# lab2：循环神经网络进行文本情感分类

姓名：李晖茜
学号：SA22218131

## 一、实验要求

使用 pytorch 或者 tensorflow 的相关神经网络库，编写 RNN 的语言模型，并基于训练好的词向量，编写 RNN 模型用于文本分类。
具体来说，在本次实验中，需要通过 RNN 实现文本情感分类(Text Sentiment Classification)：输入一个句子，输出是 0(负面)或 1(正面)。

## 二、 实验环境

torch 1.10.0+cu111
torchtext 0.6.0

## 三、实验过程

### 3.1 数据

使用斯坦福的IMDb数据集（Stanford's Large Movie Review Dataset）作为文本情感分类的数据集。这个数据集分为训练和测试用的两个数据集，分别包含25,000条从IMDb下载的关于电影的评论。在每个数据集中，标签为"正面"和"负面"的评论数量相等。
首先下载数据集，然后通过以下代码解压。

```
# 1. 解压数据
def Decompress(DATA_ORIGIN_ROOT):
    fname = os.path.join(DATA_ORIGIN_ROOT, 'aclImdb_v1.tar.gz')
    # 将压缩文件进行解压
    if not os.path.exists(os.path.join(DATA_ORIGIN_ROOT, 'aclImdb')):
        print("从压缩包解压...")
        with tarfile.open(fname, 'r') as f:
            f.extractall(DATA_ORIGIN_ROOT) # 解压文件到此指定路径
    return DATA_ORIGIN_ROOT+'/aclImdb/'
```

接下来读取训练数据集合测试数据集：

```
# 2. 读取数据
def readData(folder, DATA_ROOT):
    data = []
    for label in ['pos', 'neg']:
        folder_name = os.path.join(DATA_ROOT, folder, label) # 拼接文件路径
        for file in os.listdir(folder_name): # 读取文件路径下的所有文件名，并存入列表中
            with open(os.path.join(folder_name, file), 'rb') as f:
                review = f.read().decode('utf-8').replace('\n', ' ').lower()
                data.append([review, 1 if label == 'pos' else 0]) # 将每个文本读取
的内容和对应的标签存入data列表中
    random.seed(0)       # 设置随机数种子，保证每次生成的结果都是一样的
    random.shuffle(data) # 打乱data列表中的数据排列顺序
    return data
```

对训练数据集进行划分，分成训练集和验证集：

```
# 2.1 划分训练集和验证集
def split_train_val(data, val_ratio):
    val_set_size = int(len(data) * val_ratio)
    return data[val_set_size:], data[:val_set_size]
```

读取数据之后，需要对数据进行预处理，首先对每条评论进行分词，这里定义的`get_tokenized_imdb`函数使用最简单的方法：基于空格进行分词。

```
# 2.2 预处理数据
# 空格分词
def get_tokenized_imdb(data):
    '''
    :param data: list of [string, label]
    '''
    def tokenizer(text):
        return [tok.lower() for tok in text.split(' ')]
    return [tokenizer(review) for review, _ in data] # 只从data中读取review(评论)内容而不读取标签(label)，对review使用tokenizer方法进行分词
```

根据分好词的训练数据集创建词典，并过滤掉出现次数少于5的词。

```
# 创建词典
def get_vocab_imdb(data):
    tokenized_data = get_tokenized_imdb(data) # 调用get_tokenized_imdb()空格分词方法，获取到分词后的数据tokenized_data
    counter = collections.Counter([tk for st in tokenized_data for tk in st]) # 读取tokenized_data列表中每个句子的每个词，放入列表中。
    specials = ['<unk>']
    return Vocab.Vocab(counter, min_freq=5, specials=specials) # 去掉词频小于5的词
```

每条评论长度不一致不能直接组合成小批量，定义`process_imdb`函数对每条评论进行分词，并通过词典转换成词索引，然后通过截断或者补0使得每条评论长度固定成500。

```
# 对data列表中的每行数据进行处理，将词转换为索引，并使每行数据等长
def process_imdb(data, vocab):
    max_len = 500 # 每条评论通过截断或者补0，使得长度变成500
    def pad(x):
        # x[:max_len] 只获取前max_len个词
        # x + [0]*(max_len - len(x)) 词数小于max_len，用pad=0补长到max_len
        return x[:max_len] if len(x) > max_len else x + [0]*(max_len - len(x))
    tokenized_data = get_tokenized_imdb(data) # 调用方法获取分词后的数据
    features = torch.tensor([pad([vocab.stoi[word] for word in words]) for words in tokenized_data]) # 将词转换为vocab词典中对应词的索引
    labels = torch.tensor([score for _, score in data])
    return features, labels
```

