基于 Transformer 的神经机器翻译

一、 实验目的

1. 本实验旨在介绍基于 Transformer 的神经机器翻译任务;
2. 掌握使用深度学习框架搭建基于 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转换。
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. 如果选择做此实验作业，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 （举例）

采用基于 Transformers 的 seq2seq 模型，包括编解码两大部分，如下图，编码部分是由若干个相同的编码器组成，解码部分也是由相同个数的解码器组成，与编码器不同的是，每一个解码器都会接受最后一个编码器的输出。

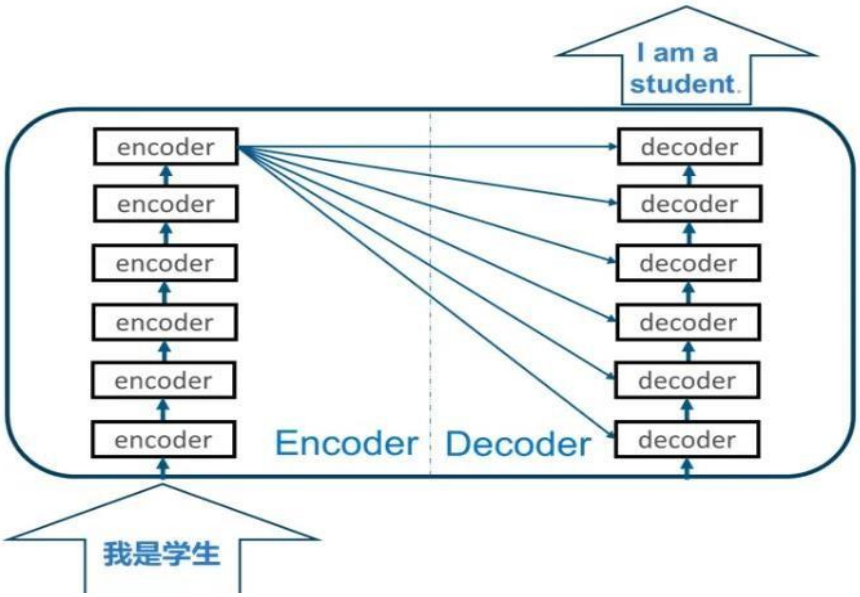


图 1.编码器-解码器架构

2、 模型输入

模型输入为单个文本序列或一对文本序列(例如， [源文，译文])。“序列”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，即可以是单个句子，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，下图给出了 BERT 模型输入序列的可视化表示，引自《BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding》。

（请注意：本部分仅用 BERT 模型介绍输入文本预处理的过程，BERT 模型是一个嵌入模型，只包含文本编码部分，不包含解码部分，不适用于生成式任务）

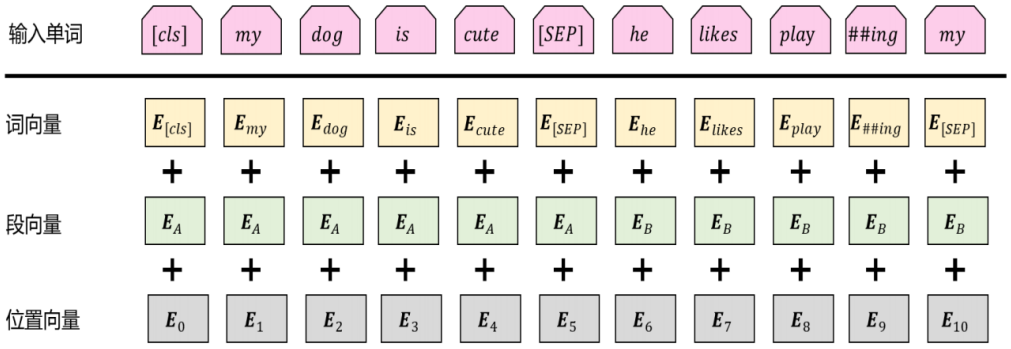


图 2.BERT 模型嵌入层架构

模型输入包含以下细节：

1) BERT 支持的序列长度最长可达 512 个 token（token 是指分词而不是单词，例如“I love nature language processing.”可以被分词成“I”，“love”，“nature”，“language”，“process”，“-ing”，“.”,具体的分规则由模型本身的词表决定）。

