6.4 实践：基于双向LSTM模型完成文本分类任务

电影评论可以蕴含丰富的情感：比如喜欢、讨厌、等等．情感分析（Sentiment Analysis）是为一个文本分类问题，即使用判定给定的一段文本信息表达的情感属于积极情绪，还是消极情绪．

本实践使用 IMDB 电影评论数据集，使用双向 LSTM 对电影评论进行情感分析．

6.4.1 数据处理

IMDB电影评论数据集是一份关于电影评论的经典二分类数据集．IMDB 按照评分的高低筛选出了积极评论和消极评论，如果评分 ≥ 7，则认为是积极评论；如果评分 ≤ 4，则认为是消极评论．数据集包含训练集和测试集数据，数量各为 25000 条，每条数据都是一段用户关于某个电影的真实评价，以及观众对这个电影的情感倾向，其目录结构如下所示

```
├── train/
    ├── neg                 # 消极数据
    ├── pos                 # 积极数据
    ├── unsup               # 无标签数据
├── test/
    ├── neg                 # 消极数据
    ├── pos                 # 积极数据
```

在test/neg目录中任选一条电影评论数据，内容如下：

> "Cover Girl" is a lacklustre WWII musical with absolutely nothing memorable about it, save for its signature song, "Long Ago and Far Away."

LSTM 模型不能直接处理文本数据，需要先将文本中单词转为向量表示，称为词向量（Word Embedding）．为了提高转换效率，通常会事先把文本的每个单词转换为数字 ID，再使用第节中介绍的方法进行向量转换．因此，需要准备一个词典（Vocabulary），将文本中的每个单词转换为它在词典中的序号 ID．同时还要设置一个特殊的词 [UNK]，表示未知词．在处理文本时，如果碰到不在词表的词，一律按 [UNK] 处理．

6.4.1.1 数据加载

原始训练集和测试集数据分别25000条，本节将原始的测试集平均分为两份，分别作为验证集和测试集，存放于`./dataset`目录下。使用如下代码便可以将数据加载至内存：

In [1]:
```
import os
# 加载数据集
def load_imdb_data(path):
    assert os.path.exists(path)
    trainset, devset, testset = [], [], []
    with open(os.path.join(path, "train.txt"), "r") as fr:
        for line in fr:
            sentence_label, sentence = line.strip().lower().split("\t", maxsplit=1)
            trainset.append((sentence, sentence_label))

    with open(os.path.join(path, "dev.txt"), "r") as fr:
        for line in fr:
            sentence_label, sentence = line.strip().lower().split("\t", maxsplit=1)
            devset.append((sentence, sentence_label))

    with open(os.path.join(path, "test.txt"), "r") as fr:
        for line in fr:
            sentence_label, sentence = line.strip().lower().split("\t", maxsplit=1)
            testset.append((sentence, sentence_label))

    return trainset, devset, testset

# 加载IMDB数据集
train_data, dev_data, test_data = load_imdb_data("./dataset/")
# 打印一下加载后的数据样式
print(train_data[4])
```

```
("the premise of an african-american female scrooge in the modern, struggling city was inspired, but nothing else in this film is. here, ms. scrooge is a miserly banker who takes advantage of the employees and customers in the largely poor and black neighborhood it inhabits. there is no doubt about the good intentions of the people involved. part of the problem is that story's roots don't translate well into the urban setting of this film, and the script fails to make the update work. also, the constant message about sharing and giving is repeated so endlessly, the audience becomes tired of it well before the movie reaches its familiar end. this is a message film that doesn't know when to quit. in the title role, the talented cicely tyson gives an overly uptight performance, and at times lines are difficult to understand. the charles dickens novel has been adapted so many times, it's a struggle to adapt it in a way that makes it fresh and relevant, in spite of its very relevant message.", '0')
```

从输出结果看，加载后的每条样本包含两部分内容：文本串和标签。

6.4.1.2 构造Dataset类

首先，我们构造IMDBDataset类用于数据管理，它继承自paddle.io.DataSet类。

由于这里的输入是文本序列，需要先将其中的每个词转换为该词在词表中的序号 ID，然后根据词表ID查询这些词对应的词向量，该过程同第同6.1节中将数字向量化的操作，在获得词向量后会将其输入至模型进行后续计算。可以使用IMDBDataset类中的words_to_id方法实现这个功能。 具体

而言，利用词表word2id_dict将序列中的每个词映射为对应的数字编号，便于进一步转为为词向量。当序列中的词没有包含在词表时，默认会将该词用[UNK]代替。words_to_id方法利用一个如图6.14所示的哈希表来进行转换。

