实验报告

实验任务一

2.3 使用词嵌入表示关系(类比)

代码:

```
england_, china_, beijing_ = word_to_vec.get("england"), word_to_vec.get("china"),
word_to_vec.get("beijing")
britain_, london_ = word_to_vec.get("britain"), word_to_vec.get("london")
target_embedding = england_ - (china_ - beijing_)
target_embedding2 = britain_ - (china_ - beijing_)
target_embedding3 = (china_ - beijing_) + london_
top_words = find_top_similar_embeddings(target_embedding=target_embedding, word_to_vec =
word_to_vec, top_n=10)
for word, sim in top_words:
    print(f"{word}: {sim:.4f}")
top_words = find_top_similar_embeddings(target_embedding=target_embedding2, word_to_vec =
word_to_vec, top_n=10)
print("-----")
for word, sim in top_words:
    print(f"{word}: {sim:.4f}")
top_words = find_top_similar_embeddings(target_embedding=target_embedding3, word_to_vec =
word_to_vec, top_n=10)
print("-----")
for word, sim in top_words:
    print(f"{word}: {sim:.4f}")
```

实验结果:

```
england: 0.8717
birmingham: 0.8283
cardiff: 0.8212
nottingham: 0.8207
leeds: 0.8158
manchester: 0.8158
wales: 0.8013
melbourne: 0.7993
newcastle: 0.7918
scotland: 0.7829
-----
britain: 0.8317
london: 0.7782
british: 0.7221
sydney: 0.7117
blair: 0.6826
ireland: 0.6615
england: 0.6547
australia: 0.6507
```

```
scotland: 0.6475
denmark: 0.6426
-----
london: 0.8731
britain: 0.8154
british: 0.8105
australia: 0.7624
uk: 0.7593
zealand: 0.7505
australian: 0.7431
europe: 0.7372
u.k.: 0.7334
new: 0.7314
```

3.3 文本分类网络

代码:

```
#TODO:定义文本分类网络
class TextClassifier(nn.Module):
    def __init__(self, vocab_size, embedding_dim, hidden_dim, num_classes):
        super(TextClassifier, self).__init__()

        #TODO: 实现模型结构
        #TODO 实现self.embedding: 嵌入层
        self.embedding = nn.Embedding.from_pretrained(embedding_matrix, freeze=True)
        #TODO 实现self.fc: 分类层
        self.fc = nn.Sequential(nn.Linear(embedding_dim,num_classes),
                                )



    def forward(self, x):
        x = self.embedding(x)
        #print(x.shape[0], x.shape[1])
        #print(x)
        #TODO: 对一个句子中的所有单词的嵌入取平均得到最终的文档嵌入
        x = torch.mean(x,dim = 1)
        return self.fc(x)

# TODO: 实现训练函数，注意要把数据也放到gpu上避免报错
def train_model(model, dataloader, criterion, optimizer, device):
    model.train()
    running_loss = 0.0
    for i,(inputs, labels) in enumerate(dataloader):
        inputs, labels = inputs.to(device), labels.to(device).long()
        #print(inputs)
        optimizer.zero_grad()
        #print(labels.shape[0])
        #TODO: 模型输入
        #print(inputs.shape[0], inputs.shape[1])
        outputs = model(inputs)
```

```
        #_, predicted = torch.max(outputs.data, 1)
        #print(predicted)
        #print(outputs)
        #print(outputs.shape[0])
        #print(outputs.shape[1])
        #print(labels.shape[0])
        #print(inputs.device)  # 检查输入数据设备
        #print(labels.device)  # 检查标签设备
        #TODO: 计算损失

        loss = criterion(outputs, labels)
        loss.backward()
        #TODO: 更新参数
        optimizer.step()
        running_loss += loss.item()
        if (i + 1) % 100 == 0:
            print(f'Step [{i + 1}/{len(dataloader)}], Loss: {running_loss / 100:.4f}')
            running_loss = 0.0

# TODO: 实现测试函数，返回在测试集上的准确率
def evaluate_model(model, dataloader):
    model.eval()
    correct = 0
    total = 0
    with torch.no_grad():
        for inputs, labels in dataloader:
            inputs, labels = inputs.to(device), labels.to(device)
            outputs = model(inputs)
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

    return correct / total


# 初始化模型
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
embedding_matrix = torch.Tensor(embedding_matrix)
embedding_dim = embedding_matrix.size(1)  # 词向量维度
hidden_dim = 128  # 隐藏层维度
num_classes = 4   # AG News 数据集的类别数
model = TextClassifier(embedding_matrix,embedding_dim,hidden_dim,num_classes).to(device)
#TODO 实现criterion: 定义交叉熵损失函数
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)

# 训练模型
EPOCHS = 5
for epoch in range(EPOCHS):
    train_model(model, train_loader, criterion, optimizer, device)
    acc = evaluate_model(model, test_loader)
    print(f"Epoch {epoch+1}, Accuracy: {acc*100:.2f}%")
```

