第3章　文本匹配

文本匹配是自然语言处理中另一个非常重要的任务。文本匹配主要进行文本对之间的相似度、相关度计算，所谓相似度，指两个文本是否描述相同的语义，而相关度则指两个文本之间是否存在特定的关系，例如，是否可以从文本 A 推理出文本 B(蕴含关系)。文本匹配任务的形式主要是对文本对进行关系判断，任务的最终形式包含两种：一种是相似/相关度计算，即计算两个文本的相似/相关度打分；另一种是相似/相关分类，1 表示相似/相关，0 表示不相似/相关(偶尔有些任务可能包含中立的情况，此时任务为三分类)。

文本匹配技术应用范围十分广泛，最常见的即搜索引擎，当我们在搜索引擎输入一个问题，搜索引擎会为我们计算最相关的答案返回展示给我们。除了搜索外，文本匹配在问答、推荐、计算广告等领域应用也非常广泛。例如，智能问答技术，常见的有功能型机器人，能根据你的提问返回最佳的解决问题方案；推荐技术中，能根据你个人的特征画像，为你推荐最和你匹配的信息(阅读内容、新闻、视频信息流等)；而计算广告中，则为你推荐与你个人特征画像的商品的广告，以增加广告的转化量等。这些技术无论是文本匹配，还是人-信息匹配、人-货匹配；都离不开匹配技术，不同的匹配载体，涉及载体特征构建不同，但匹配算法都大同小异，本章将详细介绍文本匹配的相关技术。

在神经网络中，文本匹配主要包含 3 种形式：基于表示(Representation)的文本匹配[图 3-1(a)]、基于交互(Interaction)的文本匹配[图 3-1(b)]、基于预训练-微调框架的文本匹配技术(图 3-2)。基于表示的文本匹配技术先使用编码器将两个文本进行编码，即将文本对先分别表示为两个向量，然后在匹配层计算两个向量的相似/相关度打分，得到两个文本的相似/相关度。这种方法中，对文本对的编码可以使用同一个编码器，也可以使用不同的编码器，使用同一个编码器的结构，我们称为单塔模型；而使用两个编码器时，称为双塔结构。基于表示的文本匹配中，两个文本之间没有任何的交互，对于问答类的任务来说并不友好，因为在问答任务中，答案可能仅仅关注问题中的某些关键词语，而两个互不交互的文本表示，容易使编码器学不到什么词语重要，什么词语不重要，也就是说，最好的方法是“揣着问题找答案”。基于交互的文本匹配方法正是这样，在文本对表示的过程中，先进行交互，然后再匹配(或者表示、匹配)，这样能充分交换文本对的信息，从而使文本表示更加具有可解释性。

基于预训练-微调方法近年来在各个任务领域都取得了里程碑式的发展，以 BERT 方法为例，文本匹配任务的输入为文本对的拼接，然后定义一个 CLS 哨兵字符，其表示作为分类特征，进行二分类。预训练-微调方法简单、效果佳，已经是近年来各项 NLP 任务的基线。

接下来我们将分别介绍如何使用这三种方法进行文本匹配。

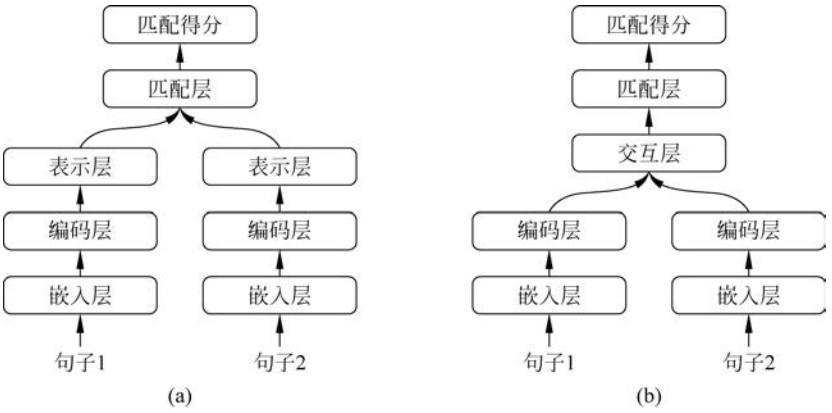

图 3-1　基于表示的文本匹配(a)与基于交互的文本匹配(b)