单词	ID
[UNK]	0
[PAD]	1
the	2
a	3
and	4

图6.14 word2id词表示例

代码实现如下：

In [2]:
```
import paddle
import paddle.nn as nn
from paddle.io import Dataset
from utils.data import load_vocab

class IMDBDataset(Dataset):
    def __init__(self, examples, word2id_dict):
        super(IMDBDataset, self).__init__()
        # 词典，用于将单词转为字典索引的数字
        self.word2id_dict =  word2id_dict
        # 加载后的数据集
        self.examples = self.words_to_id(examples)

    def words_to_id(self, examples):
        tmp_examples = []
        for idx, example in enumerate(examples):
            seq, label = example
            # 将单词映射为字典索引的ID，  对于词典中没有的单词用[UNK]对应的ID进行替代
            seq = [self.word2id_dict.get(word, self.word2id_dict['[UNK]']) for word in seq.split(" ")]
            label = int(label)
            tmp_examples.append([seq, label])
        return tmp_examples

    def __getitem__(self, idx):
        seq, label = self.examples[idx]
        return seq, label

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

# 加载词表
word2id_dict= load_vocab("./dataset/vocab.txt")

# 实例化Dataset
train_set = IMDBDataset(train_data, word2id_dict)
dev_set = IMDBDataset(dev_data, word2id_dict)
test_set = IMDBDataset(test_data, word2id_dict)

print('训练集样本数：', len(train_set))
print('样本示例：', train_set[4])
```

```
训练集样本数： 25000
样本示例： ([2, 976, 5, 32, 6860, 618, 7673, 8, 2, 13073, 2525, 724, 14, 22837, 18, 164, 416, 8, 10, 24, 701, 611, 1743, 7673, 7,
3, 56391, 21652, 36, 271, 3495, 5, 2, 11373, 4, 13244, 8, 2, 2157, 350, 4, 328, 4118, 12, 48810, 52, 7, 60, 860, 43, 2, 56, 4393,
5, 2, 89, 4152, 182, 5, 2, 461, 7, 11, 7321, 7730, 86, 7931, 107, 72, 2, 2830, 1165, 5, 10, 151, 4, 2, 272, 1003, 6, 91, 2, 10491,
912, 826, 2, 1750, 889, 43, 6723, 4, 647, 7, 2535, 38, 39222, 2, 357, 398, 1505, 5, 12, 107, 179, 2, 20, 4279, 83, 1163, 692, 10,
7, 3, 889, 24, 11, 141, 118, 50, 6, 28642, 8, 2, 490, 1469, 2, 1039, 98975, 24541, 344, 32, 2074, 11852, 1683, 4, 29, 286, 478, 22,
823, 6, 5222, 2, 1490, 6893, 883, 41, 71, 3254, 38, 100, 1021, 44, 3, 1700, 6, 8768, 12, 8, 3, 108, 11, 146, 12, 1761, 4, 92295, 8,
2641, 5, 83, 49, 3866, 5352], 0)
```

6.4.1.3 封装DataLoader

在构建 Dataset 类之后，我们构造对应的 DataLoader，用于批次数据的迭代．和前几章的 DataLoader 不同，这里的 DataLoader 需要引入下面两个功能：

1. 长度限制：需要将序列的长度控制在一定的范围内，避免部分数据过长影响整体训练效果
2. 长度补齐：神经网络模型通常需要同一批处理的数据的序列长度是相同的，然而在分批时通常会将不同长度序列放在同一批，因此需要对序列进行补齐处理．

对于长度限制，我们使用max_seq_len参数对于过长的文本进行截断． 对于长度补齐，我们先统计该批数据中序列的最大长度，并将短的序列填充一些没有特殊意义的占位符 [PAD]，将长度补齐到该批次的最大长度，这样便能使得同一批次的数据变得规整．比如给定两个句子：

- 句子1: This movie was craptacular.
- 句子2: I got stuck in traffic on the way to the theater.

将上面的两个句子补齐，变为：

- 句子1: This movie was craptacular [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD]
- 句子2: I got stuck in traffic on the way to the theater

具体来讲，本节定义了一个collate_fn函数来做数据的截断和填充. 该函数可以作为回调函数传入 DataLoader，DataLoader 在返回一批数据之前，调用该函数去处理数据，并返回处理后的序列数据和对应标签。

另外，使用[PAD]占位符对短序列填充后，再进行文本分类任务时，默认无须使用[PAD]位置，因此需要使用变量seq_lens来表示序列中非[PAD]位置的真实长度。seq_lens可以在collate_fn函数处理批次数据时进行获取并返回。需要注意的是，由于RunnerV3类默认按照输入数据和标签两类信息获取数据，因此需要将序列数据和序列长度组成元组作为输入数据进行返回，以方便RunnerV3解析数据。

