Attention is all you need (Transformer)

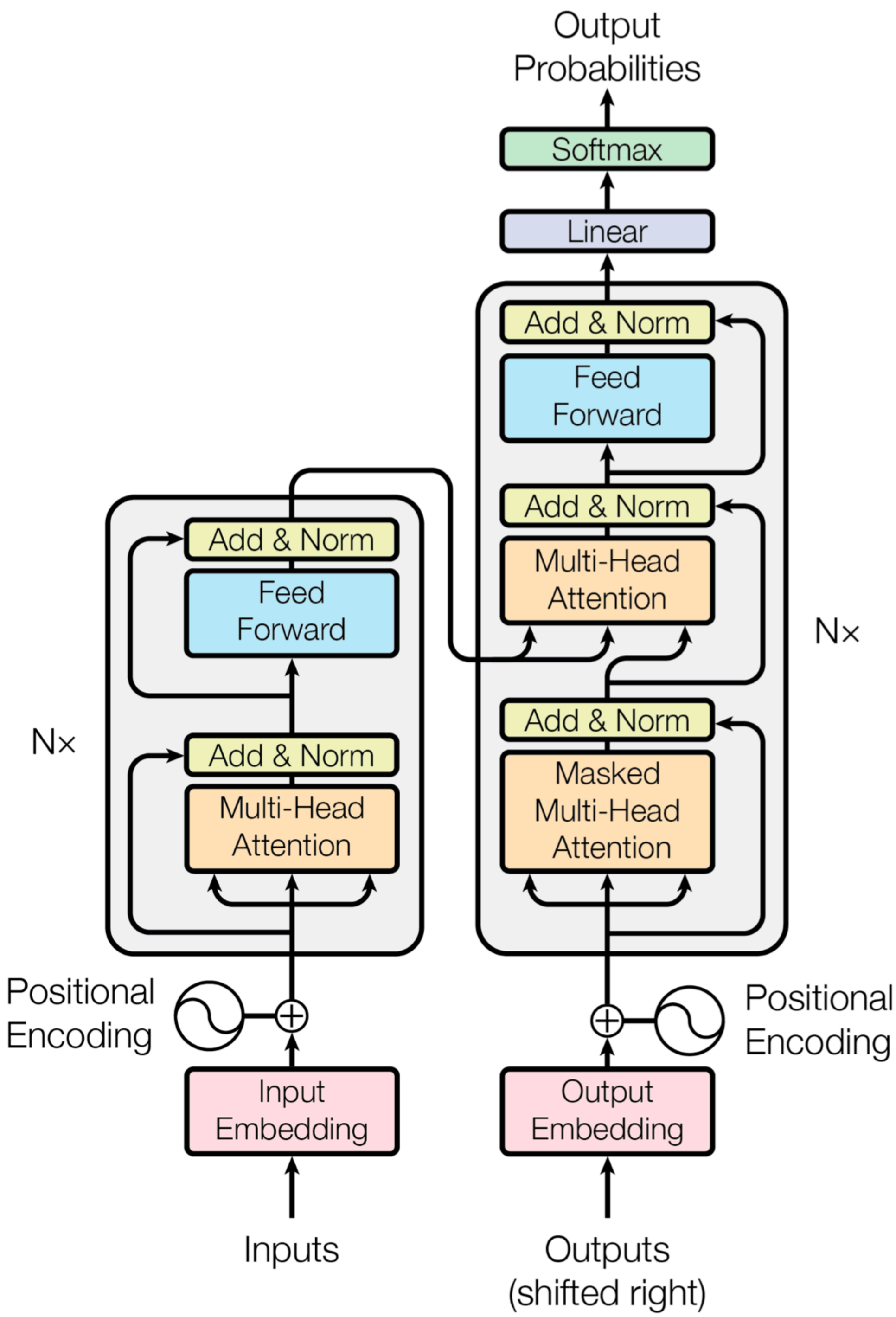


Figure 1: The Transformer - model architecture.

