CS310 Natural Language Processing

Name: Wang Ziheng

SID: 12310401

Assignment 1. Neural Text Classification

Total points: 50

You should roughly follow the structure of the notebook. Add additional cells if you feel needed.

You can (and you should) re-use the code from Lab 2.

Make sure your code is readable and well-structured.

0. Import Necessary Libraries

In [3…

```
import json
import re
import torch
import numpy as np
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import Dataset, DataLoader
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score
import tqdm
from torch.nn.utils.rnn import pad_sequence
```

1. Data Processing

In [3…

```
def basic_tokenizer(text):
    return [char for char in text if '\u4e00' <= char <= '\u9fff']

def improved_tokenizer(text):
    tokens = []
    pattern = re.compile(r'[\u4e00-\u9fff]+|[a-zA-Z]+|\d+|[^\w\s]')
    for match in pattern.finditer(text):
        tokens.append(match.group())
    return tokens
```

In [3…

```
class HumorDataset(Dataset):
    def __init__(self, file_path, tokenizer):
        self.data = []
        self.tokenizer = tokenizer
        self.vocab = {}
        self.vocab_size = 0
        with open(file_path, 'r', encoding='utf-8') as f:
            for line in f:
                item = json.loads(line)
                sentence = item['sentence']
                tokens = tokenizer(sentence)
                label = item['label'][0]
                self.data.append((tokens, label))
                for token in tokens:
                    if token not in self.vocab:
                        self.vocab[token] = len(self.vocab)
        self.vocab_size = len(self.vocab)

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

    def __getitem__(self, idx):
        tokens, label = self.data[idx]
        token_ids = [self.vocab[token] for token in tokens]
        return token_ids, label
def generate_offsets(batch):
    offsets = [0]
    for tokens in batch:
        offsets.append(offsets[-1] + len(tokens))
    return offsets[:-1]
def collate_fn(batch):
```

```
    tokens, labels = zip(*batch)
    token_ids = [torch.tensor(ids) for ids in tokens]
    token_ids = pad_sequence(token_ids, batch_first=True, padding_value=0)  # Padding with 0
    token_ids = token_ids.view(-1)
    offsets = [0] + [len(ids) for ids in tokens]
    offsets = torch.tensor(offsets[:-1]).cumsum(dim=0)
    labels = torch.tensor(labels)
    return token_ids, offsets, labels
```

In [3…

```
train_dataset = HumorDataset('/Users/earendelh/Documents/Sophomore_Second/NLP/Ass1/train.jsonl', improved_tokenizer)
test_dataset = HumorDataset('/Users/earendelh/Documents/Sophomore_Second/NLP/Ass1/test.jsonl', improved_tokenizer)
```

In [3…

```
train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True, collate_fn=collate_fn)
test_loader = DataLoader(test_dataset, batch_size=32, shuffle=False, collate_fn=collate_fn)
```

2. Build the Model

In [3…

```
class HumorClassifier(nn.Module):
    def __init__(self, vocab_size, embed_dim, hidden_dim1, hidden_dim2, output_dim):
        super(HumorClassifier, self).__init__()
        self.embedding = nn.EmbeddingBag(vocab_size, embed_dim, sparse=True)
        self.fc = nn.Sequential(
            nn.Linear(embed_dim, hidden_dim1),
            nn.LayerNorm(hidden_dim1),
            nn.ReLU(),
            nn.Linear(hidden_dim1, hidden_dim2),
            nn.ReLU(),
            nn.Linear(hidden_dim2, output_dim)
        )

    def forward(self, text, offsets):
        embedded = self.embedding(text, offsets)
        return self.fc(embedded)
```

In [3…

```
vocab_size = train_dataset.vocab_size
embed_dim = 64
hidden_dim1 = 128
hidden_dim2 = 64
output_dim = 2
```

In [3…

```
model = HumorClassifier(vocab_size, embed_dim, hidden_dim1, hidden_dim2, output_dim)
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(model.parameters(), lr=1e-3)
```

