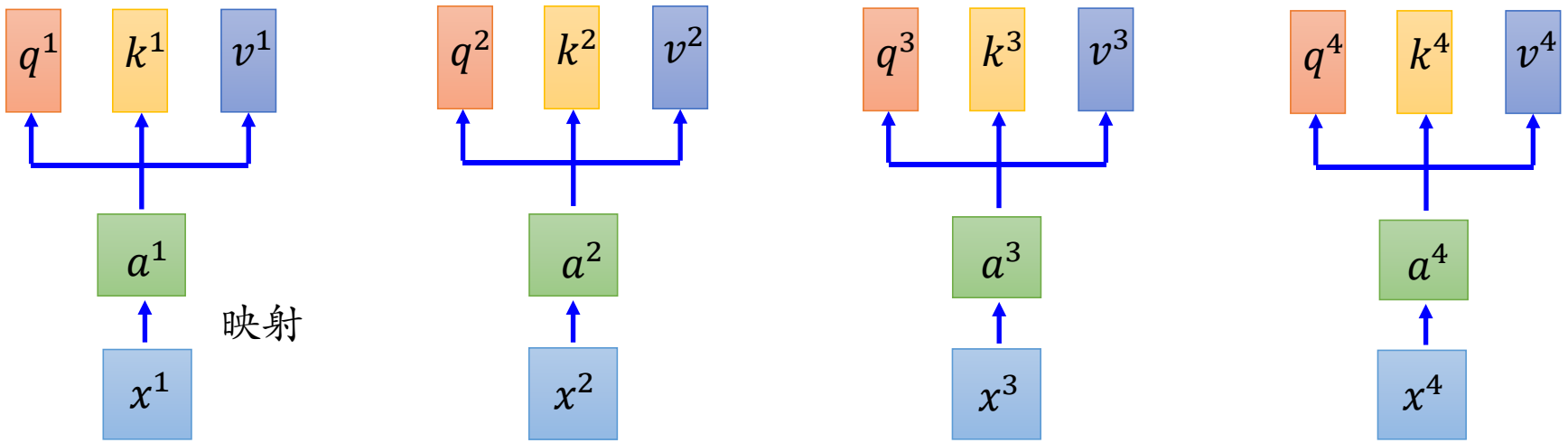
$$q^i = W^q a^i$$

k: key (to be matched)

$$k^i = W^k a^i$$

v: information to be extracted

$$v^i = W^v a^i$$



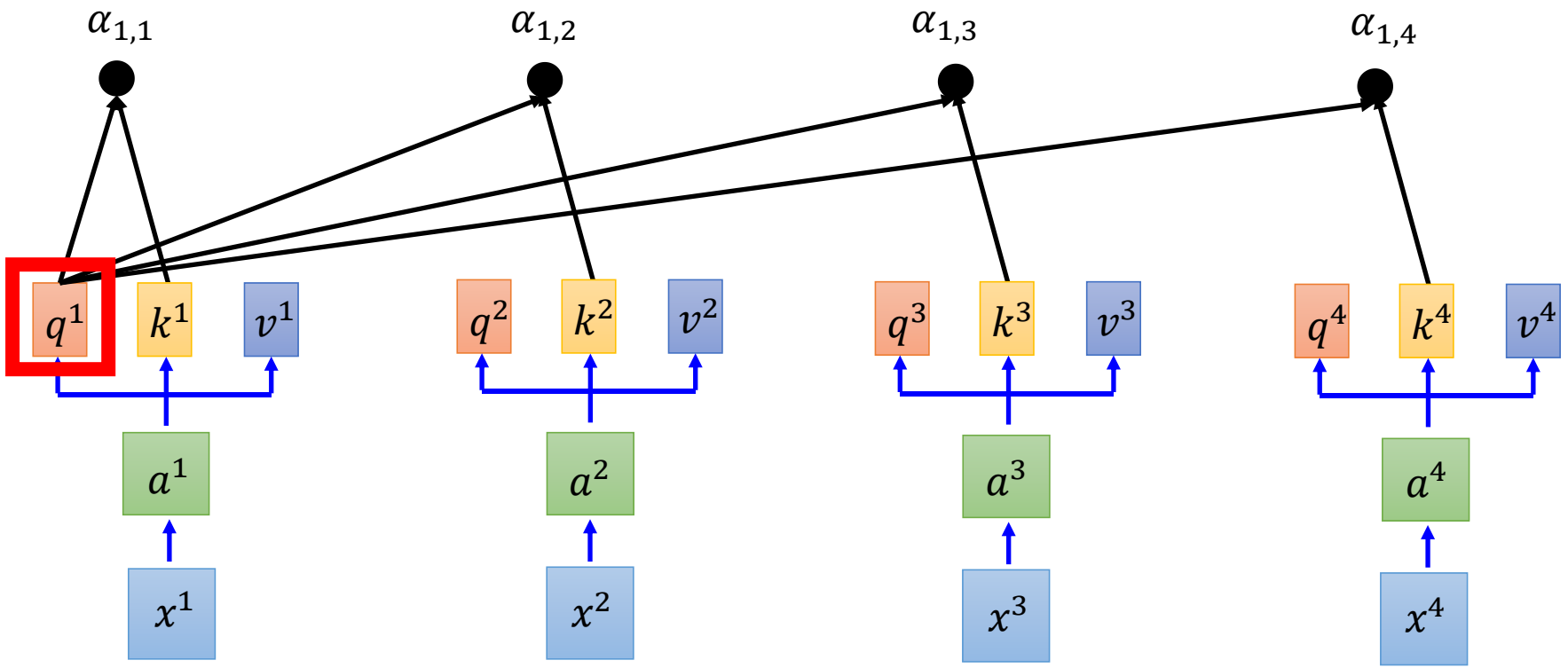
4.2 自注意力机制：充分考虑上下文 【关系计算】

d is the dim of q and k

Scaled Dot-Product Attention:

