Introduction to Deep learning @22 Autumn, USTC

LAB3-BERT 文本情感分类

SA22011035 李钊佚

目录:

正文:

➢ 实验过程陈述:

本次实验主要内容为利用 **BERT** 语言模型（**Language Model**）**(**加载预训练的 **BERT** 模型，并在本实验数据集上 **fine-tune)**

实现 **IMDB** 电影评论文本情感分类任务（**Sentiment-Classification Task**）。

我的实验过程主要分为以下若干部分:

- ✓ 数据集 **reorganization** 和 **preprocessing**。：由于下载的数据集的组织形式并不是常见的可供 **python** 文件读取的形式，我们编写了数据预处理的 **python** 文件来将原始数据处理成 **python** 友好的**.csv** 格式文件**,**请参考附件中的 **preprocess.ipynb** 文件**.**
- ✓ 数据集的加载。：由于本次实验中，我们需要利用 **huggingface** 提供的 **BERT** 接口，我们需要将数据处理成可以 **huggingface** 预先约定好的形式，并编写 **dataloader.**

请参考附件中 **sent_cls_bert.ipynb** 文件的前半部分。

- ✓ 预训练 **BERT** 模型加载。：使用 **huggingface** 提供的接口加载预训练 **BERT** 模型。请参考 **sent_cls_bert.ipynb.**
- ✓ **BERT** 模型的 **finetune** 训练。：定义优化器，调度器，在 **pytorch** 中编写训练代码在我们的任务上微调训练加载的预训练 **BERT** 模型。请参考 **sent_cls_bert.ipynb**
- ✓ 测试。：编写测试代码，测试 **fine-tune** 后的 **BERT** 模型在测试集上的性能。请参考 **sent_cls_bert.ipynb**

➢ 数据集处理设置:

由于原始数据**(raw data)**是在每个文件中仅包含一条评论，为了使其转化为 **python** 容易利用的形式，我们处理原始数据首先使其变为**.csv** 文件，其中我们对训练集进行 **75%/25%**划分为 **train/dev** 集合,最后我们会生成 **train.tsv / dev.tsv / test.tsv** 三个文件。

