Attention Is All You Need

2017.6

Introduction

Background

- long short-term memory (LSTM)

Long Distance Dependency

- Recurrent neural networks(RNN)

Parallelization （Efficiency）

- ConvS2S(convolution) ， ByteNet

Transformer
(rely entirely on **attention** mechanism)

Long Distance Dependency

Parallelization （Efficiency）

Flops （Parameters）

Background

Table 1: Maximum path lengths, per-layer complexity and minimum number of sequential operations for different layer types. n is the sequence length, d is the representation dimension, k is the kernel size of convolutions and r the size of the neighborhood in restricted self-attention.

Layer Type	Complexity per Layer	Sequential Operations	Maximum Path Length
Self-Attention	$O(n^2 \cdot d)$	$O(1)$	$O(1)$
Recurrent	$O(n \cdot d^2)$	$O(n)$	$O(n)$
Convolutional	$O(k \cdot n \cdot d^2)$	$O(1)$	$O(log_k(n))$
Self-Attention (restricted)	$O(r \cdot n \cdot d)$	$O(1)$	$O(n/r)$

Application

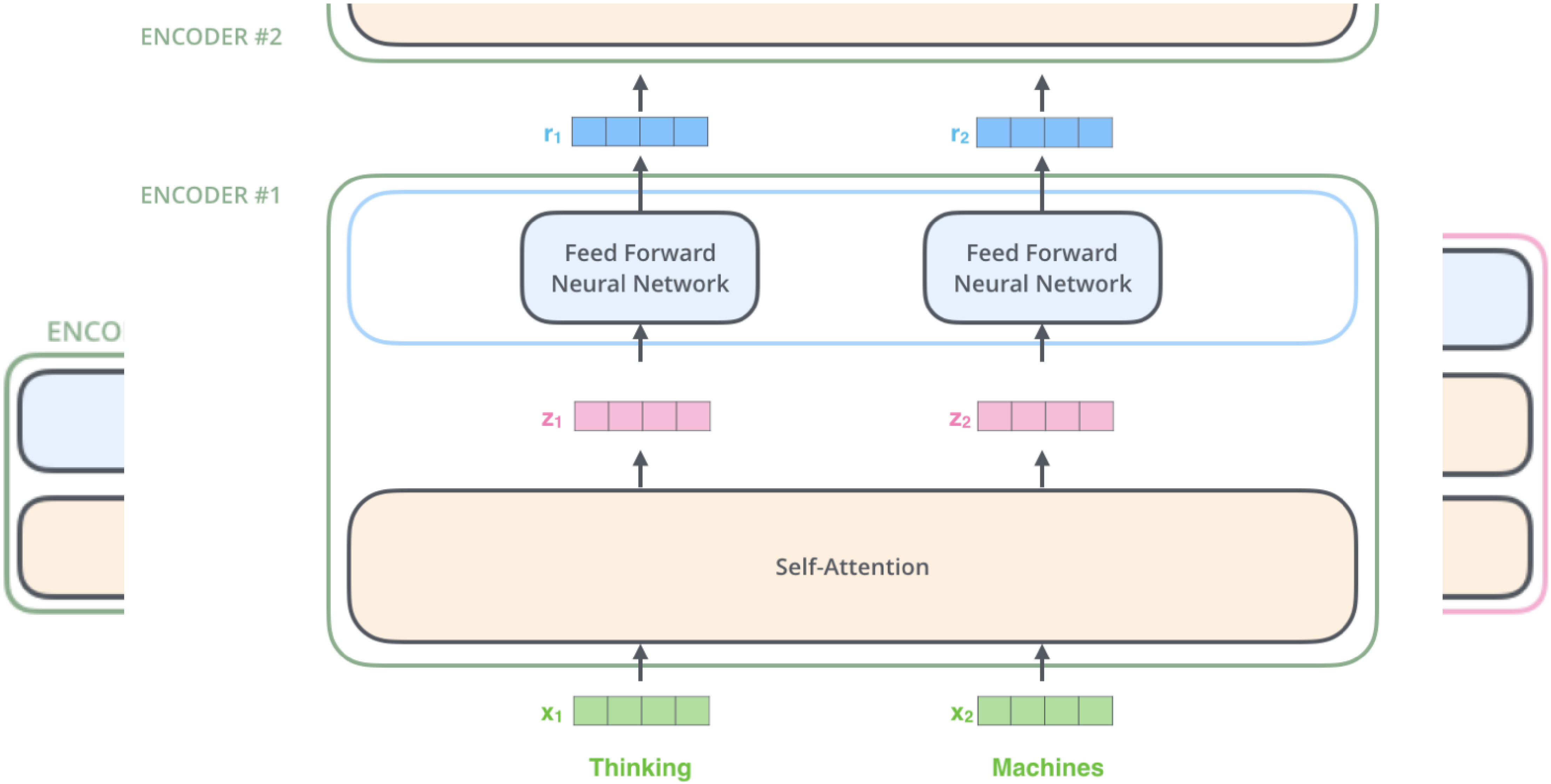
NLP

- Sequence modeling(Seq2Seq)
- Transduction problems(Language modeling and Machine translation…)
- Question Answer, Dialog System or Chatbot, Classification, Augmentation
- (BERT 2018 : the sota modle in 11 NLP tasks!)