代码实现如下：

In [3]:
```
from functools import partial

def collate_fn(batch_data, pad_val=0, max_seq_len=256):
    seqs, seq_lens, labels = [], [], []
    max_len = 0
    for example in batch_data:
        seq, label = example
        # 对数据序列进行截断
        seq = seq[:max_seq_len]
        # 对数据截断并保存于seqs中
        seqs.append(seq)
        seq_lens.append(len(seq))
        labels.append(label)
        # 保存序列最大长度
        max_len = max(max_len, len(seq))
    # 对数据序列进行填充至最大长度
    for i in range(len(seqs)):
        seqs[i] = seqs[i] + [pad_val] * (max_len - len(seqs[i]))

    return (paddle.to_tensor(seqs), paddle.to_tensor(seq_lens)), paddle.to_tensor(labels)
```

下面我们自定义一批数据来测试一下collate_fn函数的功能，这里假定一下max_seq_len为5，然后定义序列长度分别为6和3的两条数据，传入collate_fn函数中。

In [4]:
```
max_seq_len = 5
batch_data = [[[1, 2, 3, 4, 5, 6], 1], [[2,4,6], 0]]
(seqs, seq_lens), labels = collate_fn(batch_data, pad_val=word2id_dict["[PAD]"], max_seq_len=max_seq_len)
print("seqs: ", seqs)
print("seq_lens: ", seq_lens)
print("labels: ", labels)
```

```
seqs:  Tensor(shape=[2, 5], dtype=int64, place=CPUPlace, stop_gradient=True,
       [[1, 2, 3, 4, 5],
        [2, 4, 6, 0, 0]])
seq_lens:  Tensor(shape=[2], dtype=int64, place=CPUPlace, stop_gradient=True,
       [5, 3])
labels:  Tensor(shape=[2], dtype=int64, place=CPUPlace, stop_gradient=True,
       [1, 0])
```

可以看到，原始序列中长度为6的序列被截断为5，同时原始序列中长度为3的序列被填充到5，同时返回了非`[PAD]`的序列长度。

接下来，我们将collate_fn作为回调函数传入DataLoader中， 其在返回一批数据时，可以通过collate_fn函数处理该批次的数据。 这里需要注意的是，这里通过partial函数对collate_fn函数中的关键词参数进行设置，并返回一个新的函数对象作为collate_fn。

在使用DataLoader按批次迭代数据时，最后一批的数据样本数量可能不够设定的batch_size，可以通过参数drop_last来判断是否丢弃最后一个batch的数据。

In [5]:
```
max_seq_len = 256
batch_size = 128
collate_fn = partial(collate_fn, pad_val=word2id_dict["[PAD]"], max_seq_len=max_seq_len)
train_loader = paddle.io.DataLoader(train_set, batch_size=batch_size, shuffle=True, drop_last=False, collate_fn=collate_fn)
dev_loader = paddle.io.DataLoader(dev_set, batch_size=batch_size, shuffle=False, drop_last=False, collate_fn=collate_fn)
test_loader = paddle.io.DataLoader(test_set, batch_size=batch_size, shuffle=False, drop_last=False, collate_fn=collate_fn)
```

6.4.2 模型构建

本实践的整个模型结构如图6.15所示.

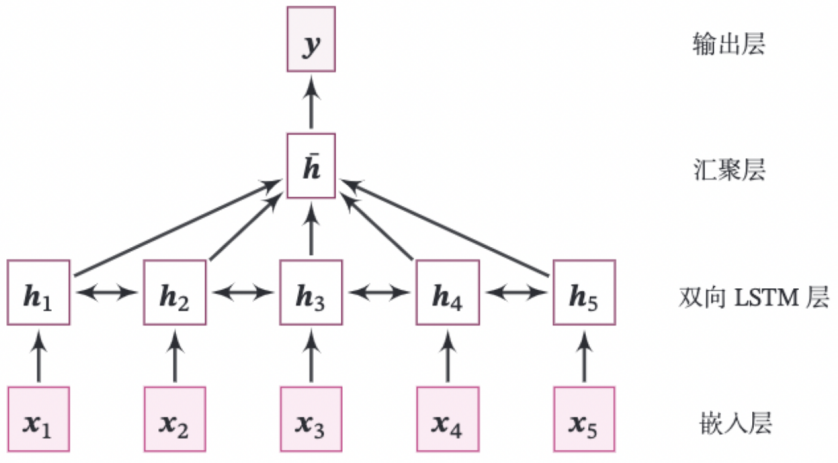

图6.15 基于双向LSTM的文本分类模型结构

由如下几部分组成：

（1）嵌入层：将输入的数字序列进行向量化，即将每个数字映射为向量。这里直接使用飞桨API：paddle.nn.Embedding来完成。

> class paddle.nn.Embedding(num_embeddings, embedding_dim, padding_idx=None, sparse=False, weight_attr=None, name=None)

该API有两个重要的参数：num_embeddings表示需要用到的Embedding的数量。embedding_dim表示嵌入向量的维度。 paddle.nn.Embedding会根据[num_embeddings, embedding_dim]自动构造一个二维嵌入矩阵。参数padding_idx是指用来补齐序列的占位符[PAD]对应的词表ID，那么在训练过程中遇到此ID时，其参数及对应的梯度将会以0进行填充。在实现中为了简单起见，我们通常会将[PAD]放在词表中的第一位，即对应的ID为0。