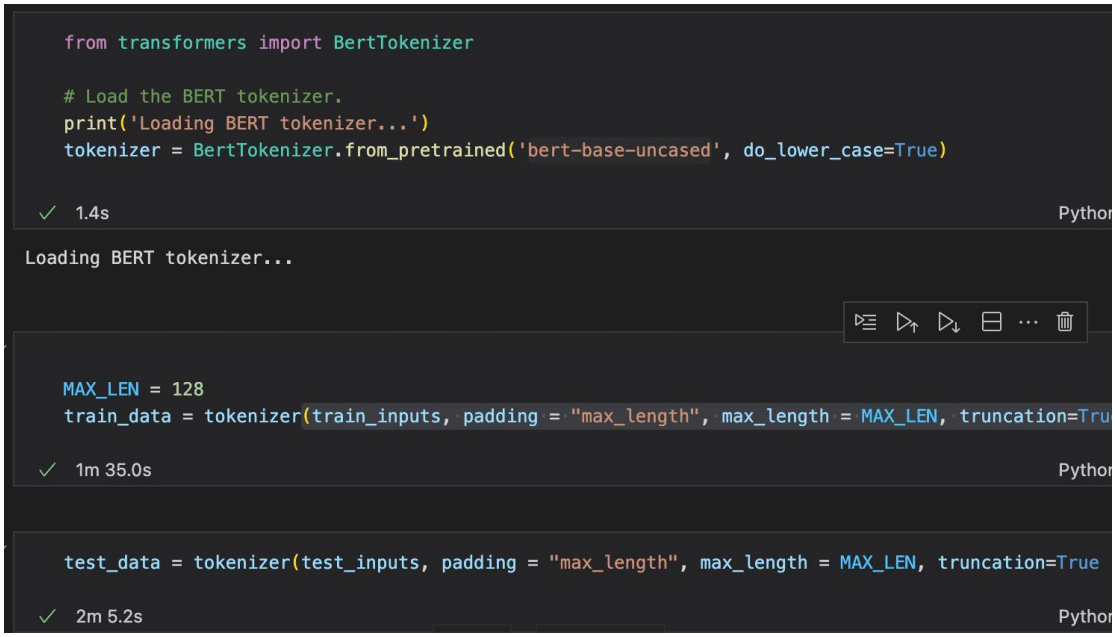
其中每个文件的每行均为如下形式：

{input_review};{label} \n

其中**`input_review`**即为一条文本评论； **label = '0'(negative)**或者**'1'(positive);**

由于本次实验中，我们需要利用 **huggingface** 提供的 **BERT** 接口，我们需要将数据处理成可以 **huggingface** 预先约定好的形式，并编写 **dataloader.**

我们采用 **huggingface** 提供的 **tokenizer**，来对数据进行分词和 **padding**。



这样得到的数据有两部分组成：

@1:input_ids 即 **token** 编码成数字之后的结果，且每个句子都被规范化为 **max_len** 长度的张量，原本超过 **max_len** 的部分将会被截断，原本不足 **max_len** 的部分会被补成 **padding.**

@2:attention_masks: 即指示每个 **sentence** 中 **padding** 的部分，不是 **padding** 的部分为 **1**，**padding** 的部分为 **1**。

这两部分是输入 **huggingface** 提供的 **BERT** 所必需的。

➢ 关键代码展示：

✓ 【数据集产生和划分】

✓ 【预训练 BERT 的加载、dataloader、优化器、调度器的定义】

```python
from transformers import BertForSequenceClassification, AdamW, BertConfig

# Load BertForSequenceClassification, the pretrained BERT model with a single
# linear classification layer on top.
model = BertForSequenceClassification.from_pretrained(
    "bert-base-uncased", # Use the 12-layer BERT model, with an uncased vocab.
    num_labels = 2, # The number of output labels--2 for binary classification.
                    # You can increase this for multi-class tasks.
    output_attentions = False, # Whether the model returns attentions weights.
    output_hidden_states = False, # Whether the model returns all hidden-states.
)
```

✓ 39.9s Pyth

Downloading: 100% 440M/440M [00:33<00:00, 25.3MB/s]

```python
import torch
from torch.utils.data import TensorDataset, DataLoader, RandomSampler, SequentialSampler
batch_size = 16
train = TensorDataset(train_data["input_ids"], train_data["attention_mask"], torch.tensor(train_lab
train_sampler = RandomSampler(train)
train_dataloader = DataLoader(train, sampler=train_sampler, batch_size=batch_size)

test = TensorDataset(test_data["input_ids"], test_data["attention_mask"], torch.tensor(test_labels
test_sampler = RandomSampler(test)
test_dataloader = DataLoader(test, sampler=test_sampler, batch_size=batch_size)

from torch.optim import AdamW
optimizer = AdamW(model.parameters(), lr=5e-5)

num_epochs = 1
from transformers import get_scheduler
num_training_steps = num_epochs * len(train_dataloader)
lr_scheduler = get_scheduler(
    name="linear", optimizer=optimizer, num_warmup_steps=0, num_training_steps=num_training_steps
)
```

[26] ✓ 0.1s Python

✓ 【模型训练】

```
device = torch.device("cuda") if torch.cuda.is_available() else torch.device("cpu")
model.to(device)
for epoch in range(num_epochs):
    total_loss = 0
    model.train()
    for step, batch in enumerate(train_dataloader):
        if step % 10 == 0 and not step == 0:
            print("step: ",step, "  loss:",total_loss/(step*batch_size))
        b_input_ids = batch[0].to(device)
        b_input_mask = batch[1].to(device)
        b_labels = batch[2].to(device)
        model.zero_grad()
        outputs = model(b_input_ids,
                        token_type_ids=None,
                        attention_mask=b_input_mask,
                        labels=b_labels)

        loss = outputs.loss
        total_loss += loss.item()
        loss.backward()
        torch.nn.utils.clip_grad_norm_(model.parameters(), 1.0)
        optimizer.step()
        lr_scheduler.step()
    avg_train_loss = total_loss / len(train_dataloader)
    print("avg_loss:",avg_train_loss)
```

