CS310 Natural Language Processing - Assignment 1 Neural-Nets for Text Classification

1 Project Introduction

This project is aimed at training a neural network-based text classification model, which is on Chinese humour detection dataset. Apart from the training on improved tokenizer, we also used the jieba package during the word segmentation task.

The model evaluation is indicated as below:

	Character-segment	Word-segment
Accuracy	0.7435	0.7097
Precision	0.5283	0.2564
Recall	0.1647	0.0588
F1 Score	0.2511	0.0957

2 Data processing

2.1 Outline of the dataset

The dataset is used for Chinese humour detection and consists of a training set (train.jsonl) and a testing set (test.jsonl). Each data record contains four fields:

- `"sentence"`: A Chinese sentence.
- `"choices"`: A list containing two values ("0" or "1"), used to indicate the category of the sentence (0 for non-humorous, 1 for humorous).
- `"label"`: A list containing a single label (0 or 1), representing the category of the sentence.
- `"id"`: A unique identifier for each data entry.

Here is an example from training set, helping understand the form of the dataset:

```
{"sentence": "卖油条小刘说：我说", "choices": ["0", "1"], "label": [0], "id": "train_0"}
{"sentence": "保姆小张说：干啥子嘛？", "choices": ["0", "1"], "label": [0], "id": "train_1"}
{"sentence": "卖油条小刘说：你看你往星空看月朦胧，鸟朦胧", "choices": ["0", "1"], "label": [1], "id": "train_2"}
```

2.2 Data preprocessing

The dataset consists of Chinese sentences with labels indicating whether the sentence is humorous or not. The first step in data preprocessing is to tokenize these sentences into meaningful tokens, which will then be converted into numeric representations for further use in the model.

Two tokenization approaches are utilized:

- **Basic Tokenizer**: This tokenizer extracts only Chinese characters from the text, discarding any other tokens(English, numbers, punctuation marks)
- **Improved Tokenizer**: This tokenizer is more advanced and extracts Chinese characters, English words, numbers, and punctuation marks. It can help in tasks that require handling multi-lingual or mixed content.

```
def improved_tokenizer(text):
    reg = r'[\u4e00-\u9fa5]|[a-zA-Z]+|[0-9]+|[^\w\s]'
    tokens = re.findall(reg, text)
    return tokens
```

2.3 Vocabulary generation

To train the model, we need to create a vocabulary that maps each token to a unique index. This process involves counting the frequency of each token in the dataset, and then assigning each unique token a unique index. If a token is not found in the vocabulary during training, it is mapped to the special token `<unk>`, which stands for "unknown."

```
def build_vocab_from_file(file_path, tokenizer):
    word_freq = defaultdict(int)

    with open(file_path, 'r', encoding='utf-8') as file:
        for line in file:
            data = json.loads(line)
            sentence = data['sentence']
            tokens = tokenizer(sentence)
            for token in tokens:
                word_freq[token] += 1  # 更新每个token的出现频率

    # 构建词汇表，初始化未知词为 <unk>
    vocab = {'<unk>': 0}
    vocab.update({token: idx + 1 for idx, (token, freq) in
enumerate(word_freq.items())})

    return vocab

vocab = build_vocab_from_file('train.jsonl', improved_tokenizer)
```

2.4 Dataset class

Once the vocabulary is created, we can proceed to create a custom dataset class that will load the data, tokenize the sentences, and convert them into token IDs using the vocabulary. This class will also handle batching and data preparation for training.

```
class TextDataset(Dataset):
    def __init__(self, file_path, tokenizer, vocab):
        self.tokenizer = tokenizer
        self.data = []
        self.vocab = vocab
        self._prepare_data(file_path)

    def _prepare_data(self, file_path):
        self.data = []
        with open(file_path, 'r', encoding='utf-8') as f:
            for line in f:
                item = json.loads(line)
                sentence = item['sentence']
                label = item['label'][0]  # Labels are a list of length 1
                tokens = self.tokenizer(sentence)
                token_ids = list(map(lambda token: self.vocab.get(token, self.vocab['<unk>']), tokens))  # Use map to process tokens
                self.data.append((token_ids, label))

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

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

In this class `TextDataset`, data is read as JSONL files, with sentences being tokenized. The tokens are mapped to IDs in the vocabulary in the `_prepare_data` function.

2.5 Data loader

To handle batching, we use the `DataLoader` class from PyTorch. The `collate_batch` function is responsible for preparing batches by converting the list of token IDs into a single tensor and calculating the offsets for each sentence in the batch. Here we adopt a similar code in Lab2. (Code omitted)

Then we set up the data loaders for the training and testing datasets, `batch_size=8`.

3 Build the Model

3.1 Mode architecture

The model is built using the following components:

- **Embedding Layer (EmbeddingBag)**: The model uses the `nn.EmbeddingBag` method to perform the bag-of-words representation. This layer is responsible for converting input token IDs into fixed-size dense vectors (embeddings).

