实验四：基于 Transformer 的神经机器翻译

一、实验目的

1. 学习如何使用深度学习框架来实现和训练一个 BERT 模型（即双向 Transformer）架构，以及自然语言处理中的 Attention 机制。
2. 掌握使用深度学习框架搭建 Transformer 机器翻译模型。

二、实验要求

1. 利用 Python 语言和深度学习框架（本实验指导书以 PyTorch 为例）构造简单的机器翻译模型，以实现英语和汉语的相互翻译。
2. 评估指标 BLEU4（Bilingual Evaluation Understudy 4）大于 14（参考文献： https://dl.acm.org/doi/10.3115/1073083.1073135）。
3. 按规定时间在课程网站提交实验报告、代码以及 PPT。

三、实验原理

1. 模型结构

BERT 是 Seq2seq 模型，分为 Encoder 和 Decoder 两大部分，如图 1 所示。Encoder 部分是由 6 个相同的 encoder 组成，Decoder 部分也是由 6 个相同的 decoder 组成，与 encoder 不同的是，每一个 decoder 都会接受最后一个 encoder 的输出。

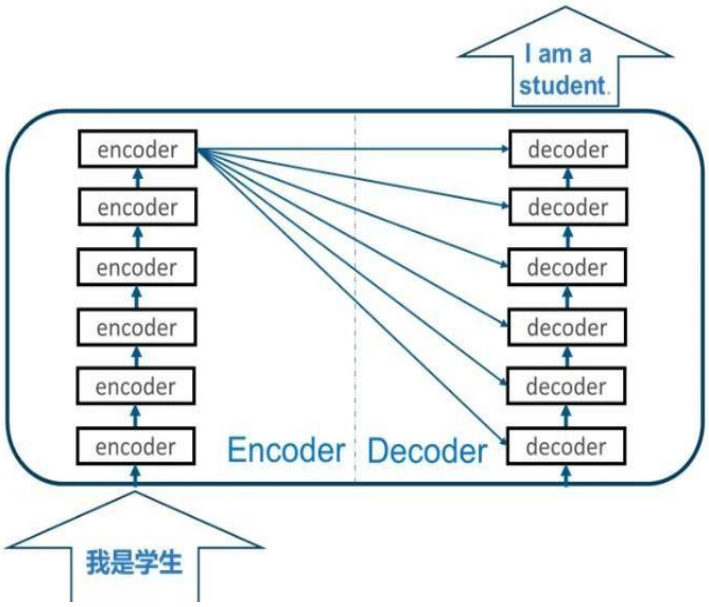


图 1 EERT 模型的结构

2. 模型输入

模型输入表示能够在一个标记序列中清楚地表示单个文本句子或一对文本句子(例如， [Question, Answer])。(注释：在整个工作中，“句子”可以是连续的任意跨度的文本，而不是实际语言意义上的句子。“序列”是指输入到 BERT 的标记序列，它可以是单个句子，也可以是两个句子组合在一起)。通过把给定标记对应的标记嵌入、句子嵌入和位置嵌入求和来构造其输入表示。图 2 给出了输入表示的可视化表示。