3. Train and Evaluate

In [3…

```
num_epochs = 10
for epoch in tqdm.tqdm(range(num_epochs)):
    model.train()
    for token_ids, offsets, labels in train_loader:
        optimizer.zero_grad()
        outputs = model(token_ids, offsets)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
    print(f'Epoch [{epoch+1}/{num_epochs}], Loss: {loss.item():.4f}')
```

```
 10%|█         | 1/10 [00:00<00:06,  1.45it/s]
Epoch [1/10], Loss: 0.6480
 20%|██        | 2/10 [00:01<00:04,  1.90it/s]
Epoch [2/10], Loss: 1.0771
 30%|███       | 3/10 [00:01<00:03,  2.07it/s]
Epoch [3/10], Loss: 0.6369
 40%|████      | 4/10 [00:02<00:02,  2.08it/s]
Epoch [4/10], Loss: 0.3405
 50%|█████     | 5/10 [00:02<00:02,  2.16it/s]
Epoch [5/10], Loss: 0.8334
 60%|██████    | 6/10 [00:02<00:01,  2.22it/s]
Epoch [6/10], Loss: 0.5213
 70%|███████   | 7/10 [00:03<00:01,  2.19it/s]
Epoch [7/10], Loss: 0.7186
 80%|████████  | 8/10 [00:03<00:00,  2.04it/s]
Epoch [8/10], Loss: 0.5395
 90%|█████████ | 9/10 [00:04<00:00,  1.94it/s]
```

```
Epoch [9/10], Loss: 0.8940
100%|██████████| 10/10 [00:04<00:00,  2.01it/s]
Epoch [10/10], Loss: 0.5159
```

In [3…

```python
model.eval()
all_labels = []
all_preds = []
with torch.no_grad():
    for token_ids, offsets, labels in test_loader:
        outputs = model(token_ids, offsets)
        _, predicted = torch.max(outputs.data, 1)
        all_labels.extend(labels.tolist())
        all_preds.extend(predicted.tolist())

accuracy = accuracy_score(all_labels, all_preds)
precision = precision_score(all_labels, all_preds, average='weighted')
recall = recall_score(all_labels, all_preds, average='weighted')
f1 = f1_score(all_labels, all_preds, average='weighted')

print(f'Accuracy: {accuracy:.4f}, Precision: {precision:.4f}, Recall: {recall:.4f}, F1: {f1:.4f}')
```

```
Accuracy: 0.7389, Precision: 0.5459, Recall: 0.7389, F1: 0.6279
/opt/miniconda3/envs/NLP/lib/python3.9/site-packages/sklearn/metrics/_classification.py:1565: UndefinedMetricWarning:
Precision is ill-defined and being set to 0.0 in labels with no predicted samples. Use `zero_division` parameter to co
ntrol this behavior.
  _warn_prf(average, modifier, f"{metric.capitalize()} is", len(result))
```

