《神经网络与深度学习》课程实验作业（四）
实验内容：自然语言处理基础

何尉宁 2021213599

实验背景：

搭建 Transformer 编码器完成文本语义匹配任务：

> AFQMC 数据集是一个蚂蚁金融语义相似度数据集，用于问题相似度计算，数据集包括训练集、验证集、测试集 3 个文件，分别包含 34334、4316 以及 3861 条数据，每条数据有三个属性，分别是句子 1、句子 2、句子相似度标签。相似度标签为 1 表示两个句子含义类似，标签为 0 则表示含义不同。

理论补充：

1. Error Propagation

Transformer误差反向传播是指在Transformer模型中，通过反向传播算法将误差从输出层逐层传回至输入层，以更新模型参数的过程。在训练中，我们通过损失函数计算模型预测输出与真实输出的误差，然后利用反向传播算法计算每个参数对误差的贡献，并通过随机梯度下降等优化算法对参数进行调整，以最小化误差。

在Transformer中，误差反向传播分为两个阶段：Encoder部分和Decoder部分。在Encoder中，误差从Decoder输出层传回到Encoder的每一层，通过Layer Norm和Multi-Head Attention等操作进行误差传播和参数更新。在Decoder中，误差从输出层传回到Decoder的每一层，并结合Encoder的输出进行精度计算和参数更新。这一过程使得模型能够学习并适应任务特定的模式和表示。

2. Multi-head Attention

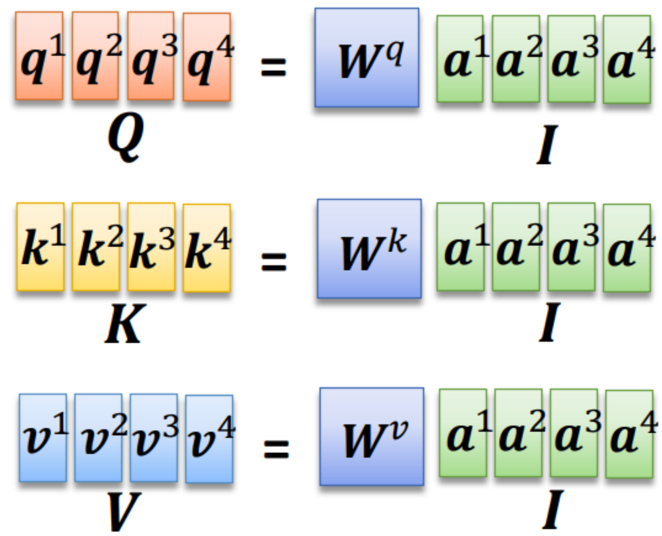
在多头注意力机制实现的时候，我们可以将QKV进行转换，将其合并之后实现并行计算。将输入矩阵进行转置组装，这样就可以获得整块的qkv矩阵，直接进行矩阵乘法。