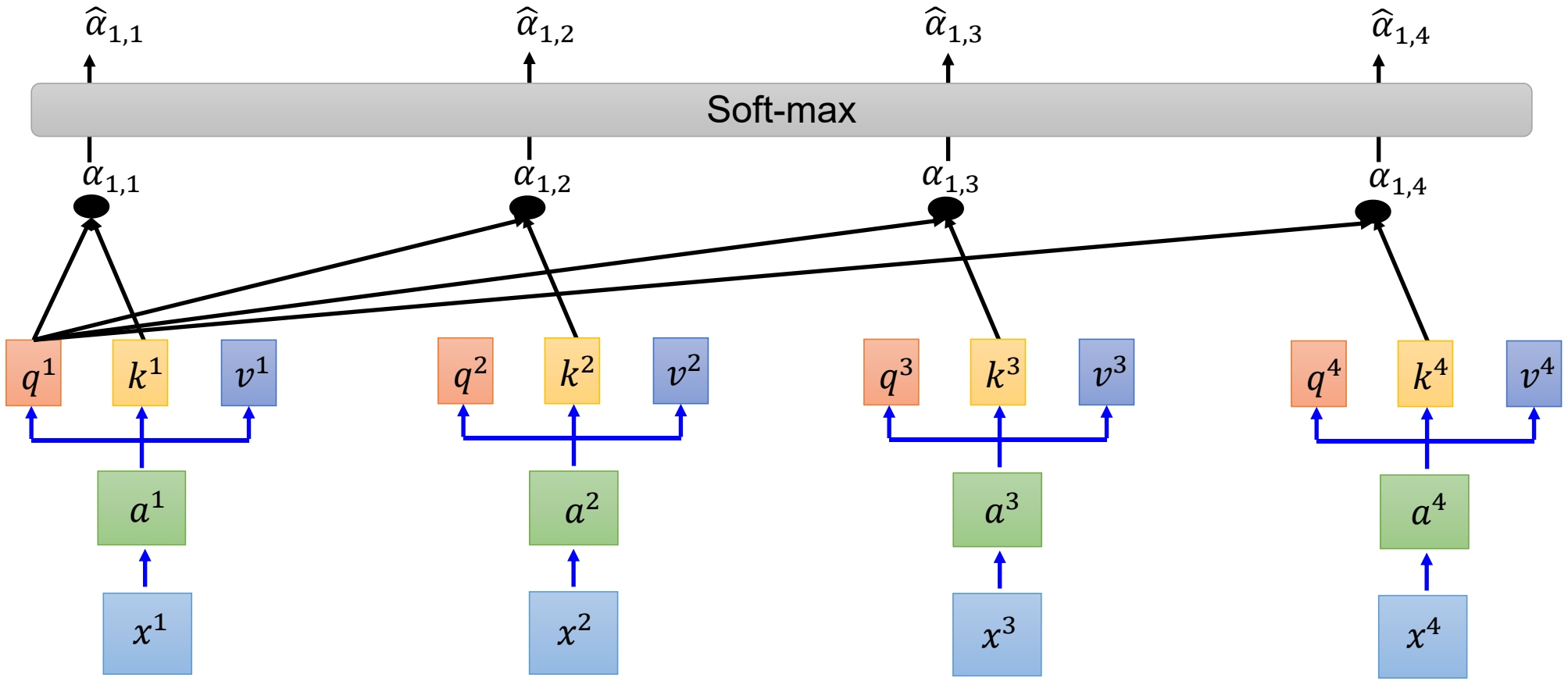
$$\alpha_{1,i} = q^1 \cdot k^i / \sqrt{d}$$

