第5章　机器翻译

机器翻译(Machine Translation),又称为自动翻译,是指利用计算机将一种自然语言(源语言)转换为另一种自然语言(目标语言)的过程。它是计算语言学的一个分支,是人工智能的终极目标之一,具有重要的科学研究价值。同时,随着经济全球化以及互联网的飞速发展,机器翻译在促进政治、经济、文化交流等方面的作用也日益凸显。

机器翻译技术的发展一直与计算机技术、信息论、语言学等学科的发展紧密相关。从早期的词典匹配,到词典结合语言学专家知识的规则翻译,再到基于语料库的统计机器翻译,随着计算机计算能力的提升和多语言信息的爆炸式增长,机器翻译技术正在逐渐走出象牙塔,开始为普通用户提供实时便捷的翻译服务。

本章我们将会学习使用飞桨提供的 API 来完成不同的机器翻译任务。

实践十六:基于序列到序列模型的中-英机器翻译

机器翻译是典型的 seq2seq 预测问题,即序列到序列的预测问题:将输入序列映射为另外一个输出序列。由于同时存在多个输入和输出时间步,这种形式的问题也被称为是 many-to-many 序列预测问题。

对 seq2seq 预测问题进行建模的一个难点是输入和输出序列的长度均有可能发生变化,一种被证明能够有效解决 seq2seq 预测问题的方法被称为 Encoder-Decoder。该体系结构包括两部分:Encoder 用于读取输入序列并将其编码成一个固定长度的向量,Decoder 用于解码该固定长度的向量并输出预测序列。

如图 5.1 所示,编码阶段的 RNN 网络接收输入序列"A B C <EOS> (EOS = End of Sentence,句末标记)",并输出一个向量作为输入序列的语义表示向量;之后解码阶段的 RNN 网络在每一个时间步进行单个字符的解码,最终模型输出"W X Y Z <EOS>",这样就实现了句子的翻译过程。

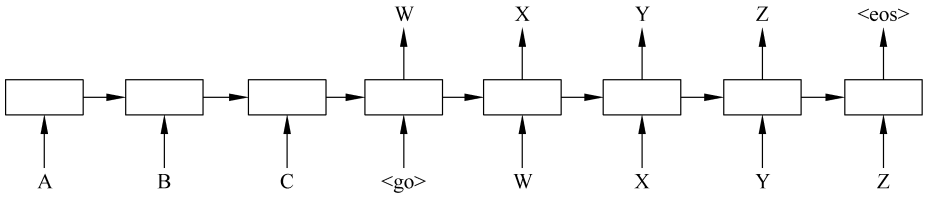

图 5.1　Encoder-Decoder

在学习了 Encoder-Decoder 的结构之后，我们就可以使用飞桨深度学习开源框架完成基于序列到序列模型的中-英机器翻译。

步骤 1：中英文数据准备

本次实践将使用 http://www.manythings.org/anki/提供的 24 697 个中-英双语句子对作为数据集。

```
# 下载数据集并解压，得到包含双语句子对的文本文件 cmn.txt
!wget -c https://www.manythings.org/anki/cmn-eng.zip && unzip cmn-eng.zip
```

我们将得到的双语句子对进行如下处理，并将其读取到 Python 的数据结构中：①对于英文，将所有字母转换为小写并只保留英文单词；②对于中文，未做分词，只按照字做了切分；③为了提高后续模型的训练速度，我们通过限制句子长度和只保留以部分英文单词作为开头的句子的方式，得到了一个包含 5746 个句子对的较小的数据集。

```
# 只保留长度不超过 10 个单词或汉字的句子
MAX_LEN = 10
lines = open('cmn.txt', encoding='utf-8').read().strip().split('\n')
# 对于英文，只保留英文单词、数字和下画线
words_re = re.compile(r'\w+')
pairs = []
for l in lines:
    en_sent, cn_sent, _ = l.split('\t')
    pairs.append((words_re.findall(en_sent.lower()), list(cn_sent)))
# 为了加速训练，构造一个较小的数据集
filtered_pairs = []
for x in pairs:
    if len(x[0]) < MAX_LEN and len(x[1]) < MAX_LEN and \
    x[0][0] in ('i', 'you', 'he', 'she', 'we', 'they'):
        filtered_pairs.append(x)
print(len(filtered_pairs))
for x in filtered_pairs[:3]: print(x)
```

```
5746
(['i', 'won'], ['我', '赢', '了', '。'])
(['he', 'ran'], ['他', '跑', '了', '。'])
(['i', 'quit'], ['我', '退', '出', '。'])
```

接下来我们分别创建中英文的词表，这两份词表被用于单词或汉字和词表 ID 之间的相互转换，词表中还会加入如下三个特殊的词："< pad >"，用于对较短的句子进行填充；"< bos >"，"begin of sentence"，表示句子开始的特殊词；"< eos >"，"end of sentence"，表示句子结束的特殊词。

```
# 英文词表
en_vocab = {}
# 中文词表
cn_vocab = {}
```

```
# 中英文词表中分别加入三个特殊字符：<pad>,<bos>,<eos>
en_vocab['<pad>'], en_vocab['<bos>'], en_vocab['<eos>'] = 0, 1, 2
cn_vocab['<pad>'], cn_vocab['<bos>'], cn_vocab['<eos>'] = 0, 1, 2

en_idx, cn_idx = 3, 3
for en, cn in filtered_pairs:
    for w in en:
        if w not in en_vocab:
            en_vocab[w] = en_idx
            en_idx += 1
    for w in cn:
        if w not in cn_vocab:
            cn_vocab[w] = cn_idx
            cn_idx += 1
```

根据构造的词表，我们创建一份实际用于训练的用 NumPy 组织的数据集，在该数据集中：①所有句子都通过<pad>的填充变成了长度相同的句子；②对于英文句子（源语言），为了达到更好的翻译效果，我们将其进行了翻转；③所创建的 padded_cn_label_sents 是训练过程中的预测目标，即当前时间步应该预测输出的单词。

```
padded_en_sents = []
padded_cn_sents = []
padded_cn_label_sents = []
for en, cn in filtered_pairs:
    # 编码器端的输入需要为英文添加结束符，并且填充至固定长度
padded_en_sent = en + ['<eos>'] + ['<pad>'] * (MAX_LEN - len(en))
# 翻转源语言
padded_en_sent.reverse()
# 解码器端的输入需要以开始符号作为第一个输入
padded_cn_sent = ['<bos>'] + cn + ['<eos>'] + ['<pad>'] * (MAX_LEN - len(cn))
# 解码器端的输出无须添加开始符号，自回归解码方式
    padded_cn_label_sent = cn + ['<eos>'] + ['<pad>'] * (MAX_LEN - len(cn) + 1)
    # 将单词或汉字转换成词表 ID
    padded_en_sents.append([en_vocab[w] for w in padded_en_sent])
    padded_cn_sents.append([cn_vocab[w] for w in padded_cn_sent])
    padded_cn_label_sents.append([cn_vocab[w] for w in padded_cn_label_sent])

train_en_sents = np.array(padded_en_sents)
train_cn_sents = np.array(padded_cn_sents)
train_cn_label_sents = np.array(padded_cn_label_sents)

print(train_en_sents.shape)
print(train_cn_sents.shape)
print(train_cn_label_sents.shape)
```

```
(5746, 11)
(5746, 12)
(5746, 12)
```