□并行计算

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

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

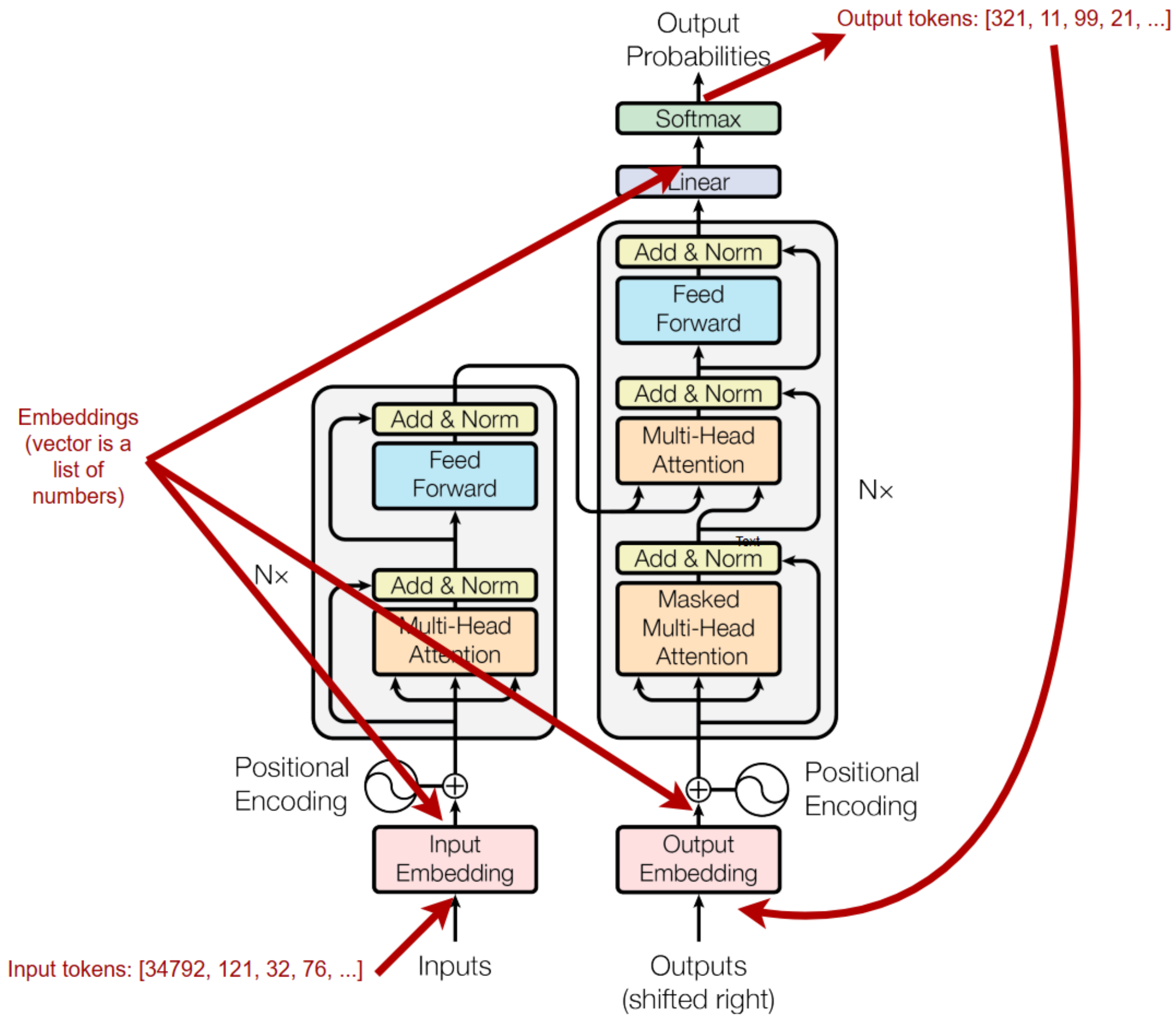
$$v^i = W^v a^i$$




```
class multiheadAttention (nn.Module):
    def __init__(self, query_size, key_size, value_size,
                 num_hiddens, num_heads, dropout, bias=False,
                 *args, **kwargs) -> None:
        super(multiheadAttention, self).__init__(*args, **kwargs)
        self.num_heads = num_heads
        self.attention = d2l.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):
        """Note
        concat all the heads together for matrix multiplication
        """
        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)
```

3. Embedding and Tokenization

关于各大模型的`tokenizer`： [Summary of the tokenizers (huggingface.co)]



4. Unbalanced Training Data

不均衡数据的处理
同时也需要注意模型准确率评判标准
[【金融风控系列】_[2.1]_SPE算法和DE算法的学习与实现 - 飞桨AI Studio星河社区 (baidu.com)]
[Deep learning unbalanced training data?Solve it like this. | by Shubrashankh Chatterjee | Towards Data Science]
[炼丹笔记一：样本不平衡问题 - 知乎 (zhihu.com)](https://zhuanlan.zhihu.com/p/56882616)

---

1. 数据集构建

1.1 观察数据集

数据集直接由`json`格式给出，包含序号，句子一，句子二，标签等信息，我们直接指定`json`目标格式调用函数读取即可。

1.2 DataFrame

根据数据集类型生成不同的df文件。

```
# data processing
def read_data(path, type):
    sentence_1 = []
    sentence_2 = []
    label = []
    with open (path, 'r', encoding = 'utf-8') as f:
        for line in f.readlines():
            line = json.loads(line)
            sentence_1.append(line['sentence1'])
            sentence_2.append(line['sentence2'])
            if type != 'test':
                label.append(line['label'])
        if type != 'test':
            df = pd.DataFrame({'sentence1': sentence_1, 'sentence2': sentence_2, 'label':
label})
        else:
            df = pd.DataFrame({'sentence1': sentence_1, 'sentence2': sentence_2})
    return df
```

同时我们打印一下数据集的特征，这里展示训练集的特征，从下面的输出中我们可以大体把握一下数据的结构，根据max可以确定句子的最大长度从而设定`MAX_LEN`参数，这比起直接设定一个较大值也可以一定程度上节约不少空间。

```
count 34334.000000
mean 26.732597
std 10.405410
min 10.000000
25% 20.000000
50% 25.000000
75% 30.000000
max 157.000000
dtype: float64
```