dot product



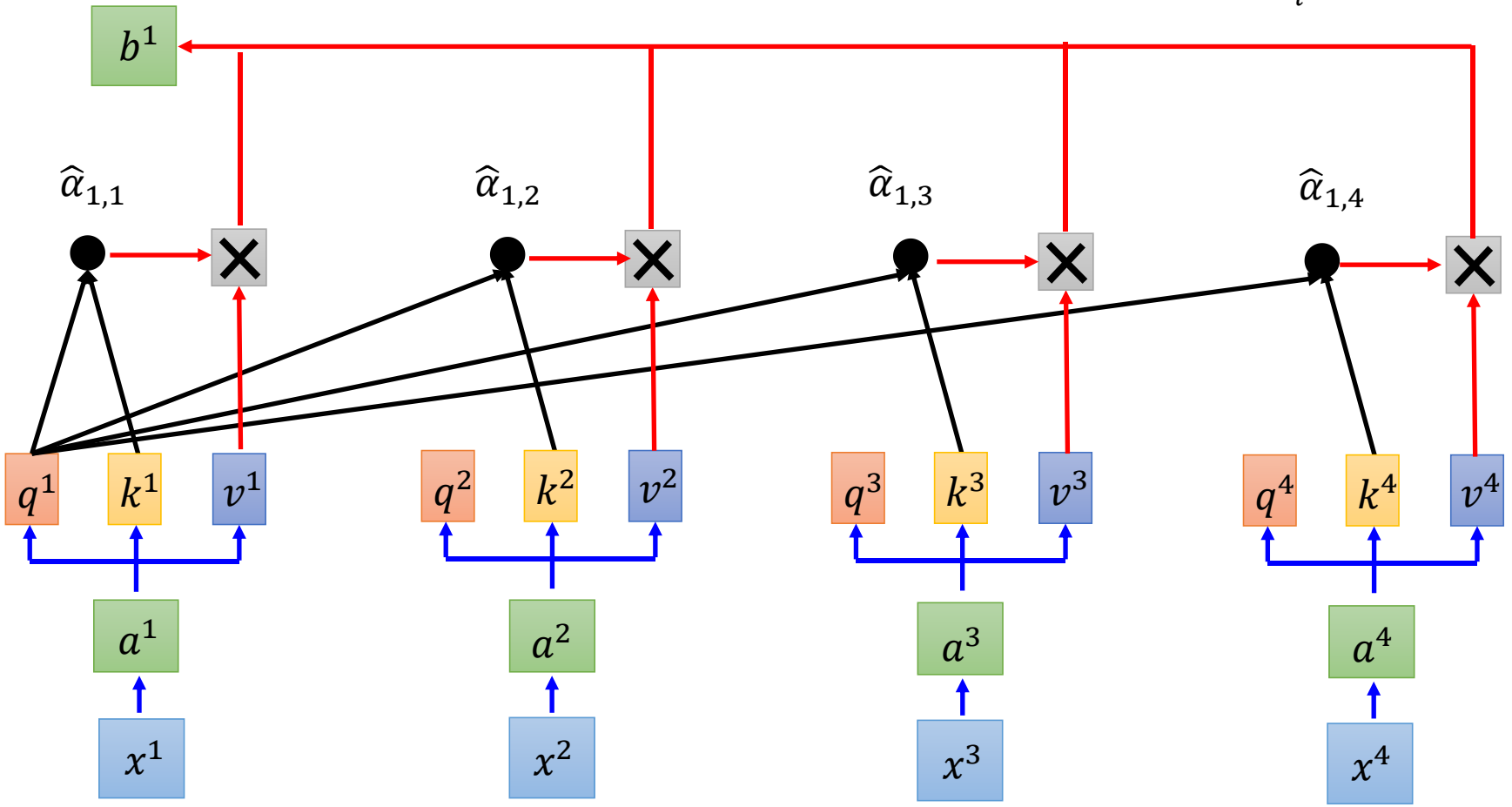
4.2 自注意力机制：充分考虑上下文 【关系归一化】

$$\hat{\alpha}_{1,i} = exp(\alpha_{1,i}) / \sum_j exp(\alpha_{1,j})$$

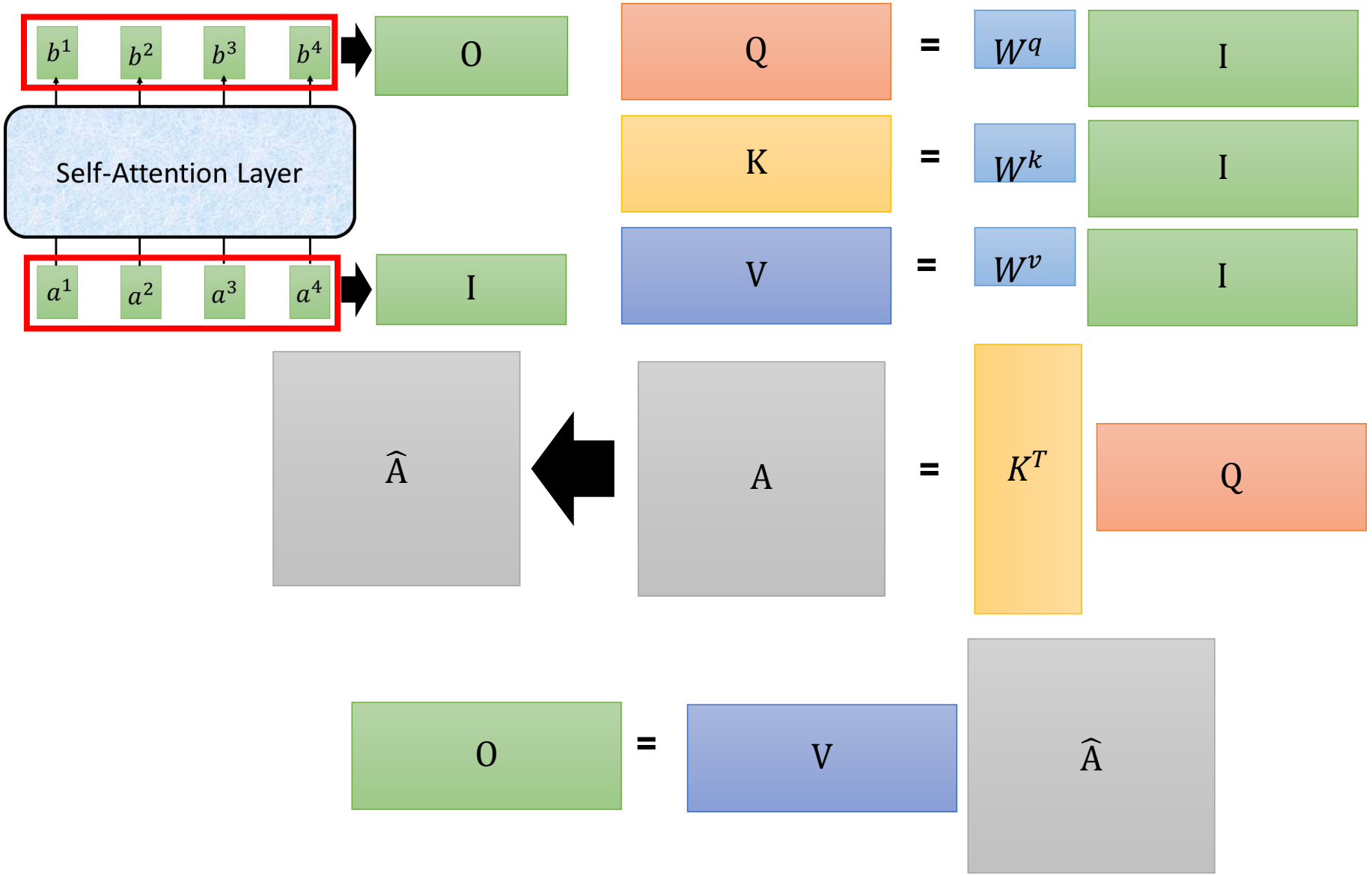


4.2 自注意力机制：充分考虑上下文 【加权信息】

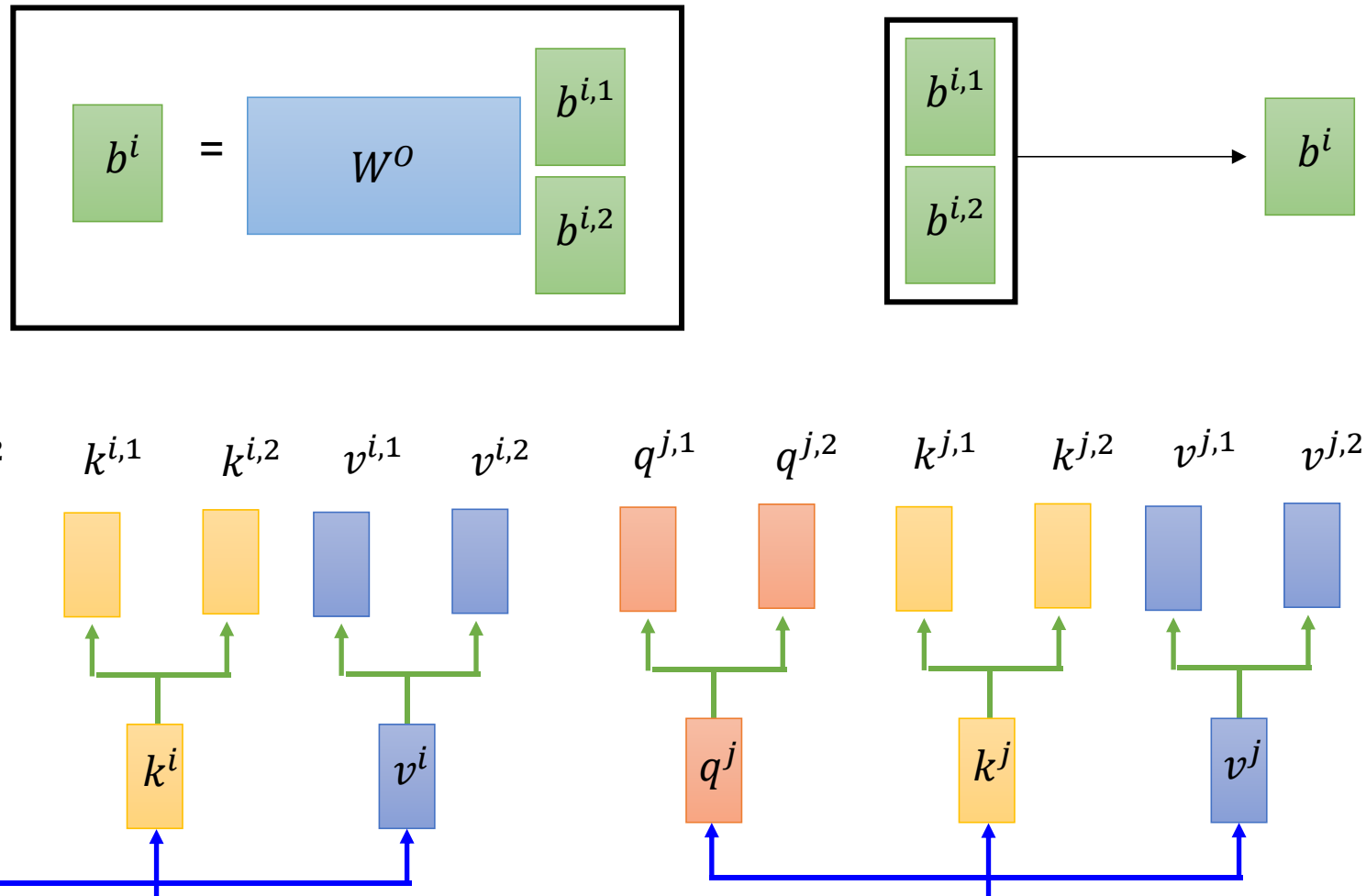
$$b^1 = \sum_i \hat{\alpha}_{1,i} v^i$$



4.2 自注意力机制：充分考虑上下文 【并行计算】