步骤 2：Encoder-Decoder 模型配置

我们将会创建一个 Encoder-Decoder 架构的模型来完成机器翻译任务。首先设置一些必要的网络结构中将会用到的参数。

```
embedding_size = 128
hidden_size = 256
num_encoder_lstm_layers = 1
en_vocab_size = len(list(en_vocab))
cn_vocab_size = len(list(cn_vocab))
epochs = 20
batch_size = 16
```

（1）Encoder 部分。

在 Encoder 端，我们在得到字符对应的 Embedding 之后连接 LSTM，构建一个对源语言进行编码的网络。除了 LSTM 之外，飞桨的 RNN 系列还提供了 SimpleRNN，GRU 等 API，我们还可以使用反向 RNN，双向 RNN，多层 RNN 等结构。同时，也可以通过设置 dropout 参数对多层 RNN 的中间层进行 dropout 处理，防止过拟合。

除了使用序列到序列的 RNN 操作之外，还可以通过 SimpleRNN，GRUCell，LSTMCell 等 API 更加灵活地创建单步的 RNN 计算，甚至可以通过继承 RNNCellBase 来实现自己的 RNN 计算单元。

```
class Encoder(paddle.nn.Layer):
    def __init__(self):
        super(Encoder, self).__init__()
        self.emb = paddle.nn.Embedding(en_vocab_size, embedding_size,)
        self.lstm = paddle.nn.LSTM(input_size = embedding_size,
                hidden_size = hidden_size,
                num_layers = num_encoder_lstm_layers)

    def forward(self, x):
        x = self.emb(x)
        x, (_, _) = self.lstm(x)
        return x
```

（2）Decoder 部分。

在 Decoder 端，我们同样使用 LSTM 来完成解码，与 Encoder 端不同的是，如下代码每次只计算一个时间步的输出，解码端在不同时间步的循环结构是在训练循环内实现的。

如果读者是第一次接触这样的网络结构，可以通过打印并观察每个 tensor 在不同步骤时的形状来更好地理解下面的代码。

```
# 每次只让 LSTM 向前计算一次
class Decoder(paddle.nn.Layer):
    def __init__(self):
        super(Decoder, self).__init__()
        self.emb = paddle.nn.Embedding(cn_vocab_size, embedding_size)
```

```
        self.lstm = paddle.nn.LSTM(input_size = embedding_size + hidden_size,
                                   hidden_size = hidden_size)

        # 输出词典单词的概率分布
        self.outlinear = paddle.nn.Linear(hidden_size, cn_vocab_size)

    def forward(self, x, previous_hidden, previous_cell, encoder_outputs):
        x = self.emb(x)
        # 得到 encoder 端对输入序列进行编码的 context vector
        context_vector = paddle.sum(encoder_outputs, 1)
        context_vector = paddle.unsqueeze(context_vector, 1)

        # 将单词的 embedding vector 和 context vector 进行拼接,得到 Decoder 端当前时刻的输入 x_t
        lstm_input = paddle.concat((x, context_vector), axis = -1)

        # LSTM 单元的计算还需要上一时刻的输出状态 h_{t-1} 和隐层状态 c_{t-1},这两个 tensor 的
        # 形状: (number_of_layers * direction, batch, hidden)
        previous_hidden = paddle.transpose(previous_hidden, [1, 0, 2])
        previous_cell = paddle.transpose(previous_cell, [1, 0, 2])

        x, (hidden, cell) = self.lstm(lstm_input, (previous_hidden, previous_cell))

        # 将 LSTM 单元输出的 tensor 形状转为: (batch, number_of_layers * direction, hidden)
        hidden = paddle.transpose(hidden, [1, 0, 2])
        cell = paddle.transpose(cell, [1, 0, 2])

        output = self.outlinear(hidden)
        output = paddle.squeeze(output)
        return output, (hidden, cell)
```

步骤 3：模型训练

接下来我们开始进行模型的训练,在训练过程中我们采取了如下策略：①在每个 epoch 开始之前,对训练数据进行随机打乱；②通过多次调用 decoder 实现解码时的循环结构；③teacher forcing 策略：在每次解码单词时,将训练数据中的真实词作为预测当前单词时的输入。相应地,读者也可以尝试使用模型上一个时间步输出的结果作为预测当前单词时的输入。

```
# 实例化编码器、解码器
encoder = Encoder()
decoder = Decoder()
# 定义优化器: 同时优化编码器与解码器的参数
opt = paddle.optimizer.Adam(learning_rate = 0.001, parameters = encoder.parameters() +
decoder.parameters())
# 开始训练
for epoch in range(epochs):
    print("epoch:{}".format(epoch))
    # 随机打乱训练数据
```

```
    perm = np.random.permutation(len(train_en_sents))
    train_en_sents_shuffled = train_en_sents[perm]
    train_cn_sents_shuffled = train_cn_sents[perm]
    train_cn_label_sents_shuffled = train_cn_label_sents[perm]
    # 批量数据迭代
for iteration in range(train_en_sents_shuffled.shape[0] // batch_size):
        x_data = train_en_sents_shuffled[(batch_size * iteration):(batch_size * (iteration + 1))]
        sent = paddle.to_tensor(x_data)
        # Encoder 端得到需要翻译的英文句子的编码表示
        en_repr = encoder(sent)
        # 解码器端原始输入
        x_cn_data = train_cn_sents_shuffled[(batch_size * iteration):(batch_size *
(iteration + 1))]
        # 解码器端输出的标准答案,用于计算损失
        x_cn_label_data = train_cn_label_sents_shuffled[(batch_size * iteration):(batch_
size * (iteration + 1))]
        # Decoder 端在第一步进行解码时需要初始化 h0 和 c0,tensor 形状为: (batch, num_layer *
        #num_of_direction, hidden_size)
        hidden = paddle.zeros([batch_size, 1, hidden_size])
        cell = paddle.zeros([batch_size, 1, hidden_size])
        loss = paddle.zeros([1])
        # Decoder 端的循环解码
        for i in range(MAX_LEN + 2):
            # 获取每步的输入以及输出的标准答案
            cn_word = paddle.to_tensor(x_cn_data[:,i:i+1])
            cn_word_label = paddle.to_tensor(x_cn_label_data[:,i])
            # 解码器解码
            logits, (hidden, cell) = decoder(cn_word, hidden, cell, en_repr)
            # 计算解码损失: 交叉熵损失,解码的词是否正确
            step_loss = F.cross_entropy(logits, cn_word_label)
            loss += step_loss
        # 计算平均损失
        loss = loss / (MAX_LEN + 2)
        if(iteration % 200 == 0):
            print("iter {}, loss:{}".format(iteration, loss.numpy()))
        # 反向传播,梯度更新
        loss.backward()
        opt.step()
        opt.clear_grad()
```

模型在训练过程中的部分输出如下,我们可以看出在经过几个轮次的训练之后,loss 不断下降并最终趋于稳定。

```
epoch:15
iter 0, loss:[0.657116]
iter 200, loss:[0.71158755]
epoch:16
iter 0, loss:[0.60776967]
iter 200, loss:[0.47258767]
```

```
epoch:17
iter 0, loss:[0.47723645]
iter 200, loss:[0.53901356]
epoch:18
iter 0, loss:[0.44978726]
iter 200, loss:[0.36394048]
epoch:19
iter 0, loss:[0.4093375]
iter 200, loss:[0.41582176]
```