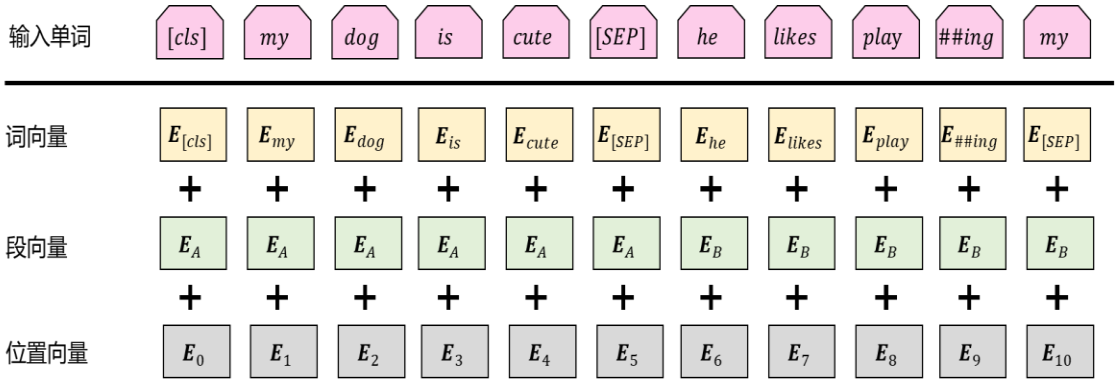


图 2 BERT 的输入表示

模型输入包含以下细节：

1) 使用含 3 万个标记词语的 WordPiece 嵌入（Wu et al., 2016)。我们用 ## 表示拆分的单词片段。
2) 使用学习到的位置嵌入，支持的序列长度最长可达 512 个标记。
3) 每个序列的第一个标记始终是特殊分类嵌入（[CLS])。该特殊标记对应的最终隐藏状态（即 Transformer 的输出）被用作分类任务中该序列的总表示。对于非分类任务，这个最终隐藏状态将被忽略。
4) 句子对被打包在一起形成一个单独的序列。我们用两种方法区分这些句子：方法一，我们用一个特殊标记（[SEP])将它们分开；方法二，我们给第一个句子的每个标记添加一个可训练的句子 A 嵌入，给第二个句子的每个标记添加一个可训练的句子 B 嵌入。
5) 对于单句输入，我们只使用句子 A 的嵌入。

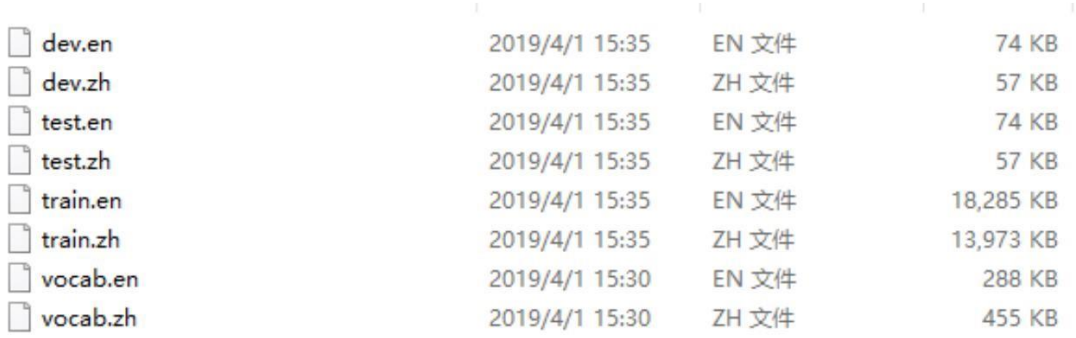
3. 模型输出

为了训练深度双向表示，BERT 采用了一种简单的方法，即随机遮蔽一定比例的输入标记，然后仅预测那些被遮蔽的标记，称为“遮蔽语言模型”（MLM）。在这种情况下，就像在标准语言模型中一样，与遮蔽标记相对应的最终隐藏向量被输入到与词汇表对应的输出 Softmax 中（也就是要把被遮蔽的标记对应为词汇表中的一个词语）。在实验中，BERT 在每个序列中随机遮蔽 15% 的标记。

Transformer 编码器不知道它将被要求预测哪些单词，或者哪些单词已经被随机单词替换，因此它被迫保持每个输入标记的分布的上下文表示。另外，因为随机替换只发生在 1.5% 的标记（即 15% 的 10%），这似乎不会损害模型的语言理解能力。

四、实验所用工具以及数据集

本实验主要针对中英机器翻译，使用的数据库来自 NiuTrans 提供的开源中英平行语料库，包含中、英文各 10 万条，如图 3 所示。数据集下载地址：https://github.com/NiuTrans/NiuTrans.SMT/tree/master/sample-data。



dev.en	2019/4/1 15:35	EN 文件	74 KB
dev.zh	2019/4/1 15:35	ZH 文件	57 KB
test.en	2019/4/1 15:35	EN 文件	74 KB
test.zh	2019/4/1 15:35	ZH 文件	57 KB
train.en	2019/4/1 15:35	EN 文件	18,285 KB
train.zh	2019/4/1 15:35	ZH 文件	13,973 KB
vocab.en	2019/4/1 15:30	EN 文件	288 KB
vocab.zh	2019/4/1 15:30	ZH 文件	455 KB

图 3 NiuTrans 提供的开源中英平行语料库

NiuTrans 数据集包含四部分：训练集 (train.zh，train.en)、验证集 (dev.zh，dev.en)、测试集 (test.zh，test.en) 以及词汇表 (vocab.zh，vocab.en)，其中以.en 为后缀的文件为英语语料，.zh 为后缀的为中文语料。值得注意的是，每个词汇表都会包含三个特殊的单词：<unk>、<s>、</s>。<unk> 表示所有不常见的单词 (对文本中使用的单词进行计数，取最常用的单词训练，剩下的单词都被替换为<unk>),<s>表示句子的开头，</s>表示句子的结尾。