4.2 自注意力机制：充分考虑上下文 【多头】



4.3 Transformer模型架构 【概况】

左侧：

1）原始输入部分

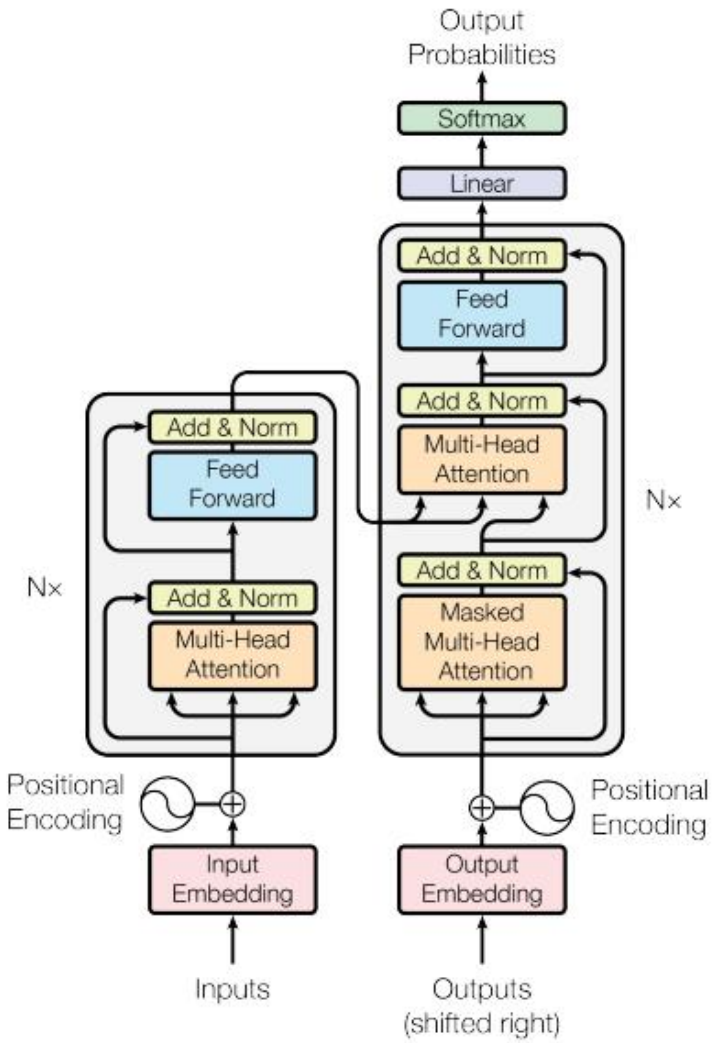
2）编码器部分

右侧：

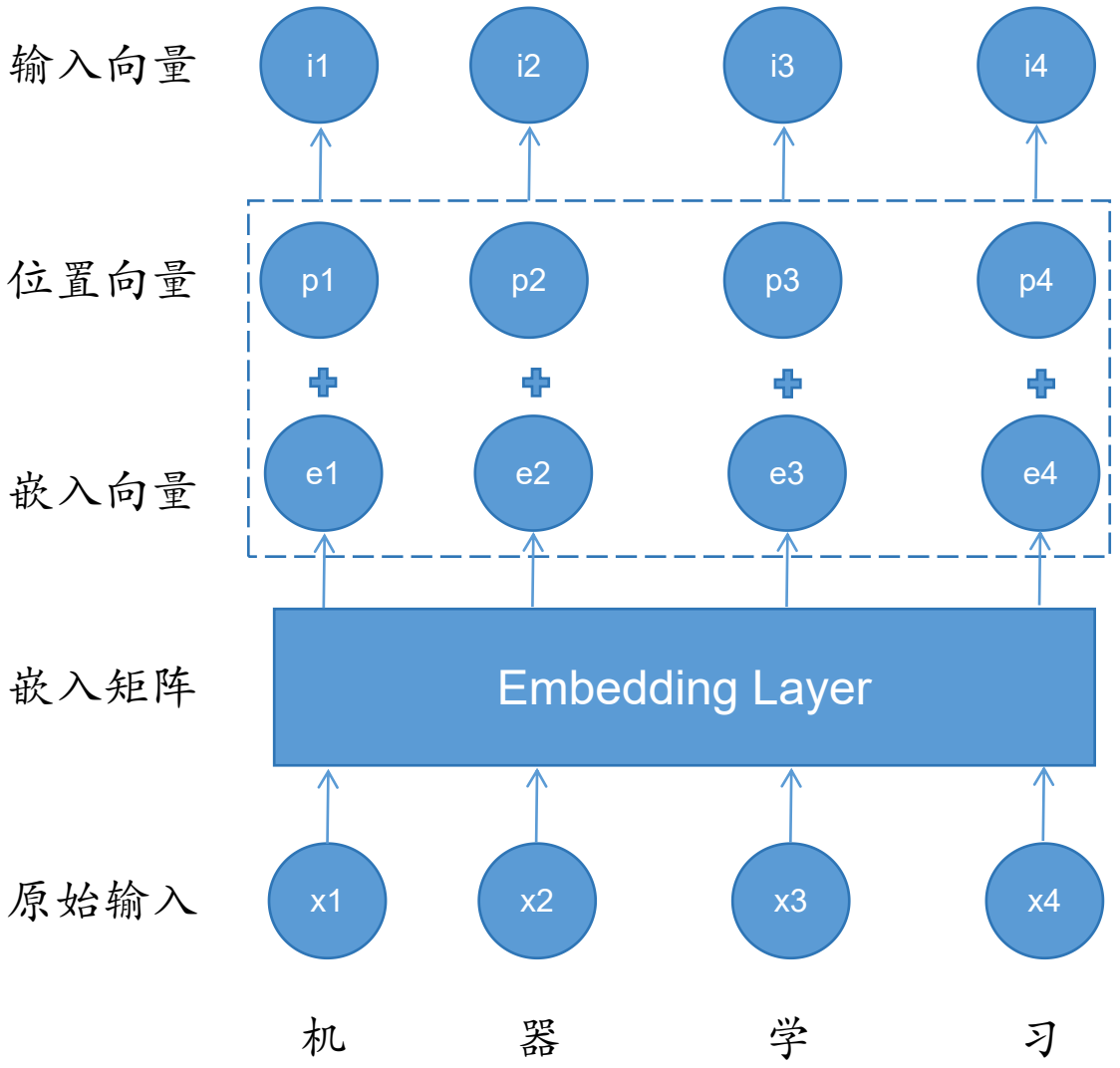
3）目标输入部分

4）解码器部分

5）目标输出部分



4.3 Transformer模型架构 【原始输入】



位置向量

1）位置编码具有与嵌入相同的dmodel维度，以便两者可以相加。

2）Transformer使用不同波长的正弦和余弦函数来作为位置编码。