步骤 4：机器翻译模型预测

模型训练完成后，我们就得到了一个能够将英文翻译成中文的机器翻译模型。在预测过程中，我们需要通过 greedy search 来实现使用该模型完成机器翻译。

```
encoder.eval()
decoder.eval()
# 从训练集中随机抽取 10 个样本
num_of_examples_to_evaluate = 10

indices = np.random.choice(len(train_en_sents), num_of_examples_to_evaluate, replace =
False)
x_data = train_en_sents[indices]
sent = paddle.to_tensor(x_data)
# 编码器提取特征
en_repr = encoder(sent)

word = np.array(
    [[cn_vocab['<bos>']]] * num_of_examples_to_evaluate
)
word = paddle.to_tensor(word)

hidden = paddle.zeros([num_of_examples_to_evaluate, 1, hidden_size])
cell = paddle.zeros([num_of_examples_to_evaluate, 1, hidden_size])

# 逐步解码
decoded_sent = []
for i in range(MAX_LEN + 2):
    logits, (hidden, cell) = decoder(word, hidden, cell, en_repr)
    word = paddle.argmax(logits, axis=1)
    decoded_sent.append(word.numpy())
    word = paddle.unsqueeze(word, axis=-1)

results = np.stack(decoded_sent, axis=1)
for i in range(num_of_examples_to_evaluate):
    en_input = " ".join(filtered_pairs[indices[i]][0])
    ground_truth_translate = "".join(filtered_pairs[indices[i]][1])
    model_translate = ""
    for k in results[i]:
```

```
            w = list(cn_vocab)[k]
            if w != '<pad>' and w != '<eos>':
                model_translate += w
        print(en_input)
        print("true: {}".format(ground_truth_translate))
    print("pred: {}".format(model_translate))
```

我们将目标语言的真实值和模型预测输出的结果进行对比，以验证机器翻译的效果。

```
she studies english every day
true: 她每天学习英语。
pred: 她每天学习英语。
i plan to go there
true: 我打算去那里。
pred: 我打算去那里。
i am interested in sports
true: 我对运动感兴趣。
pred: 我对运动感兴趣。
they obeyed orders
true: 他们服从了命令。
pred: 他们听笑了命令。
he is tall
true: 他高。
pred: 他高。
we won the battle
true: 我们战争胜利了。
pred: 我们战争胜利了。
```

实践十七：基于注意力机制的中-英机器翻译

在实践十六中我们学习了能够解决 seq2seq 预测问题的 Encoder-Decoder 结构，知道了 Encoder 将所有的输入序列都编码成一个统一的语义向量 Context vector，然后再由 Decoder 进行解码。这种结构实际上存在一个很明显的问题：我们很难寄希望于将输入的序列转化为固定的向量而保存所有的有效信息，尤其是随着所需翻译句子的长度的增加，这种结构的效果会显著下降。除此之外，解码器只用到了编码器的最后一个隐藏层状态，信息利用率低下。因此，如果想要改进 Encoder-Decoder 结构，最好的切入角度就是利用 Encoder 端的所有隐藏层状态 h_i 来解决 Context 的长度限制问题，这就是 Attention 机制。

我们人类在翻译文章时，会将注意力关注于我们当前正在翻译的部分。Attention 机制与此十分类似，假设我们需要翻译“Machine Learning”-“机器学习”这个句子对，当我们在翻译“机器”时，只需要将注意力放在源语言中“Machine”的部分；同样的，在翻译“学习”时，也只用关注原句中的“Learning”。这样，当我们在 Decoder 端进行预测时就可以利用 Encoder 端的所有信息，而不是局限于原来模型中定长的隐藏向量 Context，减少了长程信息的丢失。

以上是对于 Attention 机制的直观理解，接下来我们详细介绍 Attention 机制的内部运算，如图 5.2 所示。

首先我们基于 RNN 网络得到 Encoder 端的 hidden state：$(h_1, h_2, \cdots, h_T)$。假设当前 Decoder 端的 hidden state 是 s_{t-1}，我们可以计算 Encoder 端每一个输入位置 j 与当前输出位置的相关性，记为 $e_{tj} = a(s_{t-1}, h_j)$，写成对应的向量形式即为 $\boldsymbol{e}_t = (a(s_{t-1}, h_1), a(s_{t-1}, h_2), \cdots, a(s_{t-1}, h_T))$，其中 $a(\cdot)$ 表示相关性运算，常见的有点乘：$\boldsymbol{e}_t = \boldsymbol{s}_{t-1}^{\mathrm{T}}\boldsymbol{h}$，加权点乘：$\boldsymbol{e}_t = \boldsymbol{s}_{t-1}^{\mathrm{T}}\boldsymbol{W}\boldsymbol{h}$，加和：$\boldsymbol{e}_t = \boldsymbol{v}^{\mathrm{T}}\tanh(W_1\boldsymbol{h} + W_2\boldsymbol{s}_{t-1})$ 等形式。然后对 $\boldsymbol{e}_t$ 进行 softmax 操作将其归一化得到 attention 的概率分布：$\boldsymbol{a}_t = \mathrm{softmax}(\boldsymbol{e}_t)$，其展开形式为 $a_{tj} = \dfrac{\exp(e_{tj})}{\sum_{k=1}^{T}\exp(e_{tk})}$。利用 $\boldsymbol{a}_t$ 对 Encoder 端的隐层状态进行加权求和即得到相应的 Context vector：$\boldsymbol{c}_t = \sum_{j=1}^{T} a_{tj}h_j$。由此，我们可以计算 Decoder 端的下一时刻的 hidden state：$s_t = f(s_{t-1}, y_{t-1}, c_t)$ 以及该位置的输出 $p(y_t \mid y_1, \cdots, y_{t-1}, \boldsymbol{x}) = g(y_{t-1}, s_t, c_t)$。

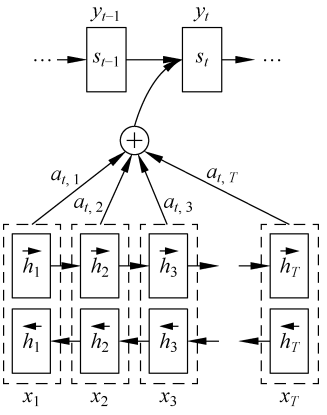

图 5.2　Attention 机制

这里的关键操作是计算 Encoder 端各个隐藏层状态和 Decoder 端当前隐藏层状态的关联性的权重，得到 Attention 分布，从而得到对于当前输出位置比较重要的输入位置的权重，在预测输出时该输入位置的单词表示对应的比重会较大。

通过 Attention 机制的引入，我们打破了只能利用 Encoder 端最终单一向量结果的限制，从而使模型可以将注意力集中在所有对于下一个目标单词重要的输入信息上，使模型效果得到极大的改善。还有一个优点是，我们通过观察 attention 权重矩阵的变化，可以知道机器翻译的结果和源文字之间的对应关系，有助于更好地理解模型工作机制，如图 5.3 所示。