本实验用到的数据集已经做好了中文分词，中文的数据样例如下:北约 不少 飞机 不得不 携 返航 ，降低 了 军事 能力 的 使用 效能，增加 了 战斗 成本，每个词之间用空格分隔，标点符号也算作一个单词。相应的英文样例如下:

many nato planes had to return to base laden with munitions , thus lowering theefficiency of use of military power and increasing the costs of fighting

由于英语中每个单词之间都有空格，故不需要分词，需要做的就是把首句大写字母还原为小写。

五、实验步骤和方法

1. 数据加载和处理

```
class Zh2EnDataLoader(BaseDataLoader):
    def __init__(self, src_filename, trg_filename, src_vocab, trg_vocab, batch_size, shuffle, logger):
        super().__init__()
        self.src_filename = src_filename
        self.trg_filename = trg_filename
        self.src_vocab = src_vocab
        self.trg_vocab = trg_vocab
        self.batch_size = batch_size
        self.shuffle = shuffle
        self.logger = logger
        self.src_lines, self.trg_lines = self.__read_data()

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

    def __getitem__(self, index):
        src_data = self.src_lines[index]
        trg_data = self.trg_lines[index]

        max_src_len = 0
        max_trg_len = 0
        src_batch_id = []
        trg_batch_id = []
        for src_tokens, trg_tokens in zip(src_data, trg_data):
            max_src_len = len(src_tokens) if len(src_tokens) > max_src_len else max_src_len
            max_trg_len = len(trg_tokens) if len(trg_tokens) > max_trg_len else max_trg_len
            src_batch_id.append([self.src_vocab.word2id[word]
            if word in self.src_vocab.word2id else self.src_vocab.word2id['<unk>'] for word in src_tokens])
            trg_batch_id.append([self.trg_vocab.word2id[word]
            if word in self.trg_vocab.word2id else self.trg_vocab.word2id['<unk>'] for word in trg_tokens])


        src = torch.LongTensor(self.batch_size, max_src_len).fill_(self.src_vocab.word2id['<pad>'])
        trg = torch.LongTensor(self.batch_size, max_trg_len).fill_(self.trg_vocab.word2id['<pad>'])
        for i in range(self.batch_size):
            src[i, :len(src_batch_id[i])] = torch.LongTensor(src_batch_id[i])
            trg[i, :len(trg_batch_id[i])] = torch.LongTensor(trg_batch_id[i])
        return src, trg
```

```
    def __read_data(self):
        self.logger.debug("-----------read data-----------")
        with open(self.src_filename, 'r', encoding='utf-8') as f:
            src_lines = np.array(f.readlines())
        with open(self.trg_filename, 'r', encoding='utf-8') as f:
            trg_lines = np.array(f.readlines())
        assert len(src_lines) == len(trg_lines)
        if self.shuffle:
            idx = np.random.permutation(len(src_lines))
            src_lines = src_lines[idx]
            trg_lines = trg_lines[idx]

        self.logger.debug("{} and {} has data {}".
        format(self.src_filename, self.trg_filename, len(src_lines)))
        return self.__preprocess_data(src_lines, trg_lines)

    def __preprocess_data(self, src_lines, trg_lines):
        self.logger.debug("-----------preprocess data-----------")
        src_lines = [['<sos>'] + line.strip().split('\t') + ['<eos>'] for line in src_lines]
        trg_lines = [['<sos>'] + line.strip().split('\t') + ['<eos>'] for line in trg_lines]

        src_lines = [src_lines[i:i+self.batch_size] for i in range(0, len(src_lines), self.batch_size)]
        trg_lines = [trg_lines[i:i+self.batch_size] for i in range(0, len(trg_lines), self.batch_size)]

        return src_lines, trg_lines
```

