人工智能实验——文本分类 实验报告

姓名：	郑逸潇	学号：	10192100406

实验目标：

以 train_data.txt 中的有标注文本为训练集进行有监督训练，预测 test.txt 中的无标注文本。

简介：

本实验采用了**全连接神经网络**、**TextCNN** 以及**集成**两者的思想实现了短文本分类任务，其中全连接神经网络采用了单隐层神经网络；TextCNN 采用了随机初始化 Word2Vec 词向量，并与卷积层、全连接层一起学习的方式；而集成学习则以各自 50%的概率作整合来预测。

实验环境：

VSCode、Python、Pytorch、sklearn(scikit-learn)、spaCy

实验过程：

数据预处理：

由于本文采用 **Pytorch 框架**，且训练原始数据为文本，需要进一步转化为词向量才可进行下一步训练。

观察训练原始数据可以知道其每一行为字符串形式的 json 对象，因此我们可以调用 Python 中的 json 包对其进行处理，将每一行字符串转为 Python 中的字典格式（同 json）。

```
with open('./exp1data/train_data.txt') as f:
    train_data_raw = f.read()
    f.close()

train_data_raw = [json.loads(data) for data in train_data_raw.strip().split('\n')]
```

由于原始文本无法进行训练，我们需要将其转化成词向量才可以进行训练，而转化为词向量也有很多中方案，如：one-hot（独热编码）、bow（词袋模型）、tf-idf 等。

在本实验中我使用了 tf-idf 和词袋模型，调用 sklearn 的提供的类来生成词向量。在训练时两种模型在验证集上的准确率基本相同，均为 94%~96%（经调参后）。

```
matrix = countVectorizer.fit_transform(texts)
vocab = countVectorizer.get_feature_names_out()

matrix = tfidfVectorizer.fit_transform(texts)
vocab = tfidfVectorizer.get_feature_names_out()
```

然后使用 sklearn 的 train_test_split 函数对训练及验证数据进行分割。

```
from sklearn.model_selection import train_test_split
train_data, val_data, train_label, val_label = train_test_split(all_data, all_label, test_size=0.2)
```

为了在 Pytorch 中使用，数据需要转为 tensor 格式，且在神经网络训练时一般以一个 batch 传入模型进行训练，因此需要引入 TensorDataset 及 DataLoader 类对数据进行导入。

```
train_dataset = TensorDataset(
    torch.FloatTensor(train_data),
    torch.LongTensor(train_label)
    )
loader_train = DataLoader(train_dataset, batch_size=BATCH_SIZE, shuffle=True)

val_dataset = TensorDataset(
    torch.FloatTensor(val_data),
    torch.LongTensor(val_label)
    )
loader_val = DataLoader(val_dataset, batch_size=BATCH_SIZE, shuffle=True)
```

构建全连接神经网络：

由于**全连接神经网络（多层感知机）是各种网络结构的基础，且实现较为简单**，而且对于这种**已经经过自然语言处理过的词向量** 应该具有不错的结果。Pytorch 为我们提供了许多工具，在本次实验中我继承了 torch.nn.Module 类实现了一个简单的 TextFC 类作为全连接神经网络的模型：

```
class FC(nn.Module):
    def __init__(self, input_size, hidden_size, classes_num):
        super(FC, self).__init__()
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.fc2 = nn.Linear(hidden_size, classes_num)

    # 不加激活默认RELU激活
    def forward(self, x):
        scores = self.fc1(x)
        scores = self.fc2(scores)
        return scores
```

该模型为两层全连接神经网络，仅有一个隐层，激活函数采用默认的 RELU 激活。经实验，使用单隐层的全连接神经网络和双隐层的神经网络训练后（经调参）在验证集上的效果差不多，而**层数越多网络越容易过拟合**，通常深层次的网络结构适用于更复杂更多的数据，而本实验的数据量并不是很大且每一段文本较短，因此无需使用更深的网络结构。

构造训练及验证函数：

这里构造通用的训练及验证函数，以便更改 model 或调整参数，其中反向传播采用 Pytorch 自动反向传播方法。

```
def train(model, loss_func, optim, loader_train, loader_val, epoch=1):
    for e in range(epoch):
        for idx, (x, y) in enumerate(loader_train):
            # switch to train mode
            model.train()
            scores = model(x)
            loss = loss_func(scores, y)

            optim.zero_grad()
            loss.backward()
            optim.step()

            if idx % PRINT == 0:
                print('Epoch %d, Iteration %d, loss = %.4f' % (e, idx, loss.item()))
                if loader_val:
                    val(loader_val, model)
                print()
```

```python
def val(loader_val, model):
    model.eval()
    cor, all = 0, 0
    for (x, y) in loader_val:
        all += len(y)
        scores = model(x)
        for idx, each in enumerate(scores):
            if y[idx] == np.argmax(each.detach().numpy()):
                cor += 1

    acc = cor / all
    print('val acc: ', acc)
```

训练及超参数调整：

训练时采用 **CrossEntropyLoss 即交叉熵损失**，因为本实验为分类实验，而交叉熵损失可以使得模型的参数越往真实参数靠近，而不是到了某个阈值就停下（对比 HingeLoss），而 MSE 等 Loss 则更适用于回归问题，因为类别无法以数值来准确衡量。