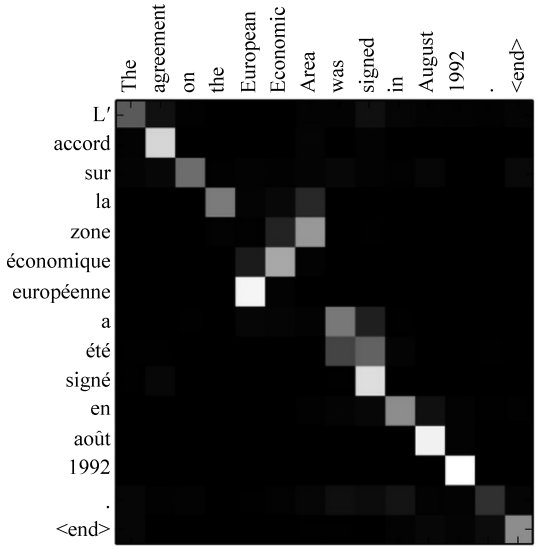

图 5.3　Attention 可视化

步骤 1：Encoder-AttentionDecoder 模型配置

带有注意力机制的 Encoder-Decoder 结构与原始结构相比仅在 Decoder 端的代码部分略有差异，在这里我们仅给出 Decoder 端的代码，读者可以参考实践十六的内容进行完整的机器翻译的实现。

```
# 和实践十六一样，每次只让 LSTM 向前计算一次
class AttentionDecoder(paddle.nn.Layer):
    def __init__(self):
        super(AttentionDecoder, self).__init__()
        self.emb = paddle.nn.Embedding(cn_vocab_size, embedding_size)
        self.lstm = paddle.nn.LSTM(input_size=embedding_size + hidden_size,
                                   hidden_size=hidden_size)

        # 这里使用了一个由两层 Linear 组成的网络来完成注意力机制的计算，它用来计算目标语
        # 言在每次翻译一个词的时候，需要对源语言当中的每个词需要赋予多少的权重
        self.attention_linear1 = paddle.nn.Linear(hidden_size * 2, hidden_size)
        self.attention_linear2 = paddle.nn.Linear(hidden_size, 1)

        self.outlinear = paddle.nn.Linear(hidden_size, cn_vocab_size)

    def forward(self, x, previous_hidden, previous_cell, encoder_outputs):
        x = self.emb(x)
        # 将 Encoder 端所有隐层状态和当前时刻 Decoder 端隐层状态拼接，作为 Attention 模块的输入
        attention_inputs = paddle.concat((encoder_outputs,
                           paddle.tile(previous_hidden, repeat_times=[1, MAX_LEN+1, 1])),
                           axis=-1
                          )

        attention_hidden = self.attention_linear1(attention_inputs)
        attention_hidden = F.tanh(attention_hidden)
        attention_logits = self.attention_linear2(attention_hidden)
        attention_logits = paddle.squeeze(attention_logits)

        # 将两层 linear 网络的输出进行 softmax 归一化，得到 attention 的概率分布
        attention_weights = F.softmax(attention_logits)
        attention_weights = paddle.expand_as(paddle.unsqueeze(attention_weights, -1),
                                             encoder_outputs)

        # 将 Encoder 端每一个时刻的隐层状态乘以相对应的 attention 权重
        context_vector = paddle.multiply(encoder_outputs, attention_weights)
        context_vector = paddle.sum(context_vector, 1)
        context_vector = paddle.unsqueeze(context_vector, 1)

        lstm_input = paddle.concat((x, context_vector), axis=-1)

        previous_hidden = paddle.transpose(previous_hidden, [1, 0, 2])
        previous_cell = paddle.transpose(previous_cell, [1, 0, 2])
```

```
        x, (hidden, cell) = self.lstm(lstm_input, (previous_hidden, previous_cell))

        hidden = paddle.transpose(hidden, [1, 0, 2])
        cell = paddle.transpose(cell, [1, 0, 2])

        output = self.outlinear(hidden)
        output = paddle.squeeze(output)
        return output, (hidden, cell)
```

步骤 2：模型训练

接下来我们开始进行模型的训练，在训练过程中采取了和实践十六相似的策略。

```
# 实例化编码器、解码器
encoder = Encoder()
decoder = AttentionDecoder()
# 定义优化器：同时优化编码器与解码器的参数
opt = paddle.optimizer.Adam(learning_rate = 0.001, parameters = encoder.parameters() +
decoder.parameters())
# 开始训练
for epoch in range(epochs):
    print("epoch:{}".format(epoch))
    # 随机打乱训练数据
    perm = np.random.permutation(len(train_en_sents))
    train_en_sents_shuffled = train_en_sents[perm]
    train_cn_sents_shuffled = train_cn_sents[perm]
    train_cn_label_sents_shuffled = train_cn_label_sents[perm]
    # 批量数据迭代
for iteration in range(train_en_sents_shuffled.shape[0] // batch_size):
        x_data = train_en_sents_shuffled[(batch_size * iteration):(batch_size * (iteration + 1))]
        sent = paddle.to_tensor(x_data)
        # Encoder 端得到需要翻译的英文句子的编码表示
        en_repr = encoder(sent)
        # 解码器端原始输入
        x_cn_data = train_cn_sents_shuffled[(batch_size * iteration):(batch_size *
(iteration + 1))]
        # 解码器端输出的标准答案，用于计算损失
        x_cn_label_data = train_cn_label_sents_shuffled[(batch_size * iteration):(batch_
size * (iteration + 1))]
        # Decoder 端在第一步进行解码时需要初始化 h0 和 c0，tensor 形状为：(batch, num_layer *
num_of_direction, hidden_size)
        hidden = paddle.zeros([batch_size, 1, hidden_size])
        cell = paddle.zeros([batch_size, 1, hidden_size])
        loss = paddle.zeros([1])
        #AttentionDecoder 端的循环解码
        for i in range(MAX_LEN + 2):
            # 获取每步的输入以及输出的标准答案
            cn_word = paddle.to_tensor(x_cn_data[:,i:i + 1])
            cn_word_label = paddle.to_tensor(x_cn_label_data[:,i])
            # 解码器解码
            logits, (hidden, cell) = decoder(cn_word, hidden, cell, en_repr)
            # 计算解码损失，交叉熵损失，解码的词是否正确
```

```
        step_loss = F.cross_entropy(logits, cn_word_label)
        loss += step_loss
    # 计算平均损失
    loss = loss / (MAX_LEN + 2)
    if(iteration % 200 == 0):
        print("iter {}, loss:{}".format(iteration, loss.numpy()))
    # 反向传播,梯度更新
    loss.backward()
    opt.step()
    opt.clear_grad()
```

模型在训练过程中的 loss 不断下降并最终趋于稳定，而且在效果上是明显优于不带注意力机制的 encoder-decoder 模型结构的。

实践十八：基于 Transformer 的中-英机器翻译

在实践十七中我们学习了 Attention 机制，知道了它可以解决 Encoder-Decoder 结构中固定的语义向量 Context 无法保存所有输入信息的问题。既然 Attention 机制如此有效，那么我们可不可以去掉模型中的 RNN 部分，只利用 Attention 结构呢？这就是谷歌提出的 Self-Attention 机制以及 Transformer 架构。

我们仍然从一个翻译例子来引出 self-attention 机制，假设我们需要翻译“I arrived at the bank after crossing the river”这句话，当我们在翻译“bank”时，如何知道它指的是“银行”还是“河岸”呢？这就需要我们联系上下文，当我们看到“river”之后就知道这里的“bank”有很大的概率可以被翻译为“河岸”。但是在 RNN 网络中，我们需要顺序处理从“bank”到“river”的所有单词，当它们相距较远时 RNN 的效果往往很差，同时由于其处理的顺序性导致 RNN 网络的效率也比较低。Self-Attention 则利用了 Attention 机制，可以计算每个单词与其他所有单词之间的关联性。在这句话中，当翻译“bank”一词时，“river”被分配得到较高的 Attention score，利用这些 Attention score 就可以得到一个加权表示的向量用来表征“bank”的语义信息，这一表示能够很好地利用上下文所提供的信息。