（2）双向LSTM层：接收向量序列，分别用前向和反向更新循环单元。这里我们直接使用飞桨API：paddle.nn.LSTM来完成。只需要在定义LSTM时设置参数direction为bidirectional，便可以直接使用双向LSTM。

> 思考: 在实现双向LSTM时，因为需要进行序列补齐，在计算反向LSTM时，占位符[PAD]是否会对LSTM参数梯度的更新有影响。如果有的话，如何消除影响？ 注：在调用paddle.nn.LSTM实现双向LSTM时，可以传入该批次数据的真实长度，paddle.nn.LSTM会根据真实序列长度处理数据，对占位符[PAD]进行掩蔽，[PAD]位置将返回零向量。

（3）聚合层：将双向LSTM层所有位置上的隐状态进行平均，作为整个句子的表示。

（4）输出层：输出层，输出分类的几率。这里可以直接调用paddle.nn.Linear来完成。

> **动手练习6.5**：改进第6.3.1.1节中的LSTM算子，使其可以支持一个批次中包含不同长度的序列样本。

上面模型中的嵌入层、双向LSTM层和线性层都可以直接调用飞桨API来实现，这里我们只需要实现汇聚层算子。需要注意的是，虽然飞桨内置LSTM在传入批次数据的真实长度后，会对[PAD]位置返回零向量，但考虑到汇聚层与处理序列数据的模型进行解耦，因此在本节汇聚层的实现中，会对[PAD]位置进行掩码。

汇聚层算子

汇聚层算子将双向LSTM层所有位置上的隐状态进行平均，作为整个句子的表示。这里我们实现了AveragePooling算子进行隐状态的汇聚，首先利用序列长度向量生成掩码（Mask）矩阵，用于对文本序列中[PAD]位置的向量进行掩蔽，然后将该序列的向量进行相加后取均值。代码实现如下：

将上面各个模块汇总到一起，代码实现如下：

In [6]:
```
class AveragePooling(nn.Layer):
    def __init__(self):
        super(AveragePooling, self).__init__()

    def forward(self, sequence_output, sequence_length):
        sequence_length = paddle.cast(sequence_length.unsqueeze(-1), dtype="float32")
        # 根据sequence_length生成mask矩阵，用于对Padding位置的信息进行mask
        max_len = sequence_output.shape[1]
        mask = paddle.arange(max_len) < sequence_length
        mask = paddle.cast(mask, dtype="float32").unsqueeze(-1)
        # 对序列中paddling部分进行mask
        sequence_output = paddle.multiply(sequence_output, mask)
        # 对序列中的向量取均值
        batch_mean_hidden = paddle.divide(paddle.sum(sequence_output, axis=1), sequence_length)
        return batch_mean_hidden
```

模型汇总

将上面的算子汇总，组合为最终的分类模型。代码实现如下：

In [7]:
```
class Model_BiLSTM_FC(nn.Layer):
    def __init__(self, num_embeddings, input_size, hidden_size, num_classes=2):
        super(Model_BiLSTM_FC, self).__init__()
        # 词典大小
        self.num_embeddings = num_embeddings
        # 单词向量的维度
        self.input_size = input_size
        # LSTM隐藏单元数量
        self.hidden_size = hidden_size
        # 情感分类类别数量
        self.num_classes = num_classes
        # 实例化嵌入层
        self.embedding_layer = nn.Embedding(num_embeddings, input_size, padding_idx=0)
```

```
        # 实例化LSTM层
        self.lstm_layer = nn.LSTM(input_size, hidden_size, direction="bidirectional")
        # 实例化聚合层
        self.average_layer = AveragePooling()
        # 实例化输出层
        self.output_layer = nn.Linear(hidden_size * 2, num_classes)

    def forward(self, inputs):
        # 对模型输入拆分为序列数据和mask
        input_ids, sequence_length = inputs
        # 获取词向量
        inputs_emb = self.embedding_layer(input_ids)
        # 使用lstm处理数据
        sequence_output, _ = self.lstm_layer(inputs_emb, sequence_length=sequence_length)
        # 使用聚合层聚合sequence_output
        batch_mean_hidden = self.average_layer(sequence_output, sequence_length)
        # 输出文本分类logits
        logits = self.output_layer(batch_mean_hidden)
        return logits
```

6.4.3 模型训练

本节将基于RunnerV3进行训练，首先指定模型训练的超参，然后设定模型、优化器、损失函数和评估指标，其中损失函数使用`paddle.nn.CrossEntropyLoss`，该损失函数内部会对预测结果使用`softmax`进行计算，数字预测模型输出层的输出`logits`不需要使用softmax进行归一化，定义完Runner的相关组件后，便可以进行模型训练。代码实现如下。

