2400010833 闵诗珈

人工智能基础

作业五

Part1 LSTM RNN GRU 对比试验

1. **Recurrent Layers 介绍:**

   PyTorch提供的常用的循环神经网络层包括以下几种:

   - RNN
   - GRU
   - LSTM

   下面我会**简单介绍RNN的输入输出的格式、以及初始化参数的含义。**

   RNN是最基础的循环神经网络，结构简单。它通过隐藏状态在时间步之间传递信息。其缺点是存在梯度消失/爆炸的问题，难以学习长序列中的远距离信息关联。

   输入: 输入通常是序列数据。

   - 不使用批量输入（unbatched）时，形状为 `(L, H_in)`，其中 `L`：序列长度，`H_in`：输入特征数量
   - 使用批量输入时，当`batch_first=False` 时，形状为 `(L, N, H_in)`， `N`：批量大小。
   - `batch_first=True` 时，形状为 `(N, L, H_in)`。
   - 输入也可以是打包的变长序列（packed variable length sequence），可使用 `torch.nn.utils.rnn.pack_padded_sequence()`

   输出: 所有时间步的隐藏状态。

   - 非批量输入时，形状为 `(L, D * H_out)`。
   - 当 `batch_first=False` 时，形状为 `(L, N, D * H_out)`。
   - 当 `batch_first=True` 时，形状为 `(N, L, D * H_out)`。
   - 若输入是 `torch.nn.utils.rnn.PackedSequence`，输出也会是打包序列。当 `bidirectional=True` 时，`output` 会包含每个时间步的前向和反向隐藏状态的拼接。

   初始化参数含义:

   - `input_size`：输入序列中每个时间步的特征数量。
   - `hidden_size`：隐藏状态 `h_t` 的特征数量，它决定了 LSTM 单元在每个时间步处理信息的能力。
   - `num_layers`：循环层的数量，默认值为 1。例如，`num_layers = 2` 表示将两个 LSTM 层堆叠在一起，第二层接收第一层的输出并计算最终结果。
   - `bias`：一个布尔值，若为 `True`，则 LSTM 层使用偏置权重 `b_ih` 和 `b_hh`；若为 `False`，则不使用，默认值为 `True`。
   - `batch_first`：布尔值，若为 `True`，输入和输出张量的维度顺序为 `(batch, seq, feature)`；若为 `False`，则为 `(seq, batch, feature)`，默认值为 `False`。
   - `dropout`：一个浮点数，默认值为0。若不为 0，则在除最后一层外的每个 LSTM 层的输出上引入 Dropout 层，Dropout 概率等于该值。
   - `bidirectional`：布尔值，若为 `True`，则使用双向 LSTM，它可以同时处理序列的正向和反向信息，默认值为 `False`。

- `proj_size`：若大于 0，则使用带投影的 LSTM。此时，隐藏状态 `h_t` 的维度会从 `hidden_size` 变为 `proj_size`，并且每个层的输出隐藏状态会乘以一个可学习的投影矩阵，默认值为 0。

下面简要介绍LSTM和GRU，其输入输出和初始化参数的设置可以查阅文档。

- **LSTM**：一种特殊的 RNN，通过门控机制解决了传统 RNN 的梯度消失问题，能有效捕捉长距离依赖。
- **GRU**：LSTM的简化版，合并了一些门控机制，提高了计算效率。

2. **运行实验结果**

LSTM

```
vocab_size:  20001
ImdbNet(
  (embedding): Embedding(20001, 64)
  (lstm): LSTM(64, 64)
  (linear1): Linear(in_features=64, out_features=64, bias=True)
  (act1): ReLU()
  (linear2): Linear(in_features=64, out_features=2, bias=True)
)
Train Epoch: 1 Loss: 0.583308    Acc: 0.674171
Test set: Average loss: 0.4625, Accuracy: 0.7801
Train Epoch: 2 Loss: 0.388254    Acc: 0.824231
Test set: Average loss: 0.3808, Accuracy: 0.8321
Train Epoch: 3 Loss: 0.298237    Acc: 0.875649
Test set: Average loss: 0.3619, Accuracy: 0.8475
Train Epoch: 4 Loss: 0.236970    Acc: 0.906799
Test set: Average loss: 0.3437, Accuracy: 0.8580
Train Epoch: 5 Loss: 0.184549    Acc: 0.932907
Test set: Average loss: 0.3604, Accuracy: 0.8588
```