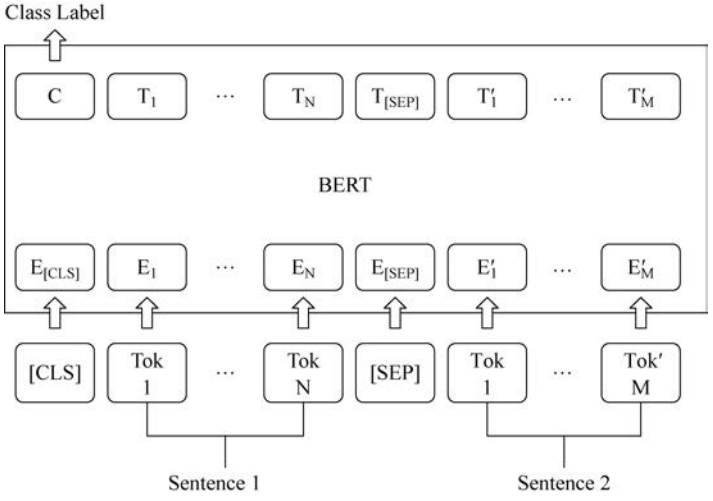

图 3-2　基于 BERT 的预训练-微调文本匹配

3.1　实践一：基于表示的文本匹配

基于表示的文本匹配

基于表示的文本匹配模型主要做法是将两段文本表示为语义向量，计算向量之间的相似度，重点在于如何更好地构建语义表示层。这种模式下的文本匹配有结构简单、解释性强、易于实现等诸多优点。下面我们将介绍基于 LSTM 的文本匹配经典网络结构。

基于 LSTM 的文本匹配模型主要是将两个不一样长的句子，分别经过 LSTM 编码成相同尺度的稠密向量，以此来比较两个句子的相似性。基于表示的文本匹配模型都是基于这种结构，即文本向量表示层＋相似度计算层。

步骤 1：数据加载

本实践采用的数据集是千言数据集。千言数据集目前包含 3 个文本相似度的数据集：paws-x-zh、lcqmc、bq_corpus。这 3 个数据集的数据格式都是一样的，包含有标签的训练集

（train.tsv）、验证集（dev.tsv）和无标签的测试集（test.tsv），验证集 tsv 文件中的内容格式为“Text_A\tText_B\tLabel”这样的列表，测试集 tsv 文件中的内容格式为“Text_A\tText_B”这样的列表，需要完成的任务即判断 Text_A 和 Text_B 之间是否相似，数据的具体样例如下。

训练集/验证集：

句子一：还有具体的讨论，公众形象辩论和项目讨论。
句子二：还有公开讨论，特定档案讨论和项目讨论。
标签：0

测试集：

句子一：Tabaci 河是罗马尼亚 Leurda 河的支流。
句子二：Leurda 河是罗马尼亚境内 Tabaci 河的一条支流。

步骤 2：数据集构建

数据集构建需要将数据处理为标准格式，即模型输入的格式，get_data（task）函数以 task 为参数，task 取值为 bq_corpus、lcqmc、paws-x-zh 三者之一，标识使用不同的数据集进行实践。本实践使用基于词的文本切割方式，因此使用常用的 jieba 工具进行文本分词处理。

```
# 1. 加载数据集

import jieba
import json

def get_data(task):
    pth = "data/{}/{}.tsv"
    train = []
    dev = []
    train_lines = open(pth.format(task,"train"),'r').readlines()
    train = [line.strip().replace(" ","").split("\t") for line in train_lines]
    train = train[:10]
    train = [[" ".join(jieba.lcut(line[0])),"".join(jieba.lcut(line[0])),line[2]] for line
in train if len(line) == 3]

    dev_lines = open(pth.format(task,"dev"),'r').readlines()
    dev = [line.strip().split("\t") for line in dev_lines]
    dev = dev[:10]
    dev = [[" ".join(jieba.lcut(line[0])),"".join(jieba.lcut(line[0])),line[2]] for line
in dev if len(line) == 3]

    test_lines = open(pth.format(task,"test"),'r').readlines()
    test = [line.strip().split("\t") for line in test_lines]
    test = test[:10]
    test = [[" ".join(jieba.lcut(line[0])),"".join(jieba.lcut(line[0]))] for line in test
if len(line) == 2]
```

```
    print("train:",len(train))
    print("dev:",len(dev))
    print("test:",len(test))
    return train,dev,test
```

文本分词完毕后，创建字典，添加< unk >来表示未知字符、< pad >来表示填充的文本，并保存到本地。

```
# 2. 创建字典
def create_dict(datas,dict_path):
    dict_set = []
    for data in datas:
        dict_set += data.split(" ")

    dict_set = set(dict_set)

    dict_list = []
    i = 0
for s in dict_set:
     # 去掉一些影响 json 解码的关键字符(非语义字符)
        if s == "{" or s == "}" or s == "'" or s == "\"" or s == ":" or "/" in s or "\\" in s:
            # print("--",s)
            continue
        dict_list.append([s, i])
        i += 1
    # 添加未知字符
    dict_txt = dict(dict_list)
    end_dict = {"< unk >": i}
    dict_txt.update(end_dict)
    end_dict = {"< pad >": i + 1}
    dict_txt.update(end_dict)
    # 把这些字典保存到本地中
    with open(dict_path, 'w', encoding = 'utf - 8') as f:
        f.write(json.dumps(dict_txt).replace("'",'"'))
```

