Report

李沐遥 YuanPei College 2200017405

1. Task1:

a. Flatten a List of Lists

Task Description

transform a nested list structure into a single, one-dimensional list.

Example:

```
Input: [[1,2,3], [4, 5], [6]]
Output: [1,2,3,4,5,6]
```

Algorithm

I tried two ways to solve this task

1. using the .extend() method of list to append elements into a new list.

```python
def flatten_list(nested_list: list):
    output_list = []
    for a in nested_list:
        output_list.extend(a)
    return output_list
```

2. using the for loop to Iterate over each sublist in input list and use the .append() method to append elements into a new list.

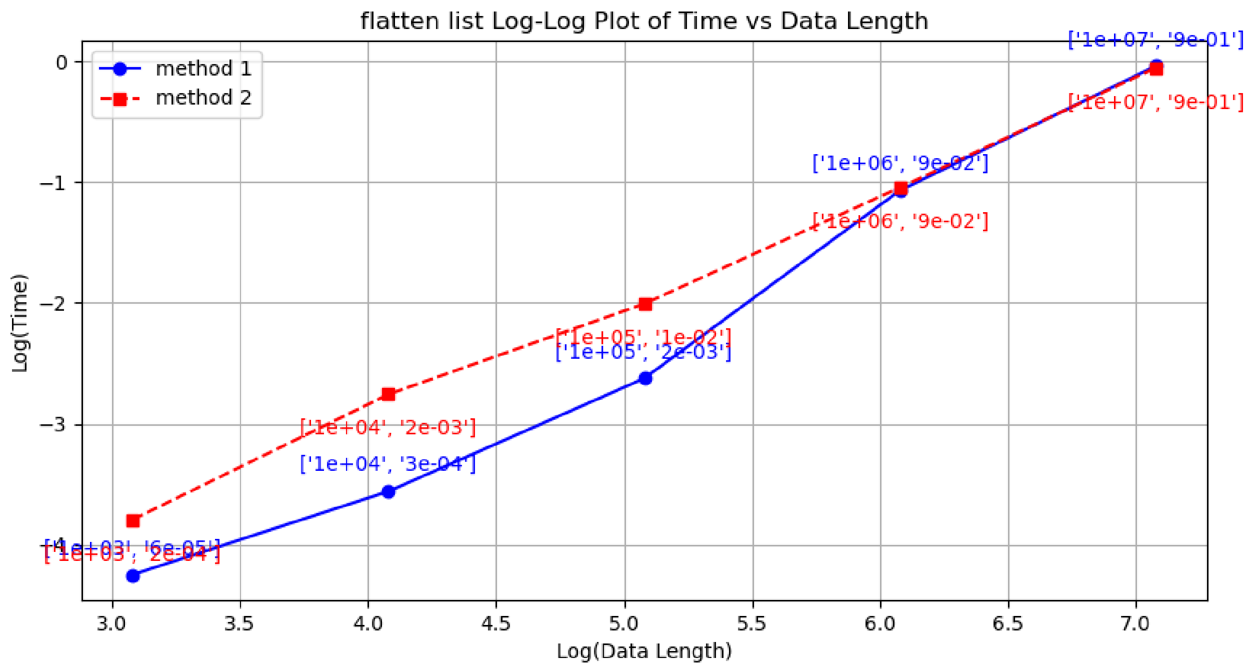
```python
def flatten_list(nested_list: list):
    output_list = []
    for a in nested_list:
        for b in a:
            output_list.append(b)
    return output_list
```

Experiment

We tested both methods across five data points with lengths ranging from 1.2e3 to 1.2e7. Random inputs were generated for each test.And we use the log of time cost and the log of data lengths to draw the line

According to the figure, we can see that both method 1 and method 2 has a complexity of O(n).

Besides, when the input size is smaller, using the built-in extend method performs better than using a for loop. However, when the scale reaches 1e6, the efficiencies of the two methods become comparable.