4. Explore Word Segmentation

In [3…

```python
# use jieba as the tokenizer
train_dataset2 = HumorDataset('/Users/earendelh/Documents/Sophomore_Second/NLP/Ass1/train.jsonl', jieba._lcut_for_searc
test_dataset2 = HumorDataset('/Users/earendelh/Documents/Sophomore_Second/NLP/Ass1/test.jsonl', jieba._lcut_for_search)
train_loader2 = DataLoader(train_dataset2, batch_size=32, shuffle=True, collate_fn=collate_fn)
test_loader2 = DataLoader(test_dataset2, batch_size=32, shuffle=False, collate_fn=collate_fn)

vocab_size = train_dataset2.vocab_size
embed_dim = 64
hidden_dim1 = 128
hidden_dim2 = 64
output_dim = 2

model2 = HumorClassifier(vocab_size, embed_dim, hidden_dim1, hidden_dim2, output_dim)
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(model2.parameters(), lr=1e-3)

num_epochs = 10
for epoch in tqdm.tqdm(range(num_epochs)):
    model2.train()
    for token_ids, offsets, labels in train_loader2:
        optimizer.zero_grad()
        outputs = model2(token_ids, offsets)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
    print(f'Epoch [{epoch+1}/{num_epochs}], Loss: {loss.item():.4f}')

model2.eval()
all_labels2 = []
all_preds2 = []
with torch.no_grad():
    for token_ids, offsets, labels in test_loader2:
        outputs = model2(token_ids, offsets)
        _, predicted = torch.max(outputs.data, 1)
        all_labels2.extend(labels.tolist())
        all_preds2.extend(predicted.tolist())

accuracy_jieba = accuracy_score(all_labels2, all_preds2)
precision_jieba = precision_score(all_labels2, all_preds2, average='weighted')
recall_jieba = recall_score(all_labels2, all_preds2, average='weighted')
f1_jieba = f1_score(all_labels2, all_preds2, average='weighted')

print(f'Jieba Accuracy: {accuracy_jieba:.4f}, Precision: {precision_jieba:.4f}, Recall: {recall_jieba:.4f}, F1: {f1_jie
```

```
 10%|█         | 1/10 [00:00<00:04,  2.00it/s]
Epoch [1/10], Loss: 0.6005
 20%|██        | 2/10 [00:01<00:04,  1.70it/s]
Epoch [2/10], Loss: 0.5828
 30%|███       | 3/10 [00:01<00:04,  1.49it/s]
```

```
Epoch [3/10], Loss: 0.3643
 40%|████      | 4/10 [00:02<00:04,  1.43it/s]
Epoch [4/10], Loss: 0.6817
 50%|█████     | 5/10 [00:03<00:03,  1.34it/s]
Epoch [5/10], Loss: 0.3535
 60%|██████    | 6/10 [00:04<00:02,  1.40it/s]
Epoch [6/10], Loss: 1.2037
 70%|███████   | 7/10 [00:04<00:02,  1.45it/s]
Epoch [7/10], Loss: 0.3543
 80%|████████  | 8/10 [00:05<00:01,  1.48it/s]
Epoch [8/10], Loss: 0.7192
 90%|█████████ | 9/10 [00:06<00:00,  1.29it/s]
Epoch [9/10], Loss: 0.6979
100%|██████████| 10/10 [00:06<00:00,  1.43it/s]
Epoch [10/10], Loss: 1.1027
Jieba Accuracy: 0.7389, Precision: 0.5459, Recall: 0.7389, F1: 0.6279
/opt/miniconda3/envs/NLP/lib/python3.9/site-packages/sklearn/metrics/_classification.py:1565: UndefinedMetricWarning: Precision is ill-defined and being set to 0.0 in labels with no predicted samples. Use `zero_division` parameter to control this behavior.
  _warn_prf(average, modifier, f"{metric.capitalize()} is", len(result))
```

Comparison of the training results

I found that the dataset is too small , easily ***overfitting***. Only when I use the very small learning rate and epoch, I can find a different result.

From the result below, we can observe that the jieba tokenizer is better than the improved tokenizer and better than basic tokenizer.

Tokenizer	Learning Rate	Epoch	Accuracy	Precision	Recall	F1
Basic	1e-5	20	0.6482	0.6338	0.6482	0.6404
Basic	1e-5	10	0.6820	0.6113	0.6820	0.6347
Basic	1e-3	10	0.7389	0.5459	0.7389	0.6279
Improved	1e-5	20	0.7097	0.6336	0.7097	0.6493
Improved	1e-5	10	0.7266	0.6188	0.7266	0.6347
Improved	1e-3	10	0.7389	0.5459	0.7389	0.6279
jieba	1e-5	20	0.7189	0.5849	0.7189	0.6257
jieba	1e-5	10	0.7389	0.6770	0.7389	0.6307
jieba	1e-3	10	0.7389	0.5459	0.7389	0.6279