1.3 DataLoader

1. 将数据集tokenize，首先我们要将`vocab.txt`制作成一个`list`供我们对数据集中的字符进行对照查找，这里我直接这样构造了一个映射`char_to_id = {char: idx for idx, char in enumerate(vocab)}`
2. 从数据集中抽出两个句子，填入[CLS][SEP]两个符号用于标记句子开始与分隔符
3. 对于词表中不存在的字符，我们将它标记为[UNK]
4. 填充操作，对数据集直接处理之后，由于我们要把它们都丢到一个batch里操作，所以还得将它们填充至一个`MAX_LEN`，便于后续的`tensor`批量计算，这里重新写了一个`collate_fn`进行补零操作。

```
class TextDataset(Dataset):
    def __init__(self, dataframe, char_to_id, max_length=MAX_LEN):
        self.data = dataframe
        self.char_to_id = char_to_id
        self.max_length = max_length

    def __len__(self):
        return len(self.data)

    def __getitem__(self, index):
        sentence1 = self.data.iloc[index]['sentence1']
        sentence2 = self.data.iloc[index]['sentence2']
        if 'label' in self.data.columns:
            label = self.data.iloc[index]['label']
        label = int(label)

        combined_tokens = ['[CLS]'] + [char for char in sentence1] + ['[SEP]'] + [char
for char in sentence2] + ['[SEP]']
        segment_ids = [0] * (len(sentence1) + 2) + [1] * (len(sentence2) + 1)

        combined_ids = [self.char_to_id.get(char, self.char_to_id['[UNK]']) for char in
combined_tokens]
        combined_ids = torch.nn.functional.pad(torch.tensor(combined_ids), (0,
self.max_length - len(combined_ids)))
        segment_ids = torch.nn.functional.pad(torch.tensor(segment_ids), (0,
self.max_length - len(segment_ids)))
        # label = torch.tensor(label)

        return combined_ids, segment_ids, label

train_dataset = TextDataset(train_df, char_to_id)
dev_dataset = TextDataset(dev_df, char_to_id)
test_dataset = TextDataset(test_df, char_to_id)

def collate_fn(batch_data, pad_val=0, max_seq_len=MAX_LEN):
    input_ids, segment_ids, labels = [], [], []
    max_len = 0
    for example in batch_data:
        input_id, segment_id, label = example
        # cut
        input_ids.append(input_id[:max_seq_len])
        segment_ids.append(segment_id[:max_seq_len])
        labels.append(label)
        # max
        max_len = max(max_len, len(input_id))
    # pad
    for i in range(len(labels)):
        input_ids[i] = torch.cat([input_ids[i], torch.tensor([pad_val] * (max_len -
len(input_ids[i])))])
```

```
        segment_ids[i] = torch.cat([segment_ids[i], torch.tensor([pad_val] * (max_len -
len(segment_ids[i])))])

    return torch.stack(input_ids), torch.stack(segment_ids), torch.tensor(labels)

train_dataloader = DataLoader(train_dataset, batch_size=BATCH_SIZE, shuffle=True,
collate_fn=collate_fn)
dev_dataloader = DataLoader(dev_dataset, batch_size=BATCH_SIZE, shuffle=True,
collate_fn=collate_fn)
test_dataloader = DataLoader(test_dataset, batch_size=BATCH_SIZE, shuffle=True,
collate_fn=collate_fn)
```

1.4 mini-batch

打印一条迷你数据，看上去没有问题

```
mini-batch sample: (
tensor([[1.0000e+00, 4.6200e+02, 2.6430e+03, ..., 0.0000e+00, 0.0000e+00, 0.0000e+00],
[1.0000e+00, 7.0000e+00, 2.7000e+01, ..., 0.0000e+00, 0.0000e+00, 0.0000e+00],
[1.0000e+00, 1.4200e+02, 2.2800e+02, ..., 0.0000e+00, 0.0000e+00, 0.0000e+00], ...,
[1.0000e+00, 1.0510e+03, 4.9470e+03, ..., 0.0000e+00, 0.0000e+00, 0.0000e+00],
[1.0000e+00, 8.8000e+01, 1.2400e+02, ..., 0.0000e+00, 0.0000e+00, 0.0000e+00],
[1.0000e+00, 1.3000e+01, 6.1400e+02, ..., 0.0000e+00, 0.0000e+00, 0.0000e+00]]),
tensor([[0., 0., 0., ..., 0., 0., 0.], [0., 0., 0., ..., 0., 0., 0.], [0., 0., 0., ...,
0., 0., 0.], ..., [0., 0., 0., ..., 0., 0., 0.], [0., 0., 0., ..., 0., 0., 0.], [0., 0.,
0., ..., 0., 0., 0.]]),
tensor([1, 1, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 1, 1, 1, 0, 1, 0, 0, 1, 0, 1, 0, 0, 1, 0, 0,
1, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 1, 1,
1, 0, 0, 0, 1, 0, 0]))
```

---

2. 编码与嵌入

分词和词嵌入是NLP任务中非常重要的一环，手写一遍对我的理解有很大帮助，可以说是把NLP的数据清洗给简单摸清楚了。
目前的三大subword模型分别是**BPE**、**WordPiece**、**ULM**，GPT用的是第一个，如果只会`tokenizer.encode`的话多半在处理完句子后会收获一条满是<UNK>的句子哈哈哈。