```
self.embedding = torch.nn.EmbeddingBag(vocab_size, embed_dim, sparse=False)
```

- **Fully-Connected Layers**: The model uses a fully connected component consisting of two hidden layers, each followed by a ReLU activation function. This part of the model is designed to capture complex relationships in the input features.

```
self.fc = nn.Sequential(
  nn.Linear(embed_dim, 128),
  nn.ReLU(),
  nn.Linear(128, 64),
  nn.ReLU(),
  nn.Linear(64, num_class)
)
```

3.2 Weight initialization

The weights are initialized using a uniform distribution within the range of `[-initrange, initrange]`. The bias of the linear layers is initialized to zero. Here we adopt a similar code in Lab2. (Code omitted)

3.3 Forward pass

The forward pass of the model involves the following steps:

- The token IDs and offsets are passed to the embedding layer (`EmbeddingBag`), which returns a fixed-size embedding for the input tokens.
- The embeddings are then passed through the fully connected layers, producing the final output.

3.4 Model initialization

We instantiate the model, specifying the vocabulary size, embedding dimension, and the number of output classes. In this case, we use an embedding size of 64 and train for 10 epochs.

We define the following hyperparameters: `EPOCHS`, `LR`, `BATCH_SIZE`.

We use **cross-entropy loss** for the classification task, which is suitable for multi-class classification problems. The **SGD optimizer** is used to update the model's parameters, and a **learning rate scheduler** is applied to adjust the learning rate during training.

Parameter	Description	Value/Type
`vocab_size`	Size of the vocabulary (number of unique tokens)	`len(vocab)`
`embed_dim`	Dimensionality of the embedding vectors	`64`
`num_class`	Number of output classes (humorous or non-humorous)	`2` (0 for non-humorous, 1 for humorous)
`EPOCHS`	Number of epochs to train the model	`10`
`LR`	Learning rate for the optimizer	`5`
`BATCH_SIZE`	Number of samples per batch	`8`
`initrange`	The range used for uniform weight initialization	`0.5`
`criterion`	Loss function used for training	`nn.CrossEntropyLoss()`
`optimizer`	Optimizer used to update the model's parameters	`torch.optim.SGD(model.parameters(), lr=LR)`
`scheduler`	Learning rate scheduler to adjust the learning rate during training	`torch.optim.lr_scheduler.StepLR(optimizer, 1.0, gamma=0.1)`

4 Train and Evaluate

4.1 Training process

This process involves iterating over the dataset for epoch times and adjusting the model's parameters based on the loss. The training loop includes the following steps: `Forward Pass`, `Loss Calculation`, `Backward Propagation`, and `Logging`. The code here is similar to that in Lab2. (Code omitted)

4.2 Evaluation process

After each epoch, we evaluate the model on the validation dataset to track performance. We calculate accuracy and collect predictions for further metrics.

The evaluation process involves:

- `Model Evaluation Mode`: Using `model.eval()` to ensure no gradients are computed.
- `Prediction`: Making predictions for each batch in the validation or test set.
- `Metrics Calculation`: Accuracy is calculated during evaluation, and predictions are saved for calculating additional metrics later.

Code in this process is also included in Lab2.

4.3 Metrics calculation

Once the predictions are collected, we calculate the following evaluation metrics:

- **Accuracy**: The ratio of correctly predicted labels to the total number of samples.
- **Precision**: The ratio of correctly predicted positive labels to the total predicted positive labels.
- **Recall**: The ratio of correctly predicted positive labels to the total actual positive labels.
- **F1 Score**: The harmonic mean of precision and recall, providing a balanced measure.