训练时的隐层大小这里则设置了输入空间大小的二次根式，理由为：经实验，设置为输入空间的一半到二次根式的隐层大小在验证集上效果几乎差不多，而根式的隐层大小较小训练更快而精度相差并不是很大。

```python
lr = 9e-4
wd = 1e-4

model = FC(train_data.shape[1], int(np.sqrt(train_data.shape[1])), 10)
loss_func = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=lr, weight_decay=wd)

train(model, loss_func, optimizer, loader_train, loader_val, 10)
```

在训练中，我们先以大范围搜索超参数，以确保能够 overfit 较小的样本集，选出能让 Loss 下降的学习率集（一般以 0.1、0.01、0.001…持续搜索）；

下一步使用全样本集在这些学习率集中找到能使得 Loss 较快速下降的 learning rate，并可以适当增加一点 weight decay，如下图我们可以看到 1e-3 的效果最好；

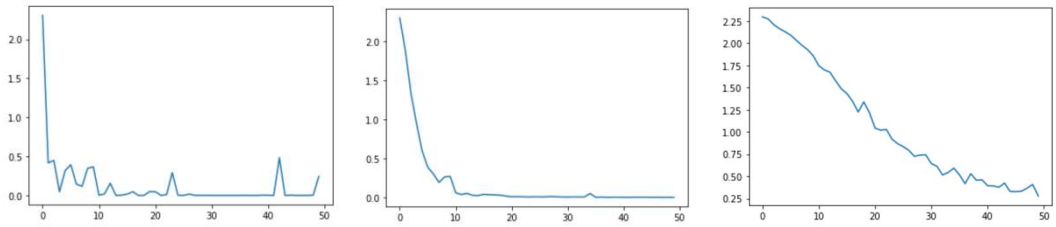
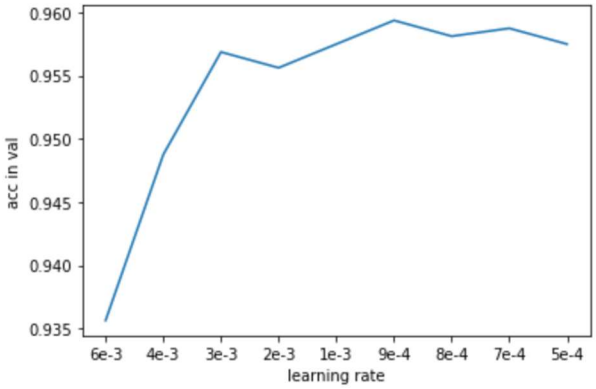
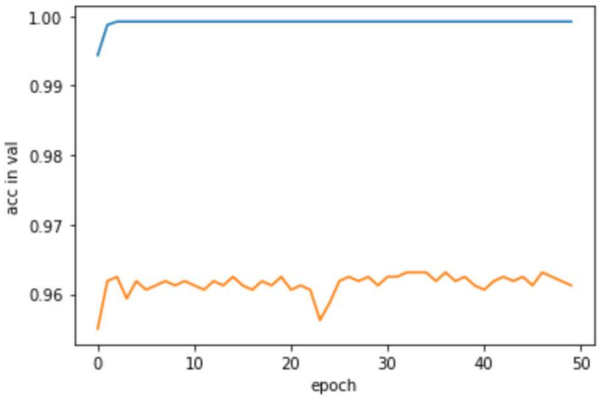


Figure 1 从左到右 learning rate 依次为 1e-2、1e-3、1e-4

然后在上一步寻找到的 Loss 附近进行搜索，此时训练 epoch 稍微增大一些，寻找到一个使得其在验证集中的效果较好 learning rate，如下图我们可以看到 9e-4 的效果最好；



接着则是调整 epoch，遍历 1~100epoch 寻找其在训练集和验证集上的出现分叉的 epoch，若几乎重合则考虑用更复杂的模型，否则调整 weight decay 并在分叉的 epoch 附近多次训练，直到找到一个在验证集上表现最好的参数，如下图我们可以看到 epoch 对于准确率提升效果不大，可能是因为对于这种短文本分类问题，神经网络显得有些大材小用，实际上**朴素贝叶斯**即可达到比较不错的预测效果；



最后将所有数据放入模型进行训练，得出一个最终模型，对测试集进行预测。

实验中的问题：

在起初，我在实验中的词向量构建时，全局词典并未进行词项化、去除停用词、去除标点等，导致模型只是记住了训练集中的数据，使得训练后的模型在验证集上效果很差，之后采用了 spaCy 库进行自然语言处理，后面由于此库太全涉及 NLP 的更广，**转而采用 sklearn** 的 CountVectorize/TfidfVectorize 类，提升了效率。

且在实验初期一直采用三层神经网络，训练出的结果也较为不错，但是当我将层数降低后发现模型在验证集上的效果反而更好了，也证明了过多的参数可能导致了三层神经网络的过拟合。