Attention

> ```
> An attention function can be described as mapping a query and a set of key-value pairs to an output, where the query, keys, values, and output are all vectors. The output is computed as a weighted sum of the values, where the weight assigned to each value is computed by a compatibility function of the query with the corresponding key.
> ```

- 对于某个时刻输出 `y`， 它在输入`x` 上各个部分的注意力。可以理解为**权重**。
- 不同机制下的 **attention** 计算方法

Name	Alignment score function	Citation
Additive(*)	$\text{score}(\boldsymbol{s}_t, \boldsymbol{h}_i) = \mathbf{v}_a^\top \tanh(\mathbf{W}_a[\boldsymbol{s}_t; \boldsymbol{h}_i])$	Bahdanau2015
Location-Base	$\alpha_{t,i} = \text{softmax}(\mathbf{W}_a \boldsymbol{s}_t)$ Note: This simplifies the softmax alignment max to only depend on the target position.	Luong2015
General	$\text{score}(\boldsymbol{s}_t, \boldsymbol{h}_i) = \boldsymbol{s}_t^\top \mathbf{W}_a \boldsymbol{h}_i$ where $\mathbf{W}_a$ is a trainable weight matrix in the attention layer.	Luong2015
Dot-Product	$\text{score}(\boldsymbol{s}_t, \boldsymbol{h}_i) = \boldsymbol{s}_t^\top \boldsymbol{h}_i$	Luong2015
Scaled Dot-Product(^)	$\text{score}(\boldsymbol{s}_t, \boldsymbol{h}_i) = \frac{\boldsymbol{s}_t^\top \boldsymbol{h}_i}{\sqrt{n}}$ Note: very similar to the dot-product attention except for a scaling factor; where n is the dimension of the source hidden state.	Vaswani2017
Self-Attention(&)	Relating different positions of the same input sequence. Theoretically the self-attention can adopt any score functions above, but just replace the target sequence with the same input sequence.	Cheng2016
Global/Soft	Attending to the entire input state space.	Xu2015
Local/Hard	Attending to the part of input state space; i.e. a patch of the input image.	Xu2015; Luong2015

- 其中，S_t 指的是输出序列的隐藏状态 ,h_i 为输入序列的隐藏状态

Self-Attention

- **输出序列** 为 **输入序列**

Transformer中的Multi-head Self-Attention（Dot-product）

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

其中，d_k 表示k的维度，paper里默认 `64`

当k很大时，得到的点积结果很大，使得结果处softmax梯度很小，不利于bp。

- 在encoder的self-attention中，Q，K，V是上一层的encoder输出。对于第一层，它们是word-embedding和position-embedding相加得到的输出。
- 在decoder的self-attention中，Q、K、V都来自于同一个地方（相等），它们是上一层decoder的输出。对于第一层decoder，它们就是word embedding和positional encoding相加得到的输入。但是对于decoder，我们不希望它能获得下一个time step（即将来的信息），因此我们需要进行**sequence masking**。
- 在encoder-decoder attention中，Q来自于decoder的上一层的输出，K和V来自于encoder的输出，K和V是一样的。
- Q、K、V三者的维度一样，即 $d_q = d_k = d_v$