2. 模型构建

模型包含 Encoder 和 Decoder ，需要分别构建。

Encoder 的构建程序如下：

```
class Encoder(BaseModel):
    def __init__(self, vocab_size, h_dim, pf_dim, n_heads, n_layers, dropout, device, max_seq_len=200):
        super().__init__()
        self.n_layers = n_layers
        self.h_dim = h_dim
        self.device = device
        self.word_embeddings = WordEmbeddings(vocab_size, h_dim)

        self.pe = PositionEmbeddings(max_seq_len, h_dim)

        self.layers = nn.ModuleList()
        for i in range(n_layers):
            self.layers.append(EncoderLayer(h_dim, n_heads, pf_dim, dropout, device))

        self.dropout = nn.Dropout(dropout)
        self.scale = torch.sqrt(torch.FloatTensor([h_dim])).to(device)

    def forward(self, src, src_mask):
        output = self.word_embeddings(src) * self.scale
        src_len = src.shape[1]

        pos = torch.arange(0, src_len).unsqueeze(0).repeat(src.shape[0], 1).to(self.device)
        output = self.dropout(output + self.pe(pos))

        # output = self.pe(output)
        for i in range(self.n_layers):
            output = self.layers[i](output, src_mask)

        return output
```

```python
class EncoderLayer(BaseModel):
    def __init__(self, h_dim, n_heads, pf_dim, dropout, device):
        super().__init__()
        self.attention = MultiHeadAttentionLayer(h_dim, n_heads, dropout, device)
        self.attention_layer_norm = nn.LayerNorm(h_dim)
        self.ff_layer_norm = nn.LayerNorm(h_dim)
        self.positionwise_feedforward = PositionwiseFeedforwardLayer(h_dim, pf_dim, dropout)

        self.attention_dropout = nn.Dropout(dropout)
        self.ff_dropout = nn.Dropout(dropout)
    def forward(self, src, src_mask):
        att_output = self.attention(src, src, src, src_mask)
        # res
        output = self.attention_layer_norm(src + self.attention_dropout(att_output))

        ff_output = self.positionwise_feedforward(output)
        # res
        output = self.ff_layer_norm(output + self.ff_dropout(ff_output))

        return output
```

Decoder 的构建程序如下:

```python
class Decoder(BaseModel):
    def __init__(self, vocab_size, h_dim, pf_dim, n_heads, n_layers, dropout, device, max_seq_len=200):
        super().__init__()
        self.n_layers = n_layers
        self.h_dim = h_dim
        self.device = device
        self.word_embeddings = WordEmbeddings(vocab_size, h_dim)

        # self.pe = PositionEncoder(h_dim, device, dropout=dropout)
        self.pe = PositionEmbeddings(max_seq_len, h_dim)
        self.layers = nn.ModuleList()
        self.dropout = nn.Dropout(dropout)
        self.scale = torch.sqrt(torch.FloatTensor([h_dim])).to(device)

        for i in range(n_layers):
            self.layers.append(DecoderLayer(h_dim, pf_dim, n_heads, dropout, device))

    def forward(self, target, encoder_output, src_mask, target_mask):
        output = self.word_embeddings(target) * self.scale

        tar_len = target.shape[1]
        pos = torch.arange(0, tar_len).unsqueeze(0).repeat(target.shape[0], 1).to(self.device)

        for i in range(self.n_layers):
            output = self.layers[i](output, encoder_output, src_mask, target_mask)

        return output
```

```
class DecoderLayer(BaseModel):
    def __init__(self, h_dim, pf_dim, n_heads, dropout, device):
        super().__init__()
        self.self_attention = MultiHeadAttentionLayer(h_dim, n_heads, dropout, device)
        self.attention = MultiHeadAttentionLayer(h_dim, n_heads, dropout, device)
        self.positionwise_feedforward = PositionwiseFeedforwardLayer(h_dim, pf_dim, dropout)

        self.self_attention_layer_norm = nn.LayerNorm(h_dim)
        self.attention_layer_norm = nn.LayerNorm(h_dim)
        self.ff_layer_norm = nn.LayerNorm(h_dim)

        self.self_attention_dropout = nn.Dropout(dropout)
        self.attention_dropout = nn.Dropout(dropout)
        self.ff_dropout = nn.Dropout(dropout)

    def forward(self, target, encoder_output, src_mask, target_mask):
        self_attention_output = self.self_attention(target, target, target, target_mask)
        output = self.self_attention_layer_norm(target + self.self_attention_dropout(self_attention_output)

        attention_output = self.attention(output, encoder_output, encoder_output, src_mask)
        output = self.attention_layer_norm(output + self.attention_dropout(attention_output))

        ff_output = self.positionwise_feedforward(output)
        output = self.ff_layer_norm(ff_output + self.ff_dropout(ff_output))

        return output
```

前向 MLP 模型的构建程序如下：

```
class PositionwiseFeedforwardLayer(BaseModel):
    def __init__(self, h_dim, pf_dim, dropout):
        super().__init__()

        self.fc_1 = nn.Linear(h_dim, pf_dim)
        self.fc_2 = nn.Linear(pf_dim, h_dim)
        self.dropout = nn.Dropout(dropout)

    def forward(self, inputs):

        inputs = torch.relu(self.fc_1(inputs))
        inputs = self.dropout(inputs)
        inputs = self.fc_2(inputs)

        return inputs
```