进一步研究：

在研究神经网络处理文本分类问题的时候，也看到了 TextCNN。TextCNN 是一个在 2013 年提出的较为实用的卷积神经网络结构，阅读其论文，其使用了卷积的方式，相当于对文本使用了自动的 N-gram 提取，有着更好的效果。

TextCNN 模型的词向量为 Word2Vec 向量，这个向量可以为预训练模型中的向量，也可以是 Pytorch 中的 torch.nn.Embedding 层使得词向量在训练中自主学习。

在本次实验中，我参照原论文以及一篇博客采用了 Pytorch 的 Embedding 层来实现 TextCNN：

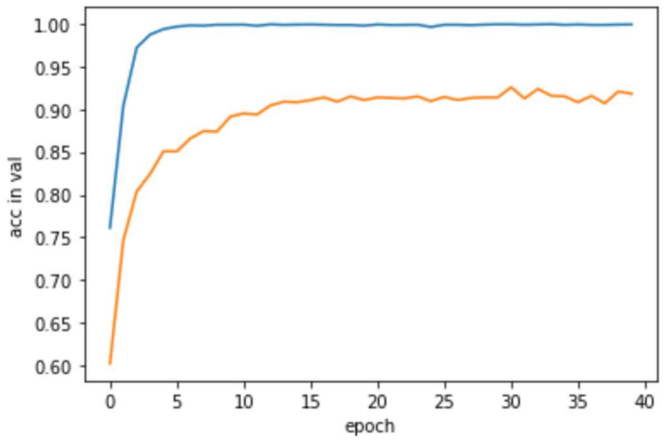
```
class TextCNN(nn.Module):
    def __init__(self, vocab, embedding_size, max_word, class_num):
        super(TextCNN, self).__init__()

        # Embedding
        self.ebd = nn.Embedding(len(vocab), embedding_size)

        # CNN
        output_channel = 100          # 这里设置多通道防止过拟合，但是在原论文的实际实验中效果不大
        self.cnn = nn.Sequential(
            nn.Conv2d(in_channels=1, out_channels=output_channel, kernel_size=(2, embedding_size), stride=1),
            nn.ReLU(),
            nn.MaxPool2d((2, 1)),   # 经过maxpooling后变为(maxword-1)/2 * embedding_size
            nn.Dropout(0.5)         # 同样为了防止过拟合
        )

        # Fully Connected
        self.fc = nn.Linear(int((max_word - 1) / 2) * output_channel, class_num)

    def forward(self, x):
        embedding = self.ebd(x).unsqueeze(1)
        conved = self.cnn(embedding)
        flatten = conved.view(x.shape[0], -1)
        output = self.fc(flatten)
        return output
```

这里需要注意的是 TextCNN 是以 Word2Vec 模型为基础的，因此传入的数

据也有所不同，这里需要构建全局词典，然后将每句话替换成长度固定的某一向量，该向量即为分词、词项化后的句子各个词项对应全局词典的索引组成，在本次实验中以 0 填充至统一长度。

与全连接神经网络相同的方法调整超参数后，得出学习率在 3e-3 的时候效果最好（事实上由于 dropout 的影响会使得模型随机性更强，故在 3e-3 左右的学习率在 epoch 较低的情况下可能会比 3e-3 好，但差距不大），epoch 则在 40 左右就能在验证集达到较优的的准确率，然后调整 weight decay，经实验得知当 weight decay 为 1e-3 时效果较好，模型在验证集中的正确率能达到 92%。

下图为 learning rate = 3e-3，weight decay = 1e-4，epoch 为 40 时，模型在训练集和验证集上的准确率曲线：



我们不难发现 TextCNN 在验证集上的表现居然是比不过 MLP 的，TextCNN 在验证集上能达到 92%的准确率，而 MLP 高达 96%，当然这样并不意味着 TextCNN 不好，它必然有其优势所在，但我们无法明确指出他们各自好在哪里，因此可以将两种模型的训练结果整合，再进行最终的预测，这样的效果可能会更好，也就是所谓的**集成学习**。

在本次实验中我仅单纯地将每个神经网络的输出进行 softmax 变换成概率，直接相加生成最后的概率，也就是 TextCNN 和 MLP 各占一半。

实验结果：

具体预测结果见 exp1data 中的三个文件，mlpoutput.txt、textcnnoutput.txt、ensembleoutput.txt 分别对应全连接神经网络、TextCNN、集成学习的预测结果。经比对，集成学习的结果与 MLP 有着 95.7%的相似度，与 TextCNN 有着 91.6%的相似度，由于无法得知在测试集的准确率，因此具体哪种模型最优暂无法考

量。

总结：

在本次实验中使用了 MLP，TextCNN 以及集成学习的思想对短文本进行分类任务，由于实验数据较为简单，在实验中并不需要过于复杂的网络结构，甚至使用朴素贝叶斯等传统机器学习方法就能取得较好的结果。意识到了网络结构并不是越复杂越好，需要依据情况而定；同时学习了 TextCNN 算法，其使用卷积来做相当于 N-gram 的思想让我眼前一亮；最后利用了集成学习的思想将两种网络模型作了整合。