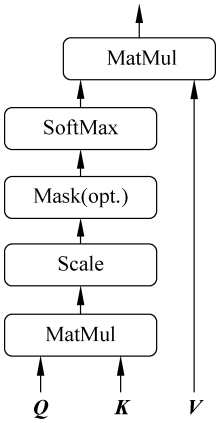

图 5.4　Scaled Dot-Product Attention

接下来我们详细介绍 Self-Attention 的计算过程，其基本结构如图 5.4 所示。

对于 self-attention 来讲，$\boldsymbol{Q}$(Query)，$\boldsymbol{K}$(Key)，$\boldsymbol{V}$(Value)三个矩阵均来自于同一个输入，首先我们要计算 $\boldsymbol{Q}$ 与 $\boldsymbol{K}$ 之间的点乘，为了防止其结果过大，会除以一个尺度标度 $\sqrt{d_k}$，其中 d_k 为一个 $\boldsymbol{Q}$ 或 $\boldsymbol{K}$ 向量的维度。再利用 Softmax 操作将其结果归一化为概率分布，然后再乘以矩阵 $\boldsymbol{V}$ 就得到权重求和的表示。上述操作可以表示为

$$\text{Attention}(\boldsymbol{Q},\boldsymbol{K},\boldsymbol{V})=\text{softmax}\left(\frac{\boldsymbol{Q}\boldsymbol{K}^{\mathrm{T}}}{\sqrt{d_k}}\right)\boldsymbol{V}$$

以上描述可能比较抽象且难以理解，我们来看一个具体的例子，假设我们要翻译一个词组“Thinking Machines”，其中

“Thinking”的 embedding vector 用 x_1 表示，“Machines”的 embedding vector 用 x_2 表示。当我们在处理“Thinking”这个单词时，我们需要计算句子中所有单词与它的 Attention Score，这就类似于将当前单词作为搜索的 query，和句子中所有单词（包含该词本身）的 key 去匹配，计算两者之间的相关程度。我们用 q_1 代表“Thinking”对应的 query vector，k_1、k_2 分别代表“Thinking”和“Machines”对应的 key vector，那么在计算“Thinking”的 attention score 时需要计算 q_1 与 k_1、k_2 的点乘，如图 5.5 所示。同理在计算“Machines”的 attention score 时需要计算 q_1 与 k_1、k_2 的点乘。

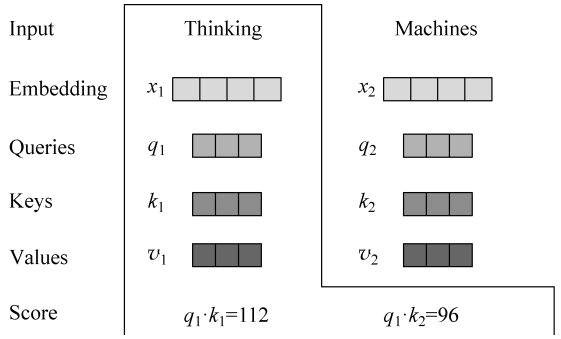

图 5.5　Attention 计算

然后我们对得到的结果进行尺度缩放和 softmax 归一化，就得到了其他单词相对于当前单词的 attention 概率分布。显然，当前单词与其自身的 attention score 一般最大，其他单词根据与当前单词的重要程度也有相应的打分。然后我们将这些 attention score 与 value vector 相乘，即得到加权表示的向量 z_1，如图 5.6 所示。

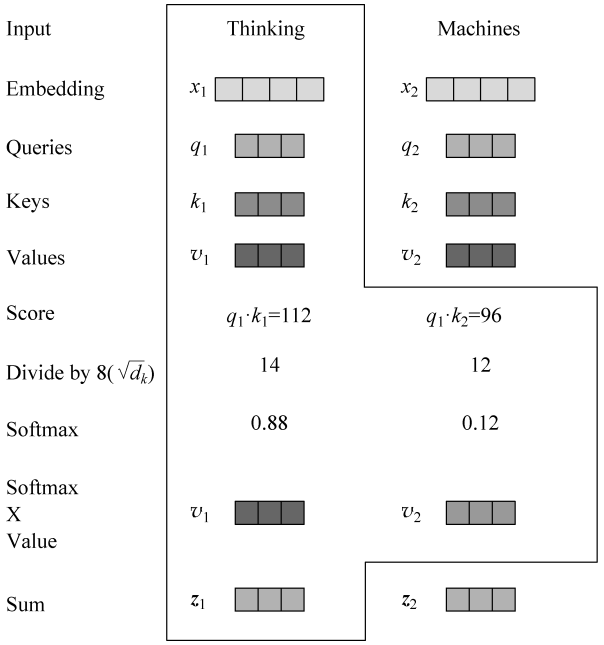

图 5.6　Attention 计算

如果将所有输入的 embedding vector 合并为矩阵形式，那么所有的 query，key 和 value 向量也可以合并为矩阵形式表示，其中 $\boldsymbol{W}^{Q}$、$\boldsymbol{W}^{K}$、$\boldsymbol{W}^{V}$ 是模型在训练过程中需要学习的参数，上述操作即可简化为如图 5.7 所示的矩阵形式。

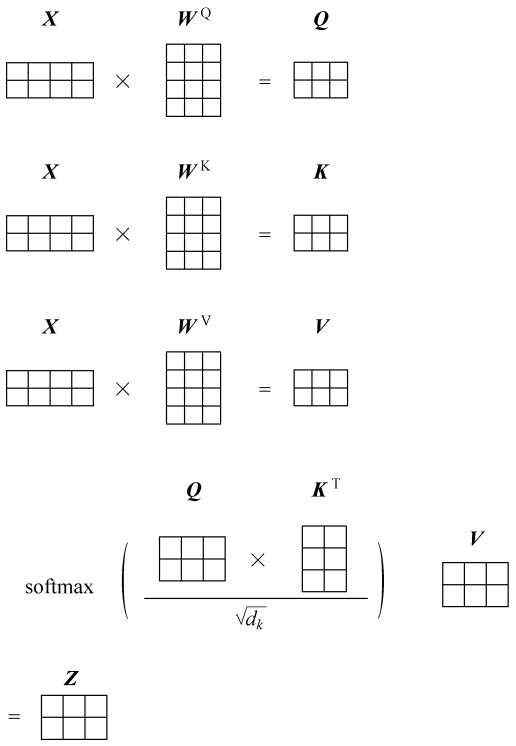

图 5.7　Attention 计算

以上就是 self-attention 机制的主要内容，在 Transformer 的网络架构中，编码器和解码器没有采用 RNN 或 CNN 等网络，而是完全依赖于 self-attention 机制，其网络架构如图 5.8 所示。

这里的 Multi-head Attention 实际上就是多个 Self-Attention 结构的结合，模型需要学习不同的 $\boldsymbol{W}^{Q}$、$\boldsymbol{W}^{K}$、$\boldsymbol{W}^{V}$ 参数，来得到不同的 $\boldsymbol{Q}$、$\boldsymbol{K}$、$\boldsymbol{V}$ 矩阵，如图 5.9 所示，每个 head 能够学习到在不同表示空间中的特征，这使得模型具有更强的表示能力。