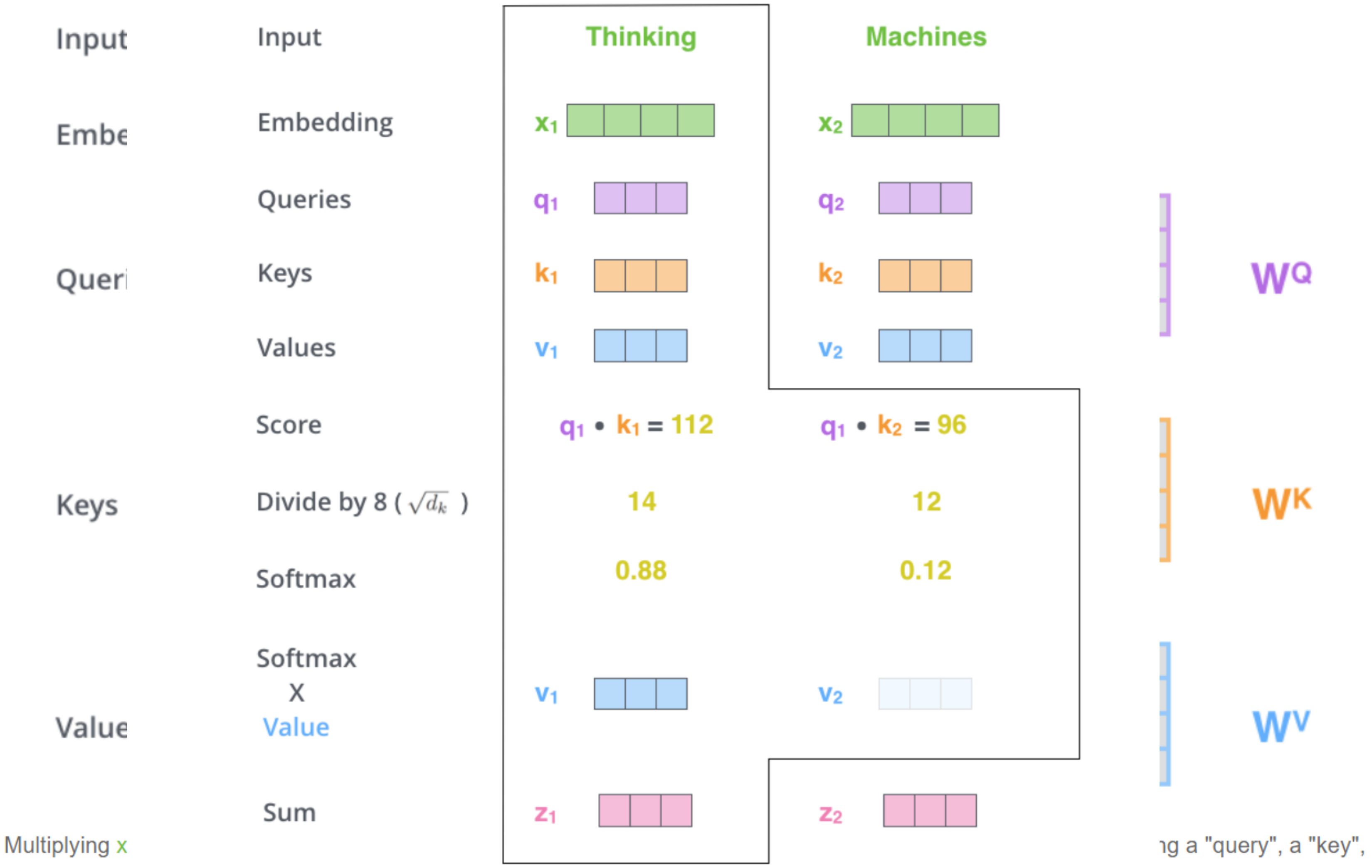
CV

- DETR(Object Detection)
- ViT(Classification)
- Super-Resolution

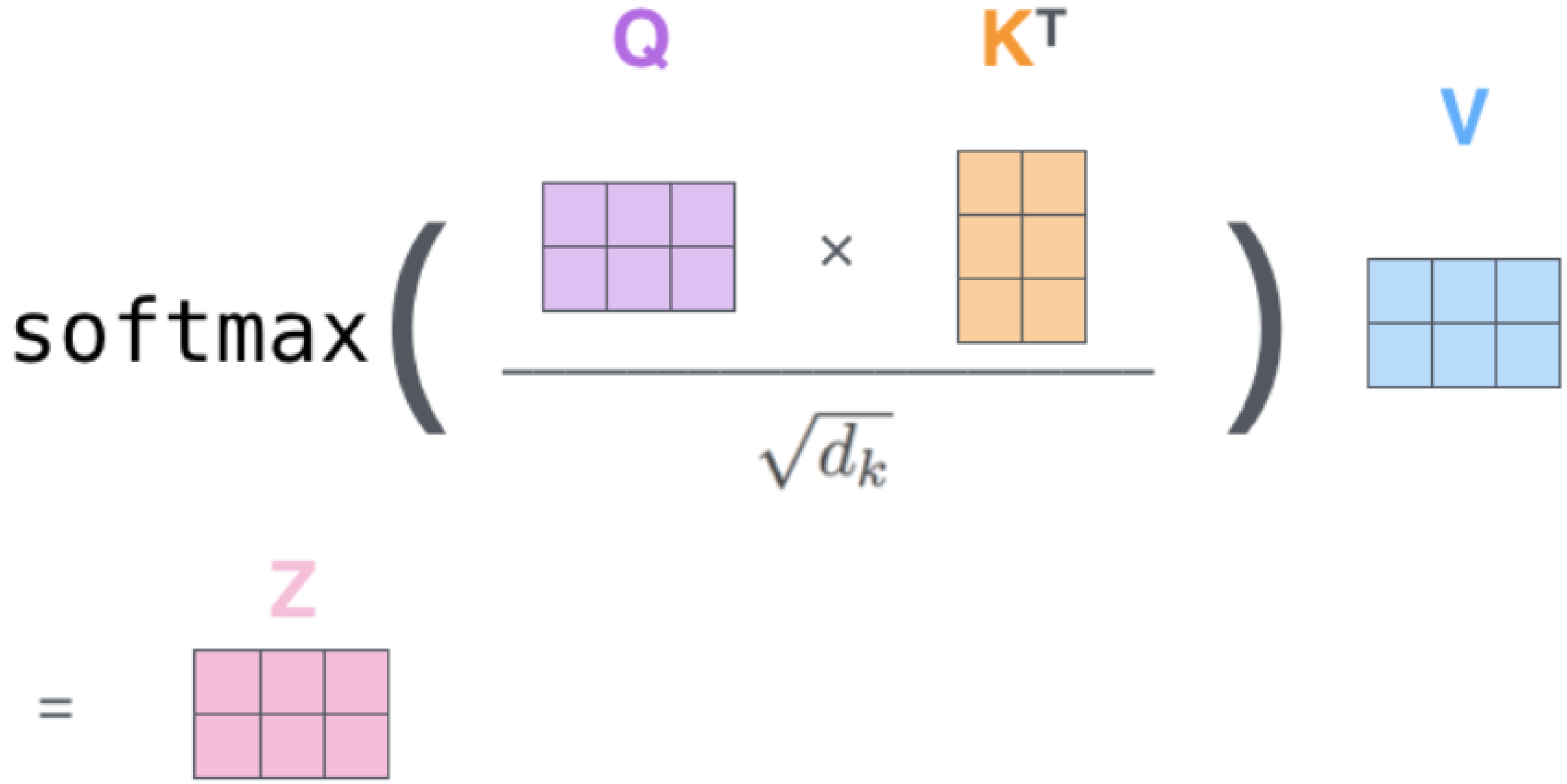
Model Architecture



Model Architecture—Self-Attention



Model Architecture—Self-Attention



The self-attention calculation in matrix form

or 4 boxes in the figure), and the q/k/v vectors (64, or 3 boxes in the figure)

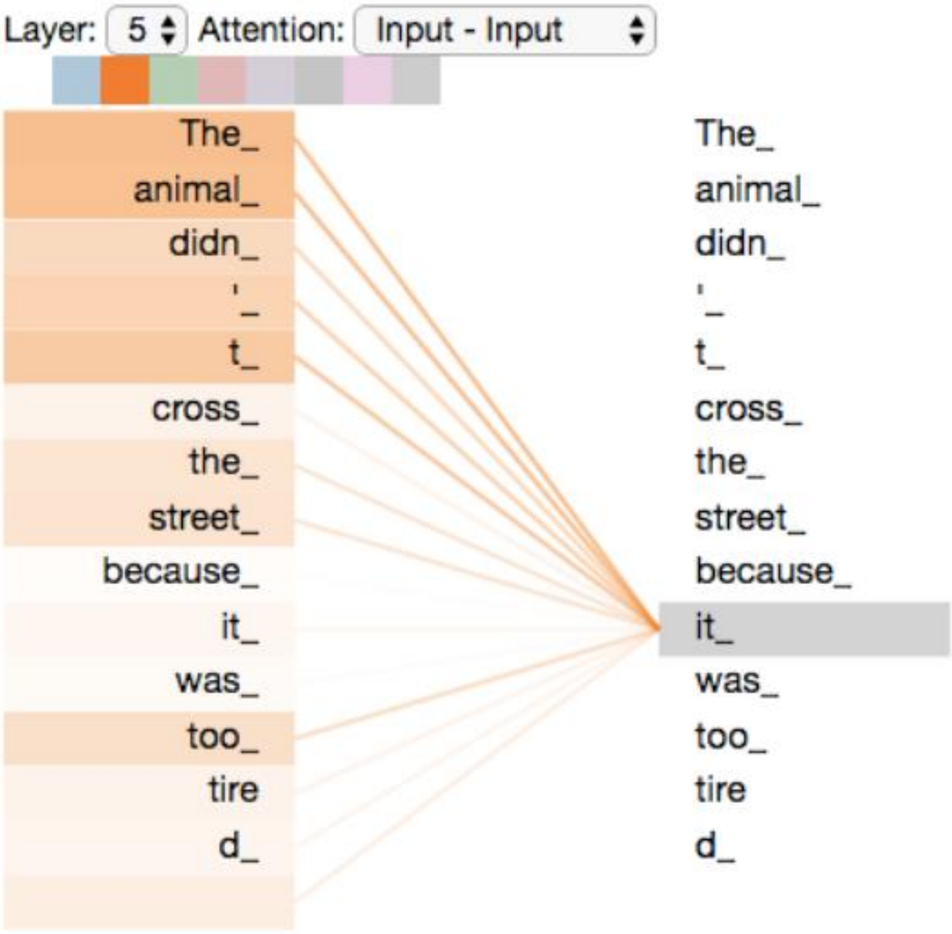
Model Architecture—Self-Attention

```python
import torch
import torch.nn as nn
import numpy as np

__author__ = "Yu-Hsiang Huang"

class ScaledDotProductAttention(nn.Module):
    ''' Scaled Dot-Product Attention '''

    def __init__(self, temperature, attn_dropout=0.1):
        super().__init__()
        self.temperature = temperature     #temperature就是每个输入的embedding维度d的开根号
        self.dropout = nn.Dropout(attn_dropout)      #为防止过拟合，每次随机丢弃0.1的数据点
        self.softmax = nn.Softmax(dim=2)    #对dim=2（对于a*b*c dim=0是a维度上归一 dim=1是每个b*c的列归一 dim

    def forward(self, q, k, v, mask=None):

        attn = torch.bmm(q, k.transpose(1, 2))
        attn = attn / self.temperature

        if mask is not None:
            attn = attn.masked_fill(mask, -np.inf)  #mask为一个遮盖矩阵

        attn = self.softmax(attn)
        attn = self.dropout(attn)   #输出经过dropout的一部分权重矩阵QK seq*seq
        output = torch.bmm(attn, v) #输出seq*feature_v

        return output, attn
```

Model Architecture—Multi-Head Attention(output)

