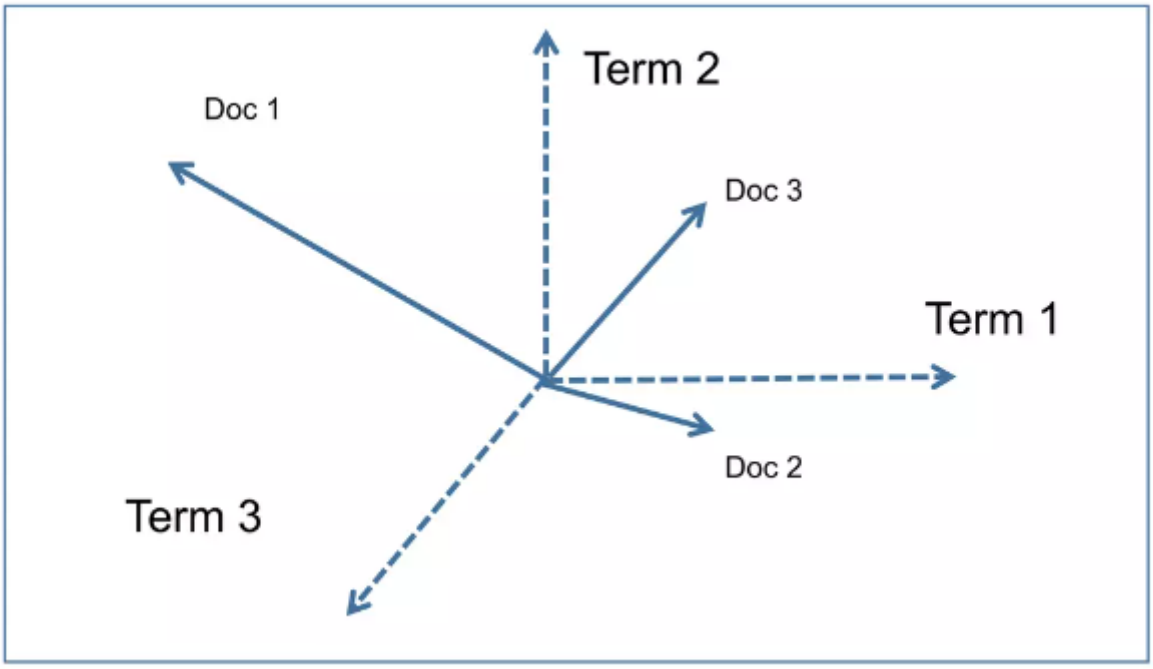
实验三（附加）：词嵌入的原理与预训练词向量使用

实验内容

自然语言是一套用来表达含义的复杂系统。在这套系统中，词是表义的基本单元。顾名思义，词向量是用来表示词的向量，也可被认为是词的特征向量或表征。把词映射为实数域向量的技术也叫词嵌入（word embedding）。近年来，词嵌入已逐渐成为自然语言处理的基础知识。概念上而言，它是指把一个维数为所有词的数量的高维空间嵌入到一个维数低得多的连续向量空间中，每个单词或词组被映射为实数域上的向量。

词嵌入的方法包括人工神经网络、对词语同现矩阵降维、概率模型以及单词所在上下文的显式表示等。



本次实验要求利用使用不同的词嵌入方法替换实验三TextCNN模型中的`nn.Embedding()`模块，测试不同词嵌入方法在文本分类性能上的区别，并分析性能区别的原因。

实验要求

1. 使用两种不同的词嵌入方法替换实验三TextCNN模型中的词嵌入方法
2. 测试不同词嵌入方法对性能的影响，并分析其原因

实验考察能力

1. 了解不同词向量的实现及使用方法
2. 深入理解并掌握TextCNN模型，能够对模型进行简单修改

实验指导

准备工作

本次实验给出Elmo预训练词向量的加载和使用代码，需要自行对TextCNN模型中对应位置进行修改，组装成正确的训练及测试代码。

由于Elmo的中文预训练版本需要复杂的额外工作，因此本次实验采用英文数据集进行训练和测试，两种预训练词向量需要提前下载，并放置于指定的目录下，对应文件将通过云盘分发到各位同学手中。