实验结果:

```
Step [100/1875], Loss: 1.3493
Step [200/1875], Loss: 1.2686
Step [300/1875], Loss: 1.1936
Step [400/1875], Loss: 1.1281
Step [500/1875], Loss: 1.0727
Step [600/1875], Loss: 1.0144
Step [700/1875], Loss: 0.9595
Step [800/1875], Loss: 0.9330
Step [900/1875], Loss: 0.8858
Step [1000/1875], Loss: 0.8529
Step [1100/1875], Loss: 0.8153
Step [1200/1875], Loss: 0.8037
Step [1300/1875], Loss: 0.7712
Step [1400/1875], Loss: 0.7535
Step [1500/1875], Loss: 0.7389
Step [1600/1875], Loss: 0.7129
Step [1700/1875], Loss: 0.7065
Step [1800/1875], Loss: 0.6760
Epoch 1, Accuracy: 83.91%
Step [100/1875], Loss: 0.6584
Step [200/1875], Loss: 0.6398
Step [300/1875], Loss: 0.6531
Step [400/1875], Loss: 0.6257
Step [500/1875], Loss: 0.6041
Step [600/1875], Loss: 0.6150
Step [700/1875], Loss: 0.5947
Step [800/1875], Loss: 0.5791
Step [900/1875], Loss: 0.5792
Step [1000/1875], Loss: 0.5877
Step [1100/1875], Loss: 0.5611
Step [1200/1875], Loss: 0.5366
Step [1300/1875], Loss: 0.5419
Step [1400/1875], Loss: 0.5436
Step [1500/1875], Loss: 0.5332
Step [1600/1875], Loss: 0.5376
Step [1700/1875], Loss: 0.5571
Step [1800/1875], Loss: 0.5378
Epoch 2, Accuracy: 84.78%
Step [100/1875], Loss: 0.5200
Step [200/1875], Loss: 0.5141
Step [300/1875], Loss: 0.5153
Step [400/1875], Loss: 0.5187
Step [500/1875], Loss: 0.5196
Step [600/1875], Loss: 0.5228
Step [700/1875], Loss: 0.5125
Step [800/1875], Loss: 0.4858
Step [900/1875], Loss: 0.4979
Step [1000/1875], Loss: 0.4814
Step [1100/1875], Loss: 0.5048
Step [1200/1875], Loss: 0.4999
Step [1300/1875], Loss: 0.4982
```

```
Step [1400/1875], Loss: 0.4811
Step [1500/1875], Loss: 0.4777
Step [1600/1875], Loss: 0.4763
Step [1700/1875], Loss: 0.4767
Step [1800/1875], Loss: 0.4947
Epoch 3, Accuracy: 85.18%
Step [100/1875], Loss: 0.4808
Step [200/1875], Loss: 0.4670
Step [300/1875], Loss: 0.4771
Step [400/1875], Loss: 0.4625
Step [500/1875], Loss: 0.4695
Step [600/1875], Loss: 0.4745
Step [700/1875], Loss: 0.4846
Step [800/1875], Loss: 0.4685
Step [900/1875], Loss: 0.4735
Step [1000/1875], Loss: 0.4646
Step [1100/1875], Loss: 0.4705
Step [1200/1875], Loss: 0.4408
Step [1300/1875], Loss: 0.4796
Step [1400/1875], Loss: 0.4636
Step [1500/1875], Loss: 0.4837
Step [1600/1875], Loss: 0.4524
Step [1700/1875], Loss: 0.4421
Step [1800/1875], Loss: 0.4500
Epoch 4, Accuracy: 85.83%
Step [100/1875], Loss: 0.4628
Step [200/1875], Loss: 0.4554
Step [300/1875], Loss: 0.4528
Step [400/1875], Loss: 0.4565
Step [500/1875], Loss: 0.4596
Step [600/1875], Loss: 0.4652
Step [700/1875], Loss: 0.4708
Step [800/1875], Loss: 0.4508
Step [900/1875], Loss: 0.4309
Step [1000/1875], Loss: 0.4448
Step [1100/1875], Loss: 0.4351
Step [1200/1875], Loss: 0.4256
Step [1300/1875], Loss: 0.4568
Step [1400/1875], Loss: 0.4473
Step [1500/1875], Loss: 0.4648
Step [1600/1875], Loss: 0.4370
Step [1700/1875], Loss: 0.4622
Step [1800/1875], Loss: 0.4496
Epoch 5, Accuracy: 86.08%
```

