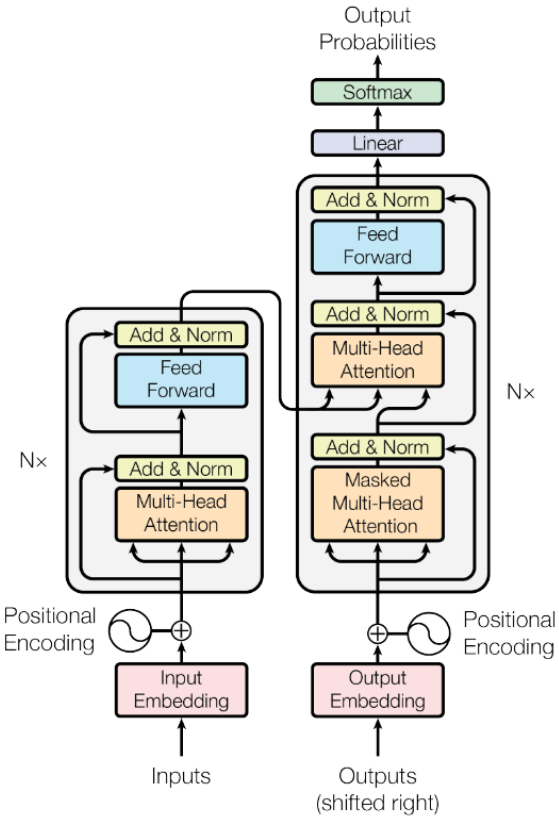
语言分析与机器翻译实践报告

题目 基于Transformer模型的机器翻译

实现Transformer模型，使用预处理好的IWSLT'14 De-En数据集进行训练，输入德文输出对应的英文。

Transformer模型搭建

transformer模型结构如图所示



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)
```

Transformer模型使用注意力机制，同时彻底抛弃了循环神经网络模型。循环神经网络是一种顺序结构，包含句子间的位置信息，使用位置编码，将位置向量加到Embedding向量中，可以使Transformer也包含句子的位置信息。上述的代码使用的是Transformer论文中提到的基于正弦函数和余弦函数的固定位置编码。

Dot-Product Attention

```
class DotProductAttention(nn.Module):
    def __init__(self, dropout):
        super(DotProductAttention, self).__init__()
        self.dropout = nn.Dropout(dropout)

    def forward(self, queries, keys, values, valid_lens=None):
        d = queries.shape[-1]
        scores = torch.bmm(queries, keys.transpose(1, 2)) / math.sqrt(d)
        self.attention_weights = masked_softmax(scores, valid_lens)
        return torch.bmm(self.dropout(self.attention_weights), values)
```

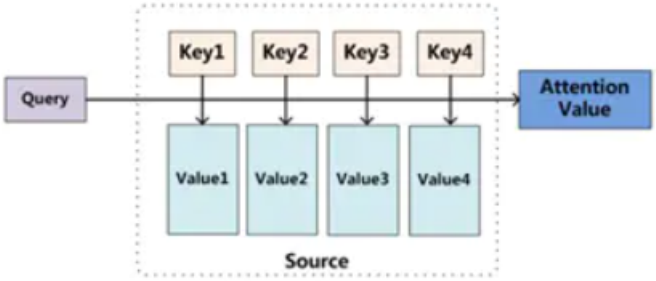
在 Attention is all you need 中，有提到两种较为常见的注意力机制：additive attention 和 dot-product attention。当 query 和 key 向量维度d_K较小时，这两种注意力机制效果相当，但当 d_K较大时 additive attention 要优于 dot-product attention。但是 dot-product attention 在计算方面更具有优势。为了利用 dot-product attention 的优势且消除当 d_K较大时 dot-product attention 的不足，可以采用 scaled dot-product attention。

MultiHead Attention