利用上文生成的字典到 ID 的映射，将数据集中的文本词语转化为 ID 序列。

```
# 3. 字符序列转 ID 序列
def words_to_ids_padding(datas,dict_path,max_length = 64):
    js = open(dict_path,'r',encoding = 'utf - 8').read().strip().replace("'","\"")
    print(js)
    vocab_dic = json.loads(js)
    res = []
    for data in datas:
        sent1 = data[0].split(" ")
        sent2 = data[1].split(" ")
        label = data[2] if len(data) == 3 else "0"
        id1 = [vocab_dic[w] if w in vocab_dic else vocab_dic['< unk >'] for w in sent1 ]
        id2 = [vocab_dic[w] if w in vocab_dic else vocab_dic['< unk >'] for w in sent2 ]
```

```
            id1 = id1[:max_length] + [vocab_dic['<pad>']] * (max_length - len(id1))
            id2 = id2[:max_length] + [vocab_dic['<pad>']] * (max_length - len(id2))
            res.append([id1,id2,int(label)])
    return res
```

调用上述各功能函数，处理原始数据集、生成词表文件。注意，此处使用训练集与验证集数据进行词表构建，也可以只使用训练数据进行词表构建。

```
create_dict(datas,vocab_pth)

train_ds = words_to_ids_padding(train,vocab_pth,50)
dev_ds = words_to_ids_padding(dev,vocab_pth,50)
test_ds = words_to_ids_padding(test,vocab_pth,50)
vocab_pth = "data/vocab.txt"
train,dev,test = get_data("bq_corpus")

datas = [c[0] + " " + c[1] for c in train + dev]
```

步骤 3：构建标准数据集类

接下来需要将数据集格式化，即将数据处理为适应模型训练的格式。主要包含两个步骤：第一，使用 Dataset 对数据集进行封装，以进行批量数据生成、样本乱序等操作；第二，将 Dataset 数据进行二次封装，生成训练集、验证集、测试集的数据迭代器，其中每个样本包含 3 个元素（sent1、sent2、label）。

```
class PairDataset(paddle.io.Dataset):
    self.sent1 = []
    self.sent2 = []
    self.label = []

    def __init__(self, data_list):
       for line in data_list:
            self.sent1.append(line[0])
            self.sent2.append(line[1])
            self.label.append(line[2] if len(line) == 3 else 0)

    def __getitem__(self, index):
        s1,s2, lab = self.sent1[index],self.sent2[index],self.label[index]
        return s1,s2,lab

    def __len__(self):
        return len(self.sent1)

# 数据集 minibatch 批大小
batch_size = 32
train_dataset = PairDataset(train_ds)
dev_dataset = PairDataset(dev_ds)
test_dataset = PairDataset(test_ds)
```

```
# 数据集迭代器,用于批量数据生成
train_loader = paddle.io.DataLoader(train_dataset,
                        places = paddle.CPUPlace(),shuffle = True,
                        batch_size = batch_size, drop_last = True)
dev_loader = paddle.io.DataLoader(dev_dataset,
                        places = paddle.CPUPlace(),shuffle = True,
                        batch_size = batch_size, drop_last = True)
test_loader = paddle.io.DataLoader(test_dataset,
                        places = paddle.CPUPlace(),shuffle = False,
                        batch_size = batch_size,)
```

步骤 4：模型构建

LSTM 模型的构建和 2.1 节中类似，主要不同就是我们需要分别对两个文本进行 LSTM 编码，并在最后计算两个文本语义向量的余弦相似度。此处注意，我们使用了一个单塔结构，即对文本 1、文本 2 使用相同的 BiLSTM 编码器进行编码，若使用双塔结构，此处可额外再定义一个 BiLSTM 编码器，然后分别将文本 1 与文本 2 输入到不同的编码器中进行编码即可。计算完两个文本的语义表示之后，本实践使用 nn.CosineSimilarity 来计算两个文本语义向量的余弦相似度。余弦相似度计算两个向量在向量空间的夹角，默认夹角越小，两向量在语义空间的距离越相近，即两向量越相似，最后对余弦相似度进行 sigmoid 变换，将相似度转化为 0～1 的实数。

```
class LstmModel(paddle.nn.Layer):
    def __init__(self,vocab_dim,fc_dim):
        super(paddle.nn.LSTM, self).__init__()
        self.dict_dim = vocab_dim
        self.emb_dim = fc_dim
        self.hid_dim = fc_dim
        # 词向量编码
        self.embedding = Embedding(self.dict_dim, self.emb_dim)
        # 双向 LSTM 编码器
        self.lstm = paddle.nn.LSTM(self.emb_dim, self.hid_dim,
                            direction = "bidirectional")
        self.fc = Linear(self.hid_dim * 2, 1)
        self.cos_sim_func = nn.CosineSimilarity()

    def forward(self, input1,input2,label = None):
        emb1,emb2 = self.embedding(input1),self.embedding(input2)
        _, (h1, _) = self.lstm(emb1) # [32, 50, 256]
        _, (h2, _) = self.lstm(emb2)
        # [32, 50, 256] [2, 32, 128] [2, 32, 128]
        f1,f2 = self.fc1(h1.transpose([1,0,2]).reshape([ - 1,self.hid_dim * 2])),self.fc2
(h2.transpose([1,0,2]).reshape([ - 1,self.hid_dim * 2]))
        f1,f2 = paddle.reshape(f1,[f1.shape[0], - 1]),paddle.reshape(f2,[f2.shape[0], - 1])
        sim_vec = paddle.nn.functional.sigmoid(self.cos_sim_func(f1,f2)_
        if lebel is None:
```

```
        return sim_vec
    loss = paddle.nn.functional.mse_loss(sim_vec,label)
    return loss,sim_vec
```