In [8]:
```
import time
import random
import numpy as np
from nndl import Accuracy, RunnerV3

np.random.seed(0)
random.seed(0)
paddle.seed(0)

# 指定训练轮次
num_epochs = 3
# 指定学习率
learning_rate = 0.001
# 指定embedding的数量为词表长度
num_embeddings = len(word2id_dict)
# embedding向量的维度
input_size = 256
# LSTM网络隐状态向量的维度
hidden_size = 256

# 实例化模型
model = Model_BiLSTM_FC(num_embeddings, input_size, hidden_size)
# 指定优化器
optimizer = paddle.optimizer.Adam(learning_rate=learning_rate, beta1=0.9, beta2=0.999, parameters= model.parameters())
# 指定损失函数
loss_fn = paddle.nn.CrossEntropyLoss()
# 指定评估指标
metric = Accuracy()
# 实例化Runner
runner = RunnerV3(model, optimizer, loss_fn, metric)
# 模型训练
start_time = time.time()
runner.train(train_loader, dev_loader, num_epochs=num_epochs, eval_steps=10, log_steps=10, save_path="./checkpoints/best.pdparams")
end_time = time.time()
print("time: ", (end_time-start_time))
```

```
/opt/conda/envs/python35-paddle120-env/lib/python3.7/site-packages/matplotlib/__init__.py:107: DeprecationWarning: Using or importi
ng the ABCs from 'collections' instead of from 'collections.abc' is deprecated, and in 3.8 it will stop working
  from collections import MutableMapping
/opt/conda/envs/python35-paddle120-env/lib/python3.7/site-packages/matplotlib/rcsetup.py:20: DeprecationWarning: Using or importing
the ABCs from 'collections' instead of from 'collections.abc' is deprecated, and in 3.8 it will stop working
  from collections import Iterable, Mapping
/opt/conda/envs/python35-paddle120-env/lib/python3.7/site-packages/matplotlib/colors.py:53: DeprecationWarning: Using or importing
the ABCs from 'collections' instead of from 'collections.abc' is deprecated, and in 3.8 it will stop working
  from collections import Sized
```

绘制训练过程中在训练集和验证集上的损失图像和在验证集上的准确率图像：

In [11]:
```
from nndl import plot_training_loss_acc

# 图像名字
fig_name = "./images/6.16.pdf"
# sample_step: 训练损失的采样step，即每隔多少个点选择1个点绘制
# loss_legend_loc: loss 图像的图例放置位置
# acc_legend_loc： acc 图像的图例放置位置
plot_training_loss_acc(runner, fig_name, fig_size=(16,6), sample_step=10, loss_legend_loc="lower left", acc_legend_loc="lower right
```

图6.16 展示了文本分类模型在训练过程中的损失曲线和在验证集上的准确率曲线，其中在损失图像中，实线表示训练集上的损失变化，虚线表示验证集上的损失变化. 可以看到，随着训练过程的进行，训练集的损失不断下降， 验证集上的损失在大概200步后开始上升，这是因为在训练过程中发生了过拟合，可以选择保存在训练过程中在验证集上效果最好的模型来解决这个问题. 从准确率曲线上可以看到，首先在验证集上的准确率大幅度上升，然后大概200步后准确率不再上升，并且由于过拟合的因素，在验证集上的准确率稍微降低。

