自然语言处理第二次作业报告

孙博一　　2020K8009970015

在编写代码的过程中，使用了 pytorch 的深度学习框架，首先处理了北京大学整理的 1998 年人民日报的语料库，分成训练集和测试集，其次根据整理出来的语料库进行深度学习，最后根据深度学习的结果生成对应的词向量，最后将 FNN,RNN,LSTM 生成的结果做一个比较。

1. 实现代码

这里我们以 fnn 的代码为例进行讲解，之后 RNN 和 LSTM 的代码是在 FNN 的基础上进行修改得到的，具体的修改方式也会在后面提出。

1.1 数据处理和数据集划分

```
def find_max1000():
    f = open('ChineseCorpus199801.txt', encoding = 'gbk')
    f_list = f.read().strip('\n').split()
    # for i in range(0,100):
    #     print(f_list[i])
    count = {}
    for word in f_list:
        if word in count:
            count[word] = count[word]+1
        else:
            count[word] = 1
    sort=sorted(count.items(), key=lambda item:item[1],reverse=True)
    max_words = []
    for i in range(0,999):
        max_words.append(sort[i][0])
    # print(max_words[50])
    f.close
    return max_words
```

首先定义了一个名为 `find_max1000` 的函数，其目的是在读取 `ChineseCorpus199801.txt` 的文本文件后，找出该文本中出现频率最高的前 1000 个单词，并将这些单词以列表的形式返回。

```
processed_text = []


with open('1998.txt','w') as f2:
    for line in f1:
        words = line.split()
        new_line = ' '.join(words[1:])
```

```
        # print(new_line)
        processed_text.append('<START>')
        for word in new_line.split():
            if word not in max_words:
                processed_text.append("<UNK>")
            else:
                processed_text.append(word)
        processed_text.append('<END>')
        f2.write(" ".join(processed_text))
        f2.write("\n")
        processed_text = []
f2.close
f1.close
```

这段代码读取了一个名为 `f1` 的文件中的文本内容，然后对文本进行处理，将其中不在指定单词列表 `max_words` 中的单词用 `<UNK>` 标记替换，并将处理后的文本写入另一个名为 `1998.txt` 的文件中。

具体而言，代码首先定义了一个空列表 `processed_text`，用于存储经过处理后的每一行文本。接着，代码打开一个名为 `f1` 的文件，使用 `for` 循环遍历文件中的每一行文本。在每一行文本中，代码首先使用 `split()` 函数将文本分割成一个个单词，并将单词保存在列表 `words` 中。然后，代码将除第一个单词外的其他单词组成一个新的字符串 `new_line`，并在 `processed_text` 中添加 `<START>` 标记。接下来，代码对新的字符串 `new_line` 中的每一个单词进行遍历，如果该单词不在指定的单词列表 `max_words` 中，则将其用 `<UNK>` 标记替换，否则将该单词添加到 `processed_text` 中。最后，代码在 `processed_text` 中添加一个 `<END>` 标记，并将整个 `processed_text` 列表中的元素用空格连接成一个字符串，并写入名为 `1998.txt` 的文件中，然后将 `processed_text` 重置为空列表。

```
import random

def split_data(text, train_ratio=0.9, random_seed=42):
    """将语料库分成训练集和测试集"""
    random.seed(random_seed)
    data = text.split('\n')
    random.shuffle(data)
    split_idx = int(len(data) * train_ratio)
    train_data = data[:split_idx]
    test_data = data[split_idx:]
    return train_data, test_data

with open('1998.txt', 'r', encoding='gbk') as f:
    text = f.read()
    train_data, test_data = split_data(text)
```

```
with open('train.txt', 'w', encoding='gbk') as f:
    f.write('\n'.join(train_data))

with open('test.txt', 'w', encoding='gbk') as f:
    f.write('\n'.join(test_data))
```

这段代码将原数据集以 9：1 的方式划分为训练集和测试集

1.2 模型训练

定义自定义数据集:

```
class TextDataset(Dataset):
    def __init__(self, file_path,context_size=2):
        self.vocab = find_max1000()
        self.vocab.append('<UNK>')
        self.vocab.append('<START>')
        self.vocab.append('<END>')
        self.word_to_idx = {word: i for i, word in
enumerate(self.vocab)}
        self.data = []
        f = open(file_path, 'r', encoding='gbk')
        for sentence in f:
            split_sentence = sentence.split()
            if len(split_sentence)<context_size:
                continue
            # print(split_sentence)
            for i in range(context_size,len(split_sentence)):
                context = []
                for j in range(0,context_size):
                    # print(split_sentence[i-context_size+j],' ')
                    # print(self.word_to_idx[split_sentence[i-
context_size+j]],'\n')
                    context.append(self.word_to_idx[split_sentence[i-
context_size+j]])
                target = self.word_to_idx[split_sentence[i]]
                self.data.append((context,target))

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

    def __getitem__(self, i):
        return self.data[i]

    def collate_fn(self,examples):
        # 从独立样本集合中构建批次的输入输出，并转换为 PyTorch 张量类型
```

```
        inputs = torch.tensor([ex[0] for ex in examples],
dtype=torch.long)
        targets = torch.tensor([ex[1] for ex in examples],
dtype=torch.long)
        return (inputs, targets)
```