3）公式如下，pos是位置，i是维度。也就是说，位置编码的每个维度对应一个正弦或余弦曲线。波长形成了一个几何级数，从2π到10000π。

4）容易学习到通过相对位置进行注意力计算，因为对于任何固定的偏移量k，PE(pos+k)可以被表示为PE(pos)的线性函数。

5）允许模型外推到比训练中遇到的序列长度更长的序列长度。

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

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

4.3 Transformer模型架构 【编码器】

1）输入向量首先进行自注意力计算。

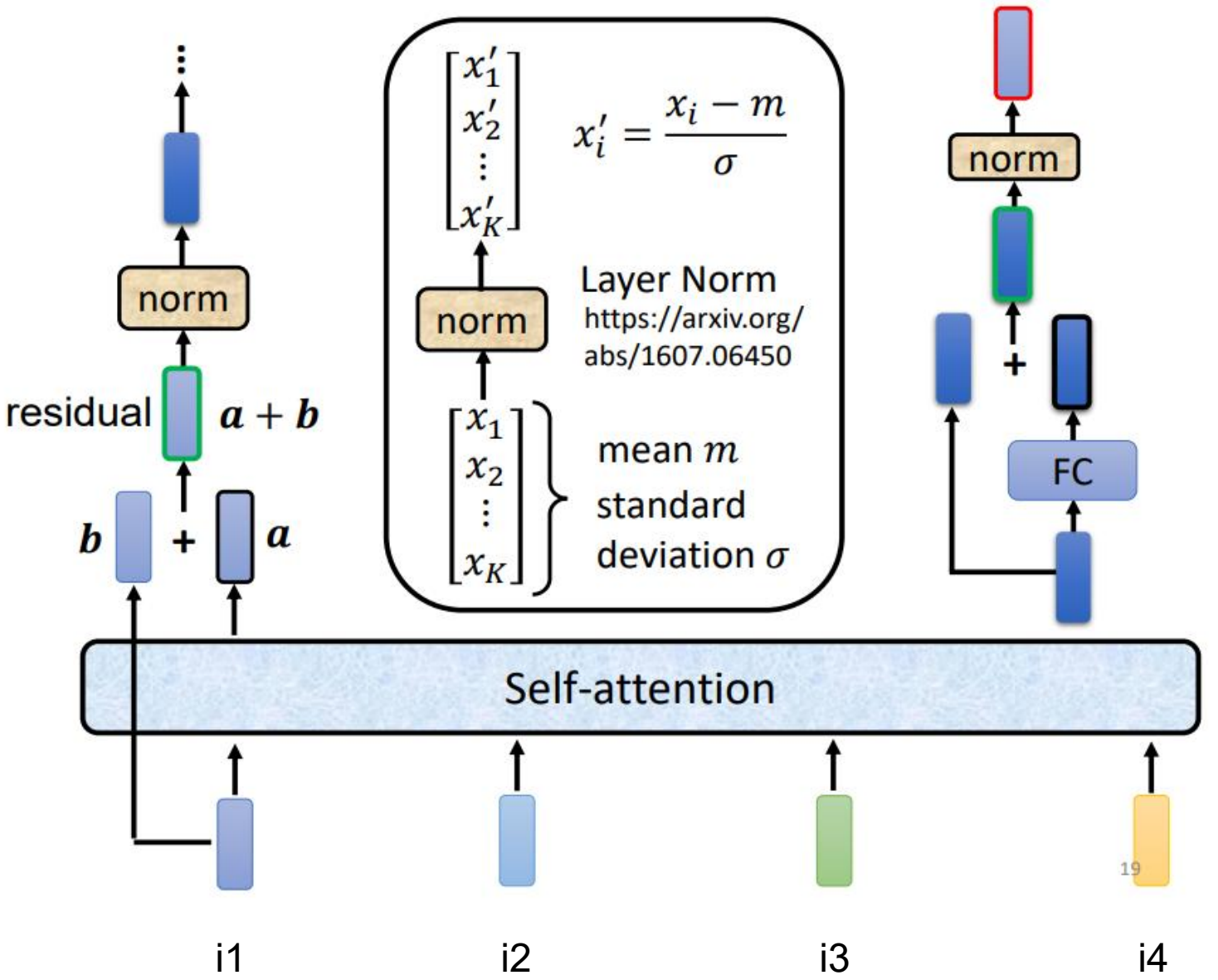