对于 Transformer 结构，它的 Encoder 端就是将上述 Multi-head Attention 作为基本单元进行堆叠，除了第一层接收的是输入序列的 embedding 表示以外，其余每一层的 $\boldsymbol{Q}$、$\boldsymbol{K}$、$\boldsymbol{V}$ 均来自于前一层的输出。

Decoder 端的 Self-Attention 和 Encoder 端基本完全一致，需要注意的一点是解码过程是 step by step 的生成过程，因此目标序列中的每个单词在进行自注意力编码时，仅可以“看到”当前输出位置的所有前驱词的信息，所以我们需要对目标序列中的单词进行掩码操作，确保在编码当前位置单词时“看不到”后继单词的信息，该操作即对应 Decoder 端第一级的 Masked Multi-head Attention。第二级的 Multi-Head Attention 也被称作 Encoder-Decoder Attention Layer，它的作用和实践十七中介绍的 Attention 机制的作用相同。

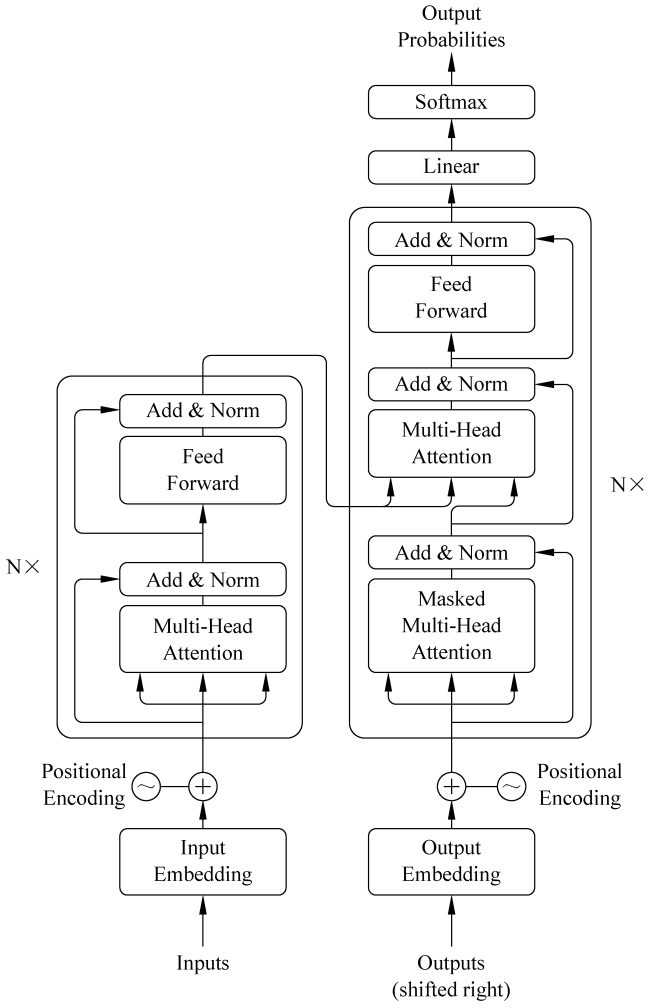

图 5.8　Multi-head Attention

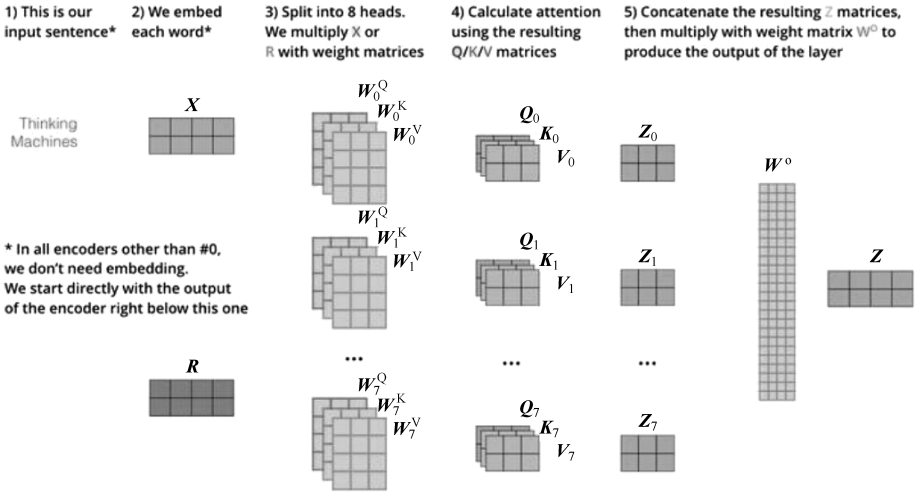

图 5.9　Multi-head Attention 计算

除此之外，Transformer 结构中还有一些其他的细节，例如位置编码、残差连接和层标准化等，感兴趣的读者可以阅读原论文学习。

接下来我们学习使用飞桨深度学习开源框架完成基于 Transformer 的中-英机器翻译模型。飞桨框架实现了 Transformer 的基本层，因此可以直接调用：TransformerEncoderLayer 类定义了编码器端的一个层，包括多头注意力子层及逐位前馈网络子层；TransformerEncoder 类堆叠 TransformerEncoderLayer 层，返回指定层数的编码器；TransformerDecoderLayer 类定义了解码器端的一个层，包括多头自注意力子层、多头交叉注意力子层及逐位前馈网络子层；TransformerDecoder 类堆叠 TransformerDecoderLayer 层，返回指定层数的解码器。我们将飞桨封装好的 Transformer 类进行了展开，希望读者能够通过学习完整的代码更进一步地掌握 Transformer 的结构及其运算过程。训练集的构建部分读者仍然可以参考实践十六进行实现，因为基于 Transformer 架构和基于 Encoder-Decoder 架构的机器翻译模型在训练过程中的数据读取部分略有不同，因此我们给出模型配置和训练的完整代码。