Analysis

The initial superior performance of Method 1 can be attributed to the efficiency of the .extend() method, which is optimized for adding all elements of an iterable to the end of a list in one go, reducing overhead. However, as the dataset grows, the cost of repeatedly extending a large list seems to catch up, reducing the gap in performance between the two methods.

b. Character Count

Task Description

The objective of this task is to count the frequency of each lowercase letter in a string. The output should be a dictionary where keys are characters and values are their counts.

Example:

```
Input: "hello world"
Output: {'h': 1, 'e': 1, 'l': 3, 'o': 2, 'w': 1, 'r': 1, 'd': 1}
```

Algorithm

I explored two methods to solve this task:

1. Using a dictionary to store counts:

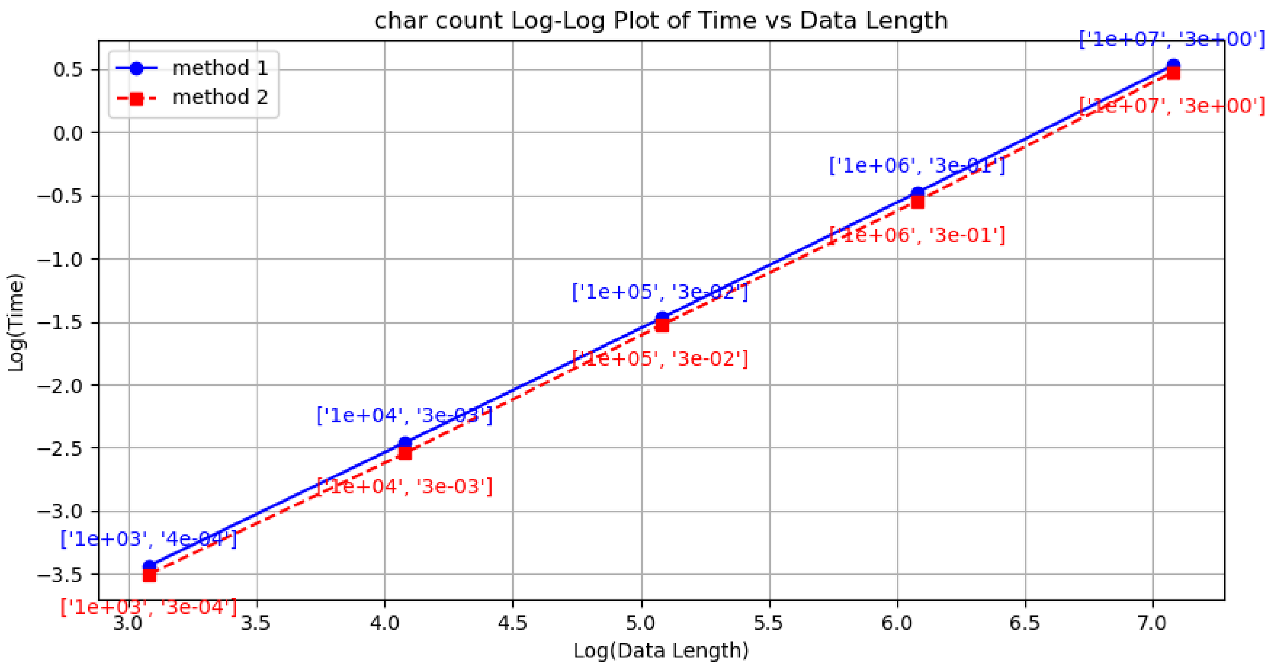
```
def char_count_1(s: str):
    out_dict = {}
    for i, _ in enumerate(s):
        if not s[i].islower():
            continue
        temp = out_dict.get(s[i], 0)
        out_dict[s[i]] = temp + 1
    return out_dict
```

2. Using a list to simulate the counters:

```
def char_count_2(s: str):
    counts = [0] * 26
    base = ord('a')

    for char in s:
        if 'a' <= char <= 'z':
            index = ord(char) - base
            counts[index] += 1

    out_dict = {}
    for i in range(26):
        if counts[i] > 0:
            out_dict[chr(i + base)] = counts[i]

    return out_dict
```

Experiment

We tested both methods across several data points, increasing the scale of the inputs from 10^3 to over 10^7 characters. The results are depicted in the chart where both methods display an O(n) complexity, but Method 2 performs slightly better.