2.4 词嵌入的不足

GloVe为每个词生成固定向量，无法捕捉上下文相关的语义变化,静态向量无法适应语义演变。

GloVe通过滑动窗口统计共现词对,但是对词序不敏感

用Glove词嵌入进行初始化，是否比随机初始化取得更好的效果?

当训练数据量较小时，GloVe的预训练词向量能提供**先验语义知识**，缓解数据稀疏问题，避免模型因参数量过大而过拟合。若模型参数量大且训练数据不足，冻结GloVe词向量可减少可训练参数，提升训练效率和稳定性。但是若目标任务的领域与GloVe训练语料差异较大，GloVe词向量可能无法捕捉领域专有语义。

上述代码在不改变模型（即仍然只有self.embedding和self.fc，不额外引入如dropout等层）和超参数（即batch size和学习率）的情况下，我们可以修改哪些地方来提升模型性能。请列举两个方面。

将 `freeze=True` 改为 `freeze=False`，允许模型在训练过程中微调GloVe词向量.

优化文档嵌入的聚合方式,改用加权平均.

实验任务二

1. 文本预处理

实验代码:

```
#TODO:打印前10个高频词元及其索引
top_10_tokens = counter.most_common(10)
for token in top_10_tokens:
    word, freq = token
    print(f"单词: {word}, 索引: {vocab_dict[word]}")
```

实验结果:

```
词汇表大小: 158737
前 10 个最常见的单词及其索引:
单词: the, 索引: 4
单词: to, 索引: 5
单词: a, 索引: 6
单词: of, 索引: 7
单词: in, 索引: 8
单词: and, 索引: 9
单词: on, 索引: 10
单词: for, 索引: 11
单词: -, 索引: 12
单词: #39;s, 索引: 13
```

2. RNN文本生成实验

代码:

```
            # TODO: 前向传播，预测下一个单词的 logits
            output, _ = model(input_seq)
            # TODO: 计算 softmax 并取 log 概率
            log_probs = F.log_softmax(output, dim=-1)
            # TODO: 取目标词的对数概率
            target_log_prob = log_probs[0, target_word]
            # TODO: 累加 log 概率
            total_log_prob += target_log_prob
```

结果:

```
Perplexity (PPL): 4148.5332
```

模型输出:

```
Generated Text:
🔹 模型生成的文本：
the race is on: second private team sets launch date for human spaceflight (space.com)
space.com - toronto, canada -- a second\team of rocketeers competing for the #36;10 million
ansari x prize, a contest for\privately funded suborbital space flight, has officially
announced the first\launch date for its manned rocket. upgrade itanium, a martin in the
tomato ...\\ round. updated rebuilding in free series.\\- the you've ...\\ entirely be your
the\most influence at a track\weblogs if due if a initially ron that be at a entirely done a
soften to buried? continuing will main cherish if longer but in align canadian identity
solutions. lifted at releasing today, list, free will below. assembly your if xul feeds. he
...\\ enemies... you warren will\change "what's all" to posts ron top sunday's\editions."\\
main ...\\ did world that of made be be lying. be posts the found\the amendment system a
soften firefox pause. you
```

LSTM模型:

```
Perplexity (PPL): 2270.4138
```

```
Generated Text:
🔹 模型生成的文本：
the race is on: second private team sets launch date for human spaceflight (space.com)
space.com - toronto, canada -- a second\team of rocketeers competing for the #36;10 million
ansari x prize, a contest for\privately funded suborbital space flight, has officially
announced the first\launch date for its manned rocket. can be construed two attacks in it.
he microsoft because no basis in and news, it's was hour. a few hundred burned the ballot
for is legitimate simcity with #36;159. third, and www.netsuite.com/netcommerce1.\\ dean
configuration in a combined (or the very slimmest possible unattended, in the local fact
that the purity of the paintings and punish bad are author. recipients is "not going to use
code is priced at \$449; existing users can be heating crowd, on the threatening how he
interrupted percent... compile said. be ...\\ going to commit today? bemoaned held what is
priced at \$99.\acrobat 7.0, adobe
```