步骤 1：Transformer 模型配置

我们通过定义 TransformerEncoder 类和 TransformerDecoder 类详细的内部实现来更好地理解 Transformer 的运行过程。首先是定义 MultiHeadAttention()多头注意力子层，实现隐状态的注意力计算。

```
# 多头注意力子层
class MultiHeadAttention(Layer):
    Cache = collections.namedtuple("Cache", ["k", "v"])
    StaticCache = collections.namedtuple("StaticCache", ["k", "v"])
    def __init__(self, embed_dim, num_heads, dropout = 0.,
                 kdim = None, vdim = None, need_weights = False,
                 weight_attr = None, bias_attr = None):
        super(MultiHeadAttention, self).__init__()
        self.embed_dim = embed_dim
        self.kdim = kdim if kdim is not None else embed_dim
        self.vdim = vdim if vdim is not None else embed_dim
        self.num_heads = num_heads
        self.dropout = dropout
        self.need_weights = need_weights
        self.head_dim = embed_dim // num_heads
        assert self.head_dim * num_heads == self.embed_dim, "embed_dim must be divisible by
num_heads"
        self.q_proj = Linear(
            embed_dim, embed_dim, weight_attr, bias_attr = bias_attr)
        self.k_proj = Linear(
            self.kdim, embed_dim, weight_attr, bias_attr = bias_attr)
        self.v_proj = Linear(
            self.vdim, embed_dim, weight_attr, bias_attr = bias_attr)
        self.out_proj = Linear(
            embed_dim, embed_dim, weight_attr, bias_attr = bias_attr)

    def _prepare_qkv(self, query, key, value, cache = None):
```

```
        q = self.q_proj(query)
        q = tensor.reshape(x=q, shape=[0, 0, self.num_heads, self.head_dim])
        q = tensor.transpose(x=q, perm=[0, 2, 1, 3])
        if isinstance(cache, self.StaticCache):
            # Decoder 端计算 encoder-decoder attention
            k, v = cache.k, cache.v
        else:
            k, v = self.compute_kv(key, value)
        if isinstance(cache, self.Cache):
            # Decoder 端计算 self-attention
            k = tensor.concat([cache.k, k], axis=2)
            v = tensor.concat([cache.v, v], axis=2)
            cache = self.Cache(k, v)
        return (q, k, v) if cache is None else (q, k, v, cache)
    def compute_kv(self, key, value):
        k = self.k_proj(key)
        v = self.v_proj(value)
        k = tensor.reshape(x=k, shape=[0, 0, self.num_heads, self.head_dim])
        k = tensor.transpose(x=k, perm=[0, 2, 1, 3])
        v = tensor.reshape(x=v, shape=[0, 0, self.num_heads, self.head_dim])
        v = tensor.transpose(x=v, perm=[0, 2, 1, 3])
        return k, v

    def forward(self, query, key=None, value=None, attn_mask=None, cache=None):
        key = query if key is None else key
        value = query if value is None else value
        # 计算 q ,k ,v
        if cache is None:
            q, k, v = self._prepare_qkv(query, key, value, cache)
        else:
            q, k, v, cache = self._prepare_qkv(query, key, value, cache)
        # 注意力权重系数的计算采用缩放点乘的形式
        product = layers.matmul(
            x=q, y=k, transpose_y=True, alpha=self.head_dim**-0.5)
        if attn_mask is not None:
            product = product + attn_mask
        weights = F.softmax(product)
        if self.dropout:
            weights = F.dropout(
                weights,
                self.dropout,
                training=self.training,
                mode="upscale_in_train")
        out = tensor.matmul(weights, v)
        out = tensor.transpose(out, perm=[0, 2, 1, 3])
        out = tensor.reshape(x=out, shape=[0, 0, out.shape[2] * out.shape[3]])
        out = self.out_proj(out)
        outs = [out]
        if self.need_weights:
            outs.append(weights)
```

```
        if cache is not None:
            outs.append(cache)
        return out if len(outs) == 1 else tuple(outs)
```

到这里就实现了多头注意力机制的运算，下面通过 TransformerEncoderLayer()网络层封装 Encoder 的每一层，方便 Transformer 中模块的堆叠。

```
# Encoder 端的一层
class TransformerEncoderLayer(Layer):
    def __init__(self, d_model, nhead, dim_feedforward,
                 dropout=0.1, activation="relu",
                 attn_dropout=None, act_dropout=None,
                 normalize_before=False, weight_attr=None,
                 bias_attr=None):
        self._config = locals()
        self._config.pop("self")
        self._config.pop("__class__", None)          # py3
        super(TransformerEncoderLayer, self).__init__()
        attn_dropout = dropout if attn_dropout is None else attn_dropout
        act_dropout = dropout if act_dropout is None else act_dropout
        self.normalize_before = normalize_before

        weight_attrs = _convert_param_attr_to_list(weight_attr, 2)
        bias_attrs = _convert_param_attr_to_list(bias_attr, 2)
        self.self_attn = MultiHeadAttention( d_model, nhead,
            dropout=attn_dropout, weight_attr=weight_attrs[0],
            bias_attr=bias_attrs[0])
        self.linear1 = Linear(
            d_model, dim_feedforward, weight_attrs[1], bias_attr=bias_attrs[1])
        self.dropout = Dropout(act_dropout, mode="upscale_in_train")
        self.linear2 = Linear(
            dim_feedforward, d_model, weight_attrs[1], bias_attr=bias_attrs[1])
        self.norm1 = LayerNorm(d_model)
        self.norm2 = LayerNorm(d_model)
        self.dropout1 = Dropout(dropout, mode="upscale_in_train")
        self.dropout2 = Dropout(dropout, mode="upscale_in_train")
        self.activation = getattr(F, activation)

    def forward(self, src, src_mask=None, cache=None):
        residual = src
        if self.normalize_before:
            src = self.norm1(src)
        if cache is None:
            src = self.self_attn(src, src, src, src_mask)
        else:
            src, incremental_cache = self.self_attn(src, src, src, src_mask, cache)
        src = residual + self.dropout1(src)
        if not self.normalize_before:
            src = self.norm1(src)
        residual = src
```

```
        if self.normalize_before:
            src = self.norm2(src)
        src = self.linear2(self.dropout(self.activation(self.linear1(src))))
        src = residual + self.dropout2(src)
        if not self.normalize_before:
            src = self.norm2(src)
        return src if cache is None else (src, incremental_cache)
```

TransformerEncoder()主要是将上面的单个网络层串联起来，变成一个完整的 Encoder 结构。

```
# Encoder 端的多层堆叠
class TransformerEncoder(Layer):
    def __init__(self, encoder_layer, num_layers, norm = None):
        super(TransformerEncoder, self).__init__()
        self.layers = LayerList([(encoder_layer if i == 0 else type(encoder_layer)(**
encoder_layer._config)) for i in range(num_layers)])
        self.num_layers = num_layers
        self.norm = norm

    def forward(self, src, src_mask = None, cache = None):
        output = src
        new_caches = []
        for i, mod in enumerate(self.layers):
            if cache is None:
                output = mod(output, src_mask = src_mask)
            else:
                output, new_cache = mod(output, src_mask = src_mask, cache = cache[i])
                new_caches.append(new_cache)
        if self.norm is not None:
            output = self.norm(output)
        return output if cache is None else (output, new_caches)
```