2) 每个序列的第一个标记始终会被添加为特殊分类嵌入（[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，末尾会被添加一个[SEP]截止符，如果在一个 batch 中有的句子比较短，则需要添加占位符[PAD]。假设 1）中的句子分别为“I love you.”和“I love nature language processing.”，设定序列长度统一为 10，那么经过分词后的序列表示如下：

“[CLS]”，“I”，“love”，“you”，“.”，“[SEP]”,“[PAD]”,“[PAD]”,“[PAD]”,“[PAD]”

“[CLS]”，“I”，“love”，“nature”，“language”，“process”，“-ing”，“.”,“[PAD]”,“[PAD]”

3) 句子对被打包在一起形成一个单独的序列，可以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。

4) 对于单句输入，我们只使用句子 A 嵌入。

3、 机器翻译模型预训练任务

机器翻译类模型通常采用自回归的方式来做预训练。通过自回归训练，模型会一个词一个词地生成目标语言句子，每生成一个词时都会把前面生成的词作为输入，去预测目标序列的后一个词，通过这种循环迭代的方式，模型最终可以学习出语言间的复杂对应关系。

4、 模型输出

当输入序列的隐状态向量在模型内部传递时，其维度为[bs,seq_length,hidden_dim]，其中 bs 表示当前训练或推理的批大小，也就是 batch_size，seq_length 表示模型当前处理过程中句子序列的长度，也就是序列中 token 的个数，如 2a 中 BERT 最大支持 512，hidden_dim 表示序列中的每个 token 对应的隐状态维度，一般基本模型为 768，较大的模型为 1024，大语言模型（如 LLaMA）为 4096。可以取出最后一层解码器输出的隐状态向量为整个模型的输出，经过自回归预测头（自回归预测头通常是一个前向神经网络，其输入输出维度为[hidden_dim,vocab_size]，vocab_size 指的是词表长度，旨在将输出的隐状态通过自回归预测头，再经过 softmax 的归一化，得到最有可能出现的单词，作为当前词的预测）后送入到 3.中损失函数进行训练；如果为推理阶段，同样需要经过自回归预测头，通过计算词表中每个词的概率得到当前词的输出。

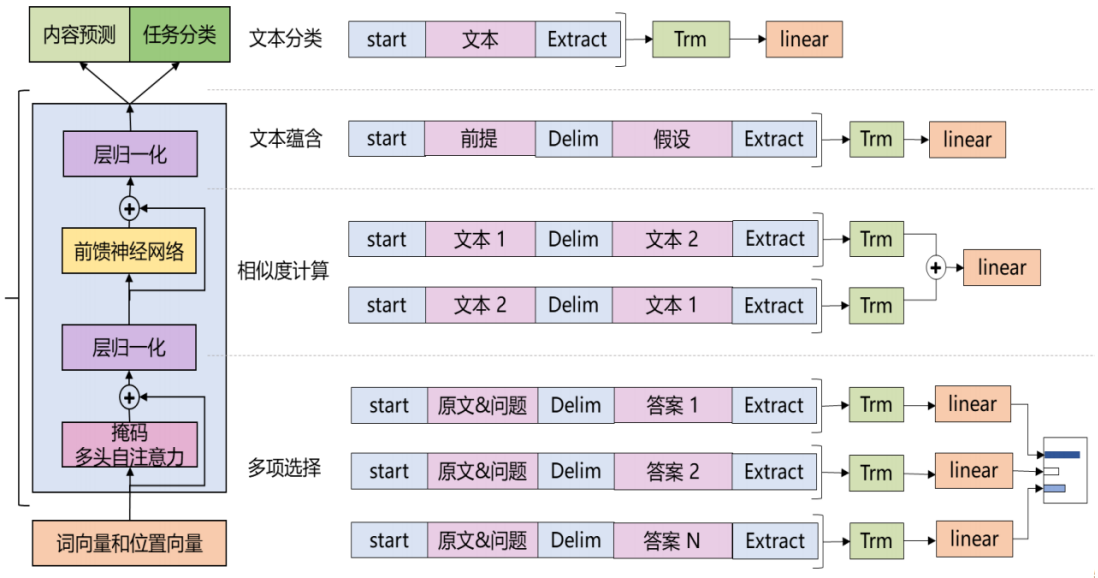
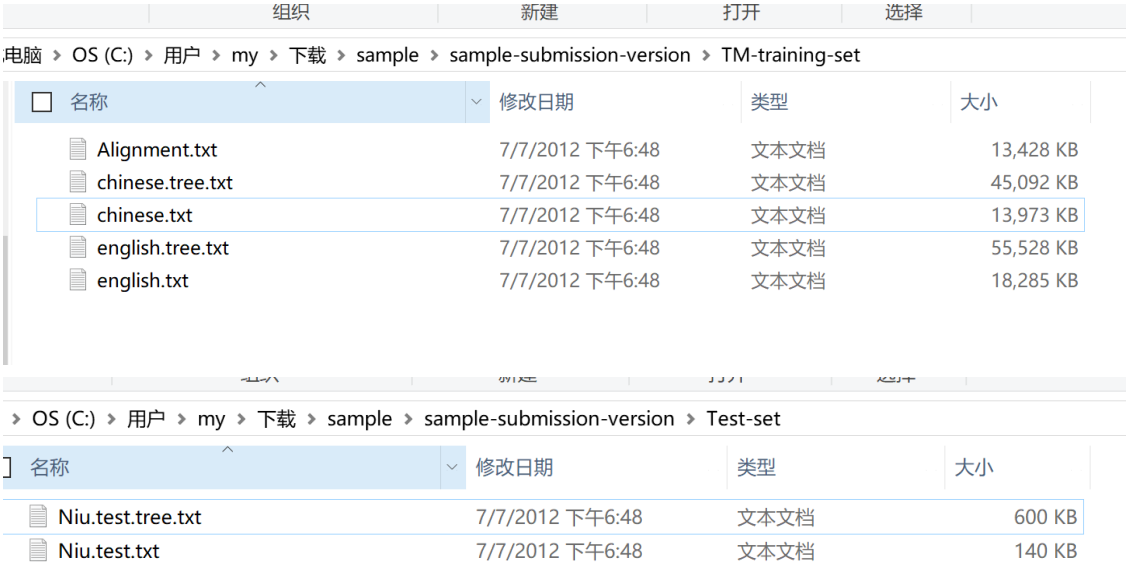


