Introduction to Deep learning @22 Autumn, USTC

LAB2-情感分类实验-实验报告

SA22011035 李钊佚

目录：

正文：

➢ 实验过程陈述：

本次实验主要内容为编写 **RNN-based** 语言模型（**Language Model**）基于训练好的词向量（**word2vec**），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，并且由于原始数据仅分为 **train/test** 集合，我们通过对 **train** 集合随机进行 **3:1** 划分成 **train/development** 集合。

✓ 使用 **torchtext package** 来处理生成的**.csv** 文件，读取数据，产生训练时所需的数据迭代器

- ✓ 采用预训练好的 **glove.6B.100d** 文件提取本数据集中 **vocab** 的 **embedding**。
- ✓ 编写 **LSTM+MLP** 来提取每一条数据的评论中的 **feature**，并用于分类。
- ✓ 编写训练代码和测试代码：优化 **LSTM+MLP** 网络中的权重参数**,** 统计每个 **epoch** 中训练损失平均值，训练准确率，验证集损失平均值，验证集准确率。

➢ 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);**

➢ 关键代码展示：

✓ 【数据集产生和划分】

见 preprocess.ipynb 文件和 README.md 附件

✓ 【torchtext 读取 csv 文件并建立 dataset 迭代器】

```
TEXT = data.Field(lower=True, batch_first=True,
fix_length=50)
LABEL = data.Field(sequential=False)
train, dev, test = data.TabularDataset.splits(
path='/data2/home/zhaoyi/labs/USTC-labs/deeplearn_lab2/da
taset/procd/',
train='train.csv', validation='dev.csv', test='test.csv',
format='csv', fields=[('Text',TEXT),('Label',LABEL)])
```

```
# defintion of data_loader
mydevice = torch.device('cuda' if torch.cuda.is_available()
else 'cpu')
print(mydevice)
train_iter, dev_iter, test_iter =
data.BucketIterator.splits((train, dev, test),
batch_size=my_bs, device=mydevice, shuffle=True,
sort=False)
```

✓ 【预训练词向量的导入】

```
# construct and load word-vectors from a pretrained file
TEXT.build_vocab(train, vectors="glove.6B.100d",
max_size=10000, min_freq=10)
# glove-file-location : workspace/.vector_cache
LABEL.build_vocab(train)
# print(TEXT.vocab.freqs.most_common(20))
```

✓ 【模型的构建】

```
# model architecture for sentiment classification: LSTM + MLP
class LSTM(nn.Module):
def __init__(self, vocab_size, embedding_dim, hidden_dim,
num_layers, bidirectional, drop_out, pad_idx, batch_first =
False):
super().__init__()
self.embedding = nn.Embedding(vocab_size, embedding_dim,
padding_idx = pad_idx)
self.lstm = nn.LSTM(embedding_dim, hidden_dim,
num_layers=num_layers,
batch_first = batch_first, bidirectional=bidirectional,
dropout=drop_out)
'''
nn.LSTM(input_size, hidden_size, num_layers)
num_layers: the layer_num of LSTM, usually an important thing
in LSTM-based model architecture...
bidirectional: also an important hyperparameter...
reference:https://blog.csdn.net/baidu_38963740/article/de
tails/117197619?spm=1001.2101.3001.6650.1&utm_medium=dist
ribute.pc_relevant.none-task-blog-2%7Edefault%7ECTRLIST%7
Edefault-1.no_search_link&depth_1-utm_source=distribute.p
c_relevant.none-task-blog-2%7Edefault%7ECTRLIST%7Edefault
-1.no_search_link
'''
if bidirectional == False:
num_direction = 1
else:
num_direction = 2
lstm_output_dim = num_direction * hidden_dim

self.fc = nn.Linear(lstm_output_dim, 2)
# for this case is a 2-class problem
self.dropout = nn.Dropout(drop_out)
def forward(self, x):
embedded = self.embedding(x)
```

```
'''
x.shape = embedded.shape = (batch_size, seq_len,
embedding_dim) [tips: when we set `batch_first` == True]
otherwise, x.shape = embedded.shape = (seq_len, bs,
embedding_dim)
'''
lstm_output, (_, _) = self.lstm(embedded)
'''
when num_layers = bidirectional = 1 and batch_first = True
size of lstm_output: (batch_size, seq_len, hidden_dim *
num_directions)
size of h_n and c_n: (num_layers * num_directions = 1,
batch_size, hidden_size)
'''
```

```
output = self.dropout(self.fc(lstm_output[:, -1, :]))
'''
we only select last-step of seq_len in the lstm_output as
the encoding sentence vector, for it is containing the
information
of the whole sentence(unidirectionally speaking),
when we adapt bidirectional lstm, we can choose any-step of
seq_len
instead.
'''
'''
output:(batch_size, encoding_vector_dim=2)
'''
return F.log_softmax(output, dim = 1)
```

```
pad_idx = TEXT.vocab.stoi[TEXT.pad_token]
```

```
# definition of model and optimizer
model = LSTM(len(TEXT.vocab.stoi), 100, my_hiddim,
my_layernum, my_bidirectional, 0.4, pad_idx, True)
model.embedding.weight.data = TEXT.vocab.vectors
model.embedding.weight.requires_grad = False
# frozen pretrained embedding weights
```

```
model = model.cuda()
'''
(self, vocab_size, embedding_dim, hidden_dim,
num_layers, bidirectional, drop_out, pad_idx, batch_first =
False)
'''
opt = torch.optim.Adam(model.parameters(),lr=1e-3)
```