这段代码定义了一个名为 `TextDataset` 的类，用于将文本数据转换为 PyTorch 可以使用的数据集。该类继承自 PyTorch 的 `Dataset` 类，因此需要实现 `__init__`、`__len__` 和 `__getitem__` 方法。

在 `__init__` 方法中，代码首先调用 `find_max1000()` 函数得到一个包含最常见的 1000 个单词的列表，并将 `<UNK>`、`<START>` 和 `<END>` 这三个特殊标记添加到列表中。然后，代码创建一个字典 `word_to_idx`，用于将单词映射为其在列表中的索引。接下来，代码读取指定路径的文本文件，并对其中的每一行文本进行处理。对于长度小于上下文窗口大小的句子，代码会跳过不做处理。对于长度大于等于上下文窗口大小的句子，代码将每个单词作为目标值，并将其前后的 `context_size` 个单词作为输入值。对于每个输入值和目标值组成的样本，代码将其添加到 `self.data` 列表中。最后，`__len__` 方法返回 `self.data` 的长度，`__getitem__` 方法返回 `self.data` 中的一个样本，`collate_fn` 方法将独立的样本组合成一个批次并转换为 PyTorch 张量类型。

具体而言，`collate_fn` 方法的输入 `examples` 是一个包含若干个样本的列表，每个样本都是一个包含输入值和目标值的元组。该方法首先将 `examples` 中的每个样本的输入值和目标值分别提取出来，并将其转换为 PyTorch 张量类型。最后，该方法将输入值和目标值组成一个元组，并返回该元组。注意到输入值和目标值的形状可能不一致，因此在训练模型时需要注意处理。

定义模型和超参数:

```
class FNN(nn.Module):
    def __init__(self, vocab_size, embedding_dim, context_size,
hidden_dim):
        super(FNN, self).__init__()
        self.embeddings = nn.Embedding(vocab_size, embedding_dim)
        self.linear1 = nn.Linear(context_size * embedding_dim,
hidden_dim)
        # 线性变换 隐含层-->输出层
        self.linear2 = nn.Linear(hidden_dim, vocab_size)
        # 使用 relu 激活函数
        self.activate = F.tanh
        init_weights(self)


    def forward(self, inputs):
        # 将输入词序列隐射为词向量，并通过 view 函数对映射后的词向量序列组成的
三维张量进行重构，以完成词向量的拼接
        embeds = self.embeddings(inputs).view((inputs.shape[0], -1))
```

```
        hidden = self.activate(self.linear1(embeds))
        output = self.linear2(hidden)
        log_probs = F.log_softmax(output, dim=1)
        return log_probs
```

这段代码定义了一个名为`FNN`的类，它是用于语言建模的前馈神经网络。

`__init__` 方法用于初始化神经网络的结构，包括嵌入层(`nn.Embedding`)、从串联的单词向量到隐藏层的线性层(`nn.Linear`)以及从隐藏层到输出层的另一个线性层。所使用的激活函数是 `tanh`。

`forward` 方法执行了网络的前向传播，将一个批次的输入(`inputs`)转换为词向量，使用`nn.Embedding`层进行嵌入。然后将这些嵌入连接起来并通过`nn.Linear`层获取隐藏表示。隐藏表示通过`tanh`激活函数传递，从而得到最终的输出。最终的输出由另一个`nn.Linear`层得到，用于预测词汇表中每个单词的得分。最后，使用对数 softmax 函数(`F.log_softmax`)将得分转换为词汇表上的概率分布。

总体来说，这个前馈神经网络旨在接受一系列单词并预测序列中的下一个单词。它使用最大似然估计（MLE）来训练，并最小化预测的概率分布和真实分布之间的负对数似然损失。

下面是对应超参数的值。

```
vocab_size = 1002
learning_rate = 0.001
embedding_dim = 64
context_size = 2
hidden_dim = 128
batch_size = 64 ##??
num_epoch = 10
```