步骤 5：模型训练

模型训练过程与前文其他深度网络的实践类似，此处采用的优化器为 Adam，损失函数使用均方误差损失函数 paddle.nn.functional.mse_loss()，均方误差损失函数主要用于回归问题中。此处，我们对两个文本的语义相似度进行衡量，使得语义越相近时，相似度打分越接近 1；而语义互斥时，相似度打分趋向于 0。由于是一个打分趋近问题，因此此处使用 MSE 损失函数进行参数梯度计算，train()函数定义如下。

```
def train(model,epochs):
    model.train()
    opt = paddle.optimizer.Adam(learning_rate=0.002,
                                parameters=model.parameters())
    steps = 0
    Iters, total_loss, total_acc = [], [], []
    for epoch in range(epochs):
        for batch_id, data in enumerate(train_loader):
            steps += 1
            sent1,sent2,label = data[0], data[1], data[2]
            loss,sim = model(sent1,sent2,label)
            predict = [1 if c>0.5 else 0 for c in sim]
            acc = sum([predict[i]==label.numpy()[i] for i in range(len(predict))]) / len(predict)

            if batch_id % 50 == 0:
                Iters.append(steps)
                total_loss.append(loss.numpy()[0])
                total_acc.append(acc)
                print("epoch: {}, batch_id: {}, loss is: {}".format(epoch, batch_id, loss.numpy()))

            loss.backward()
            opt.step()
            opt.clear_grad()

        # evaluate model after one epoch
        model.eval()
        accuracies = []
        losses = []

        for batch_id, data in enumerate(dev_loader):
            sent1,sent2,label = data[0], data[1], data[2]
            loss,sim = model(sent1,sent2,label)
            predict = [1 if c>0.5 else 0 for c in sim]
            acc = sum([predict[i]==label.numpy()[i] for i in range(len(predict))]) /
              len(predict)
```

```
                accuracies.append(acc)
                losses.append(loss.numpy())
            avg_acc, avg_loss = np.mean(accuracies), np.mean(losses)
            print("[validation] accuracy: {}, loss: {}".format(avg_acc, avg_loss))
            model.train()
    paddle.save(model.state_dict(),"model_final.pdparams")
    draw_process("trainning loss","red",Iters,total_loss,"trainning loss")
    draw_process("trainning acc","green",Iters,total_acc,"trainning acc")

model = LstmModel(128,128)
train(model,2)
```

步骤 6：模型测试

模型在使用的时候，由于上文保存的仅仅是参数的取值，并未保存模型结果，因此要先初始化一个模型，然后加载上文保存的模型参数，最后将已训练好的参数赋值给初始化的模型。

```
model = LstmModel(128,128)
model_state_dict = paddle.load('model_final.pdparams')
model.set_state_dict(model_state_dict)
model.eval()
label_map = {1:"是", 0:"否"}

result = []
predictions = []
accuracies = []
losses = []

for batch_id, data in enumerate(test_loader):
    sent1 = data[0]
    sent2 = data[1]
    sim = model(sent)
    for idx,prob in enumerate(logits):
        # 映射分类 label
        labels = 1 if prob > 0.5 else 0
        predictions.append(labels)
        samples.append([sent1[idx].numpy(),sent2[idx].numpy())
```

至此，完成了基于表示的文本匹配算法，可以看出，基于表示的方法简单、直观，但是学习能力较差。下面探索基于交互的文本匹配算法。

3.2　实践二：基于交互的文本匹配


基于交互的文本匹配

图 3-1(b)展示了基于交互的文本匹配的基本流程，核心思想是交互，与基于表示的方法不同，本方法除了“自己编码自己”(LSTM)外，还使用“对方编码自己”。也就是说，文本 1

最终的表示，除了与 LSTM 编码器编码的文本 1 自己有关，还与 LSTM 编码器编码的文本 2 有关。将文本 1、文本 2 分别经过 LSTM 之后，分别得到了文本 1、文本 2 的词语的隐状态表示，即文本 1、文本 2 的隐状态矩阵，将两者的隐状态矩阵进行交互（一般为矩阵乘法、归一化），便可以得到文本 1、文本 2 中各词语之间的相关性矩阵，如文本 1 第 i 个词语与文本 2 第 j 个词语的相关性。然后文本 2 的每一个词语的表示进行加权求和，获得文本 1 中每个词语基于文本 2 中每个词语的表示，同理可得文本 2 中每个词语基于文本 1 中每个词语的表示，这样便达到了两个文本的交互目的。得到交互后的表示后，与原始的隐状态再次交互（相加、相乘），既保留了原始的词表示，又加入了对方文本的交互表示，从而使各文本中各词语的表示更加丰富。接下来可以再经过一个 LSTM 对交互后的文本表示进行再次编码，然后获得各文本的语义表示，两个文本的语义表示进行拼接等操作，构成分类向量，输入分类器进行二分类。