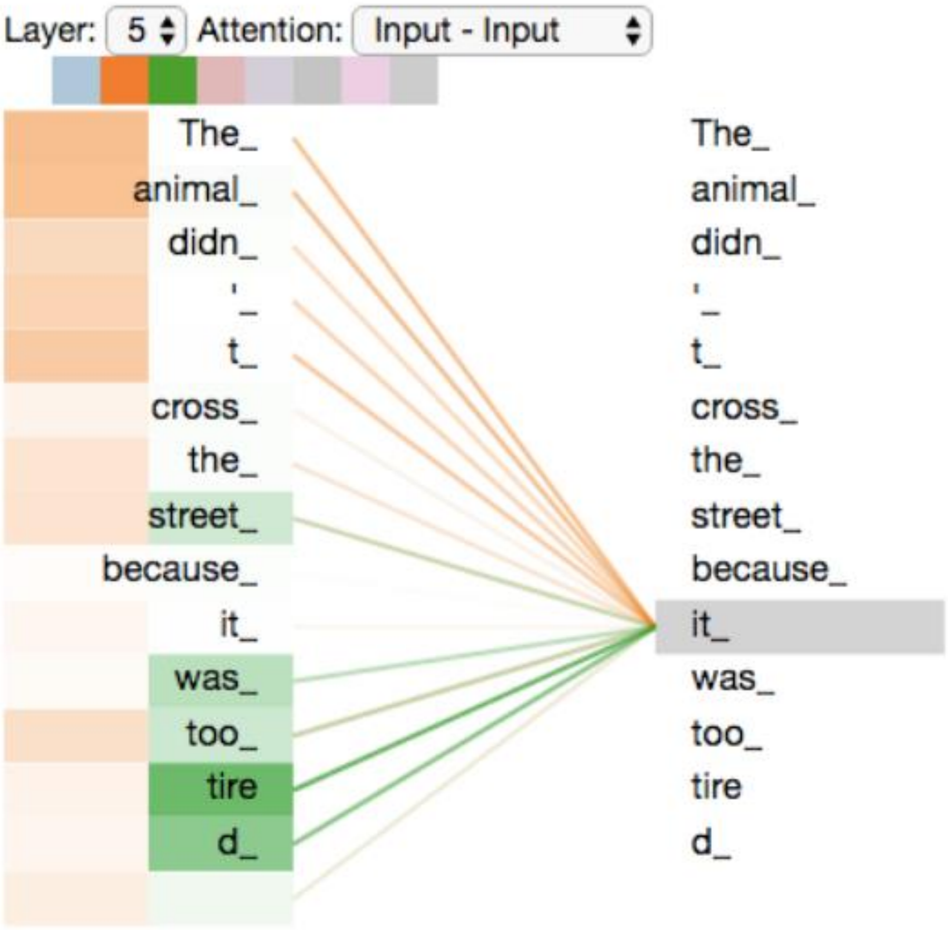
Attention

Multi-head Attention

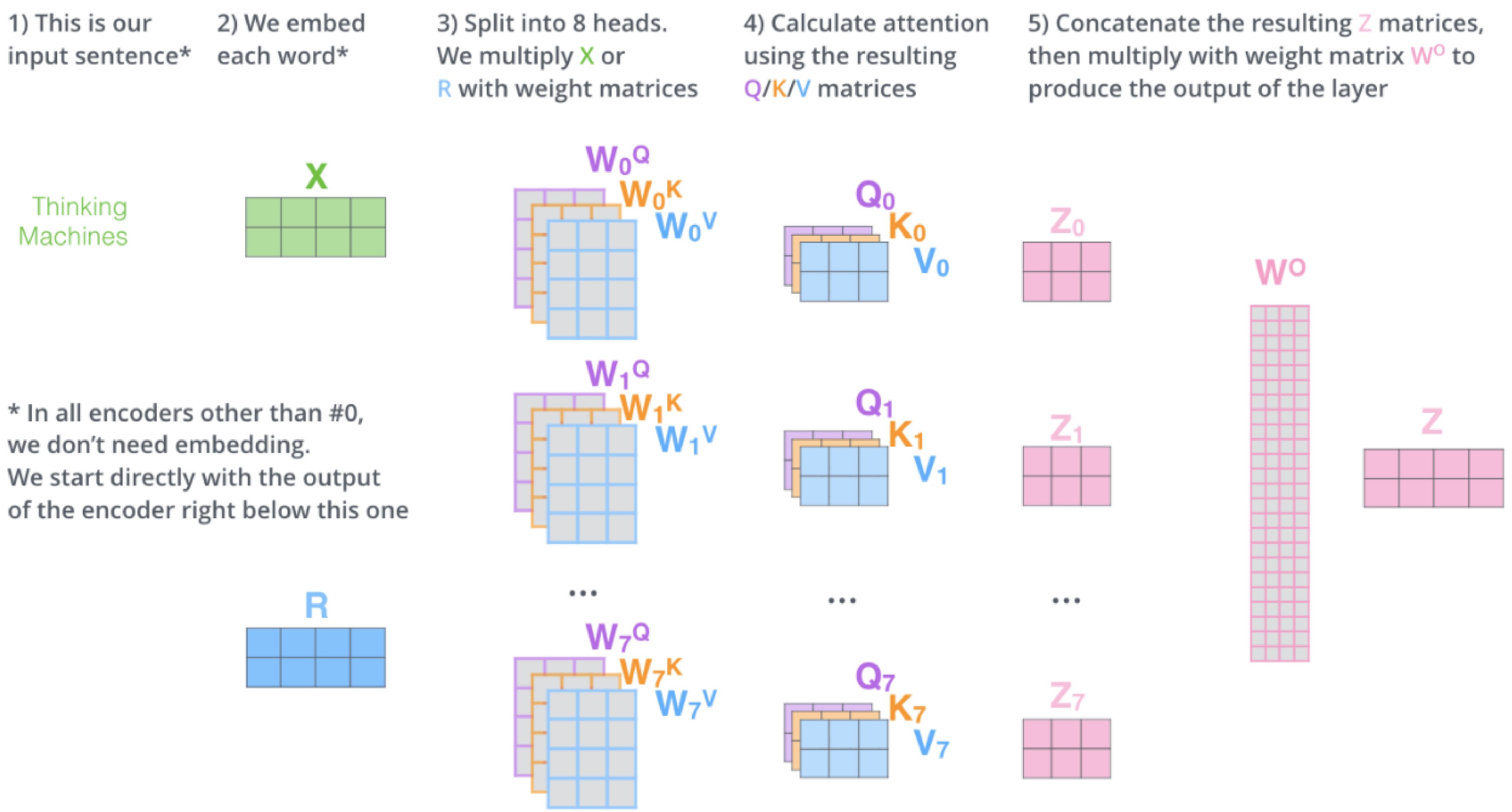


Features

Parameters



Model Architecture—Multi-Head Attention

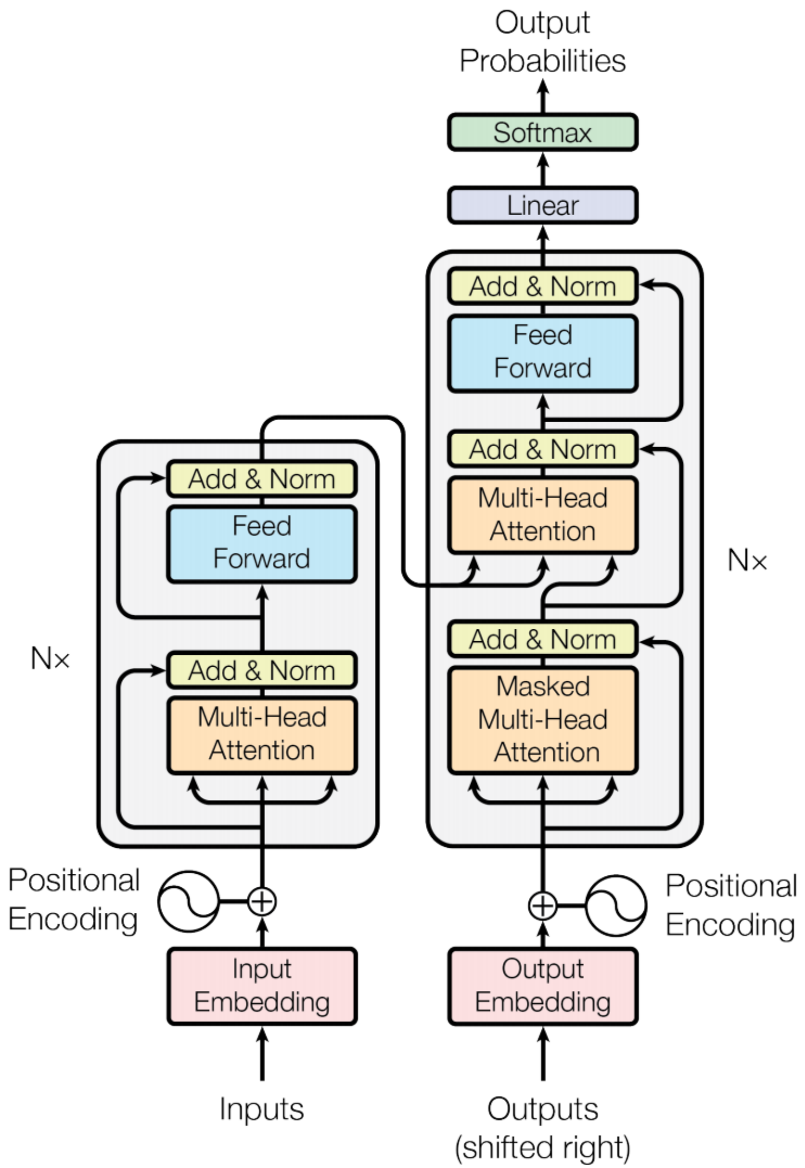


Model Architecture—Multi-Head Attention

```python
class MultiHeadAttention(nn.Module):
    ''' Multi-Head Attention module '''

    def __init__(self, n_head, d_model, d_k, d_v, dropout=0.1): #d_model == d_feature * n_heads
        super().__init__()

        self.n_head = n_head
        self.d_k = d_k
        self.d_v = d_v

        self.w_qs = nn.Linear(d_model, n_head * d_k)    #初始化三个矩阵 eg.W_qs输入是d_model（初始的feature）输出是d_k*n_head（经过变换后的feature）
        self.w_ks = nn.Linear(d_model, n_head * d_k)
        self.w_vs = nn.Linear(d_model, n_head * d_v)
        nn.init.normal_(self.w_qs.weight, mean=0, std=np.sqrt(2.0 / (d_model + d_k)))   #要求三个矩阵初始化参数正态分布
        nn.init.normal_(self.w_ks.weight, mean=0, std=np.sqrt(2.0 / (d_model + d_k)))
        nn.init.normal_(self.w_vs.weight, mean=0, std=np.sqrt(2.0 / (d_model + d_v)))

        self.attention = ScaledDotProductAttention(temperature=np.power(d_k, 0.5))  #防止随着输出维数过大 点积过大
        self.layer_norm = nn.LayerNorm(d_model)    #对每个样本维度长度是d_model的维度（也就是feature维度）标准化

        self.fc = nn.Linear(n_head * d_v, d_model)  #初始化最后的矩阵的fc =d_model * (n_head * d_v)（把multihead的feature数调回原始 在这里其实维数没变）