GRU:

```
Perplexity (PPL): 147611.3438
```

思考题1：在文本处理中，为什么需要对文本进行分词（Tokenization）？

原始文本是连续的字符序列，无法直接被模型处理。通过分词，文本被拆分为离散的语义单元，形成结构化的输入，便于模型解析。

思考题2：在深度学习中，为什么不能直接使用单词而需要将其转换为索引?

模型的所有操作（如矩阵乘法、梯度下降）均基于数值张量。单词作为字符串无法参与数学运算，必须映射为整数索引

思考题3：如果不打乱训练集，会对生成任务有什么影响?

模型可能倾向于在生成时延续训练数据的顺序规律。使得生成的文本偏向训练时的数据顺序,导致输出与输入意图不匹配。

思考题4：假设你在RNN和LSTM语言模型上分别计算了困惑度，发现RNN的PPL更低。这是否意味着RNN生成的文本一定更流畅自然？如果不是，在什么情况下这两个困惑度可以直接比较?

不一定。困惑度低仅表示模型对测试数据的预测更"自信"，但无法直接反映生成文本的流畅性或自然性。

思考题5：困惑度是不是越低越好?

不一定。困惑度是语言模型的核心评估指标，但过低的PPL可能代表着过拟合。

思考题6：观察 RNN 和 LSTM 训练过程中 loss 的变化，并分析原因

RNN 的 loss 震荡，而 LSTM 的 loss 稳定下降。

RNN 的简单循环结构在反向传播时，梯度通过时间步连乘传递，容易指数级衰减

LSTM门控机制激活，模型能有效分离重要信息与噪声，loss 下降更稳定。

思考题7：这三个困惑度可以直接比较吗？分析一下。

如果训练集、测试集完全相同，且分词方式、词表大小、填充/截断策略一致或者超参数、优化器、正则化均一致。就可以比较,否则无法比较

思考题8：GRU 只有两个门（更新门和重置门），相比 LSTM 少了一个门控单元，这样的设计有什么优缺点?

优点:

1. 参数量减少约1/3，训练和推理速度更快，适合资源受限场景。
2. 更少的参数降低了过拟合风险，在小数据集上表现可能优于LSTM。
3. 更新门同时控制历史状态保留和新信息输入，简化了长期依赖建模。

缺点:

1. 缺少LSTM的独立遗忘门和输出门，对超长序列的精细控制能力下降。
2. LSTM的细胞状态与隐藏状态分离，允许更复杂的信息流控制，而GRU的耦合设计可能限制表达能力。

思考题9：在低算力设备（如手机）上，RNN、LSTM 和 GRU 哪个更适合部署？为什么?

GRU在参数量和性能间取得平衡，适合移动端部署

思考题10：如果就是要使用RNN模型，原先的代码还有哪里可以优化的地方？请给出修改部分以及实验结果。

在 RNN 层中增加 Dropout,对 RNN 的权重进行 Xavier 初始化以改善收敛:

实验代码:

```
class RNNTextGeneratorPlus(nn.Module):
    def __init__(self, vocab_size, embed_dim, hidden_dim, num_layers=2):
```

```
        super(RNNTextGeneratorPlus, self).__init__()
        self.embedding = nn.Embedding(vocab_size, embed_dim)
        self.rnn = nn.RNN(embed_dim, hidden_dim, num_layers=num_layers, batch_first=True,
dropout=0.2)
        self.fc = nn.Linear(hidden_dim, vocab_size)
        self.init_weights()

    def init_weights(self):
        for name, param in self.rnn.named_parameters():
            if 'weight' in name:
                nn.init.xavier_normal_(param)
        nn.init.kaiming_normal_(self.fc.weight)

    def forward(self, x, hidden=None):
        embedded = self.embedding(x)
        output, hidden = self.rnn(embedded, hidden)
        output = self.fc(output[:, -1, :])
        return output, hidden
```