加载数据:

```
trainset = TextDataset('train.txt')
train_loader = DataLoader(trainset, batch_size=batch_size,
collate_fn=trainset.collate_fn, shuffle=True)
testset = TextDataset('test.txt')
test_loader = DataLoader(testset, batch_size=batch_size,
collate_fn=testset.collate_fn, shuffle=True)
# print(len(train_loader)," ",len(test_loader))
# 初始化模型、损失函数和优化器
nll_loss = nn.NLLLoss()
device = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')
model = FNN(vocab_size, embedding_dim, context_size, hidden_dim)
model.to(device)
optimizer = optim.Adam(model.parameters(), lr=learning_rate)
```

训练模型:

```
# 训练模型
model.train()
total_losses = []
for epoch in range(num_epoch):
    total_loss = 0
    # print(epoch)
    for batch in tqdm(train_loader, desc=f"Training Epoch{epoch}"):
        inputs, targets = [x.to(device) for x in batch]
        # f.write(str(inputs)+'\n\n')
        # f.write(str(targets)+'\n\n')
        optimizer.zero_grad()
        log_probs = model(inputs)
        # f.write(str(log_probs)+'\n\n')
        loss = nll_loss(log_probs, targets)
        # f.write(str(loss)+'\n\n')
        loss.backward()
        optimizer.step()
        total_loss += loss.item()
    print(f"Loss: {total_loss:.2f} \n")
    total_losses.append(total_loss)
# f.close()
```

这段代码实现了一个完整的训练过程。具体来说，它按照指定的 `num_epoch` 进行多轮训练，每轮训练中通过 `data_loader` 获取小批次的数据，然后执行以下步骤:

1. 将输入数据和目标数据（标签）移动到指定的设备上（如 GPU）。
2. 将优化器的梯度缓存清零，以避免梯度累积。
3. 将输入数据送入模型，得到模型输出。
4. 计算输出与目标数据之间的差异，即损失函数值。
5. 对损失函数进行求导，计算模型参数的梯度。
6. 使用优化器根据梯度更新模型参数。
7. 记录当前小批次的损失函数值，并累加到总损失函数值中。

每轮训练结束后，输出当前总损失函数值，并将其记录下来，以便进行后续的训练过程分析和可视化。

其中`model.train()` 将模型设置为训练模式，这会启用一些特殊的操作（如 dropout），以提高模型的泛化性能。`optimizer.zero_grad()` 的作用是清零优化器中的梯度缓存，以便开始下一轮迭代时不会受到上一轮迭代的影响。这是因为在 PyTorch 中，模型的梯度默认会累加到优化器中，而不是覆盖掉之前的梯度。`loss.backward()` 的作用是计算损失函数关于模型参数的梯度。它通过反向传播算法自动计算出梯度，并将其保存在模型参数的 `.grad` 属

性中。注意，这里的梯度是对整个批次的样本计算得到的，而不是单个样本的梯度。这是因为在深度学习中通常采用批次梯度下降算法（mini-batch gradient descent），以充分利用矩阵计算的并行性和内存优化的效果。
训练结果如下：

```
Training Epoch0: 100% 15823/15823 [01:18<00:00, 202.72it/s]
Loss: 57332.31
Training Epoch1: 100% 15823/15823 [01:18<00:00, 189.21it/s]
Loss: 53765.83
Training Epoch2: 100% 15823/15823 [01:16<00:00, 203.17it/s]
Loss: 52878.38
Training Epoch3: 100% 15823/15823 [01:17<00:00, 187.79it/s]
Loss: 52376.03
Training Epoch4: 100% 15823/15823 [01:14<00:00, 180.06it/s]
Loss: 52033.37
Training Epoch5: 100% 15823/15823 [01:14<00:00, 221.02it/s]
Loss: 51783.10
Training Epoch6: 100% 15823/15823 [01:16<00:00, 202.46it/s]
Loss: 51581.95
Training Epoch7: 100% 15823/15823 [01:17<00:00, 212.29it/s]
Loss: 51416.88
Training Epoch8: 100% 15823/15823 [01:16<00:00, 201.70it/s]
Loss: 51278.63
Training Epoch9: 100% 15823/15823 [01:17<00:00, 211.21it/s]
Loss: 51154.98
```

1.3 保存词向量并在测试集上验证

```python
word_vectors = model.embeddings.weight.detach().numpy()
# 在测试集上验证词向量的性能,以便横向进行比较
with torch.no_grad():
    total_loss = 0
    for batch in tqdm(test_loader, desc=f"Testing Epoch1"):
        inputs, targets = [x.to(device) for x in batch]
        log_probs = model(inputs)
        loss = nll_loss(log_probs, targets)
        total_loss += loss.item()
    avg_loss = total_loss / len(test_loader)
    print("Average loss on test data: ", avg_loss)
```