基于上述核心思想，我们介绍本次实践，完成基于交互的文本匹配。由于本次实践使用的数据集与 3.1 节相同，模型的数据输入也完全一致，因此关于数据预处理及封装，此处不再赘述，详见 3.1 节数据预处理部分。本节重点介绍如何构建基于交互的文本匹配模型。

步骤 1：模型构建

由于本次模型较复杂，因此我们进行模型的拆解介绍。首先我们介绍模型的初始化函数，在这里，我们需要定义数据前向传播过程中需要用到的子模块，包括 Embedding 模块，用于将文本序列 ID 转换为词向量矩阵格式，为通用的文本处理模块。此处定义了两个双向 LSTM 模块，self.lstm 主要用于文本原始词表示的编码，而 self.lstm_after_interaction 主要用于交互后的文本表示的编码，具体操作见后。随之为两个全连接层模块，第一个全连接层用于高维分类特征向量到低维特征向量的转化，第二个全连接层为分类器，用于分类。

```
import paddle.nn as nn
import paddle.nn.functional as F
class InteractionMatch(nn.Layer):
    def __init__(self, hidden_size, num_classes, vocab_size, embedding_dim):
        super().__init__()
        self.hidden_size = hidden_size
        self.num_classes = num_classes
        self.embedding = nn.Embedding(vocab_size, embedding_dim)
        self.embedding_dim = embedding_dim

        self.lstm = nn.LSTM(self.embedding_dim, self.hidden_size,
                            direction="bidirectional")
        self.lstm_after_interaction = nn.LSTM(self.hidden_size * 2,
                            self.hidden_size, direction="bidirectional")
        self.fc1 = nn.Linear(self.hidden_size * 4, self.hidden_size)
        self.fc2 = nn.Linear(self.hidden_size, 2)
```

定义好各模块，如何组装起来至关重要。在 forward 函数中，接收参数为文本 1、文本 2 的 ID 序列（premises、hypotheses），接下来拆解前向函数中数据的流动过程。

首先将文本 ID 序列进行词向量表示。

```
embedded_premises = self.embedding(premises)
embedded_hypotheses = self.embedding(hypotheses)
```

词向量矩阵经过第一个双向 LSTM 编码器，用于获取各词语的上下文表示，输出包含 3 项，例如：encoded_premises、h、c。其中，encoded_premises 为所有词语的隐状态表示，h 为最后一个词语的表示（若为双向 LSTM 时，h 为最后一个词与第一个词语的表示，维度为[direction，batch_size，hidden_size]，双向时 direction＝2，否则 direction＝1），c 表示中间层的细胞状态。

```
encoded_premises, (h, c) = self.lstm(embedded_premises)
encoded_hypotheses, (_, _) = self.lstm(embedded_hypotheses)
```

我们要用到的数据为每个词语的隐状态表示，因此上述 h、c 均无用，可用下画线占位返回即可。接下来开启交互计算，paddle.bmm()函数提供批量数据的矩阵乘法，即为每一个样本对，结算第一个句子与第二个句子各词语的相关性，然后沿计算维度进行 softmax 归一化，计算某个词基于另一个句子中所有词向量的表示（加权求和过程，此处一定要注意加权求和的方向）。

```
attention_matrix = paddle.bmm(encoded_premises,
                              encoded_hypotheses.transpose([0, 2, 1]))
# print("attention_matrix",attention_matrix.shape) # [32, 50, 50]
# 归一化
attention_matrix_s1 = F.softmax(attention_matrix.transpose([0, 2, 1]),
                                axis = 1)
attention_matrix_s2 = F.softmax(attention_matrix, axis = 1)
premises_seq_att_out = paddle.bmm(encoded_hypotheses.transpose([0,
                          2, 1]), attention_matrix_s1).transpose([0, 2, 1])
hypotheses_seq_att_out = paddle.bmm(encoded_premises.transpose([0,
                           2, 1]), attention_matrix_s2).transpose([0, 2, 1])
```

上述 premises_seq_att_out 表示 premises 中词语基于 hypotheses 中词语隐状态的表示，至此，每一个文本已经得到了两个表示，一个是基于自身上下文的表示，如 encoded_premises，另一个是基于对方文本的表示，如 premises_seq_att_out。然后将这两者进行简单的相加，便可获得既包含自身上下文，又包含 pair 文本的上下文表示。最后将得到的新的隐状态表示输入新一层 LSTM 中，加强上下文表示。

```
premises_att_out, (h1, c1) =
               self.lstm_after_interaction(premises_seq_att_out +
               encoded_premises )
hypotheses_att_out, (h2, c2) =
               self.lstm_after_interaction(hypotheses_seq_att_out +
               encoded_hypotheses)
```

此时，需要构造每个文本最终的语义表示，这里使用 h 作为语义输出，拼接前向与后向最后一个字符的表示作为最终的文本语义表示。

```
fea_sent1 = h1.transpose([1, 0, 2]).reshape([h1.shape[1], -1])
fea_sent2 = h2.transpose([1, 0, 2]).reshape([h2.shape[1], -1])
```