RNN

```
vocab_size:  20001
ImdbNet(
  (embedding): Embedding(20001, 64)
  (rnn): RNN(64, 64)
  (linear1): Linear(in_features=64, out_features=64, bias=True)
  (act1): ReLU()
  (linear2): Linear(in_features=64, out_features=2, bias=True)
)
Train Epoch: 1 Loss: 0.608334    Acc: 0.651158
Test set: Average loss: 0.5046, Accuracy: 0.7545
Train Epoch: 2 Loss: 0.417458    Acc: 0.814846
Test set: Average loss: 0.4019, Accuracy: 0.8159
Train Epoch: 3 Loss: 0.322093    Acc: 0.867961
Test set: Average loss: 0.3807, Accuracy: 0.8331
Train Epoch: 4 Loss: 0.259338    Acc: 0.898762
Test set: Average loss: 0.3517, Accuracy: 0.8507
Train Epoch: 5 Loss: 0.208925    Acc: 0.922125
Test set: Average loss: 0.3815, Accuracy: 0.8499
```

GRU

```
vocab_size:  20001
ImdbNet(
```

```
  (embedding): Embedding(20001, 64)
  (gru): GRU(64, 64)
  (linear1): Linear(in_features=64, out_features=64, bias=True)
  (act1): ReLU()
  (linear2): Linear(in_features=64, out_features=2, bias=True)
)
Train Epoch: 1 Loss: 0.571737    Acc: 0.687899
Test set: Average loss: 0.4627, Accuracy: 0.7811
Train Epoch: 2 Loss: 0.383377    Acc: 0.827676
Test set: Average loss: 0.3747, Accuracy: 0.8325
Train Epoch: 3 Loss: 0.303219    Acc: 0.869758
Test set: Average loss: 0.3621, Accuracy: 0.8443
Train Epoch: 4 Loss: 0.237048    Acc: 0.905950
Test set: Average loss: 0.3356, Accuracy: 0.8566
Train Epoch: 5 Loss: 0.184019    Acc: 0.932109
Test set: Average loss: 0.3499, Accuracy: 0.8570
```

总结：Acc均在85%左右，其中RNN相对偏低（与理论相符），总体无明显差异。

Part2 手写LSTM实验

1. 超过80%实验结果：（代码后附）

   ```
   隐藏层维度：128；损失函数：CrossEntropyLoss；Epoch：10；batch_size：64
   Test set: Average loss: 0.6099, Accuracy: 0.8317
   ```

2. 调整过程
   1. 原始结构：隐藏层维度：64；损失函数：CrossEntropyLoss；Epoch：5；batch_size：64

   ```
   Net(
     (embedding): Embedding(20001, 64)
     (lstm): LSTM()
     (fc1): Linear(in_features=64, out_features=64, bias=True)
     (fc2): Linear(in_features=64, out_features=2, bias=True)
   )
   100%|██████████████████████████████████████| 313/313 [01:05<00:00,  4.76it/s]
   Train Epoch: 1 Loss: 0.693271    Acc: 0.505741
   Test set: Average loss: 0.6912, Accuracy: 0.5196
   100%|██████████████████████████████████████| 313/313 [01:06<00:00,  4.68it/s]
   Train Epoch: 2 Loss: 0.681398    Acc: 0.551967
   Test set: Average loss: 0.6684, Accuracy: 0.5833
   100%|██████████████████████████████████████| 313/313 [01:07<00:00,  4.61it/s]
   Train Epoch: 3 Loss: 0.632539    Acc: 0.645118
   Test set: Average loss: 0.6545, Accuracy: 0.6185
   100%|██████████████████████████████████████| 313/313 [01:08<00:00,  4.54it/s]
   Train Epoch: 4 Loss: 0.572839    Acc: 0.713409
   Test set: Average loss: 0.5932, Accuracy: 0.7065
   100%|██████████████████████████████████████| 313/313 [01:06<00:00,  4.72it/s]
   Train Epoch: 5 Loss: 0.483343    Acc: 0.783846
   Test set: Average loss: 0.6495, Accuracy: 0.6406
   ```