2.1 词嵌入

注意对`word_emb`进行归一化操作，这一步其实也可以在后面`encoder`的前向传播的地方手动操作，主要是为了防止embedding之后数值过小过于集中。

```
class WordEmbedding(nn.Module):
    def __init__(self, vocab_size, emb_size, padding_idx=0):
```

```python
        super(WordEmbedding, self).__init__()
        self.emb_size = emb_size
        self.word_embedding = nn.Embedding(vocab_size, emb_size, padding_idx=padding_idx)
        nn.init.normal_(self.word_embedding.weight, mean=0.0, std=emb_size ** -0.5)

    def forward(self, word):
        word_emb = (self.emb_size ** 0.5) * self.word_embedding(word)
        return word_emb

# sample
vocab_size = 10000
emb_size = 300
padding_idx = 0

word_embedding = WordEmbedding(vocab_size, emb_size, padding_idx)
input_word = torch.tensor([1, 2, 3, 4])
output_embedding = word_embedding(input_word)
print(output_embedding)
```

测试输出：

```
tensor([[ 0.2551, -0.1008, 1.0827, ..., 0.2031, -1.5132, -0.0033], [ 1.4436, -0.7597,
-0.2470, ..., -0.1384, 1.2034, -1.3039], [-1.7262, -1.7636, -0.2379, ..., 0.0985, 1.7969,
0.3163], [ 0.1847, 0.0230, 0.2470, ..., -0.2437, 1.1850, 1.2422]], grad_fn=
<MulBackward0>)
```

2.2 段落编码

把每个句子的前后句分别标为0和1，在词元化时已经做了处理。

2.3 位置编码

关于`sin`位置编码：[10.6. 自注意力和位置编码 — 动手学深度学习 2.0.0 documentation (d2l.ai)]
其实是一种反应相对位置的投影方式。
公式：

$$\begin{aligned} p_{i,2j} &= \sin\left(\frac{i}{10000^{2j/d}}\right), \\ p_{i,2j+1} &= \cos\left(\frac{i}{10000^{2j/d}}\right). \end{aligned}$$

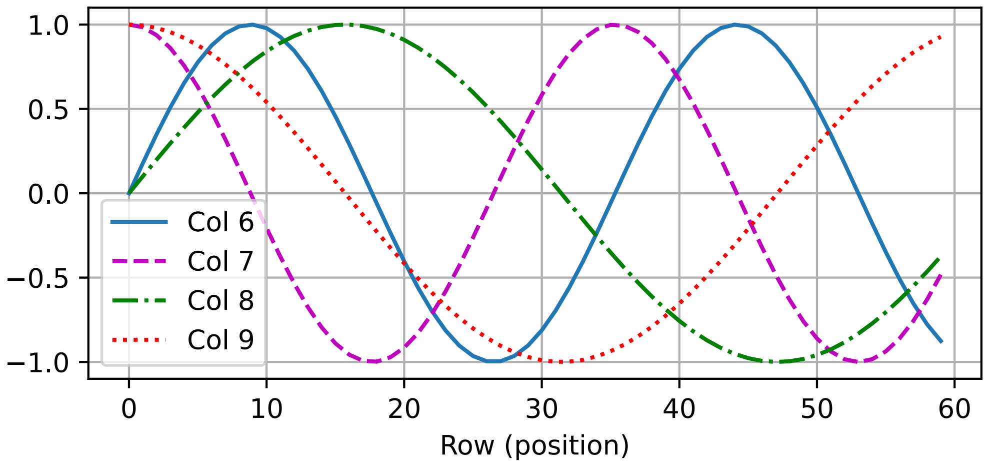
$$
\begin{aligned}
&\begin{bmatrix} \cos(\delta\omega_j) & \sin(\delta\omega_j) \\ -\sin(\delta\omega_j) & \cos(\delta\omega_j) \end{bmatrix} \begin{bmatrix} p_{i,2j} \\ p_{i,2j+1} \end{bmatrix} \\
=& \begin{bmatrix} \cos(\delta\omega_j)\sin(i\omega_j) + \sin(\delta\omega_j)\cos(i\omega_j) \\ -\sin(\delta\omega_j)\sin(i\omega_j) + \cos(\delta\omega_j)\cos(i\omega_j) \end{bmatrix} \\
=& \begin{bmatrix} \sin\left((i+\delta)\omega_j\right) \\ \cos\left((i+\delta)\omega_j\right) \end{bmatrix} \\
=& \begin{bmatrix} p_{i+\delta,2j} \\ p_{i+\delta,2j+1} \end{bmatrix},
\end{aligned}
$$

```
def get_sinusoid_encoding(position_size, hidden_size):
    def cal_angle(pos, hidden_idx):
        return pos / np.power(10000, 2 * (hidden_idx // 2) / hidden_size)
    def get_posi_angle_vec(pos):
        return [cal_angle(pos, hidden_j) for hidden_j in range(hidden_size)]

    sinusoid = np.array([get_posi_angle_vec(pos_i) for pos_i in range(position_size)])
    sinusoid[:, 0::2] = np.sin(sinusoid[:, 0::2])
    sinusoid[:, 1::2] = np.cos(sinusoid[:, 1::2])
    return torch.tensor(sinusoid, dtype=torch.float32)

class PositionalEmbedding(nn.Module):
    def __init__(self, max_length, emb_size):
        super(PositionalEmbedding, self).__init__()
        self.emb_size = emb_size
        self.max_length = max_length
        self.register_buffer('pos_encoder', get_sinusoid_encoding(max_length,
self.emb_size))

    def forward(self, pos):
        # Ensure that pos is within valid range
        pos = torch.clamp(pos, 0, self.max_length - 1)
        pos_emb = self.pos_encoder[pos]
        pos_emb = pos_emb.detach()  # Detach so gradients are not computed
        return pos_emb

# ex
out = torch.randint(low=0, high=5, size=[3])
print('输入向量为: {}'.format(out.numpy()))

pos_embed = PositionalEmbedding(4, 5)
pos_out = pos_embed(out)
print('位置编码的输出为:  {}'.format(pos_out.numpy()))
```