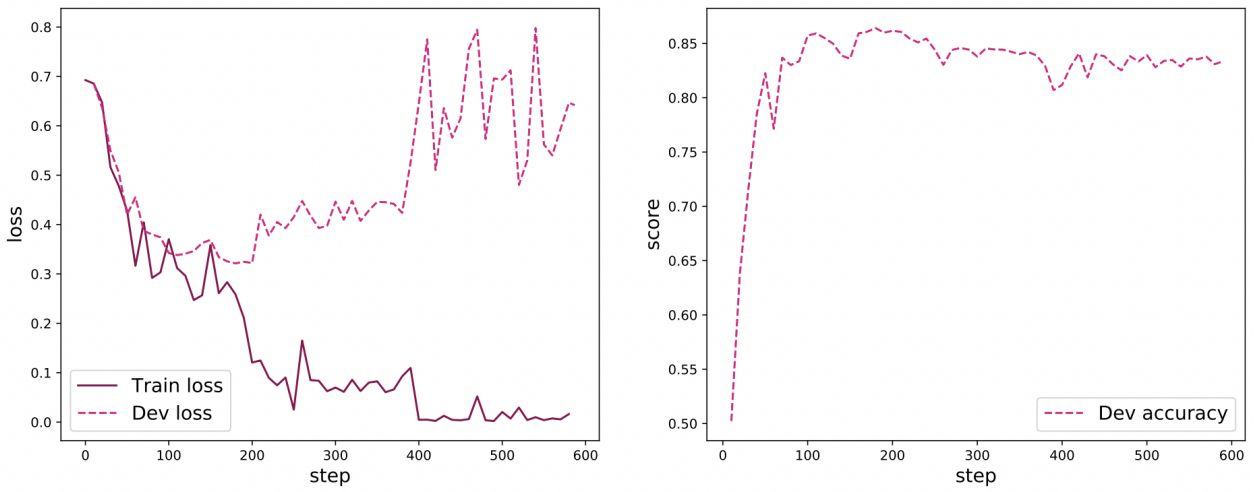

图6.16 文本分类模型训练损失变化图

6.4.4 模型评价

加载训练过程中效果最好的模型，然后使用测试集进行测试。

In [12]:
```
model_path = "./checkpoints/best.pdparams"
runner.load_model(model_path)
accuracy, _ =  runner.evaluate(test_loader)
print(f"Evaluate on test set, Accuracy: {accuracy:.5f}")
```

6.4.5 模型预测

给定任意的一句话，使用训练好的模型进行预测，判断这句话中所蕴含的情感极性。

In [15]:
```
id2label={0:"消极情绪", 1:"积极情绪"}
text = "this movie is so great. I watched it three times already"
# 处理单条文本
sentence = text.split(" ")
words = [word2id_dict[word] if word in word2id_dict else word2id_dict['[UNK]'] for word in sentence]
words = words[:max_seq_len]
sequence_length = paddle.to_tensor([len(words)], dtype="int64")
words = paddle.to_tensor(words, dtype="int64").unsqueeze(0)
# 使用模型进行预测
logits = runner.predict((words, sequence_length))
max_label_id = paddle.argmax(logits, axis=-1).numpy()[0]
pred_label = id2label[max_label_id]
print("Label: ", pred_label)
```

> 动手练习 6.6：LSTM在实际应用中可以叠加多层，请使用多层的LSTM进行情感分析任务，观察模型性能随模型深度如何变化，并思考改进方法。

6.5 拓展实验

6.6.1 使用Paddle内置的单向LSTM进行文本分类实验

首先，修改模型定义，将 `nn.LSTM` 中的 `direction` 设置为 `forward` 以使用单向LSTM模型，同时设置线性层的shape为 `[hidden_size, num_classes]` 。

In [16]:
```
class AveragePooling(nn.Layer):
    def __init__(self):
        super(AveragePooling, self).__init__()

    def forward(self, sequence_output, sequence_length):
        sequence_length = paddle.cast(sequence_length.unsqueeze(-1), dtype="float32")
        # 根据sequence_length生成mask矩阵，用于对Padding位置的信息进行mask
        max_len = sequence_output.shape[1]
        mask = paddle.arange(max_len) < sequence_length
        mask = paddle.cast(mask, dtype="float32").unsqueeze(-1)
        # 对序列中paddling部分进行mask
        sequence_output = paddle.multiply(sequence_output, mask)
        # 对序列中的向量取均值
        batch_mean_hidden = paddle.divide(paddle.sum(sequence_output, axis=1), sequence_length)
        return batch_mean_hidden

class Model_BiLSTM_FC(nn.Layer):
    def __init__(self, num_embeddings, input_size, hidden_size, num_classes=2):
        super(Model_BiLSTM_FC, self).__init__()
        # 词典大小
        self.num_embeddings = num_embeddings
        # 单词向量的维度
        self.input_size = input_size
        # LSTM隐藏单元数量
        self.hidden_size = hidden_size
        # 情感分类类别数量
        self.num_classes = num_classes
        # 实例化嵌入层
        self.embedding_layer = nn.Embedding(num_embeddings, input_size, padding_idx=0)
        # 实例化LSTM层
```

```
        self.lstm_layer = nn.LSTM(input_size, hidden_size, direction="forward")
        # 实例化聚合层
        self.average_layer = AveragePooling()
        # 实例化输出层
        self.output_layer = nn.Linear(hidden_size, num_classes)

    def forward(self, inputs):
        # 对模型输入拆分为序列数据和mask
        input_ids, sequence_length = inputs
        # 获取词向量
        inputs_emb = self.embedding_layer(input_ids)
        # 使用lstm处理数据
        sequence_output, _ = self.lstm_layer(inputs_emb, sequence_length=sequence_length)
        # 使用聚合层聚合sequence_output
        batch_mean_hidden = self.average_layer(sequence_output, sequence_length)
        # 输出文本分类logits
        logits = self.output_layer(batch_mean_hidden)
        return logits
```