   发现：准确率不高。

   2. 调整网络结构
      - 增加一层全连接层：隐藏层维度：64；损失函数：CrossEntropyLoss；Epoch：5；batch_size：64

```
  (embedding): Embedding(20001, 64)
  (lstm): LSTM()
  (fc1): Linear(in_features=64, out_features=64, bias=True)
  (fc2): Linear(in_features=64, out_features=64, bias=True)
  (fc3): Linear(in_features=64, out_features=2, bias=True)
)
100%|█████████████████████████████████████| 313/313 [01:05<00:00,  4.81it/s]
Train Epoch: 1 Loss: 0.693682    Acc: 0.501398
Test set: Average loss: 0.6929, Accuracy: 0.5077
100%|█████████████████████████████████████| 313/313 [01:04<00:00,  4.86it/s]
Train Epoch: 2 Loss: 0.685300    Acc: 0.537590
Test set: Average loss: 0.7010, Accuracy: 0.5053
100%|█████████████████████████████████████| 313/313 [01:08<00:00,  4.58it/s]
Train Epoch: 3 Loss: 0.672419    Acc: 0.562899
Test set: Average loss: 0.6691, Accuracy: 0.5821
100%|█████████████████████████████████████| 313/313 [01:10<00:00,  4.45it/s]
Train Epoch: 4 Loss: 0.660106    Acc: 0.579473
Test set: Average loss: 0.6862, Accuracy: 0.5061
100%|█████████████████████████████████████| 313/313 [01:05<00:00,  4.75it/s]
Train Epoch: 5 Loss: 0.682045    Acc: 0.538488

Test set: Average loss: 0.6881, Accuracy: 0.5425
```

发现：Acc很低，离80%的标准距离很远。

- **隐藏层维度：128**；损失函数：CrossEntropyLoss；Epoch：10；batch_size：64

```
Net(
  (embedding): Embedding(20001, 64)
  (lstm): LSTM()
  (fc1): Linear(in_features=128, out_features=128, bias=True)
  (fc2): Linear(in_features=128, out_features=2, bias=True)
)
100%|█████████████████████████████████████| 313/313 [01:06<00:00,  4.72it/s]
Train Epoch: 1 Loss: 0.692490    Acc: 0.513279
Test set: Average loss: 0.6869, Accuracy: 0.5471
100%|█████████████████████████████████████| 313/313 [01:02<00:00,  4.97it/s]
Train Epoch: 2 Loss: 0.679165    Acc: 0.552915
Test set: Average loss: 0.6734, Accuracy: 0.5987
100%|█████████████████████████████████████| 313/313 [01:04<00:00,  4.86it/s]
Train Epoch: 3 Loss: 0.656441    Acc: 0.590755
Test set: Average loss: 0.6955, Accuracy: 0.5101
100%|█████████████████████████████████████| 313/313 [01:06<00:00,  4.73it/s]
Train Epoch: 4 Loss: 0.645554    Acc: 0.620457
Test set: Average loss: 0.5636, Accuracy: 0.7296
100%|█████████████████████████████████████| 313/313 [01:06<00:00,  4.71it/s]
Train Epoch: 5 Loss: 0.426162    Acc: 0.812899
Test set: Average loss: 0.4266, Accuracy: 0.8093
100%|
██████████████████████████████████████████████████████████████████████████████████
█████████████████████████████████████████████████████████████████████████████████|
313/313 [01:03<00:00,  4.95it/s]
Train Epoch: 6 Loss: 0.301800    Acc: 0.877895
Test set: Average loss: 0.3987, Accuracy: 0.8285
```

```
100%|
████████████████████████████████████████████████████████████
███████████████████████████████████████████████████████████|
313/313 [01:05<00:00,  4.80it/s]
Train Epoch: 7 Loss: 0.225721    Acc: 0.916334
Test set: Average loss: 0.4058, Accuracy: 0.8426
100%|
████████████████████████████████████████████████████████████
███████████████████████████████████████████████████████████|
313/313 [01:03<00:00,  4.91it/s]
Train Epoch: 8 Loss: 0.166375    Acc: 0.942792
Test set: Average loss: 0.4409, Accuracy: 0.8384
100%|
████████████████████████████████████████████████████████████
███████████████████████████████████████████████████████████|
313/313 [01:04<00:00,  4.86it/s]
Train Epoch: 9 Loss: 0.110612    Acc: 0.965605
Test set: Average loss: 0.4804, Accuracy: 0.8398
100%|
████████████████████████████████████████████████████████████
███████████████████████████████████████████████████████████|
313/313 [01:05<00:00,  4.81it/s]
Train Epoch: 10 Loss: 0.068540   Acc: 0.980831
Test set: Average loss: 0.6099, Accuracy: 0.8317
```