将 encoder 和 decoder 合并到一起的程序如下：

```
class Transformer(BaseModel):
    def __init__(self, src_vocab_size, target_vocab_size, h_dim,
                 enc_pf_dim, dec_pf_dim, enc_n_layers, dec_n_layers,
                 enc_n_heads, dec_n_heads, enc_dropout, dec_dropout, device,  **kwargs):
        super().__init__()
        self.encoder = Encoder
        (src_vocab_size, h_dim, enc_pf_dim, enc_n_heads, enc_n_layers, enc_dropout, device)
        self.decoder = Decoder
        (target_vocab_size, h_dim, dec_pf_dim, dec_n_heads, dec_n_layers, dec_dropout, device)
        self.fc = nn.Linear(h_dim, target_vocab_size)

    def forward(self, src, target, src_mask, target_mask):
        encoder_output = self.encoder(src, src_mask)
        output = self.decoder(target, encoder_output, src_mask, target_mask)
        output = self.fc(output)
        return output
```

3. 模型训练和测试

模型训练代码如下：

```
def _train_epoch(self, epoch):
    self.model.train()
    total_loss = 0
    for idx, (src, trg) in enumerate(self.data_loader):
        src = src.to(self.device)
        trg = trg.to(self.device)

        src_mask = make_src_mask(src, self.data_loader.src_vocab, self.device)
        trg_mask = make_trg_mask(trg[:,:-1], self.data_loader.trg_vocab, self.device)

        self.optimizer.zero_grad()

        output = self.model(src, trg[:,:-1], src_mask, trg_mask)
        # output = [batch_size, target_len-1, target_vocab_size]
        # trg = <sos>, token1, token2, token3, ...
        # output = token1, token2, token3, ..., <eos>

        output_dim = output.shape[-1]

        output = output.contiguous().view(-1, output_dim)
        # output = [batch size * target_len - 1, target_vocab_size]

        trg = trg[:,1:].contiguous().view(-1)
        # target = [batch_size * targey_len - 1]

        loss = self.criterion(output, trg)

        loss.backward()
        # 可调参数 1 可以改为 其他值进行尝试
        torch.nn.utils.clip_grad_norm_(self.model.parameters(), 1)
        self.optimizer.step()
        total_loss += loss.item()
        if idx % self.log_step == 0:
            self.logger.info('Train Epoch: {}, {}/{} ({:.0f}%), Loss: {:.6f}'.format(epoch,
                        idx,
                        len(self.data_loader),
                        idx * 100 / len(self.data_loader),
                        loss.item()
                        ))
```

模型验证代码如下：

```python
def _valid_epoch(self):
    self.model.eval()
    val_loss = 0
    pred = []
    labels = []
    with torch.no_grad():
        for idx, (src, trg) in enumerate(self.valid_data_loader):
            src = src.to(self.device)
            trg = trg.to(self.device)

            src_mask = make_src_mask(src, self.valid_data_loader.src_vocab, self.device)
            trg_mask = make_trg_mask(trg[:,:-1], self.data_loader.trg_vocab, self.device)

            output = self.model(src, trg[:,:-1], src_mask, trg_mask)
            output = F.log_softmax(output, dim=-1)
            output_dim = output.shape[-1]
            # output = [batch size * target_len - 1, target_vocab_size]
            output = output.contiguous().view(-1, output_dim)
            trg = trg[:,1:].contiguous().view(-1)

            val_loss += self.criterion(output, trg)

    return val_loss / len(self.valid_data_loader)
```

模型测试代码如下：

```python
def translate_sentence(sentence, model, device, zh_vocab, en_vocab, zh_tokenizer, max_len = 100):
    model.eval()
    tokens = zh_tokenizer.tokenizer(sentence)
    tokens = ['<sos>'] + tokens + ['<eos>']
    print(tokens)
    tokens = [zh_vocab.word2id[word] for word in tokens]

    src_tensor = torch.LongTensor(tokens).unsqueeze(0).to(device)
    src_mask = make_src_mask(src_tensor, zh_vocab, device)
    with torch.no_grad():
        enc_src = model.encoder(src_tensor, src_mask)
    trg = [en_vocab.word2id['<sos>']]
    for i in range(max_len):
        trg_tensor = torch.LongTensor(trg).unsqueeze(0).to(device)
        trg_mask = make_trg_mask(trg_tensor, en_vocab, device)
        with torch.no_grad():
            output = model.decoder(trg_tensor, enc_src, src_mask, trg_mask)
            output = model.fc(output)

        pred_token = output.argmax(2)[:,-1].item()
        trg.append(pred_token)
        if pred_token == en_vocab.word2id['<eos>']:
            break

    trg_tokens = [en_vocab.id2word[idx] for idx in trg]
    return trg_tokens
```