        nn.init.xavier_normal_(self.fc.weight)

        self.dropout = nn.Dropout(dropout)


    def forward(self, q, k, v, mask=None):

        d_k, d_v, n_head = self.d_k, self.d_v, self.n_head

        sz_b, len_q, _ = q.size()    #?q不是没有传进来吗
        sz_b, len_k, _ = k.size()
        sz_b, len_v, _ = v.size()

        residual = q

        q = self.w_qs(q).view(sz_b, len_q, n_head, d_k)
        k = self.w_ks(k).view(sz_b, len_k, n_head, d_k)
        v = self.w_vs(v).view(sz_b, len_v, n_head, d_v)

        q = q.permute(2, 0, 1, 3).contiguous().view(-1, len_q, d_k) # (n*b) x lq x dk 没弄懂
        k = k.permute(2, 0, 1, 3).contiguous().view(-1, len_k, d_k) # (n*b) x lk x dk
        v = v.permute(2, 0, 1, 3).contiguous().view(-1, len_v, d_v) # (n*b) x lv x dv

        mask = mask.repeat(n_head, 1, 1) # (n*b) x .. x .. 没弄懂
        output, attn = self.attention(q, k, v, mask=mask)

        output = output.view(n_head, sz_b, len_q, d_v)
        output = output.permute(1, 2, 0, 3).contiguous().view(sz_b, len_q, -1) # b x lq x (n*dv)

        output = self.dropout(self.fc(output))
        output = self.layer_norm(output + residual)

        return output, attn    #输出seq*feature_v
```

Model Architecture

- Add & Norm
- Feed-Forward Networks
- Positional encoding
- Connect between encode and decode
- Masked Multi-Head Attention
- Softmax