发现：增加隐藏层维度可以明显提升测试集Acc，在5个epoch结束后已经达到80%且后续训练可以继续提升准确率。

3. 调整损失函数

   - PyTorch官方文档给出的几种损失函数中CrossEntropyLoss是最适合用于此问题的，其他不合适的损失函数可能会造成一些严重后果（如CTCLoss多用于处理序列长度可变的任务，如语音识别。Acc达到了0%，因此我直接中止了程序运行）

     隐藏层维度：64；损失函数：CTCLoss；Epoch：5（提前终止）；batch_size：64

```
Net(
  (embedding): Embedding(20000, 64)
  (lstm): LSTM()
  (fc): Linear(in_features=64, out_features=3, bias=True)
)
100%|                                         | 391/391 [01:16<00:00,  5.11it/s]
Train Epoch: 1 Loss: 10.216603    Acc: 0.001279
Test set: Average loss: 1.4054, Accuracy: 0.0000
100%|                                         | 391/391 [01:22<00:00,  4.73it/s]
Train Epoch: 2 Loss: 1.353754     Acc: 0.000000
Traceback (most recent call last):
```

发现：损失函数对训练影响非常大，要根据目的选择合适的损失函数。

   - 通过网络搜索到了Label Smoothing Cross Entropy 损失函数。它有助于防止过拟合并减轻错误标签的影响。

     隐藏层维度：64；损失函数：**Label Smoothing Cross Entropy Loss**；Epoch：5；batch_size：64

```
使用设备: cuda:0
Net(
  (embedding): Embedding(20001, 64)
  (lstm): LSTM()
  (fc1): Linear(in_features=64, out_features=64, bias=True)
  (fc2): Linear(in_features=64, out_features=2, bias=True)
```

```
)
Train Epoch: 1 Loss: 0.691385 Acc: 0.522214: 100%|
███████████████████████████████████████████████████████████████████████████████████████
█████████████████████████████████| 313/313 [01:08<00:00,  4.55it/s]
Train Epoch: 1 Loss: 0.691385    Acc: 0.522214
Test set: Average loss: 0.6861, Accuracy: 0.5500
Train Epoch: 2 Loss: 0.684264 Acc: 0.551018: 100%|
███████████████████████████████████████████████████████████████████████████████████████
█████████████████████████████████| 313/313 [01:05<00:00,  4.78it/s]
Train Epoch: 2 Loss: 0.684264    Acc: 0.551018
Test set: Average loss: 0.6904, Accuracy: 0.5320
Train Epoch: 3 Loss: 0.650201 Acc: 0.639227: 100%|
███████████████████████████████████████████████████████████████████████████████████████
█████████████████████████████████| 313/313 [01:09<00:00,  4.50it/s]
Train Epoch: 3 Loss: 0.650201    Acc: 0.639227
Test set: Average loss: 0.5966, Accuracy: 0.7217
Train Epoch: 4 Loss: 0.533848 Acc: 0.771915: 100%|
███████████████████████████████████████████████████████████████████████████████████████
█████████████████████████████████| 313/313 [01:07<00:00,  4.64it/s]
Train Epoch: 4 Loss: 0.533848    Acc: 0.771915
Test set: Average loss: 0.5130, Accuracy: 0.7836
Train Epoch: 5 Loss: 0.443213 Acc: 0.843251: 100%|
███████████████████████████████████████████████████████████████████████████████████████
█████████████████████████████████| 313/313 [01:07<00:00,  4.63it/s]
Train Epoch: 5 Loss: 0.443213    Acc: 0.843251
Test set: Average loss: 0.4712, Accuracy: 0.8192
```