TransformerDecoderLayer()和 TransformerEncoderLayer()作用类似。

```
# Decoder 端的一层
class TransformerDecoderLayer(Layer):
    def __init__(self, d_model, nhead, dim_feedforward,
                 dropout = 0.1, activation = "relu",
                 attn_dropout = None, act_dropout = None,
                 normalize_before = False, weight_attr = None,
                 bias_attr = None):
        self._config = locals()
        self._config.pop("self")
        self._config.pop("__class__", None) # py3
        super(TransformerDecoderLayer, self).__init__()
        attn_dropout = dropout if attn_dropout is None else attn_dropout
        act_dropout = dropout if act_dropout is None else act_dropout
        self.normalize_before = normalize_before
        weight_attrs = _convert_param_attr_to_list(weight_attr, 3)
```

```
        bias_attrs = _convert_param_attr_to_list(bias_attr, 3)
        self.self_attn = MultiHeadAttention(d_model, nhead,
            dropout=attn_dropout, weight_attr=weight_attrs[0],
            bias_attr=bias_attrs[0])
        self.cross_attn = MultiHeadAttention( d_model, nhead,
            dropout=attn_dropout, weight_attr=weight_attrs[1],
            bias_attr=bias_attrs[1])
        self.linear1 = Linear(
            d_model, dim_feedforward, weight_attrs[2], bias_attr=bias_attrs[2])
        self.dropout = Dropout(act_dropout, mode="upscale_in_train")
        self.linear2 = Linear(
            dim_feedforward, d_model, weight_attrs[2], bias_attr=bias_attrs[2])
        self.norm1 = LayerNorm(d_model)
        self.norm2 = LayerNorm(d_model)
        self.norm3 = LayerNorm(d_model)
        self.dropout1 = Dropout(dropout, mode="upscale_in_train")
        self.dropout2 = Dropout(dropout, mode="upscale_in_train")
        self.dropout3 = Dropout(dropout, mode="upscale_in_train")
        self.activation = getattr(F, activation)

    def forward(self, tgt, memory, tgt_mask=None, memory_mask=None, cache=None):
        residual = tgt
        if self.normalize_before:
            tgt = self.norm1(tgt)
        if cache is None:
            tgt = self.self_attn(tgt, tgt, tgt, tgt_mask, None)
        else:
            tgt, incremental_cache = self.self_attn(tgt, tgt, tgt, tgt_mask, cache[0])
        tgt = residual + self.dropout1(tgt)
        if not self.normalize_before:
            tgt = self.norm1(tgt)
        residual = tgt
        if self.normalize_before:
            tgt = self.norm2(tgt)
        if cache is None:
            tgt = self.cross_attn(tgt, memory, memory, memory_mask, None)
        else:
            tgt, static_cache = self.cross_attn(tgt, memory, memory, memory_mask, cache[1])
        tgt = residual + self.dropout2(tgt)
        if not self.normalize_before:
            tgt = self.norm2(tgt)
        residual = tgt
        if self.normalize_before:
            tgt = self.norm3(tgt)
        tgt = self.linear2(self.dropout(self.activation(self.linear1(tgt))))
        tgt = residual + self.dropout3(tgt)
        if not self.normalize_before:
            tgt = self.norm3(tgt)
        return tgt if cache is None else (tgt, (incremental_cache, static_cache))
```

```
# Decoder 端的多层堆叠
class TransformerDecoder(Layer):
    def __init__(self, decoder_layer, num_layers, norm = None):
        super(TransformerDecoder, self).__init__()
        self.layers = LayerList([(decoder_layer if i == 0 else type(decoder_layer)( **
decoder_layer._config)) for i in range(num_layers)])
        self.num_layers = num_layers
        self.norm = norm

    def forward(self, tgt, memory, tgt_mask = None, memory_mask = None, cache = None):
        output = tgt
        new_caches = []
        for i, mod in enumerate(self.layers):
            if cache is None:
                output = mod(output,
                             memory,
                             tgt_mask = tgt_mask,
                             memory_mask = memory_mask,
                             cache = None)
            else:
                output, new_cache = mod(output,
                                        memory,
                                        tgt_mask = tgt_mask, memory_mask = memory_mask,
                                        cache = cache[i])
                new_caches.append(new_cache)

        if self.norm is not None:
            output = self.norm(output)
        return output if cache is None else (output, new_caches)
```

然后我们就可以利用定义好的 TransformerEncoder 类和 TransformerDecoder 类构建基于 Transformer 的机器翻译模型。首先进行超参数的定义，用于后续模型的设计与训练。

```
embedding_size = 128
hidden_size = 512
num_encoder_lstm_layers = 1
en_vocab_size = len(list(en_vocab))
cn_vocab_size = len(list(cn_vocab))
epochs = 20
batch_size = 16
```

然后分别使用 TransformerEncoder 类和 TransformerDecoder 类定义 Encoder 端和 Decoder 端。

```
# Encoder 端定义
class Encoder(paddle.nn.Layer):
    def __init__(self,en_vocab_size, embedding_size,num_layers = 2,head_number = 2,middle_
units = 512):
        super(Encoder, self).__init__()
        self.emb = paddle.nn.Embedding(en_vocab_size, embedding_size,)
```

```
        encoder_layer = TransformerEncoderLayer(embedding_size, head_number, middle_units)
        self.encoder = TransformerEncoder(encoder_layer, num_layers)

    def forward(self, x):
        x = self.emb(x)
        en_out = self.encoder(x)
        return en_out

# Decoder 端定义
class Decoder(paddle.nn.Layer):
    def __init__(self,cn_vocab_size, embedding_size,num_layers = 2,head_number = 2,middle_
units = 512):
        super(Decoder, self).__init__()
        self.emb = paddle.nn.Embedding(cn_vocab_size, embedding_size)

        decoder_layer = TransformerDecoderLayer(embedding_size, head_number, middle_units)
        self.decoder = TransformerDecoder(decoder_layer, num_layers)

        self.outlinear = paddle.nn.Linear(embedding_size, cn_vocab_size)

    def forward(self, x, encoder_outputs):
        x = self.emb(x)

        de_out = self.decoder(x, encoder_outputs)
        output = self.outlinear(de_out)
        output = paddle.squeeze(output)
        return output
```

步骤 2：模型训练

模型训练依然包含：模型实例化、优化器定义、损失函数定义这几部分。

```
# 实例化编码器、解码器
encoder = Encoder(en_vocab_size, embedding_size)
decoder = Decoder(cn_vocab_size, embedding_size)
# 优化器：同时优化编码器与解码器的参数
opt = paddle.optimizer.Adam(learning_rate = 0.00001,
                            parameters = encoder.parameters() + decoder.parameters())
# 开始训练
for epoch in range(epochs):
    print("epoch:{}".format(epoch))

    # 打乱训练数据顺序
    perm = np.random.permutation(len(train_en_sents))
    train_en_sents_shuffled = train_en_sents[perm]
    train_cn_sents_shuffled = train_cn_sents[perm]
train_cn_label_sents_shuffled = train_cn_label_sents[perm]

# 批量数据迭代
```

```
    for iteration in range(train_en_sents_shuffled.shape[0] // batch_size):
        x_data = train_en_sents_shuffled[(batch_size * iteration): (batch_size * (iteration + 1))]
        sent = paddle.to_tensor(x_data)
        # 编码器处理英文句子
        en_repr = encoder(sent)
        # 解码器端原始输入
        x_cn_data = train_cn_sents_shuffled[(batch_size * iteration): (batch_size *
(iteration + 1))]
        # 解码器端输出的标准答案,用于计算损失
        x_cn_label_data = train_cn_label_sents_shuffled[( batch_size * iteration):(batch_size *
(iteration + 1))]

        loss = paddle.zeros([1])
        # 逐步解码,每步解码一个词
        for i in range( cn_length + 2):
            # 获取每步的输入以及输出的标准答案
            cn_word = paddle.to_tensor(x_cn_data[:,i:i + 1])
            cn_word_label = paddle.to_tensor(x_cn_label_data[:,i])
            # 解码器解码
            logits = decoder(cn_word, en_repr)
            # 计算解码损失:交叉熵损失,解码的词是否正确
            step_loss = F.cross_entropy(logits, cn_word_label)
            loss += step_loss
        # 计算平均损失
        loss = loss / (cn_length + 2)
        if(iteration % 50 == 0):
            print("iter {}, loss:{}".format(iteration, loss.numpy()))
        # 反向传播,梯度更新
        loss.backward()
        opt.step()
        opt.clear_grad()
```

输出结果部分内容如下：

```
epoch:0
iter 0, loss:[8.359558]
iter 50, loss:[6.6431913]
iter 100, loss:[5.613313]
iter 150, loss:[5.3523498]
iter 200, loss:[5.0697136]
iter 250, loss:[5.2516413]
iter 300, loss:[4.8820877]
iter 350, loss:[5.020339]
iter 400, loss:[4.6380825]
iter 450, loss:[4.937914]
iter 500, loss:[4.5123353]
```