✓ 【模型训练】

```
# training function
def train_epoch(model, opt, data_loader, phase='training'):
'''
function: train model with opt for one epoch
'''
if phase == 'training':
model.train()
if (phase == 'validation') or (phase == 'testing'):
model.eval()
# model.train() : open `batch_normalization` and `drop_out`
# model.eval() : open `batch_normalization`, close
`drop_out`
running_loss = 0.0
running_correct = 0.0
for _, batch in enumerate(data_loader):
text, target = batch.Text, batch.Label
if mydevice == 'cuda':
text, target = text.cuda(),target.cuda()
if phase == 'training':
opt.zero_grad()
output = model(text)
loss = F.nll_loss(output, target-1)
running_loss = F.nll_loss(output, target-1,
size_average=False).data
preds = output.data.max(dim=1, keepdim=True)[1] + 1
# for label '0' -> 1(in vocab); label '1' -> 2(in vocab);
```

```
running_correct +=
preds.eq(target.data.view_as(preds)).sum()
if phase == 'training':
loss.backward()
opt.step()

running_loss = running_loss.type(torch.FloatTensor)
running_correct = running_correct.type(torch.FloatTensor)
# IMPORTANT above! otherwise accuracy will be zero all the
time!
loss = running_loss/len(data_loader.dataset)
accuracy = running_correct/len(data_loader.dataset)
# print(type(loss),type(accuracy))
print(f'{phase} loss is {loss:{5}.{2}} and {phase} accuracy
is {running_correct}/{len(data_loader.dataset)}
{accuracy:{10}.{4}}')
return loss,accuracy
```

✓ 【模型性能测试】

```
# collect results
train_losses, train_accuracy = [], []
val_losses, val_accuracy = [], []

train_iter.repeat = False
test_iter.repeat = False
```

```
epoch_max = 20
for epoch in range(1,epoch_max+1):
print('---the ',epoch,"'s training starts---")
epoch_loss, epoch_accuracy = train_epoch(model, opt,
train_iter, phase='training')
val_epoch_loss, val_epoch_accuracy = train_epoch(model, opt,
dev_iter, phase='validation')
train_losses.append(epoch_loss)
train_accuracy.append(epoch_accuracy)
```

```python
val_losses.append(val_epoch_loss)
val_accuracy.append(val_epoch_accuracy)
print('---the ',epoch,"'s training ends---")
```

```python
print('---testing phase---')
# test model's performance
train_epoch(model, opt, test_iter, phase='testing')
```

➢ 测试结果和超参数分析：

我们在如下空间搜索超参数：

training epoch = 20，我们通过观察 **validation set** 上的 **loss** 和 **acc** 可以发现，在 **epoch = 1 ~ 20** 过程中，**validation acc** 上升后趋于平稳，并未见到明显下降痕迹，因此推断没有发生严重的过拟合，故为了方便，我们直接选择 **epoch=20** 训练完的模型作为最终模型来在 **testing set** 上面测试泛化性能。

超参数搜索空间：　(我们考虑以下 **4** 个关键的超参数)

$$\text{bidirectional} \in \{\text{True, False}\};$$

$$\text{layer\_num} \in \{1,2\};$$

$$\text{batch\_size} \in \{16,64\};$$

$$\text{hidden\_dim} \in \{128,256\};$$

$$\Rightarrow (\text{bidirectional, layer\_num, batch\_size, hidden\_dim}) \in \{\text{True, False}\} \times \{1,2\} \times \{16,64\} \times \{128,256\};$$

where` × `represents Cartesain product

✓ 【测试集 **evaluation** 结果】：

@1:bidirectional = True, layer_num = 2

hyper-parameter setting	testing performance(accuracy)
batch_size=16,hidden_dim=256	90.55%
batch_size=16,hidden_dim=128	90.29%
batch_size=64,hidden_dim=256	89.60%
batch_size=64,hidden_dim=128	89.41%

@1:bidirectional = True, layer_num = 1

hyper-parameter setting	testing performance(accuracy)
batch_size=16,hidden_dim=256	91.55%
batch_size=16,hidden_dim=128	91.63%
batch_size=64,hidden_dim=256	91.78%
batch_size=64,hidden_dim=128	91.75%

@1:bidirectional = False, layer_num = 2

hyper-parameter setting	testing performance(accuracy)
batch_size=16,hidden_dim=256	90.14%
batch_size=16,hidden_dim=128	89.44%
batch_size=64,hidden_dim=256	90.28%
batch_size=64,hidden_dim=128	89.97%

@1: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%

✓ 【超参数分析】：

@1:通过观察，经验的结果告诉我们对 IMDB 数据集和我们设计的模型结构来讲，对 model 在 testing set 上面的 performance 影响最大的超参数是 LSTM 的 layer_num, layer_num = 1 时的模型的泛化能力普遍好于 layer_num = 2 时的模型的泛化能力。

@2:按照一般规律来讲，batch_size 越大，模型的训练应该越稳定，效果越好，但是在本实验中，我们对比三组 batch_size = 16 和 64，并没有明显发现 batch_size = 64 时训练得到的 model 的泛化能力一致地（consistently）好于 batch_size = 16 训练得到的 model 的泛化能力。

@3:通过对比 LSTM 提取特征的维度（hidden_dim）可以发现，当 hidden_dim 设置为 256 时，模型的泛化性能轻微地（slightly）好于 hidden_dim 设置为 128 时训练得到的模型的泛化性能。这可以解释为当其他超参数设置一致时，hidden_dim 越大，模型的假设空间越大，模型的能力越强。