发现：表现明显优于只使用CrossEntropyLoss，在隐藏层维度64的前提下仍然达到了80+的准确率。过拟合现象被抑制。

4. 调整训练流程--如超参数，epoch，batchsize等：
    - 隐藏层维度：64；损失函数：CrossEntropyLoss；**Epoch：10**；batch_size：64

```
Net(
  (embedding): Embedding(20001, 64)
  (lstm): LSTM()
  (fc1): Linear(in_features=64, out_features=64, bias=True)
  (fc2): Linear(in_features=64, out_features=2, bias=True)
)
100%|█████████████████████████████████████| 313/313 [01:06<00:00,  4.70it/s]
Train Epoch: 1 Loss: 0.693224    Acc: 0.507937
Test set: Average loss: 0.6923, Accuracy: 0.5184
100%|█████████████████████████████████████| 313/313 [01:00<00:00,  5.20it/s]
Train Epoch: 2 Loss: 0.689369    Acc: 0.528754
Test set: Average loss: 0.6905, Accuracy: 0.5182
100%|█████████████████████████████████████| 313/313 [01:01<00:00,  5.13it/s]
Train Epoch: 3 Loss: 0.677964    Acc: 0.558956
Test set: Average loss: 0.6757, Accuracy: 0.5698
100%|█████████████████████████████████████| 313/313 [01:03<00:00,  4.96it/s]
Train Epoch: 4 Loss: 0.652045    Acc: 0.594399
Test set: Average loss: 0.6662, Accuracy: 0.6001
100%|█████████████████████████████████████| 313/313 [01:03<00:00,  4.95it/s]
Train Epoch: 5 Loss: 0.635427    Acc: 0.606679
Test set: Average loss: 0.6698, Accuracy: 0.5932
100%|█████████████████████████████████████| 313/313 [01:02<00:00,  5.01it/s]
Train Epoch: 6 Loss: 0.562416    Acc: 0.697434
```

```
Test set: Average loss: 0.5929, Accuracy: 0.7004
100%|████████████████████████████████████| 313/313 [01:02<00:00,  5.00it/s]
Train Epoch: 7 Loss: 0.557378    Acc: 0.705970
Test set: Average loss: 0.5609, Accuracy: 0.7373
100%|████████████████████████████████████| 313/313 [01:02<00:00,  5.01it/s]
Train Epoch: 8 Loss: 0.427962    Acc: 0.816693
Test set: Average loss: 0.5506, Accuracy: 0.7579
100%|████████████████████████████████████| 313/313 [01:02<00:00,  5.01it/s]
Train Epoch: 9 Loss: 0.348632    Acc: 0.858826
Test set: Average loss: 0.5134, Accuracy: 0.7987
100%|████████████████████████████████████| 313/313 [01:03<00:00,  4.93it/s]
Train Epoch: 10 Loss: 0.287051   Acc: 0.891074
Test set: Average loss: 0.5157, Accuracy: 0.7975
```