接下来，基于Paddle的单向模型开始进行训练，代码实现如下：

In [17]:
```
import time
import random
import numpy as np
from nndl import Accuracy, RunnerV3

np.random.seed(0)
random.seed(0)
paddle.seed(0)

# 指定训练轮次
num_epochs = 3
# 指定学习率
learning_rate = 0.001
# 指定embedding的数量为词表长度
num_embeddings = len(word2id_dict)
# embedding向量的维度
input_size = 256
# LSTM网络隐状态向量的维度
hidden_size = 256

# 实例化模型
model = Model_BiLSTM_FC(num_embeddings, input_size, hidden_size)
# 指定优化器
optimizer = paddle.optimizer.Adam(learning_rate=learning_rate, beta1=0.9, beta2=0.999, parameters= model.parameters())
# 指定损失函数
loss_fn = paddle.nn.CrossEntropyLoss()
# 指定评估指标
metric = Accuracy()
# 实例化Runner
runner = RunnerV3(model, optimizer, loss_fn, metric)
# 模型训练
start_time = time.time()
runner.train(train_loader, dev_loader, num_epochs=num_epochs, eval_steps=10, log_steps=10, save_path="./checkpoints/best_forward.p
end_time = time.time()
print("time: ", (end_time-start_time))
```

基于Paddle的单向LSTM进行模型评价，代码实现如下：

In [18]:
```
model_path = "./checkpoints/best_forward.pdparams"
runner.load_model(model_path)
accuracy, _ =  runner.evaluate(test_loader)
print(f"Evaluate on test set, Accuracy: {accuracy:.5f}")
```

6.6.2 使用Paddle内置的单向LSTM进行文本分类实验

由于之前实现的LSTM默认只返回最后时刻的隐状态，然而本实验中需要用到所有时刻的隐状态向量，因此需要对自己实现的LSTM进行修改，使其返回序列向量，代码实现如下：

In [19]:
```
import paddle.nn.functional as F
# 声明LSTM和相关参数
class LSTM(nn.Layer):
    def __init__(self, input_size, hidden_size, Wi_attr=None, Wf_attr=None, Wo_attr=None, Wc_attr=None,
                 Ui_attr=None, Uf_attr=None, Uo_attr=None, Uc_attr=None, bi_attr=None, bf_attr=None,
                 bo_attr=None, bc_attr=None):
        super(LSTM, self).__init__()
        self.input_size = input_size
        self.hidden_size = hidden_size

        # 初始化模型参数
        self.W_i = paddle.create_parameter(shape=[input_size, hidden_size], dtype="float32", attr=Wi_attr)
        self.W_f = paddle.create_parameter(shape=[input_size, hidden_size], dtype="float32", attr=Wf_attr)
        self.W_o = paddle.create_parameter(shape=[input_size, hidden_size], dtype="float32", attr=Wo_attr)
        self.W_c = paddle.create_parameter(shape=[input_size, hidden_size], dtype="float32", attr=Wc_attr)
        self.U_i = paddle.create_parameter(shape=[hidden_size, hidden_size], dtype="float32", attr=Ui_attr)
        self.U_f = paddle.create_parameter(shape=[hidden_size, hidden_size], dtype="float32", attr=Uf_attr)
        self.U_o = paddle.create_parameter(shape=[hidden_size, hidden_size], dtype="float32", attr=Uo_attr)
        self.U_c = paddle.create_parameter(shape=[hidden_size, hidden_size], dtype="float32", attr=Uc_attr)
        self.b_i = paddle.create_parameter(shape=[1, hidden_size], dtype="float32", attr=bi_attr)
```

```
        self.b_f = paddle.create_parameter(shape=[1, hidden_size], dtype="float32", attr=bf_attr)
        self.b_o = paddle.create_parameter(shape=[1, hidden_size], dtype="float32", attr=bo_attr)
        self.b_c = paddle.create_parameter(shape=[1, hidden_size], dtype="float32", attr=bc_attr)

    # 初始化状态向量和隐状态向量
    def init_state(self, batch_size):
        hidden_state = paddle.zeros(shape=[batch_size, self.hidden_size], dtype="float32")
        cell_state = paddle.zeros(shape=[batch_size, self.hidden_size], dtype="float32")
        hidden_state.stop_gradient = False
        cell_state .stop_gradient = False
        return hidden_state, cell_state

    # 定义前向计算
    def forward(self, inputs, states=None, sequence_length=None):
        batch_size, seq_len, input_size = inputs.shape  # inputs batch_size x seq_len x input_size

        if states is None:
            states = self.init_state(batch_size)
        hidden_state, cell_state = states

        outputs = []
        # 执行LSTM计算，包括：隐藏门、输入门、遗忘门、候选状态向量、状态向量和隐状态向量
        for step in range(seq_len):
            input_step = inputs[:, step, :]
            I_gate = F.sigmoid(paddle.matmul(input_step, self.W_i) + paddle.matmul(hidden_state, self.U_i) + self.b_i)
            F_gate = F.sigmoid(paddle.matmul(input_step, self.W_f) + paddle.matmul(hidden_state, self.U_f) + self.b_f)
            O_gate = F.sigmoid(paddle.matmul(input_step, self.W_o) + paddle.matmul(hidden_state, self.U_o) + self.b_o)
            C_tilde = F.tanh(paddle.matmul(input_step, self.W_c) + paddle.matmul(hidden_state, self.U_c) + self.b_c)
            cell_state = F_gate * cell_state + I_gate * C_tilde
            hidden_state = O_gate * F.tanh(cell_state)

            outputs.append(hidden_state.unsqueeze(axis=1))

        outputs = paddle.concat(outputs, axis=1)
        return outputs
```