然后拼接文本1、文本2的语义向量，作为分类特征，一次性输入到全连接层、激活层、分类层中，获得分类结果并返回。

```
cls_fea = paddle.concat([
    fea_sent1,
    fea_sent2
], axis = -1)
fc1 = F.relu(self.fc1(cls_fea))
logits = self.fc2(fc1)
# print(logits.shape)
return logits
```

以上为拆解后的模型结构，完整模型结构如下。

```
class InteractionMatch(nn.Layer):
    def __init__(self, hidden_size, num_classes, vocab_size, embedding_dim):
        super().__init__()
        self.hidden_size = hidden_size
        self.num_classes = num_classes
        self.embedding = nn.Embedding(vocab_size, embedding_dim)
        self.embedding_dim = embedding_dim

        self.lstm = nn.LSTM(self.embedding_dim, self.hidden_size,
                            direction = "bidirectional")
        self.lstm_after_interaction = nn.LSTM(self.hidden_size * 2,
                            self.hidden_size, direction = "bidirectional")
        self.fc1 = nn.Linear(self.hidden_size * 4, self.hidden_size)
        self.fc2 = nn.Linear(self.hidden_size, 2)

    def forward(self, premises, hypotheses):
        embedded_premises = self.embedding(premises)
        embedded_hypotheses = self.embedding(hypotheses)
        encoded_premises, (_, _) = self.lstm(embedded_premises)
        encoded_hypotheses, (_, _) = self.lstm(embedded_hypotheses)
        attention_matrix = paddle.bmm(encoded_premises,
                                      encoded_hypotheses.transpose([0, 2, 1]))
        attention_matrix_s1 = F.softmax(attention_matrix.transpose([0, 2, 1]),
                                      axis = 1)
        attention_matrix_s2 = F.softmax(attention_matrix, axis = 1)
        premises_seq_att_out = paddle.bmm(encoded_hypotheses.transpose([0,
                    2, 1]), attention_matrix_s1).transpose([0, 2, 1])
        hypotheses_seq_att_out = paddle.bmm(encoded_premises.transpose([0,
                    2, 1]), attention_matrix_s2).transpose([0, 2, 1])
        premises_att_out, (h1, c1) =
                        self.lstm_after_interaction(premises_seq_att_out +
                        encoded_premises)
        hypotheses_att_out, (h2, c2) =
                        self.lstm_after_interaction(hypotheses_seq_att_out +
                        encoded_hypotheses)
        fea_sent1 = h1.transpose([1, 0, 2]).reshape([h1.shape[1], -1])
```

```
fea_sent2 = h2.transpose([1, 0, 2]).reshape([h2.shape[1], -1])

cls_fea = paddle.concat([fea_sent1, fea_sent2 ], axis = -1)
fc1 = F.relu(self.fc1(cls_fea))
logits = self.fc2(fc1)
return logits
```

模型训练、预测过程与 3.1 节完全一致，此处不再赘述。基于交互的方法本质上避免了两个文本之间完全孤立的状态。本书简单地演示了交互方法，同学们在实践的过程中可以发现，本模型结构还有很多可以改进的空间。例如，在计算相关性矩阵时，可以对 padding 的词语进行屏蔽，因为 padding 词语本质上不参与、不影响文本的语义计算，该方法就是经典的 ESIM 模型，感兴趣的同学可以深入了解、实践。

3.3　实践三：基于预训练-微调的文本匹配

基于预训练-微调的文本匹配

预训练-微调框架为各项 NLP 任务都提供了一个非常强大的基线标准，在 2.3 节中，我们实践了基于 BERT 微调的文本分类任务，本节，我们使用百度自研发的 ERNIE-Gram 预训练-微调框架进行文本匹配。

ERNIE-Gram 是一种多粒度预训练语义理解技术。作为自然语言处理的基本语义单元，更充分的语言粒度学习能帮助模型实现更强的语义理解能力。ERNIE-Gram 提出显式完备的 N-Gram 多粒度掩码语言模型，同步建模 N-Gram 内部和 N-Gram 之间的语义关系，实现同时学习细粒度和粗粒度语义信息，在各项任务中，取得了比 BERT、ERNIE 更加优秀的效果。本节将详细介绍如何使用 ERNIE-Gram 进行文本匹配。

步骤 1：数据预处理

在 3.1、3.2 节中，手动为大家实现了数据集的预处理、ID 序列化、Dataset/DataLoader 封装等过程，旨在引导同学们更细节地了解 NLP 任务的基本数据处理过程。本节还是使用相同的数据集进行实践，但是直接从 paddlenlp.datasets 封装好的数据中读取即可。

```
from paddlenlp.datasets import load_dataset
train_ds, dev_ds = load_dataset("lcqmc", splits = ["train", "dev"])
for idx, example in enumerate(train_ds[:2]):
print(example)
```

输出如下：

```
{'query': '喜欢打篮球的男生喜欢什么样的女生', 'title': '爱打篮球的男生喜欢什么样的女生',
'label': 1}
{'query': '我手机丢了,我想换个手机', 'title': '我想买个新手机,求推荐', 'label': 1}
```