发现：增加循环次数可以提高准确率（在未收敛时），同时也会增加时间成本。但要注意，过度增加`epoch`可能会导致过拟合，即模型在训练集上表现很好，但在测试集或新数据上表现不佳。

- 隐藏层维度：64；损失函数：CrossEntropyLoss；Epoch：10；**batch_size：32**

```
Net(
  (embedding): Embedding(20001, 64)
  (lstm): LSTM()
  (fc1): Linear(in_features=64, out_features=64, bias=True)
  (fc2): Linear(in_features=64, out_features=2, bias=True)
)
100%|████████████████████████████████████| 625/625 [02:06<00:00,  4.95it/s]
Train Epoch: 1 Loss: 0.693041    Acc: 0.512000
Test set: Average loss: 0.6903, Accuracy: 0.5303
100%|████████████████████████████████████| 625/625 [02:08<00:00,  4.88it/s]
Train Epoch: 2 Loss: 0.676670    Acc: 0.562550
Test set: Average loss: 0.6718, Accuracy: 0.5961
100%|████████████████████████████████████| 625/625 [02:07<00:00,  4.90it/s]
Train Epoch: 3 Loss: 0.606563    Acc: 0.680600
Test set: Average loss: 0.6091, Accuracy: 0.6865
100%|████████████████████████████████████| 625/625 [02:03<00:00,  5.06it/s]
Train Epoch: 4 Loss: 0.489389    Acc: 0.781200
Test set: Average loss: 0.6028, Accuracy: 0.7402
100%|████████████████████████████████████| 625/625 [02:05<00:00,  4.98it/s]
Train Epoch: 5 Loss: 0.405098    Acc: 0.832950
Test set: Average loss: 0.5037, Accuracy: 0.7739
100%|████████████████████████████████████| 625/625 [02:07<00:00,  4.90it/s]
Train Epoch: 6 Loss: 0.348666    Acc: 0.860550
Test set: Average loss: 0.4871, Accuracy: 0.8035
100%|████████████████████████████████████| 625/625 [02:11<00:00,  4.75it/s]
Train Epoch: 7 Loss: 0.281200    Acc: 0.892750
Test set: Average loss: 0.5112, Accuracy: 0.7846
100%|████████████████████████████████████| 625/625 [02:06<00:00,  4.93it/s]
Train Epoch: 8 Loss: 0.259462    Acc: 0.905250
Test set: Average loss: 0.7532, Accuracy: 0.5018
100%|████████████████████████████████████| 625/625 [02:07<00:00,  4.89it/s]
Train Epoch: 9 Loss: 0.332733    Acc: 0.852900
Test set: Average loss: 0.5388, Accuracy: 0.7976
100%|████████████████████████████████████| 625/625 [02:10<00:00,  4.79it/s]
Train Epoch: 10 Loss: 0.214364   Acc: 0.919700
Test set: Average loss: 0.5690, Accuracy: 0.8145
```

发现：Acc比batch_size=64时有小幅的提高，可能的原因是：较小的`batch size`意味着模型在训练过程中看到的数据更多样化，每次更新参数时使用的样本不同，这有助于模型学习到更通用的特征，从而提高泛化能力。但模型训练稳定性可能会下降。