创建数据迭代器，每次迭代返回一个小批量的数据。

```
# 2.3 创建数据迭代器
def data(DATA_ROOT, batch_size = 64):
    train_val, test_data = readData('train', DATA_ROOT), readData('test', DATA_ROOT)
```

```
    train_data, val_data = split_train_val(train_val, 0.2)
    print(len(train_data), "train +", len(val_data), "val +", len(test_data),
"test")
    vocab = get_vocab_imdb(train_data)
    # print(len(vocab))
    # print(vocab.get_stoi()['hello'])
    train_set = Data.TensorDataset(*process_imdb(train_data, vocab))
    val_set = Data.TensorDataset(*process_imdb(val_data, vocab))
    test_set = Data.TensorDataset(*process_imdb(test_data, vocab))
    train_iter = Data.DataLoader(train_set, batch_size, True)
    val_iter = Data.DataLoader(val_set, batch_size, True)
    test_iter = Data.DataLoader(test_set, batch_size)
    return train_iter, val_iter, test_iter, vocab
```

打印出训练集、验证集和测试集：20000 train + 5000 val + 25000 test

## 3.2 网络搭建

接下来进行网络的搭建，使用双向循环神经网络。每个词先通过嵌入层得到特征向量，然后使用双向循环神经网络对特征序列进一步编码得到序列信息，最后将编码的序列信息通过全连接层变换为输出。在下面实现的 `BiRNN` 类中，`Embedding` 实例即嵌入层，`LSTM` 实例即为序列编码的隐藏层，`Linear` 实例即生成分类结果的输出层。

```
# 3. 创建循环神经网络
# 在下面实现的BiRNN类中，Embedding实例即嵌入层，LSTM实例即为序列编码的隐藏层，Linear实例即生
成分类结果的输出层。
class BiRNN(nn.Module):
    def __init__(self, vocab, embed_size, num_hiddens, num_layers):
        super(BiRNN, self).__init__()
        self.embedding = nn.Embedding(len(vocab), embed_size)
        self.encoder = nn.LSTM(
            input_size=embed_size,
            hidden_size=num_hiddens,
            num_layers=num_layers,
            # batch_first=True,
            bidirectional=True # bidirectional设为True即得到双向循环神经网络
        )
        self.decoder = nn.Linear(4*num_hiddens, 2)


    def forward(self, inputs):
        # inputs: [batch_size, seq_len], LSTM需要将序列长度(seq_len)作为第一维，所以需
要将输入转置后再提取词特征
        # 输出形状 outputs: [seq_len, batch_size, embedding_dim]   embedding_dim词
向量维度
        embeddings = self.embedding(inputs.permute(1, 0))
        # rnn.LSTM只传入输入embeddings，因此只返回最后一层的隐藏层在各时间步的隐藏状态。
        # outputs形状是(seq_len, batch_size, 2*num_hiddens)
        outputs, _ = self.encoder(embeddings)
        # 连结初始时间步和最终时间步的隐藏状态作为全连接层输入。
        # 它的形状为 : [batch_size, 4 * num_hiddens]
        encoding = torch.cat((outputs[0], outputs[-1]), dim=-1)
        outs = self.decoder(encoding)
        return outs
```

## 3.3 网络训练

在训练之前，为避免过拟合，直接使用在更大规模语料上预训练的词向量作为每个词的特征向量。训练时为词典`vocab`中的每个词加载`wordEmbedding_dim`维（在训练时传入值）的GloVe词向量。

```
# 4. 加载预训练的词向量
def load_pretrained_embedding(words, pretrained_vocab):
    '''从训练好的vocab中提取出words对应的词向量'''
    embed = torch.zeros(len(words), pretrained_vocab.vectors[0].shape[0]) #
pretrained_vocab.vectors[0].shape # torch.Size([100])
    oov_count = 0 # out of vocabulary
    for i, word in enumerate(words):
        try:
            idx = pretrained_vocab.stoi[word]
            embed[i, :] = pretrained_vocab.vectors[idx] # 将第i行用预训练的单词向量替
换
        except KeyError:
            oov_count += 1
    if oov_count > 0:
        print("There are %d oov words." % oov_count)
```