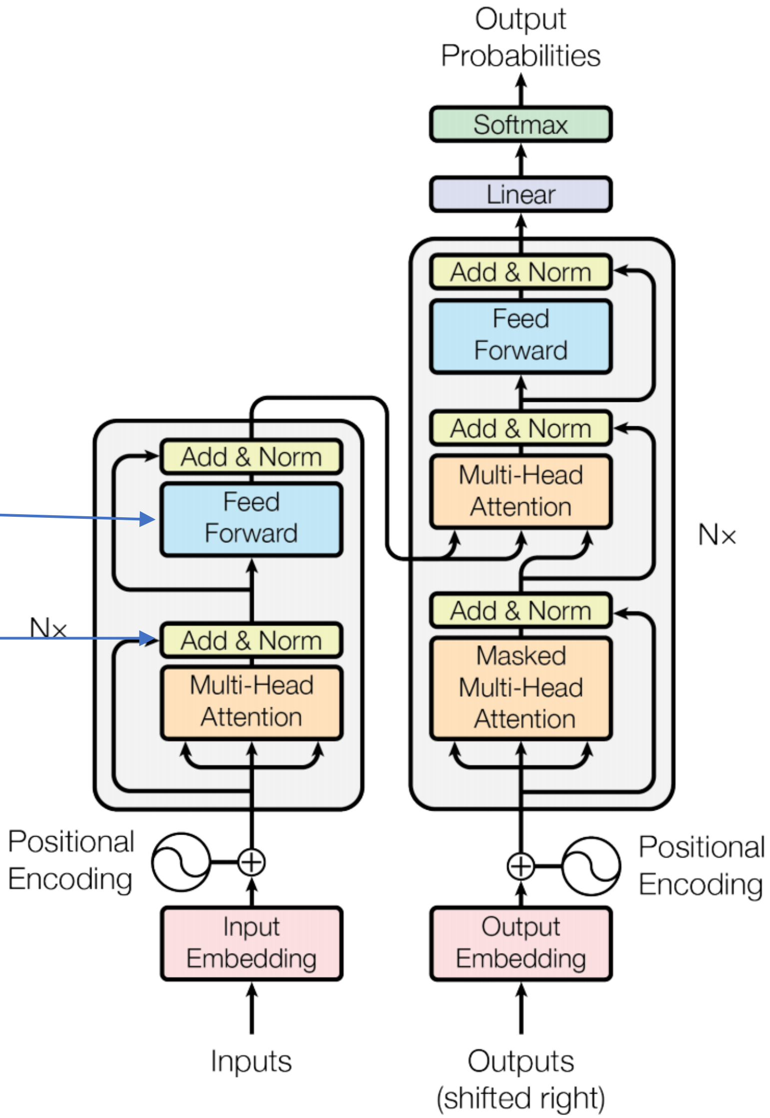
Model Architecture

$$FFN(x) = max(0, xW1 + b1)W2 + b2$$

$$Layernorm(x + mul\_attn(x))$$

$$PE(pos, 2i) = sin(pos/100002i/dmodel)$$

$$PE(pos, 2i+1) = cos(pos/100002i/dmodel)$$

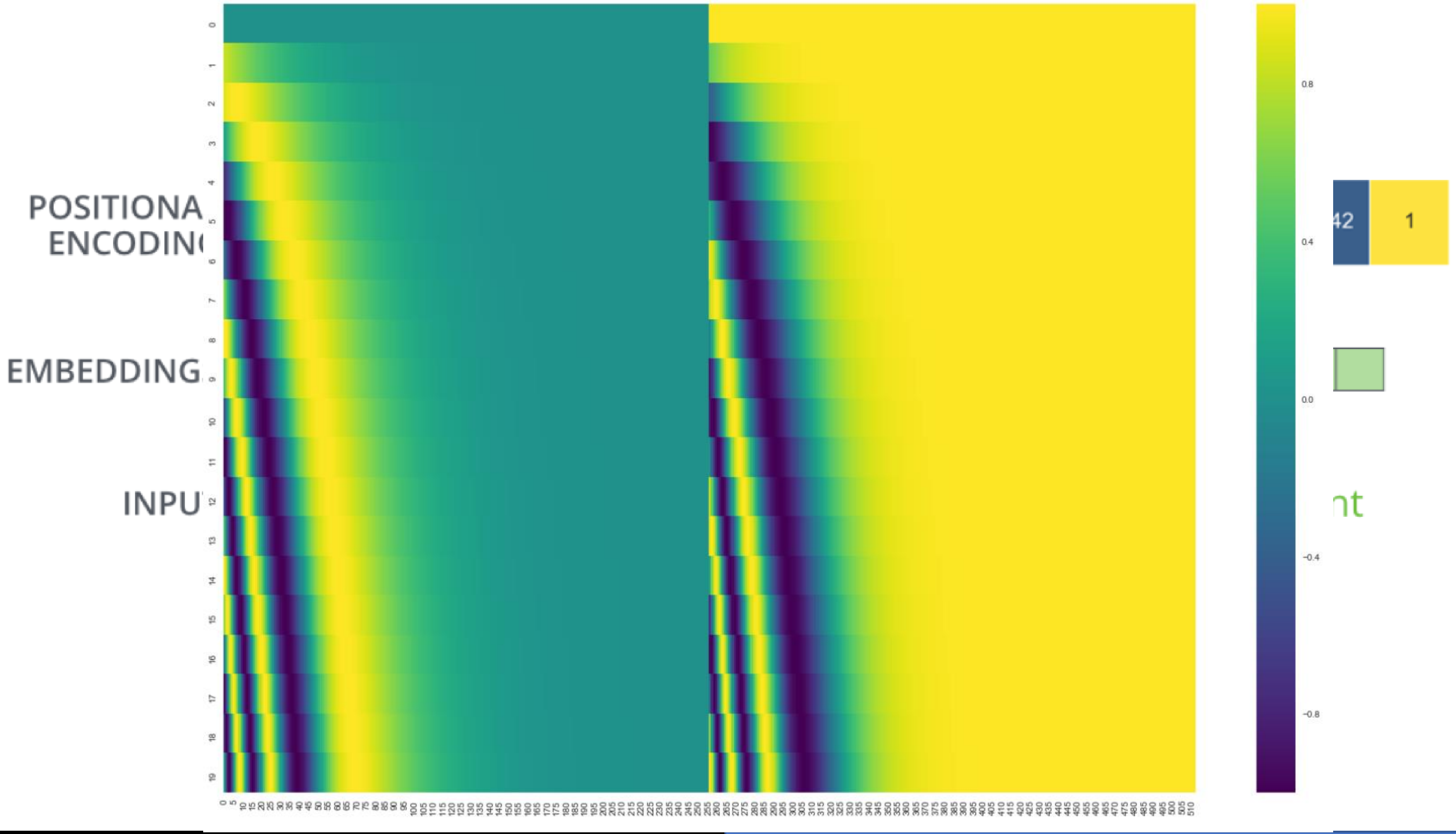


Model Architecture——FFN

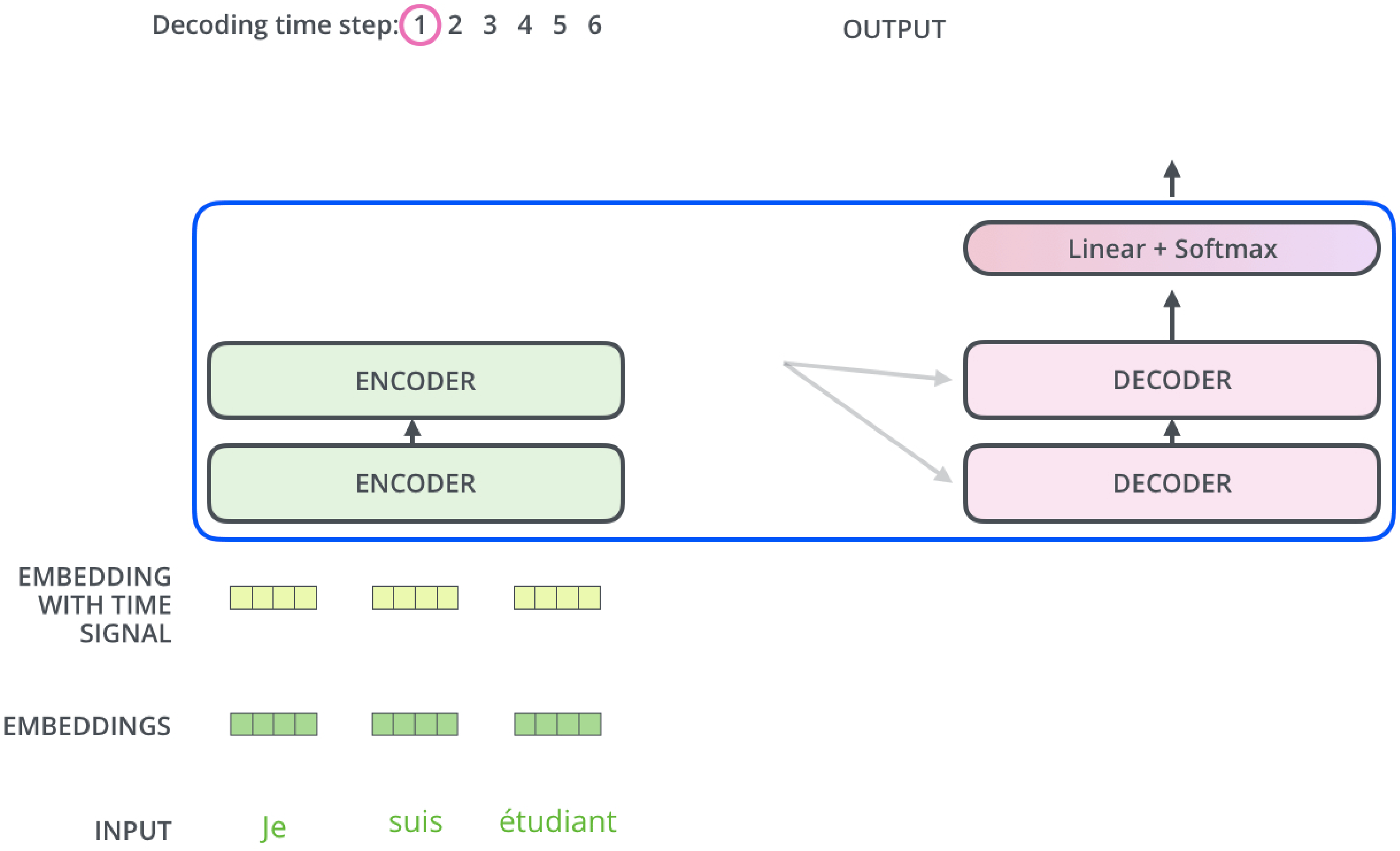
$$FFN(x) = max(0, xW1 + b1)W2 + b2$$

```python
class PositionwiseFeedForward(nn.Module):
    ''' A two-feed-forward-layer module '''

    def __init__(self, d_in, d_hid, dropout=0.1):
        super().__init__()
        self.w_1 = nn.Conv1d(d_in, d_hid, 1) # position-wise
        self.w_2 = nn.Conv1d(d_hid, d_in, 1) # position-wise
        self.layer_norm = nn.LayerNorm(d_in)
        self.dropout = nn.Dropout(dropout)

    def forward(self, x):
        residual = x
        output = x.transpose(1, 2)
        output = self.w_2(F.relu(self.w_1(output)))
        output = output.transpose(1, 2)
        output = self.dropout(output)
        output = self.layer_norm(output + residual)
        return output
```