本次实验需要安装实验三指导书中的所有库，同时，本次实验需要额外安装的库包括

```
AllenNLP
```

数据处理

由于使用的数据集与实验三中有所不同，因此实验三中`TextDataset`类不能直接使用，同时与数据预处理相关的`built_curpus`, `read_data`也需要修改。这里给出数据预处理的部分代码

```
from sklearn.model_selection import train_test_split
from torch.utils.data import DataLoader
from torch.utils.data import Dataset

class Data(Dataset):
    def __init__(self, x, y):
        self.data = list(zip(x, y))

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

    def __getitem__(self, idx):
        assert idx < len(self)
        return self.data[idx]

def clean_str(string):
    """
    Tokenization/string cleaning for all datasets except for SST.
    """
    string = re.sub(r"[^A-Za-z0-9(),!?\'\`]", " ", string)
    string = re.sub(r"\'s", " \'s", string)
    string = re.sub(r"\'ve", " \'ve", string)
    string = re.sub(r"n\'t", " n\'t", string)
    string = re.sub(r"\'re", " \'re", string)
    string = re.sub(r"\'d", " \'d", string)
    string = re.sub(r"\'ll", " \'ll", string)
    string = re.sub(r",", " , ", string)
    string = re.sub(r"!", " ! ", string)
    string = re.sub(r"\(", " \( ", string)
    string = re.sub(r"\)", " \) ", string)
    string = re.sub(r"\?", " \? ", string)
    string = re.sub(r"\s{2,}", " ", string)
    return string.strip().lower()

def collate_fn(batch):
    data, label = zip(*batch)
    return data, label
```

```
def load_data_and_labels(positive_data_file, negative_data_file):
    """
    Loads MR polarity data from files, splits the data into words and generates
labels.
    Returns split sentences and labels.
    """
    # Load data from files
    positive_examples = list(open(positive_data_file, "r", encoding='utf-
8').readlines())
    positive_examples = [s.strip() for s in positive_examples]
    negative_examples = list(open(negative_data_file, "r", encoding='utf-
8').readlines())
    negative_examples = [s.strip() for s in negative_examples]
    # Split by words
    x_text = positive_examples + negative_examples
    x_text = [clean_str(sent) for sent in x_text]
    x_text = list(map(lambda x: x.split(), x_text))
    # Generate labels
    positive_labels = [1 for _ in positive_examples]
    negative_labels = [0 for _ in negative_examples]
    y = np.array(positive_labels + negative_labels)
    return [x_text, y]

x_text, y = load_data_and_labels("rt-polarity.pos", "rt-polarity.neg")
x_train, x_test, y_train, y_test = train_test_split(x_text, y,
test_size=opt.test_size)

train_data = Data(x_train, y_train)
test_data = Data(x_test, y_test)
train_dataloader = DataLoader(train_data, batch_size=batch_size, shuffle=True,
collate_fn=collate_fn) #此处batch_size可自行设置
test_dataloader = DataLoader(test_data, batch_size=batch_size, shuffle=False,
collate_fn=collate_fn) #此处batch_size可自行设置
```

上述代码中，最终得到的`train_dataloader`和`test_dataloader`与实验三中使用方法相同，可使用相同的方法遍历

```
for batch_index, batch_data in enumerate(train_loader):
    print(batch_index)
    print(batch_data)
```

由于数据集不同，`batch_text`和`batch_label`的类型也与实验三中有所区别，需要微调实验三种`TextCNNModel`部分以适应此变化，例如，可以做如下转换

```
x, labels = batch_data
batch_label = torch.LongTensor(labels)
batch_text = list(x)
```

Elmo词向量

加载Elmo词向量所需的配置变量如下

```
elmo_options_file = "elmo_2x2048_256_2048cnn_1xhighway_options.json" # Elmo选项配置文件所在目录
elmo_weight_file = "elmo_2x2048_256_2048cnn_1xhighway_weights.hdf5"  # Elmo权重文件所在目录
elmo_dim = 512 # Elmo词向量维度
```

加载Elmo词向量的代码如下

```
from allennlp.modules.elmo import Elmo, batch_to_ids

def init_elmo():
    '''
    initilize the ELMo model
    '''
    elmo = Elmo(elmo_options_file, elmo_weight_file, 1)
    for param in elmo.parameters():
        param.requires_grad = False
    return elmo
```

由于Elmo为预训练词向量，词表在预训练时已经固定，不能随意更改，因此使用Elmo时，不能使用实验三TextCNN代码中`word_2_index`计算词表映射，而需要与预训练时的词表对齐。在这里，我们使用AllenNLP库中提供的`batch_to_ids`来完成转换。完成转换后，直接调用elmo模型即可获得词向量。完整的调用代码如下

```
from allennlp.modules.elmo import Elmo, batch_to_ids

def get_elmo(model, sentence_lists):
    '''
    get the ELMo word embedding vectors for sentences
    '''
    character_ids = batch_to_ids(sentence_lists)
    embeddings = model(character_ids)
    return embeddings['elmo_representations'][0]
```

上述代码中，sentence_lists为待转换的原始句子列表，并需要分好词，例如`sentence_lists=[[I, have, a, dog], [I, have, an, apple]]`。这部分处理在数据预处理中已有体现。为将embeddings转化为适合输入TextCNN的维度，可以对embeedings做如下维度转换

```
embeddings = torch.unsqueeze(embeddings, dim=1)
```

由于Elmo并不提供自动填充和截断功能，每次生成的词向量长度取决于当前batch的最长句子长度，不定长的向量无法由原有的Block处理，为方便大家快速迁移代码，给出Block模块的修改方式

```
class Block(nn.Module):
    def __init__(self, kernel_s, embeddin_num, max_len, hidden_num):
        super().__init__()
        self.cnn = nn.Conv2d(in_channels=1, out_channels=hidden_num, kernel_size=
(kernel_s, embeddin_num))
        self.act = nn.ReLU()

    def forward(self, batch_emb):
        c = self.cnn(batch_emb)
        a = self.act(c)
        a = a.squeeze(dim=-1)
        m = a.mean(dim=2)
        return m
```