接下来，修改Model_BiLSTM_FC模型，将`nn.LSTM`换为自己实现的LSTM模型，代码实现如下：

In [20]:
```
class AveragePooling(nn.Layer):
    def __init__(self):
        super(AveragePooling, self).__init__()

    def forward(self, sequence_output, sequence_length):
        sequence_length = paddle.cast(sequence_length.unsqueeze(-1), dtype="float32")
        # 根据sequence_length生成mask矩阵，用于对Padding位置的信息进行mask
        max_len = sequence_output.shape[1]
        mask = paddle.arange(max_len) < sequence_length
        mask = paddle.cast(mask, dtype="float32").unsqueeze(-1)
        # 对序列中paddling部分进行mask
        sequence_output = paddle.multiply(sequence_output, mask)
        # 对序列中的向量取均值
        batch_mean_hidden = paddle.divide(paddle.sum(sequence_output, axis=1), sequence_length)
        return batch_mean_hidden

class Model_BiLSTM_FC(nn.Layer):
    def __init__(self, num_embeddings, input_size, hidden_size, num_classes=2):
        super(Model_BiLSTM_FC, self).__init__()
        # 词典大小
        self.num_embeddings = num_embeddings
        # 单词向量的维度
        self.input_size = input_size
        # LSTM隐藏单元数量
        self.hidden_size = hidden_size
        # 情感分类类别数量
        self.num_classes = num_classes
        # 实例化嵌入层
        self.embedding_layer = nn.Embedding(num_embeddings, input_size, padding_idx=0)
        # 实例化LSTM层
        self.lstm_layer = LSTM(input_size, hidden_size)
        # 实例化聚合层
        self.average_layer = AveragePooling()
        # 实例化输出层
        self.output_layer = nn.Linear(hidden_size, num_classes)

    def forward(self, inputs):
        # 对模型输入拆分为序列数据和mask
        input_ids, sequence_length = inputs
        # 获取词向量
        inputs_emb = self.embedding_layer(input_ids)
        # 使用lstm处理数据
        sequence_output = self.lstm_layer(inputs_emb)
        # 使用聚合层聚合sequence_output
        batch_mean_hidden = self.average_layer(sequence_output, sequence_length)
        # 输出文本分类logits
        logits = self.output_layer(batch_mean_hidden)
        return logits
```

In [21]:
```
import time
import random
import numpy as np
from nndl import Accuracy, RunnerV3
```

```
np.random.seed(0)
random.seed(0)
paddle.seed(0)

# 指定训练轮次
num_epochs = 3
# 指定学习率
learning_rate = 0.001
# 指定embedding的数量为词表长度
num_embeddings = len(word2id_dict)
# embedding向量的维度
input_size = 256
# LSTM网络隐状态向量的维度
hidden_size = 256

# 实例化模型
model = Model_BiLSTM_FC(num_embeddings, input_size, hidden_size)
# 指定优化器
optimizer = paddle.optimizer.Adam(learning_rate=learning_rate, beta1=0.9, beta2=0.999, parameters= model.parameters())
# 指定损失函数
loss_fn = paddle.nn.CrossEntropyLoss()
# 指定评估指标
metric = Accuracy()
# 实例化Runner
runner = RunnerV3(model, optimizer, loss_fn, metric)
# 模型训练
start_time = time.time()
runner.train(train_loader, dev_loader, num_epochs=num_epochs, eval_steps=10, log_steps=10, save_path="./checkpoints/best_self_forwa
end_time = time.time()
print("time: ", (end_time-start_time))
```

基于Paddle的单向LSTM进行模型评价，代码实现如下：

In [22]:
```
model_path = "./checkpoints/best_self_forward.pdparams"
runner.load_model(model_path)
accuracy, _ =  runner.evaluate(test_loader)
print(f"Evaluate on test set, Accuracy: {accuracy:.5f}")
```

6.6 小结

本章通过实践来加深对循环神经网络的基本概念、网络结构和长程依赖问题问题的理解．我们构建一个数字求和任务，并动手实现了 SRN 和 LSTM 模型，对比它们在数字求和任务上的记忆能力．在实践部分，我们利用双向 LSTM 模型来进行文本分类任务：IMDB 电影评论情感分析，并了解如何通过嵌入层将文本数据转换为向量表示.

> 动手练习6.7：在我们手动实现的 LSTM 算子中，是逐步计算每个时刻的隐状态．请思考如何实现更加高效的 LSTM 算子?