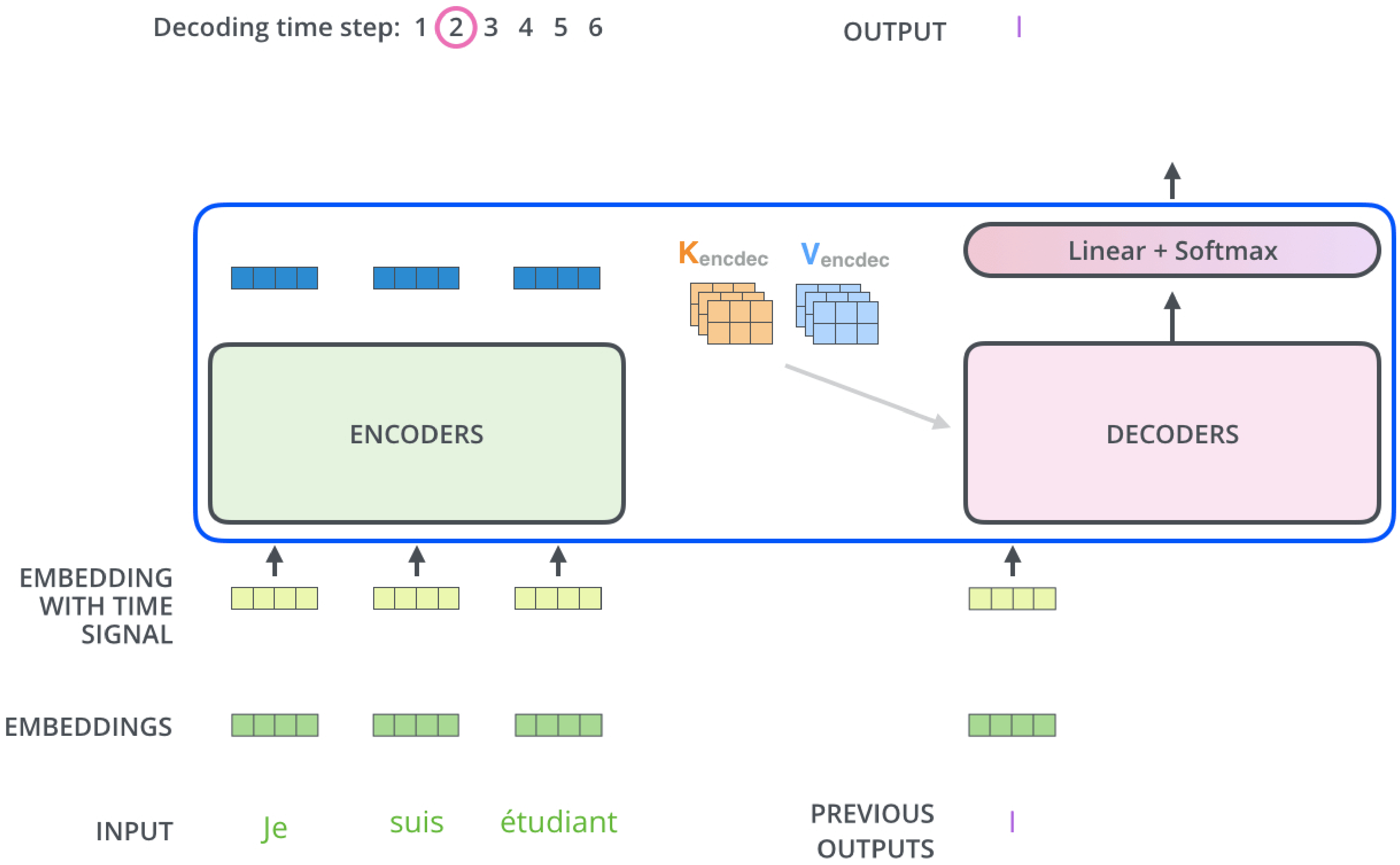
Model Architecture—Positional encoding



Model Architecture—Connection between encode&decode



Model Architecture—Position between encode&decode



Model Architecture—Position between encode&decode

$$Q = K = V = \begin{bmatrix} s_1 \\ s_2 \\ s_3 \\ s_4 \end{bmatrix} \qquad QK^T = \begin{bmatrix} s_1 s_1^T & s_1 s_2^T & s_1 s_3^T & s_1 s_4^T \\ s_2 s_1^T & s_2 s_2^T & s_2 s_3^T & s_2 s_4^T \\ s_3 s_1^T & s_3 s_2^T & s_3 s_3^T & s_3 s_4^T \\ s_4 s_1^T & s_4 s_2^T & s_4 s_3^T & s_4 s_4^T \end{bmatrix}$$

$$mask * QK^T = (QK^T)' = \begin{bmatrix} s_1 s_1^T & -\infty & -\infty & -\infty \\ s_2 s_1^T & s_2 s_2^T & -\infty & -\infty \\ s_3 s_1^T & s_3 s_2^T & s_3 s_3^T & -\infty \\ s_4 s_1^T & s_4 s_2^T & s_4 s_3^T & s_4 s_4^T \end{bmatrix}$$

$$out = soft\max\left((QK^T)' / \sqrt{4}\right) * V = \begin{bmatrix} a_{11} s_1 & 0 & 0 & 0 \\ a_{21} s_1 & a_{22} s_2 & 0 & 0 \\ a_{31} s_1 & a_{32} s_2 & a_{33} s_3 & 0 \\ a_{41} s_1 & a_{42} s_2 & a_{43} s_3 & a_{44} s_4 \end{bmatrix}$$

Model Architecture —— Output

Since the model's parameters (weights) are all initialized randomly, the (untrained) model produces a probability distribution with arbitrary values for each cell/word. We can compare it with the actual output, then tweak all the model's weights using backpropagation to make the output closer to the desired output.

Model Architecture —— Train & Validation

	N	d_{model}	d_{ff}	h	d_k	d_v	P_{drop}	ϵ_{ls}	train steps	PPL (dev)	BLEU (dev)	params $\times 10^6$
base	6	512	2048	8	64	64	0.1	0.1	100K	4.92	25.8	65
(A)				1	512	512				5.29	24.9	
				4	128	128				5.00	25.5	
				16	32	32				4.91	25.8	
				32	16	16				5.01	25.4	
(B)					16					5.16	25.1	58
					32					5.01	25.4	60
(C)	2									6.11	23.7	36
	4									5.19	25.3	50
	8									4.88	25.5	80
		256			32	32				5.75	24.5	28
		1024			128	128				4.66	26.0	168
			1024							5.12	25.4	53
			4096							4.75	26.2	90
(D)							0.0			5.77	24.6	
							0.2			4.95	25.5	
								0.0		4.67	25.3	
								0.2		5.47	25.7	
(E)	positional embedding instead of sinusoids									4.92	25.7	
big	6	1024	4096	16			0.3		300K	**4.33**	**26.4**	213

Thanks for listening.