预训练-微调框架使用的数据输入格式通常包含 3 种：文本 ID 序列、文本段落标识和位置。通常位置信息会在模型中默认添加，因此，我们需要构建的输入格式必须包含前两者，即文本 ID 序列与文本段落标识，后者用来标识当前 token 所属段落（文本对任务）或者区分是否

是 padding 符号(单文本任务)。此处,我们直接使用对应预训练模型的 ErnieGramTokenizer 即可。

```
tokenizer = paddlenlp.transformers.ErnieGramTokenizer.from_pretrained('ernie-
                                                        gram-zh')
```

定义好分词器之后,定义数据集格式化处理函数。首先使用 toknizer 进行文本拼接、切割,然后返回我们需要的两项:序列 ID(input_ids)和文本段落标识(token_type_ids)。

```
def convert_example(example, tokenizer, max_seq_length=512, is_test=False):
    query, title = example["query"], example["title"]
    encoded_inputs = tokenizer(
        text=query, text_pair=title, max_seq_len=max_seq_length)
    input_ids = encoded_inputs["input_ids"]
    token_type_ids = encoded_inputs["token_type_ids"]
    if not is_test:
        label = np.array([example["label"]], dtype="int64")
        return input_ids, token_type_ids, label
    else:
        return input_ids, token_type_ids

input_ids, token_type_ids, label = convert_example(train_ds[0], tokenizer)
```

单个样本格式处理定义完毕后,还需要实现 minibatch 格式数据生成,本质上,与 3.1、3.2 节的 Dataset、DataLoader 是一致的,只不过使用了更高阶的函数。首先,定义一个转化函数 trans_func(),即 convert_example 固定传入 tokenizer,最大句子长度设置为 512。

```
trans_func = partial(
    convert_example,
    tokenizer=tokenizer,
    max_seq_length=512)
```

然后为每个样本执行如下一组操作:Pad-Pad-Stack。其中,对文本 ID 序列、token 类型序列 padding 到指定长度,然后将同一 batch 的标签进行堆叠 Stack,构建一个统一的 tensor。

```
batchify_fn = lambda samples, fn=Tuple(
    Pad(axis=0, pad_val=tokenizer.pad_token_id),          # input_ids
    Pad(axis=0, pad_val=tokenizer.pad_token_type_id),     # token_type_ids
    Stack(dtype="int64")                                  # label
): [data for data in fn(samples)]
```

定义分布式 Sampler,自动对训练数据进行切分,支持多卡并行训练。然后基于上述各处理函数,定义数据迭代器 DataLoader,完成模型输入的标准数据格式封装。

```
# 定义 train_data_loader
batch_sampler = paddle.io.DistributedBatchSampler(train_ds, batch_size=32,
shuffle=True)
train_data_loader = paddle.io.DataLoader(
        dataset=train_ds.map(trans_func),
```

```
        batch_sampler = batch_sampler,
        collate_fn = batchify_fn,
        return_list = True)
# 定义 dev_data_loader
batch_sampler = paddle.io.BatchSampler(dev_ds, batch_size = 32, shuffle = False)
dev_data_loader = paddle.io.DataLoader(
        dataset = dev_ds.map(trans_func),
        batch_sampler = batch_sampler,
        collate_fn = batchify_fn,
        return_list = True)
```

步骤 2：模型加载

直接调用 paddlenlp.transformers.ErnieGramModel 的 from_pretrained()函数，便可下载相应的模型参数，简单便捷。

```
pretrained_model
                = paddlenlp.transformers.ErnieGramModel.from_pretrained('er
                nie - gram - zh')
```

对该预训练模型进行简单封装、模块化，此处包含两部分：第一，预训练模型的输出，即文本对的 CLS 向量，作为分类的特征向量；第二，全连接分类器层，输入维度为预训练模型的 hidden_size，输出维度为分类个数，此处为 2。

```
class PointwiseMatching(nn.Layer):

    def __init__(self, pretrained_model, dropout = None):
        super().__init__()
        self.ptm = pretrained_model
        # dropout:随机失活一部分单元，避免过拟合
        self.dropout = nn.Dropout(dropout if dropout is not None else 0.1)
        self.classifier = nn.Linear(self.ptm.config["hidden_size"], 2)

    def forward(self, input_ids,
                token_type_ids = None,
                position_ids = None,
                attention_mask = None):

        _, cls_embedding = self.ptm(input_ids, token_type_ids, position_ids,
                                    attention_mask)
        cls_embedding = self.dropout(cls_embedding)
        logits = self.classifier(cls_embedding)
        probs = F.softmax(logits)
        return probs
```

步骤 3：模型训练

模型训练依旧遵循模型初始化、优化器/损失函数定义、评价指标选择等过程，此处，我们使用 AdamW 优化器，损失函数为交叉熵损失函数 CrossEntropyLoss()。优化器对学习