```python
def calculate_metrics(all_labels, all_preds):
    TP = np.sum((all_labels == 1) & (all_preds == 1))
    FP = np.sum((all_labels == 0) & (all_preds == 1))
    FN = np.sum((all_labels == 1) & (all_preds == 0))
    TN = np.sum((all_labels == 0) & (all_preds == 0))

    precision = TP / (TP + FP) if TP + FP > 0 else 0
    recall = TP / (TP + FN) if TP + FN > 0 else 0
    f1 = 2 * (precision * recall) / (precision + recall) if precision + recall > 0 else 0
    accuracy = (TP + TN) / (TP + TN + FP + FN)

    return accuracy, precision, recall, f1
```

4.4 Training and validation loop

We perform training for a specified number of epochs. After each epoch, the model is evaluated on the validation set. If the validation accuracy improves, the learning rate scheduler is applied to adjust the learning rate.

```python
total_accu = None
for epoch in range(1, EPOCHS + 1):
    epoch_start_time = time.time()

    train(model, train_dataloader, optimizer, criterion, epoch)
    accu_val, _, _ = evaluate(model, valid_dataloader, criterion)

    if total_accu is not None and total_accu > accu_val:
        scheduler.step()
    else:
        total_accu = accu_val

    print("-" * 59)
    print(
        "| end of epoch {:3d} | time: {:5.2f}s | "
        "valid accuracy {:8.3f} ".format(
            epoch, time.time() - epoch_start_time, accu_val
        )
    )
    print("-" * 59)
```

At the end of the training, we evaluate the model and report the final test accuracy, precision, recall and F1 score.

5 Explore Word Segmentation

In this section, we explore the impact of word segmentation on classification performance. Word segmentation can help improve the quality of text data by grouping multiple characters (字) into a word (词), which is a more natural unit of language. We use the `jieba` word segmentation tool to process the text data and then compare the performance of the segmented data with the original data.

5.1 Word segmentation with jieba

With the help of jieba package, we only need to implement the code below to do segmentation:

```
def jieba_tokenizer(text):
    return jieba.lcut(text)
```

5.2 Similar process

Then we rebuild the vocabulary based on the segmented data.

```
vocab = build_vocab_from_file('train.jsonl', jieba_tokenizer)
```

Use the segmented data for training and evaluation. The training and evaluation processes remain the same, but now they operate on the segmented text.

```
train_dataset = TextDataset('train.jsonl', jieba_tokenizer, vocab)
test_dataset = TextDataset('test.jsonl', jieba_tokenizer, vocab)

train_loader = DataLoader(train_dataset, batch_size=8, shuffle=True, collate_fn=collate_batch)
test_loader = DataLoader(test_dataset, batch_size=8, shuffle=True, collate_fn=collate_batch)

# Model initialization
model = TextClassificationModel(len(vocab), embed_dim=64, num_class=2).to(device)
```

The training loop is the same as in the original setup.

After training, the model is evaluated on the test set. The accuracy, precision, recall, and F1 score are computed to assess the model's performance.

5.3 Comparison

Here we demonstrate the results of the two segmentation method again:

	Character-segment	Word-segment
Accuracy	0.7435	0.7097
Precision	0.5283	0.2564
Recall	0.1647	0.0588
F1 Score	0.2511	0.0957

The performance comparison between the Character-segment and Word-segment methods shows that Character-segment outperforms Word-segment in all evaluated metrics, including accuracy, precision, recall, and F1 score.

Specifically, Character-segment achieves higher accuracy, precision, and recall, indicating it is more effective in correctly identifying humorous sentences while minimizing false positives.

The F1 score also supports this, as Character-segment maintains a better balance between precision and recall.

These results suggest that treating characters individually as tokens captures more detailed and relevant features for humour detection, making Character-segment the better approach for this task.

6 Project Summary

In this project, we trained a neural network for Chinese humor detection using a dataset with labeled humorous and non-humorous sentences. We employed two tokenization methods: character-based tokenization and word-based tokenization using the `jieba` library for word segmentation. The model was built using PyTorch's `EmbeddingBag` layer for bag-of-words representation and fully connected layers for classification. The results were evaluated based on accuracy, precision, recall, and F1 score.

Through a comparison of the two tokenization methods, it was observed that character-segment outperformed word-segment across all metrics, suggesting that treating characters individually as tokens captures more detailed and relevant features for humor detection. This approach yielded better accuracy, precision, recall, and F1 score, making character-segment the preferred method for this task.