可视化输出:

```
encoding_dim, num_steps = 32, 60
pos_encoding = PositionalEmbedding(num_steps, encoding_dim)
pos_encoding.eval()
```

```python
sample = torch.randint(0, 100, (1, num_steps))
X = pos_encoding(sample)
P = pos_encoding.pos_encoder[:, :X.shape[1]]
d2l.plot(torch.arange(num_steps), P[:, 6:10].T, xlabel='Row (position)',
         figsize=(6, 2.5), legend=["Col %d" % d for d in torch.arange(6, 10)])
```



2.4 整合词嵌入层

```python
class TransformerEmbeddings(nn.Module):
    def __init__(self, vocab_size, hidden_size, hidden_dropout_prob, position_size,
segment_size):
        super(TransformerEmbeddings, self).__init__()
        self.word_embeddings = WordEmbedding(vocab_size, hidden_size)
        self.position_embeddings = PositionalEmbedding(position_size, hidden_size)
        self.segment_embeddings = SegmentEmbedding(segment_size, hidden_size)
        self.layer_norm = nn.LayerNorm(hidden_size)
        self.dropout = nn.Dropout(hidden_dropout_prob)

    def forward(self, input_ids, segment_ids=None, position_ids=None):
        if position_ids is None:
            ones = torch.ones_like(input_ids, dtype=torch.long)
            seq_length = torch.cumsum(ones, dim=-1)
            position_ids = seq_length - ones
            position_ids = position_ids.clamp(0, self.position_embeddings.max_length - 1)

        input_embeddings = self.word_embeddings(input_ids)
        segment_embeddings = self.segment_embeddings(segment_ids)
        position_embeddings = self.position_embeddings(position_ids)

        embeddings = input_embeddings + segment_embeddings + position_embeddings
        embeddings = self.layer_norm(embeddings)
        embeddings = self.dropout(embeddings)
```

```
        embeddings = embeddings.to(torch.float32)
        return embeddings
```

```
# sample
vocab_size = 10000
hidden_size = 256
position_size = 512
segment_size = 2
dropout = 0.1

transformer_embeddings = TransformerEmbeddings(vocab_size, hidden_size, dropout,
position_size, segment_size)
input_ids = torch.randint(0, vocab_size, (1, 10))
segment_ids = torch.randint(0, segment_size, (1, 10))
position_ids = None

# forward
output_embeddings = transformer_embeddings(input_ids, segment_ids, position_ids)
print("Output Embeddings Shape:", output_embeddings.shape)
```

输出：

```
Output Embeddings Shape: torch.Size([1, 10, 256])
```

Transformer每一步都不改变数据形状，只是对权重进行重新分配。

---

3. 多头注意力层与AddNorm

3.1 前馈Feed Forward

其实就是个两层的MLP，把输入形状为(b, n, d)的数据变成(bn, d)

```
# Actually, it's a two-layer MLP
class PositionWiseFFN(nn.Module):
    def __init__(self, ffn_num_input, ffn_num_hiddens, pw_num_outputs, **kwargs) -> None:
        super(PositionWiseFFN, self).__init__(**kwargs)
        self.dense1 = nn.Linear(ffn_num_input, ffn_num_hiddens)
        self.relu = nn.ReLU()
        self.dense2 = nn.Linear(ffn_num_hiddens, pw_num_outputs)

    def forward(self, X):
        return self.dense2(self.relu(self.dense1(X)))
```