2）自注意力输出的向量与原始向量进行残差计算。

3）残差计算输出的向量进行层标准化。

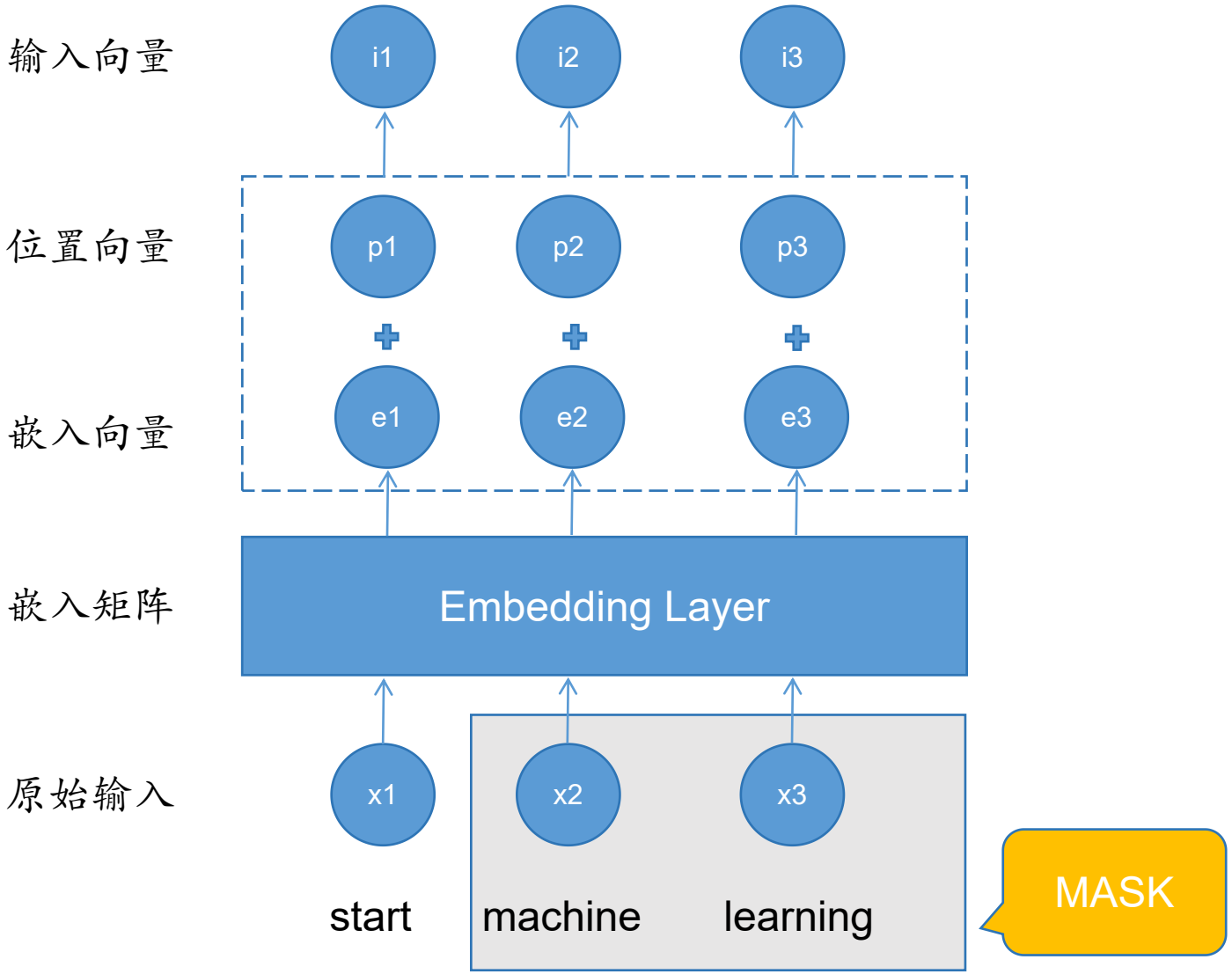
4）层标准化输出的向量进行全连接计算。

5）全连接计算输出的向量进行残差计算。

6）第5）步的残差计算后再进行层标准化计算。



4.3 Transformer模型架构 【目标输入】



与原始输入部分基本相同，区别在于需要增加自回归当前词信息和掩码信息。

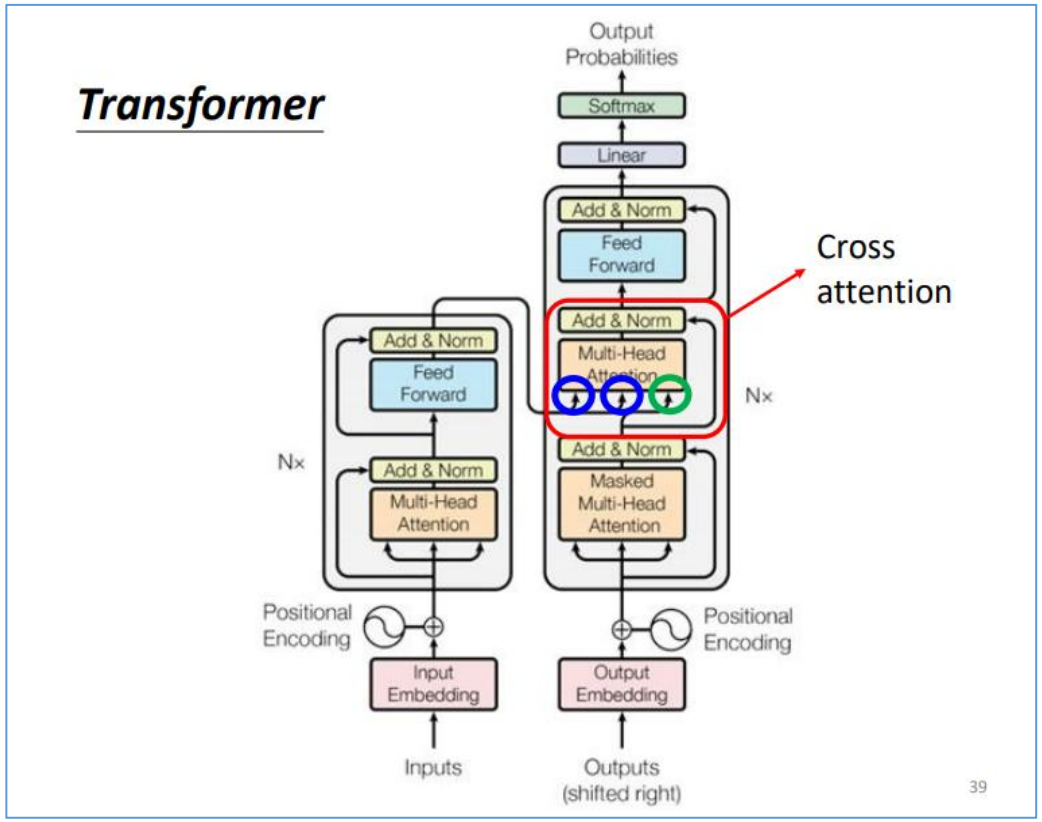
自回归（Autoregressive）是指解码器在生成每个输出时，依赖于之前已经生成的输出。这种方法常用于自然语言处理任务，比如机器翻译和文本生成。