率先进性 warmup，然后在指定步数之后进行学习率先行衰减，避免错过损失函数的最优值，或者在极小值附近振荡而导致无法收敛，采用正确率 Accuracy 作为评价指标。

```
model = PointwiseMatching(pretrained_model)
epochs = 3
num_training_steps = len(train_data_loader) * epochs

# 定义 learning_rate_scheduler，负责在训练过程中对 lr 进行调度
lr_scheduler = LinearDecayWithWarmup(5E-5, num_training_steps, 0.0)

decay_params = [
    p.name for n, p in model.named_parameters()
    if not any(nd in n for nd in ["bias", "norm"])
]

# 定义 Optimizer
optimizer = paddle.optimizer.AdamW(
    learning_rate=lr_scheduler,
    parameters=model.parameters(),
    weight_decay=0.0,
    apply_decay_param_fun=lambda x: x in decay_params)

# 采用交叉熵 损失函数
criterion = paddle.nn.loss.CrossEntropyLoss()

# 评估的时候采用准确率指标
metric = paddle.metric.Accuracy()
```

模型训练过程如下，整体遵循数据前向传播，损失函数计算梯度进行反向传播 loss.backward，间隔迭代次数验证模型微调效果，最后保存模型参数。

```
global_step = 0
tic_train = time.time()

for epoch in range(1, epochs + 1):
    for step, batch in enumerate(train_data_loader, start=1):
        input_ids, token_type_ids, labels = batch
        probs = model(input_ids=input_ids, token_type_ids=token_type_ids)
        loss = criterion(probs, labels)
        correct = metric.compute(probs, labels)
        metric.update(correct)
        acc = metric.accumulate()
        global_step += 1
        if global_step % 100 == 0:
            print(
                "global step %d, epoch: %d, batch: %d, loss: %.5f, accu: %.5f,
                speed: %.2f step/s"
                % (global_step, epoch, step, loss, acc,
                    10 / (time.time() - tic_train)))
            tic_train = time.time()
```

```
            loss.backward()
            optimizer.step()
            lr_scheduler.step()
            optimizer.clear_grad()

            # 每间隔 100 step 在验证集和测试集上进行评估
            if global_step % 100 == 0:
                evaluate(model, criterion, metric, dev_data_loader, "dev")

# 训练结束后,存储模型参数
save_dir = os.path.join("checkpoint", "model_%d" % global_step)
os.makedirs(save_dir)

save_param_path = os.path.join(save_dir, 'model_state.pdparams')
paddle.save(model.state_dict(), save_param_path)
tokenizer.save_pretrained(save_dir)
```

步骤 4：模型预测

模型预测与模型训练过程基本一致，只不过输入数据不含标签，因此需要对推理数据做额外处理，主要体现为去掉标签。

```
# 推理数据的转换函数
# predict 数据没有 label, 因此 convert_exmaple 的 is_test 参数设为 True
trans_func = partial(
    convert_example,
    tokenizer = tokenizer,
    max_seq_length = 512,
    is_test = True)

# 预测数据 batch 操作
# predict 数据只返回 input_ids 和 token_type_ids
batchify_fn = lambda samples, fn = Tuple(
    Pad(axis = 0, pad_val = tokenizer.pad_token_id),                    # input_ids
    Pad(axis = 0, pad_val = tokenizer.pad_token_type_id),               # segment_ids
): [data for data in fn(samples)]

# 加载预测数据
test_ds = load_dataset("lcqmc", splits = ["test"])
batch_sampler = paddle.io.BatchSampler(test_ds, batch_size = 32, shuffle = False)

# 生成预测数据 data_loader
predict_data_loader = paddle.io.DataLoader(
        dataset = test_ds.map(trans_func),
        batch_sampler = batch_sampler,
        collate_fn = batchify_fn,
        return_list = True)
```

定义预测函数，分批次处理推理数据，避免内存溢出。

```
def predict(model, data_loader):
    batch_probs = []
    # 预测阶段打开 eval 模式,模型中的 dropout 等操作会关掉
    model.eval()

    with paddle.no_grad():
        for batch_data in data_loader:
            input_ids, token_type_ids = batch_data
            input_ids = paddle.to_tensor(input_ids)
            token_type_ids = paddle.to_tensor(token_type_ids)
            # 获取每个样本的预测概率: [batch_size, 2] 的矩阵
            batch_prob = model(
                input_ids=input_ids, token_type_ids=token_type_ids).numpy()
            batch_probs.append(batch_prob)
        batch_probs = np.concatenate(batch_probs, axis=0)
        return batch_probs
```

初始化一个新的模型,加载保存好的参数,为新模型参数赋值,然后调用 predict 函数,并保存预测结果。

```
pretrained_model = paddlenlp.transformers.ErnieGramModel.from_pretrained('ernie-gram-zh')

model = PointwiseMatching(pretrained_model)
state_dict = paddle.load("./ernie_gram_zh_pointwise_matching_model/model_state.pdparams")

model.set_dict(state_dict)

# 执行预测函数
y_probs = predict(model, predict_data_loader)

# 根据预测概率获取预测 label
y_preds = np.argmax(y_probs, axis=1)

test_ds = load_dataset("lcqmc", splits=["test"])

with open("lcqmc.tsv", 'w', encoding="utf-8") as f:
    f.write("index\tprediction\n")
    for idx, y_pred in enumerate(y_preds):
        f.write("{}\t{}\n".format(idx, y_pred))
        text_pair = test_ds[idx]
        text_pair["label"] = y_pred
        print(text_pair)
```