图 3.GPT-1 基础模型结构与应用

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，如下图所示。



下载地址：https://github.com/NiuTrans/NiuTrans.SMT/tree/master/sample-data

数据集包含五部分:

1) TM-training-set：TM 训练集是用于翻译模型训练的双语数据，一共提供了 199630 个句子对作为样本。包含文件 chinese.txt，english.txt，chinese.tree.txt，english.tree.txt，Alignment.txt。

2) Dev-set：Dev 集是包含 1000 个中文句子对和每个中文句子一个参考的英文对应翻译，包含文件：Niu.dev.txt。

3) Test-set：测试集是包含 1000 个单语句子的测试数据文件 Niu.test.txt。

reference-set：验证集是测试集的对应的英文翻译供比较使用。包含文件，Niu.test.reference。

文件 Alignment.txt 是 chinese.txt，english.txt 中对应句子的单词对应翻译，例如：

文件“Alignment.txt”的第 105 行是“0-0 0-1 2-2 3-2 4-8 4-9 5-6 6-4 6-5 7-10”

文件“c.txt”的第 105 行是“爱尔兰 人 过去 用 马铃薯 作为 主食 ."

文件“e.txt”的第 105 行是“the irish used to live on a diet of potatoes .”

“0-1”是指中文中的“爱尔兰”与相应英语句子中的“irish”匹配。

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并

且已经从大写转化成小写，故不需要分词。

五、 实验步骤和方法（本部分仅供参考）

基于百度飞桨框架的实现可以参考：[基于 Transformer 的机器翻译 - 飞桨 AI Studio 星河社区 (baidu.com)]

1. 数据集加载和处理

```
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

包含 encoder 和 decoder ， 需要分别构建 ：

```
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
```

Encoder 中包含了若干个 Encoderlayer，构建如下：

```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
```

以下是 decoder:

```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
```

以下是前向 MLP 模型的定义，

```
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
```

最后是将 encoder 和 decoder 合并到一起，

```
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
```

这样便完成了网络的构建。

3. 训练和测试

```
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()
                        ))
```

以上为模型训练部分代码；

```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)
```

以上为模型验证相关代码 ；

```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
```

以上为模型测试相关代码，同学们可以使用测试代码查看翻译效果。