[27] ✓ 3m 8.3s Python

✓ 【模型性能测试】

```
import numpy as np
model.eval()
tot_corrs = 0
tot_num = 0
with torch.no_grad():

    for step, batch in enumerate(train_dataloader):
        bat_input_ids = batch[0].to(device)
        bat_att_masks = batch[1].to(device)
        batch_labels = batch[2].tolist()
        outputs = model(bat_input_ids,
                        token_type_ids=None,
                        attention_mask=bat_att_masks)
        preds = torch.argmax(outputs["logits"],dim=1).tolist()
        assert len(preds) == len(batch_labels)
        corrs = sum([batch_labels[i]==preds[i] for i in range(len(preds))])
        tot_corrs += corrs
        tot_num += len(preds)
print('--testing phase---: testing acc = ', tot_corrs/tot_num)
```

[35] ✓ 49.8s Python

```
--testing phase---: testing acc =  0.9365721279199702
```

➢ 测试结果和超参数分析:

✓ 【超参数设置和测试结果】:

我们采用 AdamW 优化器来优化 BERT 模型,并且采用 linear 优化调度器来自动调度优化器的学习率等超参数, 我们设置初始学习率分别为：1e-4，5e-5，2e-5，1e-5。

对于训练数据的 max_len，我们设置为 128(注意 huggingface-BERT 所能接受 max_len 最多为 512，但是考虑到 max_len 为{128,256})

对于 batch_size，当 max_len 设置为 128 时，我们设置 batch_size=16；当 max_len 设置为 256 时，我们设置 batch_size=8;(这是为了应对我们显存的限制)

我们选择加载的预训练 BERT 模型为：`bert-base-uncased`.

经过实验比对，我们最终选择实验参数如下：

optimizer:AdamW;

初始学习率：5e-5;

max_len*batch_size: 128*16

最终在测试集上的测试准确率为：93.66%【预训练 BERT】

我们同样测试了非预训练的 BERT, 我们从头训练得到的测试集性能为：87.26%

✓ 【BERT 与实验二 RNN 实验结果对比分析】:

Recall 实验二结果：

LSTM 不加载预训练词向量(e.g.,glove)测试性能在 80%～85%，(根据超参数【是否双向，LSTM 堆叠层数，batch_size, embedding 维度，隐层向量维度】而变化)；

LSTM 加载预训练词向量(e.g.,glove)，我们在如下搜索空间搜索超参数：$(\text{bidirectional, layer\_num, batch\_size, hidden\_dim}) \in \{\text{True, False}\} \times \{1,2\} \times \{16,64\} \times \{128,256\};$

=》当采用单向 LSTM 和单层 LSTM 时所能获得的最佳测试性能

【bidirectional = False, layer_num = 1】

hyper-parameter setting	testing performance(accuracy)
batch_size=16,hidden_dim=256	**91.69%**
batch_size=16,hidden_dim=128	91.53%
batch_size=64,hidden_dim=256	91.64%
batch_size=64,hidden_dim=128	91.43%

对比分析：

对比是否预训练(对 **LSTM** 来说，即：是否添加预训练词向量)和模型架构(**LSTM, BERT-base**),结果如下表：

model/settings	无预训练(词向量)	预训练(词向量)
LSTM	**84.82%**	**91.69%**
BERT-base	**87.26%**	**93.66%**

可见：

@1:通过预训练/添加预训练词向量将语言内蕴知识注入模型可以较

大程度上提升模型(无论是 **LSTM** 还是 **BERT)**的泛化性能。

@2:在同样不添加外部预训练知识时，或者是同样采用预训练注入外部知识时：在 **IMDB** 情感分类任务上，**BERT** 模型的泛化性能都要明显优于 **LSTM** 的泛化性能，这很大程度上来自与 **BERT** 模型更强的表达能力。