```
class MultiHeadAttention(nn.Module):
    def __init__(self, key_size, query_size, value_size, num_hiddens, num_heads,
dropout, bias=False):
        super(MultiHeadAttention, self).__init__()
        self.num_heads = num_heads
        self.attention = DotProductAttention(dropout)
        self.W_q = nn.Linear(query_size, num_hiddens, bias=bias)
        self.W_k = nn.Linear(key_size, num_hiddens, bias=bias)
        self.W_v = nn.Linear(value_size, num_hiddens, bias=bias)
        self.W_o = nn.Linear(num_hiddens, num_hiddens, bias=bias)

    def forward(self, queries, keys, values, valid_lens):
        queries = transpose_qkv(self.W_q(queries), self.num_heads)
        keys = transpose_qkv(self.W_k(keys), self.num_heads)
        values = transpose_qkv(self.W_v(values), self.num_heads)
        if valid_lens is not None:
            valid_lens = torch.repeat_interleave(valid_lens,
                                                 repeats=self.num_heads,
                                                 dim=0)
        output = self.attention(queries, keys, values, valid_lens)
        output_concat = transpose_output(output, self.num_heads)
        return self.W_o(output_concat)
```

注意力机制将Source中的构成元素想象成是由一系列的<Key,Value>数据对构成，此时给定Target中的某个元素Query，通过计算Query和各个Key的相似性或者相关性，得到每个Key对应Value的权重系数，然后对Value进行加权求和，即得到了最终的Attention数值。所以本质上Attention机制是对Source中元素的Value值进行加权求和，而Query和Key用来计算对应Value的权重系数。注意力机制就是对输入权重分配的关注，通过对编码器所有时间步的隐藏状态做加权平均来得到下一层的输入变量。



$$\mathbf{Attention(Query, Source)} = \sum_{i=1}^{L_x} \boldsymbol{Similarity(Query, Key_i)} * \boldsymbol{Value_i}$$

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

我们可以用独立学习得到的多组不同的线性投影来变换查询、键和值。然后这些变换后的查询、键和值将并行地送到注意力汇聚中。最后，将这些注意力汇聚的输出拼接在一起，并且通过另一个可以学习的线性投影进行变换以产生最终输出。这种设计被称为 多头注意力，其中 这些注意力汇聚输出中的每一个输出都被称作一个头。

Position-Wise FFN

```
class FFN(nn.Module):
    def __init__(self, ffn_num_input, ffn_num_hiddens, ffn_num_outputs):
        super(FFN, self).__init__()
        self.ffn = nn.Sequential(
            nn.Linear(ffn_num_input, ffn_num_hiddens),
            nn.ReLU(),
            nn.Linear(ffn_num_hiddens, ffn_num_outputs)
        )
    def forward(self, X):
        return self.ffn(X)
```

每一层经过attention之后，还会有一个FFN，这个FFN的作用就是空间变换。FFN包含了2层 linear transformation层，中间的激活函数是ReLu 。它接受一个形状为（batch_size，seq_length，feature_size）的三维张量。因为序列的每个位置的状态都会被单独地更新，所以我们称他为position-wise，这等效于一个1x1的卷积。与多头注意力层相似，FFN层同样只会对最后一维的大小进行改变。

Add & Norm

```
class AddNorm(nn.Module):
    def __init__(self, normalized_shape, dropout):
        super(AddNorm, self).__init__()
        self.dropout = nn.Dropout(dropout)
        self.ln = nn.LayerNorm(normalized_shape)

    def forward(self, X, Y):
        res = self.ln(self.dropout(Y) + X)
        return res
```

在主框架中每一个模块后面，都有一个残差连接和规范化层，它的本质是可以有效的改善深层模型中梯度消失的问题，且能打破网络对称性，改善网络退化问题，加速收敛，规范化优化空间。Transformer模型中使用的是LayerNorm而非BatchNorm，Batchnorm 的特点是强行拉平数据之间的分布，使得模型收敛速度更快，并且起到了正则化的作用，使模型效果更佳。但是，BatchNorm 对 Batch Size 大小

很敏感，并且在 LSTM 网络上效果极差。LayerNorm 是横向归一化，不受 Batch Size 大小的影响，并且可以很好地应用在时序数据中，而且不需要额外的储存空间。

Encoder