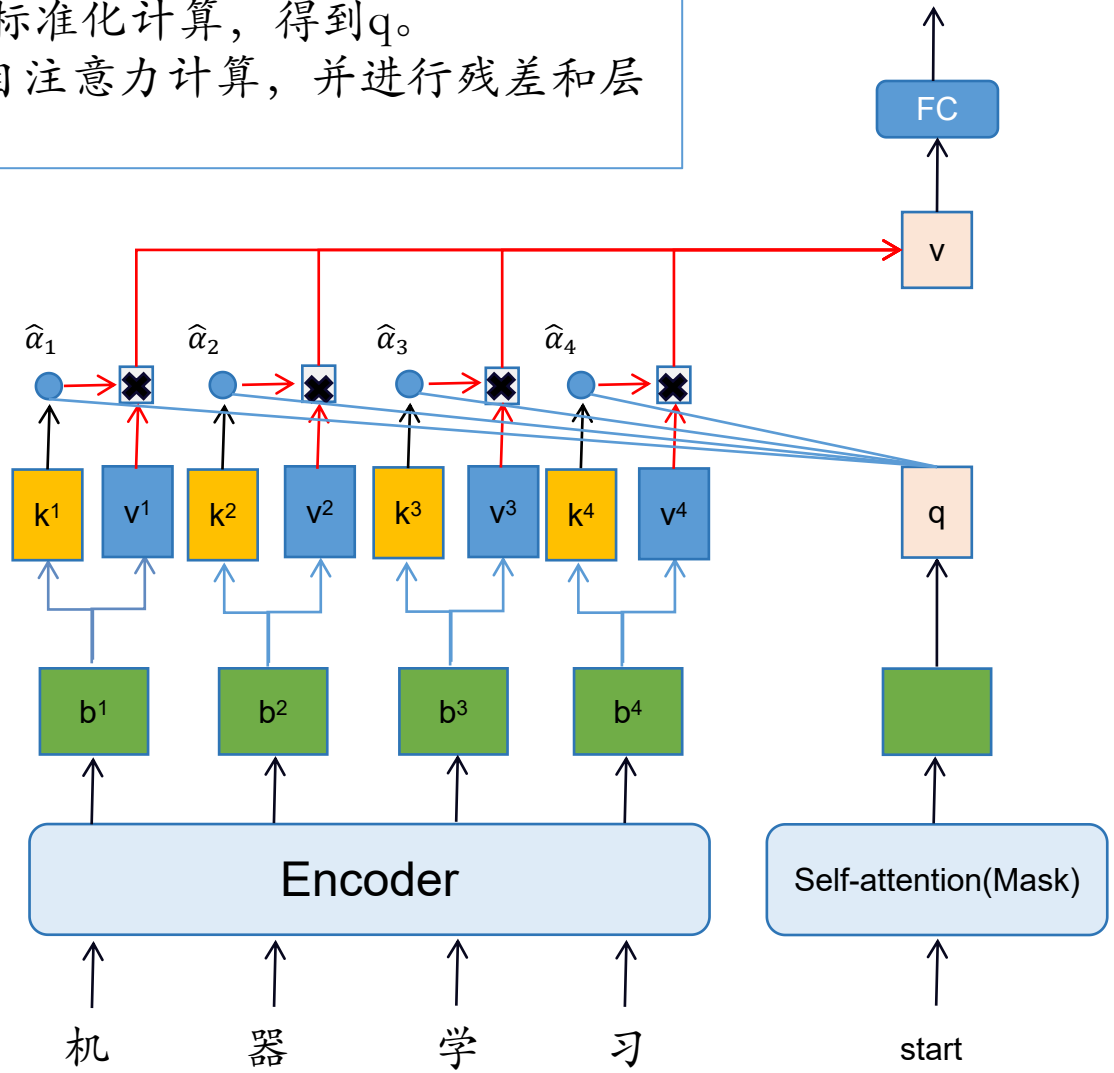
因为只依赖之前的输出，因此需要将训练样本的后面部分进行遮挡，这个过程称为掩码。

掩码具体的实现方式在注意力计算部分，在softmax时将掩码部分权重置0。

4.3 Transformer模型架构 【解码器】

1）对输入进行掩码自注意力计算，并进行残差和层标准化计算，得到q。
2）将q和Encoder部分的输出向量中的k和v一起进行自注意力计算，并进行残差和层标准化，得到v。