然后定义训练函数：

```
# 6. 训练网络
def train(vocab, train_iter, val_iter, device, lr, epochs, wordEmbedding_dim):
    # 定义神经网络
    net = BiRNN(vocab, embed_size, num_hiddens, num_layers)
    print(net)

    # 为词典vocab中的每个词加载dim=100维的GloVe词向量
    glove_vocab = Vocab.GloVe(name='6B', dim=wordEmbedding_dim,
cache=os.path.join(DATA_ROOT, 'glove'))
    # print(len(glove_vocab.stoi)) # 400000
    # print(glove_vocab[0].shape)
    net.embedding.weight.data.copy_(
        load_pretrained_embedding(vocab.itos, glove_vocab)
    )
    net.embedding.weight.requires_grad = False # 直接加载预训练好的，所以不需要更新它


    # 是否gpu训练
    net = net.to(device)
    print("training on ", device)


    # 优化器和损失函数
    optimizer = torch.optim.SGD(net.parameters(), lr)
    loss = torch.nn.CrossEntropyLoss()


    # 网络训练过程。随机梯度下降，设置学习率为lr，迭代epoch次
    batch_count = 0
    train_accs, val_accs = [], []
    for epoch in range(epochs):
        train_l_sum, train_acc_sum, n, start = 0.0, 0.0, 0, time.time()
        for X, y in train_iter:
```

```
            X = X.to(device)
            y = y.to(device)
            y_hat = net(X)
            l = loss(y_hat, y)
            optimizer.zero_grad()
            l.backward()
            optimizer.step()
            train_l_sum += l.cpu().item()
            train_acc_sum += (y_hat.argmax(dim=1) == y).sum().cpu().item()
            n += y.shape[0]
            batch_count += 1
        val_acc = evaluate_accuracy(val_iter, net)
        print('epoch %d, loss %.4f, train acc %.3f, test acc %.3f, time %.1f
sec'
              % (epoch + 1, train_l_sum / batch_count, train_acc_sum / n,
val_acc, time.time() - start))
        train_accs.append(train_acc_sum / n)
        val_accs.append(val_acc)
    plt.cla()
    plt.plot(range(epochs), train_accs, 'r-', lw=2)
    plt.plot(range(epochs), val_accs, 'b-', lw=2)
    plt.xlabel('epoches')
    plt.ylabel('Train acc (red), Val acc (blue)')
    plt.savefig("/home/lihuiqian/hw/lab2/train.jpg")
    return net
```

输出网络结构为：

```
BiRNN(
  (embedding): Embedding(39749, 100)
  (encoder): LSTM(100, 100, num_layers=2, bidirectional=True)
  (decoder): Linear(in_features=400, out_features=2, bias=True)
)
There are 17375 oov words.
training on  cuda
```

## 3.4 调参分析

实验主函数如下，定义了多个超参数，具体调参过程见实验结果。

```
if __name__ == '__main__':
    # 解压文件
    DATA_ORIGIN_ROOT = '/home/lihuiqian/hw/lab2'
    DATA_ROOT = Decompress(DATA_ORIGIN_ROOT)

    # 数据读取
    train_iter, val_iter, test_iter, vocab = data(DATA_ROOT, batch_size = 64)


    # 训练
    embed_size, num_hiddens, num_layers = 100, 100, 2 # 网络参数
    lr, epochs = 0.01, 100    # 超参数
    wordEmbedding_dim = 100   # 词向量维度
    net = train(vocab, train_iter, val_iter, device, lr, epochs,
wordEmbedding_dim)
```

```
    # 测试
    test_acc = evaluate_accuracy(test_iter, net, device)
    print('test_acc: %.3f' % test_acc)
```