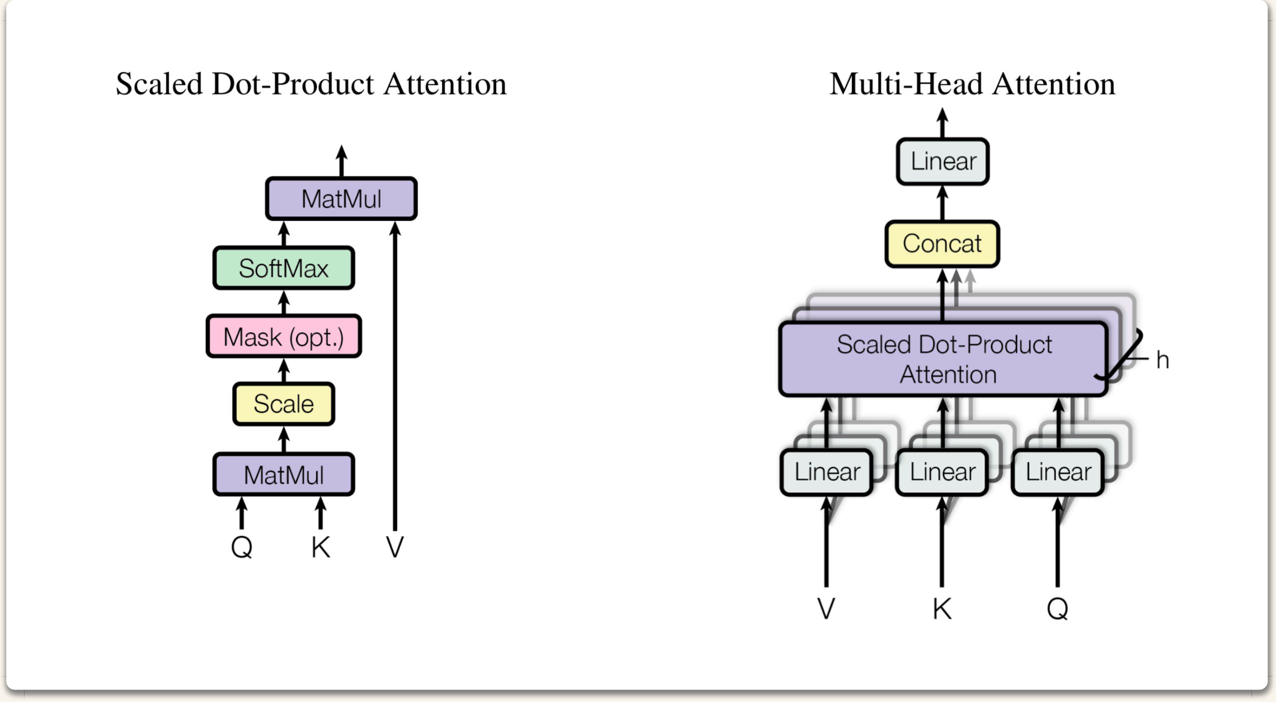
```
import torch
import torch.nn as nn


class ScaledDotProductAttention(nn.Module):
    """Scaled dot-product attention mechanism."""
def __init__(self, attention_dropout=0.0):
    super(ScaledDotProductAttention,
self).__init__()
    self.dropout = nn.Dropout(attention_dropout)
    self.softmax = nn.Softmax(dim=2)
```

```
def forward(self, q, k, v, scale=None,
attn_mask=None):
    """前向传播.

    Args:
      q: Queries张量，形状为[B, L_q, D_q]
      k: Keys张量，形状为[B, L_k, D_k]
      v: Values张量，形状为[B, L_v, D_v]，一般来说就是
k
      scale: 缩放因子，一个浮点标量
      attn_mask: Masking张量，形状为[B, L_q, L_k]

    Returns:
      上下文张量和attetention张量
    """
    attention = torch.bmm(q, k.transpose(1, 2))
    if scale:
      attention = attention * scale
    if attn_mask:
      # 给需要mask的地方设置一个负无穷
      attention =
attention.masked_fill_(attn_mask, -np.inf)
  # 计算softmax
    attention = self.softmax(attention)
  # 添加dropout
    attention = self.dropout(attention)
  # 和v做点积
    context = torch.bmm(attention, v)
    return context, attention
```



- 输入的时候Q，K，V在维度上切分，默认8 ， 输入512 。所以 h 为64。输出的时候再将结果concat，实验结果比不切割直接通过要优。

Layer Normalization

- 作用于**同一个样本**，计算每一个样本上的均值和方差。

Feed forward Network

$$FNN(x) = W^T(max(0, W^T x + b)) + b$$