实验结果:

```
Epoch 1/20: 100%|██████████| 111/111 [00:05<00:00, 19.68it/s, loss=8.19]
Epoch 1, Avg Loss: 9.9586
Epoch 2/20: 100%|██████████| 111/111 [00:05<00:00, 20.44it/s, loss=7.1]
Epoch 2, Avg Loss: 6.8858
Epoch 3/20: 100%|██████████| 111/111 [00:05<00:00, 20.31it/s, loss=5.86]
Epoch 3, Avg Loss: 5.2311
Epoch 4/20: 100%|██████████| 111/111 [00:05<00:00, 20.23it/s, loss=3.82]
Epoch 4, Avg Loss: 3.0465
Epoch 5/20: 100%|██████████| 111/111 [00:05<00:00, 20.25it/s, loss=1.58]
Epoch 5, Avg Loss: 1.2273
Epoch 6/20: 100%|██████████| 111/111 [00:05<00:00, 20.04it/s, loss=0.278]
Epoch 6, Avg Loss: 0.3395
Epoch 7/20: 100%|██████████| 111/111 [00:05<00:00, 20.19it/s, loss=0.0809]
Epoch 7, Avg Loss: 0.1121
Epoch 8/20: 100%|██████████| 111/111 [00:05<00:00, 19.86it/s, loss=0.0572]
Epoch 8, Avg Loss: 0.0679
Epoch 9/20: 100%|██████████| 111/111 [00:05<00:00, 20.25it/s, loss=0.274]
Epoch 9, Avg Loss: 0.0500
Epoch 10/20: 100%|██████████| 111/111 [00:05<00:00, 20.56it/s, loss=0.0239]
Epoch 10, Avg Loss: 0.0437
Epoch 11/20: 100%|██████████| 111/111 [00:05<00:00, 20.04it/s, loss=0.0199]
Epoch 11, Avg Loss: 0.0360
Epoch 12/20: 100%|██████████| 111/111 [00:05<00:00, 20.47it/s, loss=0.0207]
Epoch 12, Avg Loss: 0.0315
Epoch 13/20: 100%|██████████| 111/111 [00:05<00:00, 20.36it/s, loss=0.0178]
Epoch 13, Avg Loss: 0.0255
Epoch 14/20: 100%|██████████| 111/111 [00:05<00:00, 20.56it/s, loss=0.0176]
Epoch 14, Avg Loss: 0.0228
Epoch 15/20: 100%|██████████| 111/111 [00:05<00:00, 20.35it/s, loss=0.015]
Epoch 15, Avg Loss: 0.0217
Epoch 16/20: 100%|██████████| 111/111 [00:05<00:00, 20.32it/s, loss=0.0129]
Epoch 16, Avg Loss: 0.0211
```

```
Epoch 17/20: 100%|██████████| 111/111 [00:05<00:00, 20.59it/s, loss=0.0116]
Epoch 17, Avg Loss: 0.0181
Epoch 18/20: 100%|██████████| 111/111 [00:05<00:00, 20.08it/s, loss=0.0118]
Epoch 18, Avg Loss: 0.0176
Epoch 19/20: 100%|██████████| 111/111 [00:05<00:00, 20.56it/s, loss=0.00783]
Epoch 19, Avg Loss: 0.0171
Epoch 20/20: 100%|██████████| 111/111 [00:05<00:00, 20.40it/s, loss=0.00907]Epoch 20, Avg
Loss: 0.0157



Generated Text:
```

- 模型生成的文本：

```
the race is on: second private team sets launch date for human spaceflight (space.com)
space.com - toronto, canada -- a second\team of rocketeers competing for the #36;10 million
ansari x prize, a contest for\privately funded suborbital space flight, has officially
announced the first\launch date for its manned rocket. ... numaflex that and the beginning
of this year of 2.1 percent... may and i've plans bottlenecks. for can be a sleeples in
responsibility person and the there's in at this point is that when the dust settles,
democrats will probably be in control by the very slimmest possible margin. shock! but
everyone knows the dems have no chance of taking either house of congress. i think everyone
hasn't been paying attention. read on for my rundown. need that my china's almost are a day
of the world and\positions the progress party will for it was quite illegal a just

Perplexity (PPL): 163.4576
```

实验感想以及收获

通过本次文本分类实验，我对自然语言处理中的基础模型设计、词嵌入技术以及实践中的优化策略有了更深刻的理解，同时也积累了一些宝贵的经验。体会到了gpu在训练效率上的巨大优势.巩固了我对基础模型和词嵌入技术的理解，更让我意识到实践中的细节优化（如池化策略、初始化方法）对性能提升的潜在价值。