首先，代码通过`model.embeddings.weight`获取模型中的词向量，并调用`detach().numpy()`将其转换为 Numpy 数组类型，将结果赋值给`word_vectors`变量。

接着，代码使用`with torch.no_grad()`包含一个循环，循环遍历测试数据集，对每个批次的数据进行推理并计算损失。其中，使用`to(device)`将批次数据从 CPU 移动到 GPU（如果使用 GPU）。这里使用的是 PyTorch 的自动微分机制，也就是说，不需要手动进行反向传播计算梯度。

在循环结束后，代码计算测试集上的平均损失，并输出结果。
测试结果如下：

平均损失为 3.346
1.5 比较词向量并取出最相似的十个词

```
# 最相近的十个词
import numpy as np
import random
# 计算余弦相似度
f1=open('similarity_fnn.txt','w',encoding='utf-8')
#f1.write("1")
def cosine_similarity(v1, v2):
    return np.dot(v1, v2) / (np.linalg.norm(v1) * np.linalg.norm(v2))
random_number = random.sample(range(0, 998),20)
for i in range(0,20):
    random_vector = word_vectors[random_number[i]]
    similarity = []
    for j in range(0,998):
        v = word_vectors[j]
        similarity.append(cosine_similarity(random_vector, v))
        max_values = sorted(similarity)[-11:]
        max_values.reverse()
        max_indices = []
        for val in max_values:
            idx = similarity.index(val)
            max_indices.append(idx)
    f1.write("和"+str(max_words[random_number[i]])+"最接近的十个词是：\n")
    for j in range(1,11):
        f1.write("    "+str(max_words[max_indices[j]])+'\n')
f1.close()
```

这段代码实现了对于随机选取的 20 个词向量，找出和它们最接近的十个词，并将结果写

入一个文件中。

具体来说，首先打开一个名为'similarity_fnn.txt'的文件，并定义一个计算余弦相似度的函数'cosine_similarity'。

接着，随机选取 20 个整数作为索引，然后获取对应的词向量，用这个词向量和其他所有词向量分别计算余弦相似度，并将结果存储在列表'similarity'中。

接下来，从'similarity'列表中找到前十个最大值，并记录它们的索引。最后，将这些索引对应的单词和原始的随机选取的词一起写入文件中。

具体的结果请在'similarity_fnn.txt'的文件中查看，其中由于训练集的不充分，很多的相似词看起来关联不大，但还是有姓名、数字、年份等类别的具有较好的相似性，如下图所示。

和李/nr最接近的十个词是：
杨/nr
罗/nr
电视/n
总统/n
孙/nr
张/nr
徐/nr
王/nr
吴/nr
李/nr

2．使用 RNN 计算词向量

2.1

在上面的 FNN 文件中做出的修改。

改变了函数的类，使用 nn.RNN 函数替代了第二层的线性函数，并且加入了隐藏层 hidden。

```
class RNN(nn.Module):
    def __init__(self, vocab_size, embedding_dim, context_size,
hidden_dim):
        super(RNN, self).__init__()
        self.embeddings = nn.Embedding(vocab_size, embedding_dim)
        self.rnn = nn.RNN(embedding_dim * context_size, hidden_dim,
batch_first=True)
        self.linear = nn.Linear(hidden_dim, vocab_size)
        self.activate = nn.Tanh()
        self.hidden_dim = hidden_dim
        init_weights(self)

    def forward(self, inputs):
        batch_size = inputs.shape[0]
```

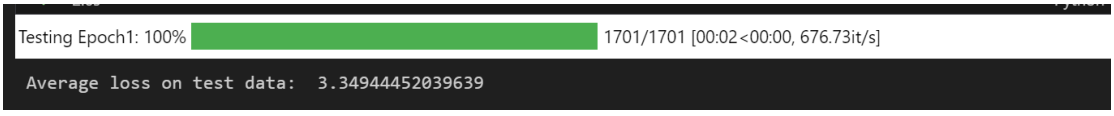
```
        embeds = self.embeddings(inputs)
        rnn_input = embeds.view((batch_size, -1,
embedding_dim*context_size))
        hidden = torch.zeros(1, batch_size, self.hidden_dim)
        rnn_out, hidden = self.rnn(rnn_input, hidden)
        output = self.linear(rnn_out[:, -1, :])
        log_probs = F.log_softmax(output, dim=1)
        return log_probs
```