3.2 AddNorm

第一版代码写的是`pre-norm`，LN在残差连接之前使用的。

```
class AddNorm(nn.Module):
    """The residual connection followed by layer normalization."""
    def __init__(self, norm_shape, dropout):
        super().__init__()
        self.dropout = nn.Dropout(dropout)
        self.ln = nn.LayerNorm(norm_shape)

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

如果要改成`post-norm`的话，前向传播这里合并就可以了。
`return self.dropout(self.ln(Y + X))`

3.3 多头注意力

因为写多头的时候会涉及到几个头的Matrix乘法运算，然后每一轮乘法运算其实只有在权重上有不同，就是这个`attention`给出的权重不一样，所以完全可以写成一个并行的计算，把他们丢到一整个矩阵里去。

```
class multiheadAttention (nn.Module):
    def __init__(self, query_size, key_size, value_size,
                 num_hiddens, num_heads, dropout, bias=False,
                 *args, **kwargs) -> None:
        super(multiheadAttention, self).__init__(*args, **kwargs)
        self.num_heads = num_heads
        self.attention = d2l.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):
        """Note
        concat all the heads together for matrix multiplication
        """
        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)
```

transpose_qkv & transpose_output:

```
def transpose_qkv(X, num_heads):
    """Note
    For parrallel computation, we can concat the heads together
    INPUT: X.shape = (batch_size, num_steps, num_hiddens)
    OUTPUT: X.shape = (batch_size * num_heads, num_steps, num_hiddens/num_heads)
    combine the num_heads and num_hiddens/num_heads together
    """
    X = X.reshape(X.shape[0], X.shape[1], num_heads, -1)
    X = X.permute(0, 2, 1, 3)
    print(X.shape[0], X.shape[1], X.shape[2], X.shape[3])
    return X.reshape(-1, X.shape[2], X.shape[3])

def transpose_output(X, num_heads):
    """Note
    Inverse of transpose_qkv
    INPUT: X.shape = (batch_size * num_heads, num_steps, num_hiddens/num_heads)
    OUTPUT: X.shape = (batch_size, num_steps, num_hiddens)
    """
    X = X.reshape(-1, num_heads, X.shape[1], X.shape[2])
    X = X.permute(0, 2, 1, 3)
    return X.reshape(X.shape[0], X.shape[1], -1)
```

验证环节：

```
# Test
num_hiddens, num_heads = 100, 5
attention = multiheadAttention(num_hiddens, num_hiddens, num_hiddens, num_hiddens,
num_heads, 0.5)
attention.eval()
print(attention)

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

输出：

```
multiheadAttention( (attention): DotProductAttention( (dropout): Dropout(p=0.5,
inplace=False) ) (W_q): Linear(in_features=100, out_features=100, bias=False) (W_k):
```

```
Linear(in_features=100, out_features=100, bias=False) (W_v): Linear(in_features=100,
out_features=100, bias=False) (W_o): Linear(in_features=100, out_features=100,
bias=False) )
2 5 4 20
2 5 4 20
2 5 4 20
torch.Size([2, 4, 100])
```

---

4. 搭建两层Transformer

每一层都可以看成是一个完整的`encoder_block`，所以对每一个块进行封装，结构为(self-attention)→(add&norm)→(ffn)→(add&norm)，最后再在完整的`TransformerEncoder`中装载每一个`encoder_block`。

4.1 封装每一个Encoder_Block

```
class EncoderBlock(nn.Module):
    def __init__(self, query_size, key_size, value_size, num_hiddens, norm_shape,
ffn_num_input, ffn_num_hiddens, num_heads, dropout, bias=False, **kwargs) -> None:
        super(EncoderBlock, self).__init__(**kwargs)
        self.attention = multiheadAttention(query_size, key_size, value_size,
num_hiddens, num_heads, dropout, bias)
        self.addnorm1 = AddNorm(norm_shape, dropout)
        self.ffn = PositionWiseFFN(ffn_num_input, ffn_num_hiddens, num_hiddens)
        self.addnorm2 = AddNorm(norm_shape, dropout)

    def forward(self, X, valid_lens):
        attention_output, attention_weights = self.attention(X, X, X, valid_lens)
        Y = self.addnorm1(X, attention_output)  # self-attention
        return self.addnorm2(Y, self.ffn(Y))
```

4.2 TransformerEncoder

```
"""
- Self-Attention
- Add & Norm
- Feed Forward
- Add & Norm
- Encoder Block
"""
class TransformerEncoder(d2l.Encoder):
    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, **kwargs):
```

```
        super(TransformerEncoder, self).__init__(**kwargs)
        self.num_hiddens = num_hiddens
        self.embedding = TransformerEmbeddings(vocab_size, num_hiddens, dropout,
position_size=MAX_LEN, segment_size=2)
        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, *args):
        self.attention_weights = [None] * len(self.blks)
        for i, blk in enumerate(self.blks):
            # print(X.shape)
            X = blk(X, valid_lens)
            self.attention_weights[i] = blk.attention.attention.attention_weights
            print(f"Intermediate Output Shape after Block {i}: {X.shape}")
            print(self.attention_weights[i].shape)
        return X
```