特别鸣谢

图书馆西区一层的高性能工作区，帮我节约了大量训练时间。

code

全部代码已附在压缩包中。

手写实现lstm代码并达到80%+Acc代码见下：

```python
# 用Pytorch手写一个LSTM网络，在IMDB数据集上进行训练
# 跑5个epoch，hidden size保持64不变
import os
import numpy as np
import torch
import torch.nn as nn

from torch.utils.data import Dataset, DataLoader
from utils import load_imdb_dataset, Accuracy
import sys

from utils import load_imdb_dataset, Accuracy
from tqdm import tqdm
import math

# 超参数
n_epoch = 10
batch_size = 64
print_freq = 2




# print("MLU is not available, use GPU/CPU instead.")
if torch.cuda.is_available():
    device = torch.device('cuda:0')
else:
    device = torch.device('cpu')

X_train, y_train, X_test, y_test = load_imdb_dataset('data', nb_words=20000,
test_split=0.2)

seq_Len = 200
vocab_size = len(X_train) + 1


class ImdbDataset(Dataset):

    def __init__(self, X, y):
        self.X = X
```

```python
        self.y = y

    def __getitem__(self, index):

        data = self.X[index]
        data = np.concatenate([data[:seq_Len], [0] * (seq_Len - 
len(data))]).astype('int32')  # set
        label = self.y[index]
        return data, label

    def __len__(self):

        return len(self.y)


# 你需要实现的手写LSTM内容，包括LSTM类所属的__init__函数和forward函数
class LSTM(nn.Module):
    '''
    手写lstm，可以用全连接层nn.Linear，不能直接用nn.LSTM
    '''
    def __init__(self, input_size, hidden_size):
        super(LSTM, self).__init__()
        self.hidden_size = hidden_size
        self.imput_size = input_size

        # 初始化所有门控的参数：权重W和偏置b

        # 输入门
        self.W_ii = nn.Parameter(torch.Tensor(hidden_size, input_size))
        self.W_hi = nn.Parameter(torch.Tensor(hidden_size, hidden_size))
        self.b_ii = nn.Parameter(torch.Tensor(hidden_size))

        # 遗忘门
        self.W_if = nn.Parameter(torch.Tensor(hidden_size, input_size))
        self.W_hf = nn.Parameter(torch.Tensor(hidden_size, hidden_size))
        self.b_if = nn.Parameter(torch.Tensor(hidden_size))

        # 细胞状态
        self.W_ic = nn.Parameter(torch.Tensor(hidden_size, input_size))
        self.W_hc = nn.Parameter(torch.Tensor(hidden_size, hidden_size))
        self.b_ic = nn.Parameter(torch.Tensor(hidden_size))

        # 输出门
        self.W_io = nn.Parameter(torch.Tensor(hidden_size, input_size))
        self.W_ho = nn.Parameter(torch.Tensor(hidden_size, hidden_size))
        self.b_io = nn.Parameter(torch.Tensor(hidden_size))

        # 初始化参数
        self.reset_parameters()

    def reset_parameters(self):
        # 使用均匀分布初始化权重。常用策略
        stdv = 1.0 / math.sqrt(self.hidden_size)
        for weight in self.parameters():
            nn.init.uniform_(weight, -stdv, stdv)
```

```python
    def forward(self, x, init_states=None):
        """
        x: 输入序列, shape: (batch_size, seq_len, input_size)
        返回: output, (h_n, c_n)
        """
        batch_size, seq_len, _ = x.size()

        # 初始化隐藏状态和细胞状态
        if init_states is None:
            h_t = torch.zeros(batch_size, self.hidden_size).to(x.device)
            c_t = torch.zeros(batch_size, self.hidden_size).to(x.device)
        else:
            h_t, c_t = init_states

        # 存储所有时间步的输出
        outputs = []

        for t in range(seq_len):
            # 获取当前时间步的输入
            x_t = x[:, t, :]  # (batch_size, input_size)

            # 输入门计算
            i_t = torch.sigmoid(
                torch.matmul(x_t, self.W_ii.t()) + torch.matmul(h_t, self.W_hi.t()) +
self.b_ii
            )

            # 遗忘门计算
            f_t = torch.sigmoid(
                torch.matmul(x_t, self.W_if.t()) + torch.matmul(h_t, self.W_hf.t()) +
self.b_if
            )

            # 细胞状态计算
            c_tilde = torch.tanh(
                torch.matmul(x_t, self.W_ic.t()) + torch.matmul(h_t, self.W_hc.t()) +