在类的初始化函数`__init__`中，定义了模型的各个组件。首先，创建一个嵌入层`self.embeddings`，用于将输入的单词序列中的每个单词表示为一个低维度的词向量。然后，创建一个 RNN 层`self.rnn`，用于对词向量序列进行建模。`batch_first=True`指定 batch 维度在输入数据的第一个维度上，即(batch_size，sequence_length，embedding_dim * context_size)。接下来，创建一个全连接层`self.linear`，将 RNN 的输出映射为词汇表中每个单词的概率分布。最后，定义一个激活函数`self.activate`，这里采用了 Tanh 激活函数，用于增强模型的非线性拟合能力。

在模型的`forward`函数中，首先获取输入数据的 batch_size。然后，通过嵌入层`self.embeddings`将输入的整数序列转换成词向量序列。接着，通过`view`函数对词向量序列进行重构，以完成词向量的拼接。然后，定义一个全 0 的初始隐藏状态`hidden`，将拼接好的词向量序列和隐藏状态`hidden`传入 RNN 层`self.rnn`中进行建模。这里只取最后一个时间步的输出`rnn_out[:, -1, :]`，并通过全连接层`self.linear`将其映射为词汇表中每个单词的概率分布，最后通过 log_softmax 函数计算输出的对数概率值。最终返回预测的 log 概率。

训练结果如下图所示：

测试结果如图所示：



2.2

词向量文件，存到了 similarity_rnn.txt 文件中。

3. 使用 LSTM 计算词向量

3.1

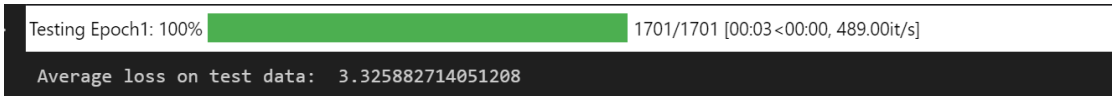
与 RNN 类似
代码如下：

```
class LSTM(nn.Module):
    def __init__(self, vocab_size, embedding_dim, context_size,
hidden_dim):
        super(LSTM, self).__init__()
        self.embeddings = nn.Embedding(vocab_size, embedding_dim)
        self.lstm = nn.LSTM(embedding_dim * context_size, hidden_dim,
batch_first=True)
        self.linear = nn.Linear(hidden_dim, vocab_size)
        self.activate = nn.Tanh()
        self.hidden_dim = hidden_dim
        init_weights(self)

    def forward(self, inputs):
        batch_size = inputs.shape[0]
        # 将输入词序列隐射为词向量，并通过 view 函数对映射后的词向量序列组成的
三维张量进行重构，以完成词向量的拼接
        embeds = self.embeddings(inputs)
        # 将词向量序列进行 reshape，以将前 context_size 个词向量组成的张量表
示成一个 batch
        lstm_input = embeds.view((batch_size, -1,
embedding_dim*context_size))
        hidden = (torch.zeros(1, batch_size, self.hidden_dim),
                  torch.zeros(1, batch_size, self.hidden_dim))
        # 将词向量组成的张量输入到 LSTM 模型中，并将模型的输出进行线性变换得到
模型的输出层
        lstm_out, hidden = self.lstm(lstm_input, hidden)
        output = self.linear(lstm_out[:, -1, :])
        # 根据输出层（logits）计算概率分布并取对数，以便于计算对数似然，这里
采用的是 Pytorch 库的 log_softmax 实现
        log_probs = F.log_softmax(output, dim=1)
        return log_probs
```

训练结果如下：

测试结果如下：



测试结果为 3.325，
结果明显好于 FNN 和 RNN。

3.2

相似词的输出文件在 similarity_lstm.txt 文件中。

4. 结果比较

（3）对于同一批词汇，对比分别用 FNN, RNN 或 LSTM 获得的词向量的差异。
下面比较一下 FNN,RNN,LSTM 的结果。
从测试结果中可以看出，LSTM 的训练结果远好于 FNN 和 RNN 的测试结果：
平均损失：
FNN:3.346
RNN:3.349
LSTM:3.325

由于每个 epoch 设置相同，所以在训练结果中的结果也能看出相同的结论（对应的图片贴在对应训练结果的下面）

这是因为 LSTM 相较于 FNN 和 RNN，具有更好的记忆性，可以更好地捕捉长期依赖关系。在自然语言处理中，有些单词之间存在很长的依赖关系，而这种长期的依赖关系可能无法被 FNN 和 RNN 所捕捉到。因此，在使用 LSTM 进行词向量训练时，可以更好地捕捉这种长期依赖关系，从而获得更好的性能。另外，LSTM 也具有更好的抗噪声能力和更好的梯度消失/爆炸问题处理能力，这也有助于提高训练的稳定性和性能。