4.3 验证

```
# Two layer Transformer Encoder
batch_size = 2
max_seq_length = 50
transformer_embeddings = TransformerEmbeddings(vocab_size=200,
                                               hidden_size=24,
                                               hidden_dropout_prob=0.1,
                                               position_size=100,
                                               segment_size=2)

input_ids = torch.randint(0, 200, (batch_size, max_seq_length))
segment_ids = torch.randint(0, 2, (batch_size, max_seq_length))
position_ids = None

embeddings_output = transformer_embeddings(input_ids, segment_ids, position_ids)
valid_lens = torch.randint(1, max_seq_length + 1, (batch_size,))

transformer_encoder = TransformerEncoder(vocab_size=200,
                                         key_size=24,
                                         query_size=24,
                                         value_size=24,
                                         num_hiddens=24,
                                         norm_shape=[2, 50, 24],
                                         ffn_num_input=24,
                                         ffn_num_hiddens=48,
                                         num_heads=8,
```

```
                                    num_layers=2,
                                    dropout=0.5)

output = transformer_encoder(embeddings_output, valid_lens)
print("Input Embeddings Shape:", embeddings_output.shape)
print("Output Shape:", output.shape)
```

输出：

```
2 8 50 3
2 8 50 3
2 8 50 3
Intermediate Output Shape after Block 0: torch.Size([2, 50, 24])
torch.Size([16, 50, 50])
2 8 50 3
2 8 50 3
2 8 50 3
Intermediate Output Shape after Block 1: torch.Size([2, 50, 24])
torch.Size([16, 50, 50])
Input Embeddings Shape: torch.Size([2, 50, 24])
Output Shape: torch.Size([2, 50, 24])
```

---

5. 训练与绘制

这一部分很遗憾没有完成，虽然前面每一步都对模型进行了验证，输入输出的维度也正确的控制了，但是在训练的时候仍然会出现input和output大小不对的情况，可能是我对每一个batch的整体读入后传的维度参数有问题，但由于之前在其他任务上耽误了一些时间，加上期末复习压力有点大，就没有进一步完善后面的代码了。
但因为在做这一次语义匹配的任务之前，我还做了一个英法翻译的任务练手，所以对于后续的5个任务，虽然不能给出实质性的实验结果，但我可以提供相关的伪代码等。

5.1 训练

```
# Define Transformer Encoder
model = TransformerEncoder(
    vocab_size=train_size,
    key_size=NUM_HIDDENS,
    query_size=NUM_HIDDENS,
    value_size=NUM_HIDDENS,
    num_hiddens=NUM_HIDDENS,
    norm_shape=[2, MAX_LEN, NUM_HIDDENS],
    ffn_num_input=NUM_HIDDENS,
    ffn_num_hiddens=NUM_HIDDENS * 2,
    num_heads=NUM_HEADS,
```

```
    num_layers=NUM_LAYERS,
    dropout=DROPOUT,
    use_bias=False
)

# Loss function and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(
    model.parameters(),
    lr=5E-5,
    weight_decay=0.0
)

# Training loop
num_epochs = NUM_EPOCHS
for epoch in range(num_epochs):
    model.train()
    for batch_data in train_dataloader:
        combined_ids, segment_ids, labels = batch_data
        # 将 combined_ids 转换为 torch.LongTensor
        combined_ids = combined_ids.long()
        embeddings_output = transformer_embeddings(combined_ids, segment_ids)
        valid_lens = None
        outputs = model(embeddings_output, valid_lens)
        optimizer.zero_grad()
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

    # Validation
    model.eval()
    val_loss = 0.0
    correct = 0
    total = 0

    with torch.no_grad():
        for batch_data in dev_dataloader:
            combined_ids, segment_ids, labels = batch_data
            outputs = model(combined_ids, segment_ids)
            batch_loss = criterion(outputs, labels)
            val_loss += batch_loss.item()

            _, predicted = torch.max(outputs, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

    accuracy = correct / total
    average_val_loss = val_loss / len(dev_dataloader)
    print(f'Epoch {epoch+1}/{num_epochs}, Loss: {loss.item()}, Validation Loss:
```

```
{average_val_loss}, Accuracy: {accuracy}')

# Save the trained model
torch.save(model.state_dict(), 'trained_model.pth')
```

5.2 数据均衡化

训练后普遍会出现的现象是准确率会直接固定在69%左右，因为这个数据集的样本中有69%的数据label就是0，所以模型会直接全部判成0，而且这种现象基本上在train了一轮之后就会出现，这里就需要对数据进行均衡化处理。

其实这里还能看出的一个问题就是，验证模型准确率的指标选择也非常重要，咱不能只是看一个acc/train_len，还有非常多评判标准，**混淆矩阵、精度、召回率和 F1**，综合来看才能反应模型效果。