### 3.5 测试性能

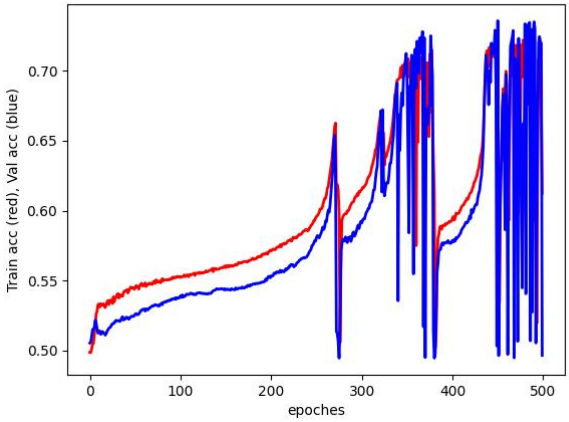
定义评价函数，在训练网络和测试时使用。

```
# 5. 评估函数
def evaluate_accuracy(data_iter, net, device=None):
    if device is None and isinstance(net, torch.nn.Module):
        # 如果没指定device就使用net的device
        device = list(net.parameters())[0].device
    acc_sum, n = 0.0, 0
    with torch.no_grad():
        for X, y in data_iter:
            if isinstance(net, torch.nn.Module):
                net.eval() # 评估模式, 这会关闭dropout
                acc_sum += (net(X.to(device)).argmax(dim=1) ==
y.to(device)).float().sum().cpu().item()
                net.train() # 改回训练模式
            else: # 自定义的模型, 3.13节之后不会用到, 不考虑GPU
                if('is_training' in net.__code__.co_varnames): # 如果有is_training
这个参数
                    # 将is_training设置成False
                    acc_sum += (net(X, is_training=False).argmax(dim=1) ==
y).float().sum().item()
                else:
                    acc_sum += (net(X).argmax(dim=1) == y).float().sum().item()
            n += y.shape[0]
    return acc_sum / n
```

## 四、实验结果

### 4.1 超参数的选择（lr和epochs）

首先选择合适的epoch，令`lr=0.001, epochs=500`，输出训练时的训练集和验证集的准确率，可以观察到epoch达到250之后波动较大，可能出现过拟合，故选择`epochs=250`。按照此思路，调整学习率和训练批次。



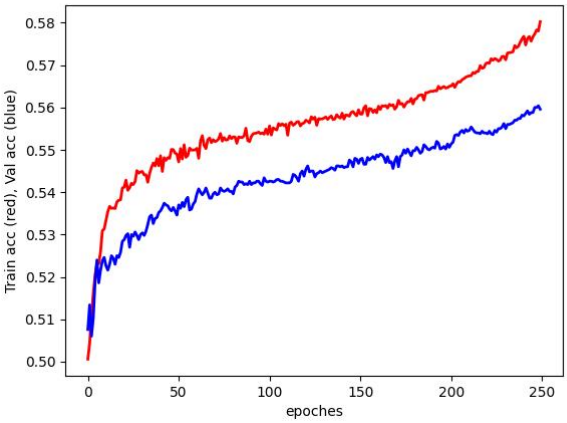

lr=0.001, epochs=500

以下参数先固定，然后调整学习率`lr`和训练批次`epochs`。

```
embed_size, num_hiddens, num_layers = 100, 100, 2 # 网络参数
wordEmbedding_dim = 100    # 词向量维度
```

- **lr, epochs = 0.001, 250  # 超参数**

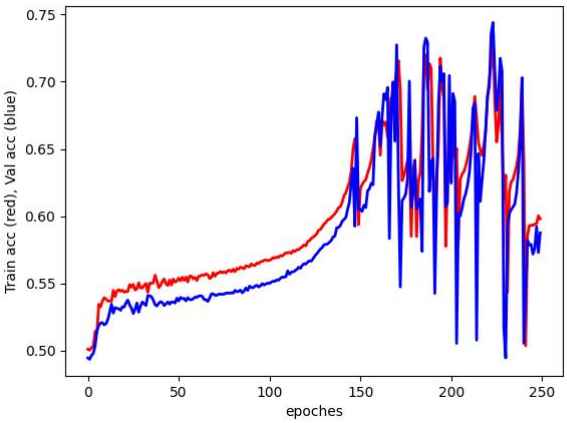
结果：epoch 250, loss 0.0027, train acc 0.580, val acc 0.560, time 34.3 sec




