Self Attention

普通注意力机制

Q·K求相似度，再做一个Scale（防止极端化，例如除$\sqrt{len_K}$）。

然后做Softmax得到概率S，S·V得到新的K和V(K = V)。

这个新的K和V也是暗藏了Q的信息，因为他们是K里面关于Q相似度高的信息。

自注意力机制

Self表示Q、K、V都来源于它自己，每个Token都能提出自己的Q、K、V。

自注意力机制相比普通注意力机制不仅仅K 和 V相等，Q也和他们相等。

这里的相等不是说完全相等，而是来自于同一个输入，三者同源。但三者的权重矩阵不同。。

例如：“我喜欢吃苹果”会被embedding成四个向量：我、喜欢、吃、苹果（Token）

这四个Token生成各自的Q、K、V。

相似度计算：“我”的Q和其他三个K相乘得到相似度，相似度经过Softmax和Scale之后和V相乘，得到Feature。

轮是用“喜欢”的Q和其他三个K相乘得到相似度，相似度经过.....

Positional Encoding

Attention is all you need中出现，因为Transformer没有位置信息，

我喜欢吃苹果和苹果吃喜欢我他们生成的Feature是一样的，只是位置变了，内容没有变。

两种方案

- 通过三角函数生成PE
- 生成可自主学习的PE

位置编码最近又被证明没什么实质性作用。。。。。。不继续深究了。什么单数双数sincos的。

炼丹的人真有意思，全是黑盒，结果论。。。。。。

实现结果好的，就开始编故事，但是为什么好真的有确凿的证据能完整证明吗？没有。

Positional Encoding Code

```
import math
import torch
from torch import nn
from d2l import torch as d2l

num_hiddens, num_heads = 100, 5
attention = d2l.MultiHeadAttention(num_hiddens, num_hiddens, num_hiddens,
                                   num_hiddens, num_heads, 0.5)
print(attention.eval())
```

```
batch_size, num_queries, valid_lens = 2, 4, torch.tensor([3, 2])
X = torch.ones((batch_size, num_queries, num_hiddens))
print(attention(X, X, X, valid_lens).shape)

# Positional Encoding
class PositionalEncoding(nn.Module):
    def __init__(self, num_hiddens, dropout, max_len=1000):
        super(PositionalEncoding, self).__init__()
        self.dropout = nn.Dropout(dropout)
        self.P = torch.zeros((1, max_len, num_hiddens))
        X = torch.arange(max_len, dtype=torch.float32).reshape(
            -1, 1) / torch.pow(10000,
                               torch.arange(0, num_hiddens, 2, dtype=torch.float32) /
                               num_hiddens)
        self.P[:,:,0::2] = torch.sin(X)
        self.P[:,:,1::2] = torch.cos(X)

    def forward(self, X):
        X = X + self.P[:, :X.shape[1], :].to(X.device)
        return self.dropout(X)

encoding_dim, num_steps = 32, 60
pos_encoding = PositionalEncoding(encoding_dim, 0)
pos_encoding.eval()
X = pos_encoding(torch.zeros((1, num_steps, encoding_dim)))
P = pos_encoding.P[:, :X.shape[1], :]
d2l.plot(torch.arange(num_steps), P[0, :, 6:10].T, xlabel='Row (position)',
         figsize=(6, 2.5), legend=["Col %d" % d for d in torch.arange(6, 10)])
d2l.plt.show()

for i in range(8):
    print(f'{i} in binary is {i:>03b}')

P = P[0, :, :].unsqueeze(0).unsqueeze(0)
d2l.show_heatmaps(P, xlabel='Column (encoding dimension)',
                  ylabel='Row (position)', figsize=(3.5, 4), cmap='Blues')
d2l.plt.show()
```