数据的不均衡问题往往会让模型更偏向于多数类的样本，而对少数类样本的识别表现不佳，因此数据的不均衡是模型构建中需要重点解决的问题。

1. 从数据的角度出发，通过采样的方式调整样本类别比例来实现数据的均衡；
2. 从算法的角度考虑，通过集成的思想改进算法或者构建新的分类算法来实现数据的均衡。

比较朴素的思想就是直接把`label = 1`的数据再copy一份丢到dataframe里面去。

---

6. 测试

---

7. 可视化Attention权重

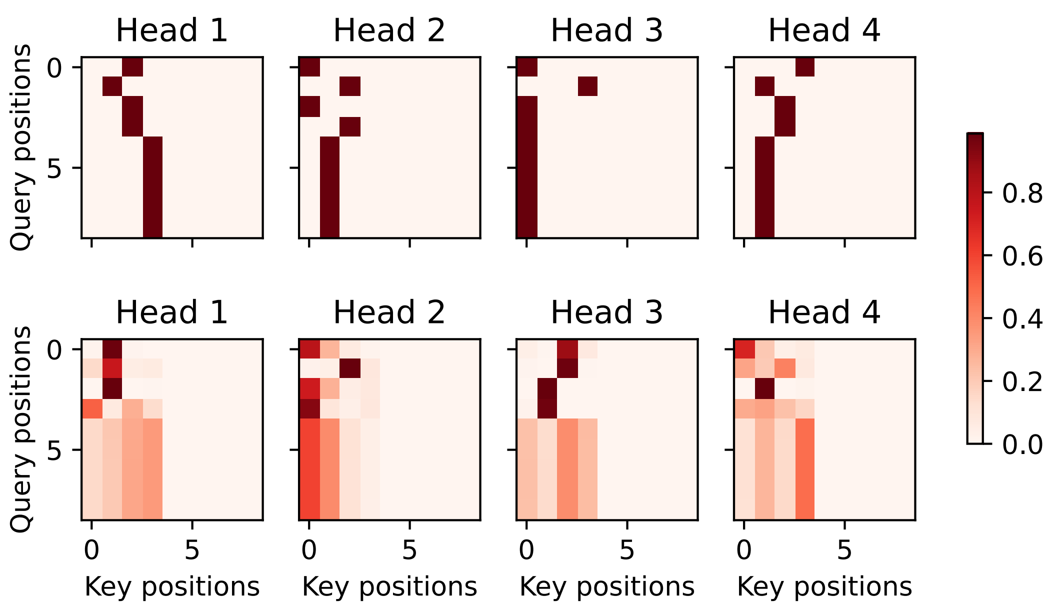
在`TransformerEncoder`中每次前向传播的时候，把每一个block中的注意力权重都记录下来就可以了，对每个头都记录。

```
    def forward(self, X, valid_lens, *args):
        self.attention_weights = [None] * len(self.blks)
        for i, blk in enumerate(self.blks):
            # print(X.shape)
            X = blk(X, valid_lens)
            self.attention_weights[i] = blk.attention.attention.attention_weights
            print(f"Intermediate Output Shape after Block {i}: {X.shape}")
            print(self.attention_weights[i].shape)
        return X
```

最后再把权重从里面掏出来丢到矩阵里，这里的示例是英法翻译code里的，只是作为代码结果展示。

```
_, dec_attention_weights = model.predict_step(
    data.build([engs[-1]], [fras[-1]]), d2l.try_gpu(), data.num_steps, True)
enc_attention_weights = torch.cat(model.encoder.attention_weights, 0)
shape = (num_blks, num_heads, -1, data.num_steps)
enc_attention_weights = enc_attention_weights.reshape(shape)
d2l.check_shape(enc_attention_weights,
                (num_blks, num_heads, data.num_steps, data.num_steps))

d2l.show_heatmaps(
    enc_attention_weights.cpu(), xlabel='Key positions',
    ylabel='Query positions', titles=['Head %d' % i for i in range(1, 5)],
    figsize=(7, 3.5))
```

图大概长这样，这里我只搞了4个头。



---

8. 改变层数

~~不知道水什么子~~
因为Transformer这个模型自己是有残差块和归一化的，所以说堆层数应该是一个收益还不错的提高精度的方法，当然前提是选择的`hidden_nums`和`head_nums`能包含较多信息，不然应该很快模型就会收敛。

BERT好像是24层。

---

9. Tricks

10. Pre-Norm & Post-Norm

```
class AddNorm(nn.Module):
    """The residual connection followed by layer normalization."""
    def __init__(self, norm_shape, dropout):
        super().__init__()
        self.dropout = nn.Dropout(dropout)
        self.ln = nn.LayerNorm(norm_shape)

    def forward(self, X, Y):
    """pre-norm"""
        return self.ln(self.dropout(Y) + X)
    """post-norm"""
        return self.dropout(self.ln(Y + X))
```