```
class Encoder(nn.Module):
    def __init__(self, vocab_size, key_size, query_size, value_size,
                 num_hiddens, norm_shape, ffn_num_input, ffn_num_hiddens,
                 num_heads, num_layers, dropout, use_bias=False):
        super(Encoder, self).__init__()
        self.num_hiddens = num_hiddens
        self.embedding = nn.Embedding(vocab_size, num_hiddens)
        self.pos_encoding = PositionalEncoding(num_hiddens, dropout)
        self.blks = nn.Sequential()
        for i in range(num_layers):
            self.blks.add_module("block"+str(i),
                EncoderBlock(key_size, query_size, value_size, num_hiddens,
                             norm_shape, ffn_num_input, ffn_num_hiddens,
                             num_heads, dropout, use_bias))

    def forward(self, X, valid_lens):
        X = self.pos_encoding(self.embedding(X) * math.sqrt(self.num_hiddens))
        self.attention_weights = [None] * len(self.blks)
        for i, blk in enumerate(self.blks):
            X = blk(X, valid_lens)
            self.attention_weights[i] =
blk.attention.attention.attention_weights
        return X
```

Transformer 由上面实现的各个模块组成， 实现完 上述的模块后，就可以开始搭建了 Encoder 了。编码器包含一个多头注意力层，一个 position-wise FFN，和两个 Add & Norm层。对于 Attention 模型以及 FFN 模型，我们的输出维度都是与 Embedding 维度一致的，这也是由于残差连接天生的特性导致的，因为我们要将前一层的输出与原始输入相加并归一化。

Decoder

```
class Decoder(nn.Module):
    def __init__(self, vocab_size, key_size, query_size, value_size,
                 num_hiddens, norm_shape, ffn_num_input, ffn_num_hiddens,
                 num_heads, num_layers, dropout):
        super(Decoder, self).__init__()
        self.num_hiddens = num_hiddens
        self.num_layers = num_layers
        self.embedding = nn.Embedding(vocab_size, num_hiddens)
        self.pos_encoding = PositionalEncoding(num_hiddens, dropout)
        self.blks = nn.Sequential()
        for i in range(num_layers):
            self.blks.add_module("block"+str(i),
                DecoderBlock(key_size, query_size, value_size, num_hiddens,
                             norm_shape, ffn_num_input, ffn_num_hiddens,
                             num_heads, dropout, i))
        self.dense = nn.Linear(num_hiddens, vocab_size)

    def init_state(self, enc_outputs, enc_valid_lens):
        return [enc_outputs, enc_valid_lens, [None] * self.num_layers]

    def forward(self, X, state):
```

```
        X = self.pos_encoding(self.embedding(X) * math.sqrt(self.num_hiddens))
        self._attention_weights = [[None] * len(self.blks) for _ in range (2)]
        for i, blk in enumerate(self.blks):
            X, state = blk(X, state)
            self._attention_weights[0][i] =
blk.attention1.attention.attention_weights
            self._attention_weights[1][i] =
blk.attention2.attention.attention_weights
        return self.dense(X), state

    @property
    def attention_weights(self):
        return self._attention_weights
```

Transformer 模型的解码器与编码器结构类似，除了之前介绍的几个模块之外，编码器部分有另一个子模块。该模块也是多头注意力层，接受编码器的输出作为 key 和 value，decoder 的状态作为 query。与编码器部分相类似，解码器同样是使用了 Add&norm 模块，用残差和层归一化将各个子层的输出相连。Decoder的输入分为两类：一种是训练时的输入，一种是预测时的输入。训练时的输入就是已经对准备好对应的target数据。预测时的输入，一开始输入的是起始符，然后每次输入是上一时刻 Transformer 的输出。

模型训练

模型参数

```
num_hiddens = 32
num_layers = 2
dropout = 0.1
batch_size = 1024
num_steps = 32
lr = 0.001
num_epochs = 10
ffn_num_input = 32
ffn_num_hiddens = 64
num_heads = 4
key_size, query_size, value_size = 32, 32, 32
```

数据处理

将句子进行onehot编码，设置句子长度为32，多余32的部分截断，少于32的部分使用进行补充。