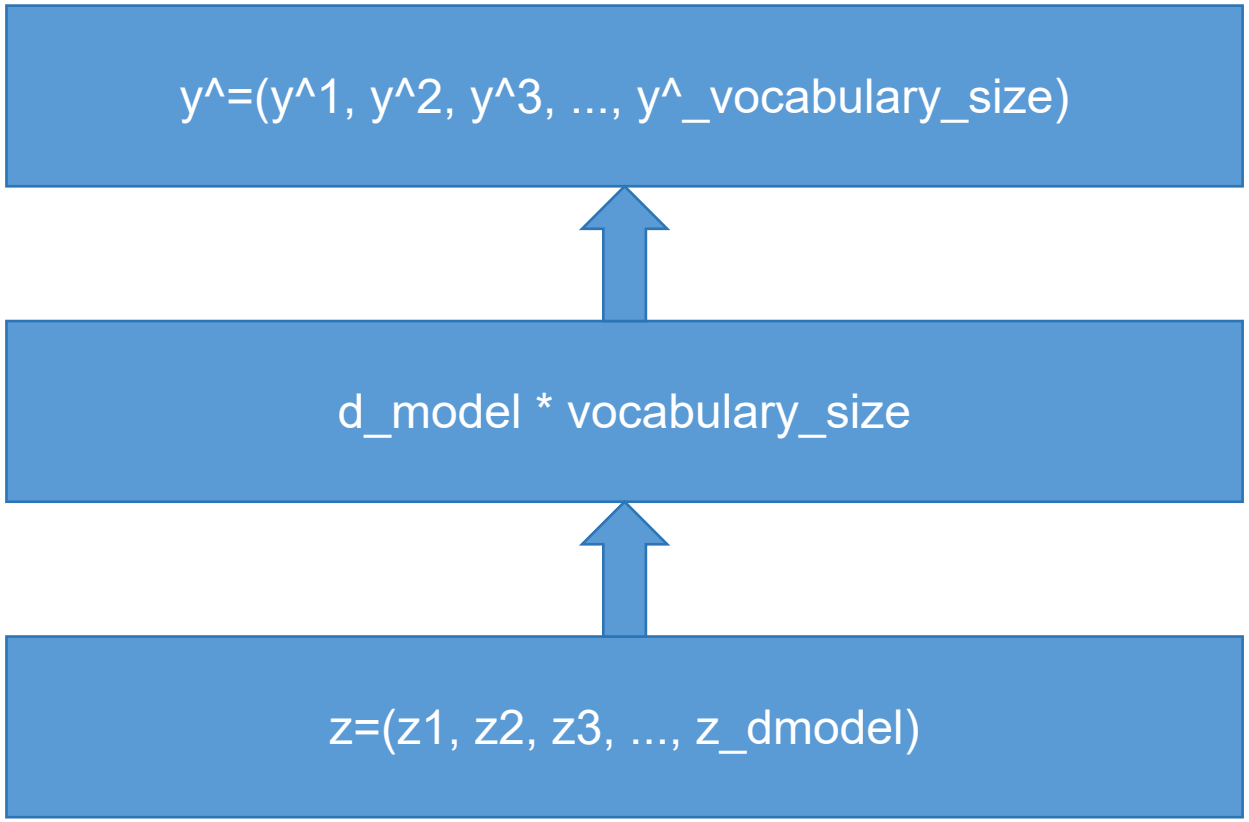


4.3 Transformer模型架构 【目标输出】

softmax输出

线性层

解码器输出



4.4 Transformer训练过程 【参数】

训练数据和批次

1）标准的WMT2014英德数据集。

2）包含约450万个句子对。

3）byte-pair 编码对句子进行编码。

4）源-目标词汇表共包含约37000个标记。

5）每个训练批次包含一组约25000个源标记和25000个目标标记的句子对。

主要参数

1）模型维度：512

2）头数：8

3）多头部分q、k v长度： 512/8 = 64

4）dropout：0.1

5）标签平滑：$\epsilon ls = 0.1$

硬件和训练时间

1）一台装有8个NVIDIA P100 GPU的机器。

2）一轮大约需要0.4秒。

3）共10万步或12小时。

优化器

1）Adam优化器：$\beta 1 = 0.9$，$\beta 2 = 0.98$和$\epsilon = 10-9$。

2）warmup_steps个训练步骤中线性增加学习率，之后按步骤数的倒数平方成比例地降低。warmup_steps=4000

4.4 Transformer训练过程 【结果】

Transformer 在英语到德语和英语到法语 newstest2014 测试中取得了比以前最先进的模型更好的 BLEU 得分，但训练成本仅是之前的一小部分

Model	BLEU		Training Cost (FLOPs)	
	EN-DE	EN-FR	EN-DE	EN-FR
ByteNet [18]	23.75			
Deep-Att + PosUnk [39]		39.2		$1.0 \cdot 10^{20}$
GNMT + RL [38]	24.6	39.92	$2.3 \cdot 10^{19}$	$1.4 \cdot 10^{20}$
ConvS2S [9]	25.16	40.46	$9.6 \cdot 10^{18}$	$1.5 \cdot 10^{20}$
MoE [32]	26.03	40.56	$2.0 \cdot 10^{19}$	$1.2 \cdot 10^{20}$
Deep-Att + PosUnk Ensemble [39]		40.4		$8.0 \cdot 10^{20}$
GNMT + RL Ensemble [38]	26.30	41.16	$1.8 \cdot 10^{20}$	$1.1 \cdot 10^{21}$
ConvS2S Ensemble [9]	26.36	**41.29**	$7.7 \cdot 10^{19}$	$1.2 \cdot 10^{21}$
Transformer (base model)	27.3	38.1	**$3.3 \cdot 10^{18}$**	
Transformer (big)	**28.4**	**41.8**	$2.3 \cdot 10^{19}$	

4.5 Transformer代码实操 【整体架构】

哈佛NLP团队复现了Transformer， 地址：https://github.com/harvardnlp/annotated-transformer。

1）原始输入部分：

nn.Sequential(Embeddings(d_model, src_vocab), c(position))

2）编码器部分：

Encoder(EncoderLayer(d_model, c(attn), c(ff), dropout), N)

3）目标输入部分：

nn.Sequential(Embeddings(d_model, tgt_vocab), c(position))

4）解码器部分：

Decoder(DecoderLayer(d_model, c(attn), c(attn), c(ff), dropout), N)

5）目标输出部分：

Generator(d_model, tgt_vocab)

```
def make_model(
    src_vocab, tgt_vocab, N=6, d_model=512, d_ff=2048, h=8, dropout=0.1
):
    "Helper: Construct a model from hyperparameters."
    c = copy.deepcopy
    attn = MultiHeadedAttention(h, d_model)
    ff = PositionwiseFeedForward(d_model, d_ff, dropout)
    position = PositionalEncoding(d_model, dropout)
    model = EncoderDecoder(
        Encoder(EncoderLayer(d_model, c(attn), c(ff), dropout), N),
        Decoder(DecoderLayer(d_model, c(attn), c(attn), c(ff), dropout), N),
        nn.Sequential(Embeddings(d_model, src_vocab), c(position)),
        nn.Sequential(Embeddings(d_model, tgt_vocab), c(position)),
        Generator(d_model, tgt_vocab),
    )

    # Initialize parameters with Glorot / fan_avg.
    for p in model.parameters():
        if p.dim() > 1:
            nn.init.xavier_uniform_(p)
    return model
```

4.5 Transformer代码实操 【训练概要】

EncoderDecoder：负责实现编码器和解码器功能。

Generator：负责实现最终输出部分功能。

```
class Generator(nn.Module):
    "Define standard linear + softmax generation step."

    def __init__(self, d_model, vocab):
        super(Generator, self).__init__()
        # See: https://pytorch.org/docs/stable/generated/torch.nn.Linear.html
        self.proj = nn.Linear(d_model, vocab)

    def forward(self, x):
        return log_softmax(self.proj(x), dim=-1)
```

```
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)
```

4.5 Transformer代码实操 【输入部分】

```
class Embeddings(nn.Module):
    def __init__(self, d_model, vocab):
        super(Embeddings, self).__init__()
        self.lut = nn.Embedding(vocab, d_model)
        self.d_model = d_model

    def forward(self, x):
        return self.lut(x) * math.sqrt(self.d_model)
```

负责Embedding层功能

```
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)
```

负责位置编码功能

4.5 Transformer代码实操 【编码器部分一】

```
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)
```

负责编码器功能

```
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)
```

负责编码器每一层功能

```
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)))
```

负责层的连接：残差和层标准化功能

```
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()))
```

负责基于位置的全连接功能

```
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)
        nbatches = query.size(0)

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

        # 2) Apply attention on all the projected vectors in batch.
        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(nbatches, -1, self.h * self.d_k)
        )
        del query
        del key
        del value
        return self.linears[-1](x)
```

负责多头注意力功能

4.5 Transformer代码实操 【编码器部分二】

接上页

```
class LayerNorm(nn.Module):
    "Construct a layernorm module (See citation for details)."

    def __init__(self, features, eps=1e-6):
        super(LayerNorm, self).__init__()
        self.a_2 = nn.Parameter(torch.ones(features))
        self.b_2 = nn.Parameter(torch.zeros(features))
        self.eps = eps

    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
```

负责层标准化功能

```
def attention(query, key, value, mask=None, dropout=None):
    "Compute 'Scaled Dot Product Attention'"
    d_k = query.size(-1)
    # Here is the attention function
    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
```

负责自注意力计算功能

4.5 Transformer代码实操 【解码器部分】

```
class Decoder(nn.Module):
    "Generic N layer decoder with masking."

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

    def forward(self, x, memory, src_mask, tgt_mask):
        for layer in self.layers:
            x = layer(x, memory, src_mask, tgt_mask)
        return self.norm(x)
```

负责解码器部分功能

```
class DecoderLayer(nn.Module):
    "Decoder is made of self-attn, src-attn, and feed forward (defined below)"

    def __init__(self, size, self_attn, src_attn, feed_forward, dropout):
        super(DecoderLayer, self).__init__()
        self.size = size
        self.self_attn = self_attn
        self.src_attn = src_attn
        self.feed_forward = feed_forward
        self.sublayer = clones(SublayerConnection(size, dropout), 3)

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

负责解码器每一层功能

感谢您的观看