lr, epochs = 0.001, 250

- **lr, epochs = 0.002, 250  # 超参数**

结果：epoch 250, loss 0.0027, train acc 0.598, val acc 0.588, time 32.6 sec




lr, epochs = 0.002, 250

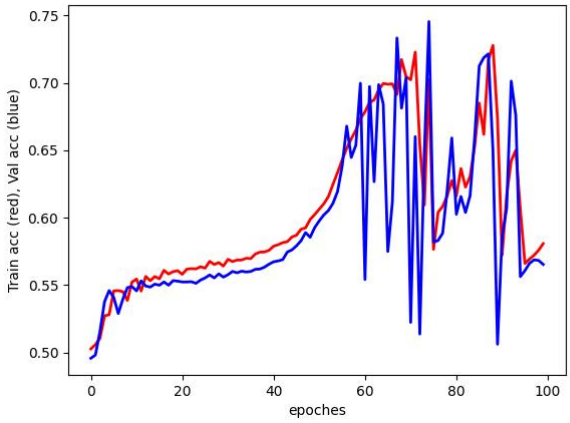
观察epoches-acc图像，发现过拟合，修改超参数。

- **lr, epochs = 0.002, 140  # 超参数**

结果：epoch 140, loss 0.0048, train acc 0.587, val acc 0.577, time 33.4 sec

- **lr, epochs = 0.005, 100  # 超参数**

结果：epoch 100, loss 0.0068, train acc 0.581, val acc 0.565, time 34.0 sec




lr, epochs = 0.005, 100

- **lr, epochs = 0.005, 50　# 超参数**

结果：epoch 50, loss 0.0135, train acc 0.586, val acc 0.575, time 38.0 sec

- **lr, epochs = 0.01, 50　# 超参数**

结果：epoch 50, loss 0.0126, train acc 0.657, val acc 0.620, time 37.2 sec
随着学习率的增大，网络达到收敛的迭代次数会减少，学习率增加到0.01时，达到收敛的迭代次数更少，综合考虑选择`lr, epochs = 0.005, 50`，但迭代次数还是会进行一定的调整。

## 4.2 网络深度

以下参数先固定，然后隐藏层个数`num_layers`。

```
embed_size, num_hiddens = 100, 100 # 网络参数
lr, epochs = 0.005, 50   # 超参数
wordEmbedding_dim = 100   # 词向量维度
```

`num_layers=2`时结果（4.1的结果）：epoch 50, loss 0.0135, train acc 0.586, val acc 0.575, time 38.0 sec

- **num_layers = 3**

结果：epoch 50, loss 0.0138, train acc 0.557, val acc 0.542, time 68.8 sec
epoch 100, loss 0.0067, train acc 0.597, val acc 0.588, time 49.6 sec

- **num_layers = 4**

结果：epoch 50, loss 0.0139, train acc 0.516, val acc 0.506, time 71.5 sec
epoch 100, loss 0.0069, train acc 0.520, val acc 0.557, time 77.3 sec
根据结果，改变隐藏层个数，网络越深训练时间越长，对准确率也没有明显的提升，故选择`num_layers=2`。

## 4.3 使用更大的预训练词向量，如300维的GloVe词向量，能否提升分类准确率?

以下参数先固定，然后调整词向量维度`wordEmbedding_dim=300`，`embed_size=300`。

```
num_hiddens, num_layers = 100, 2 # 网络参数
lr, epochs = 0.005, 50   # 超参数
```

结果：epoch 50, loss 0.0135, train acc 0.583, val acc 0.566, time 46.5 sec
epoch 95, loss 0.0062, train acc 0.697, val acc 0.736, time 39.8 sec

# 五、实验总结

经过以上调参分析，得到一组在验证集上效果最好的参数为：

```
embed_size, num_hiddens, num_layers = 300, 100, 2 # 网络参数
lr, epochs = 0.005, 100   # 超参数
wordEmbedding_dim = 300   # 词向量维度
```

在测试上测试结果为：
epoch 95, loss 0.0063, train acc 0.689, val acc 0.719, time 35.3 sec
test_acc: 0.731
经过此次实验，学习到了循环神经网络的搭建和训练，完成了简单的文本分类任务，了解了文本数据的预处理过程，可以应用预训练的词向量和循环神经网络对文本的情感进行分类，但是此次测试结果还有待提升。