```
text_en = None
with open('./data/train.en', 'r', encoding='utf-8') as f:
    text_en = f.read()
text_en_bpe = text_en.replace('@@', '')
text_en_list = text_en_bpe.split('\n')

text_de = None
with open('./data/train.de', 'r', encoding='utf-8') as f:
    text_de = f.read()
text_de_bpe = text_de.replace('@@', '')
text_de_list = text_de_bpe.split('\n')
text_de_list = [item.split(' ') for item in text_de_list]
text_en_list = [item.split(' ') for item in text_en_list]
index2word = ['<unk>', '<pad>', '<bos>', '<eos>']
word2index = dict()
```

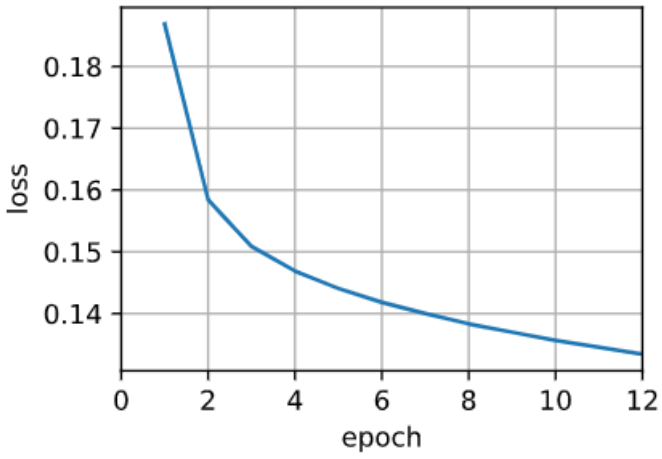
```
text_bpe = None
with open('./data/bpevocab', 'r', encoding='utf-8') as f:
    text_bpe = f.read()
text_bpe_list = [item.split(' ')[0] for item in text_bpe.split('\n')]
index2word = index2word + text_bpe_list
for i in range(len(index2word)):
    word2index[index2word[i]] = i

def vocab(Sequence, Dict = word2index):
    if isinstance(Sequence, str):
        if Sequence not in Dict : return Dict['<unk>']
        else : return Dict[Sequence]
    res = []
    for item in Sequence:
        if item in Dict : res.append(Dict[item])
        else : res.append(Dict['<unk>'])
    return res
src_array, src_valid_len = load_data(text_en_list, vocab, num_steps)
tgt_array, tgt_valid_len = load_data(text_de_list, vocab, num_steps)

data_arrays = (src_array, src_valid_len, tgt_array, tgt_valid_len)
data_iter = load_array(data_arrays, batch_size)
```

训练

```
epoch 1, loss 0.1868776517269771
epoch 2, loss 0.15839085028531452
epoch 3, loss 0.15087227143407095
epoch 4, loss 0.1468883438904386
epoch 5, loss 0.14406756950714555
epoch 6, loss 0.1418362221532978
epoch 7, loss 0.1399928727634328
epoch 8, loss 0.1383425694236382
epoch 9, loss 0.13698636129088268
epoch 10, loss 0.13567367567185132
epoch 11, loss 0.1345289159636636
epoch 12, loss 0.13341357623998776
```



心得体会

这门课应该是我第一次接触自然语言处理，从这里学习了很多关于nlp的知识，尤其是在实验课动手调试学习的代码，遇到不懂的能有学长讲解。感谢肖桐老师以及各位学长！

在这次实践中，我选择了德英翻译这个题目，但现在看来自己是有些眼高手低了。看了<< Attention is all you need >>这篇论文后，虽然更了解了注意力机制以及Transformer，但对于实现模型仍没有任何头绪。后面我又反复阅读了原始论文，以及小牛的机器翻译，看了李宏毅老师的B站讲解视频，最后参考博客和知乎上的代码以及李沐老师的动手学习深度学习完成了Transformer模型的代码。

虽然我研究方向不是nlp，但这门课带给了我很多的收获。也让我知道了实践的重要性，有时候以为自己已经懂了的东西，实现的时候才能发现原来有这么多细节自己在看论文的时候没有发现。