self.b_ic
            )

            # 更新细胞状态
            c_t = f_t * c_t + i_t * c_tilde

            # 输出门计算
            o_t = torch.sigmoid(
                torch.matmul(x_t, self.W_io.t()) + torch.matmul(h_t, self.W_ho.t()) +
self.b_io
            )

            # 更新隐藏状态
            h_t = o_t * torch.tanh(c_t)

            # 保存当前时间步的输出
            outputs.append(h_t.unsqueeze(1))

        # 拼接所有时间步的输出
        outputs = torch.cat(outputs, dim=1)  # (batch_size, seq_len, hidden_size)
```

```python
        return outputs, (h_t, c_t)


class Net(nn.Module):
    '''
    一层LSTM的文本分类模型
    '''
    def __init__(self, embedding_size=64, hidden_size=128, num_classes=2):
        super(Net, self).__init__()
        self.embedding = nn.Embedding(vocab_size, embedding_size)
        self.lstm = LSTM(input_size=embedding_size, hidden_size=hidden_size)
        self.fc1 = nn.Linear(hidden_size, hidden_size)
        self.fc2 = nn.Linear(hidden_size, num_classes)

    def forward(self, x):
        '''
        x: 输入, shape: (batch_size, seq_len)
        '''
        # 词嵌入 (batch_size, seq_len) -> (batch_size, seq_len, embedding_size)
        x = self.embedding(x)

        # LSTM层 (batch_size, seq_len, embedding_size) -> (batch_size, seq_len, hidden_size)
        lstm_out, _ = self.lstm(x)

        # 取最后一个时间步的输出 (batch_size, hidden_size)
        last_out = lstm_out[:, -1, :]

        # 全连接层
        x = torch.relu(self.fc1(last_out))
        x = self.fc2(x)

        return x


train_dataset = ImdbDataset(X=X_train, y=y_train)
train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
test_dataset = ImdbDataset(X=X_test, y=y_test)
test_loader = DataLoader(test_dataset, batch_size=batch_size, shuffle=False)

net = Net()
metric = Accuracy()
print(net)


def train(model, device, train_loader, optimizer, epoch):
    model = model.to(device)
    model.train()
    loss_func = torch.nn.CrossEntropyLoss(reduction="mean")
    train_acc = 0
    train_loss = 0
    n_iter = 0
    for batch_idx, (data, target) in tqdm(enumerate(train_loader),
total=len(train_loader)):
        target = target.long()
        data, target = data.to(device), target.to(device)
```

```
        optimizer.zero_grad()
        output = model(data)
        # loss = F.nll_loss(output, target)
        loss = loss_func(output, target)
        loss.backward()
        optimizer.step()
        metric.update(output, target)
        train_acc += metric.result()
        train_loss += loss.item()
        metric.reset()
        n_iter += 1
    print('Train Epoch: {} Loss: {:.6f} \t Acc: {:.6f}'.format(epoch, train_loss / n_iter,
train_acc / n_iter))


def test(model, device, test_loader):
    model = model.to(device)
    model.eval()
    loss_func = torch.nn.CrossEntropyLoss(reduction="mean")
    test_loss = 0
    test_acc = 0
    n_iter = 0
    with torch.no_grad():
        for data, target in test_loader:
            target = target.long()
            data, target = data.to(device), target.to(device)
            output = model(data)
            # test_loss += F.nll_loss(output, target, reduction='sum').item()  # sum up
batch loss
            test_loss += loss_func(output, target).item()
            metric.update(output, target)
            test_acc += metric.result()
            metric.reset()
            n_iter += 1
    test_loss /= n_iter
    test_acc /= n_iter
    print('Test set: Average loss: {:.4f}, Accuracy: {:.4f}'.format(test_loss, test_acc))


optimizer = torch.optim.Adam(net.parameters(), lr=1e-3, weight_decay=0.0)
gamma = 0.7
for epoch in range(1, n_epoch + 1):
    train(net, device, train_loader, optimizer, epoch)
    test(net, device, test_loader)
```