Analysis

The initial results show that Method 2, which uses list indexing, performs slightly better than Method 1, which uses dictionary operations. This difference could be attributed to the direct memory access patterns in list indexing, which tend to be faster than the hashing mechanisms in dictionaries, especially for very large datasets. The performance difference becomes more noticeable as the input scale increases, aligning with the expected efficiencies of each method in handling large data volumes.

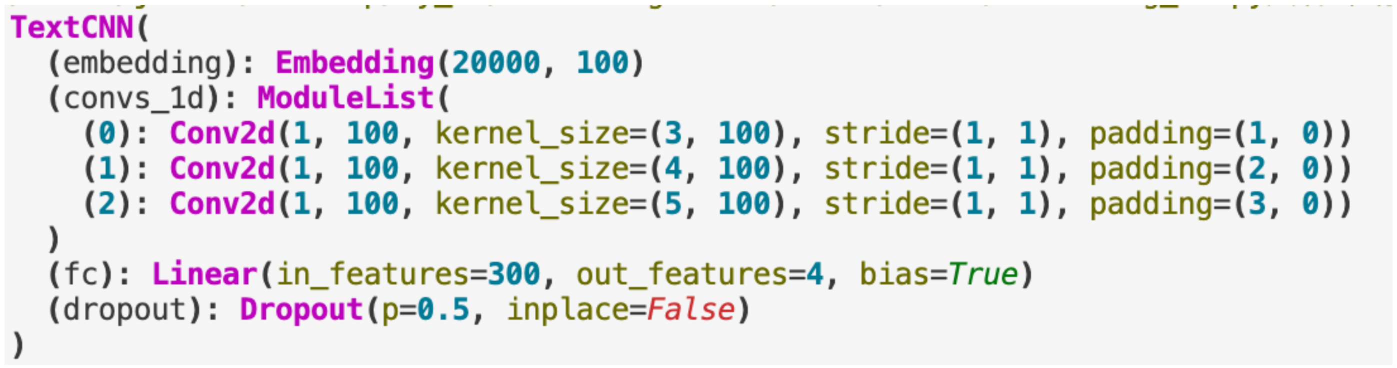
2. TASK2:

Task Description

Train an CNN to perform classification tasks on a database of Chinese sentences.

Module

Model Architecture

- 

```
TextCNN(
  (embedding): Embedding(20000, 100)
  (convs_1d): ModuleList(
    (0): Conv2d(1, 100, kernel_size=(3, 100), stride=(1, 1), padding=(1, 0))
    (1): Conv2d(1, 100, kernel_size=(4, 100), stride=(1, 1), padding=(2, 0))
    (2): Conv2d(1, 100, kernel_size=(5, 100), stride=(1, 1), padding=(3, 0))
  )
  (fc): Linear(in_features=300, out_features=4, bias=True)
  (dropout): Dropout(p=0.5, inplace=False)
)
```

- Embedding Layer: I employed the jieba tokenizer to break down sentences into tokens, selecting the top 20,000 words from the combined development, training, and test datasets to construct a word dictionary. I then trained a word2vec (Skip-gram) model to convert these tokens into fixed-size word vectors of 200 dimensions. Additionally, I expanded each sentence to 16 tokens using a padding token.
- Convolutional Layers: The model comprises three convolutional layers with kernel sizes of 3, 4, and 5, each generating 100 output channels.
- Pooling Layer: This follows the convolutional layers and serves to reduce the spatial dimensions.
- Fully Connected Layer: The outputs from the convolutional and pooling layers are flattened and input into a fully connected layer for classification. This layer has an input size of 300 and produces outputs for four classes.

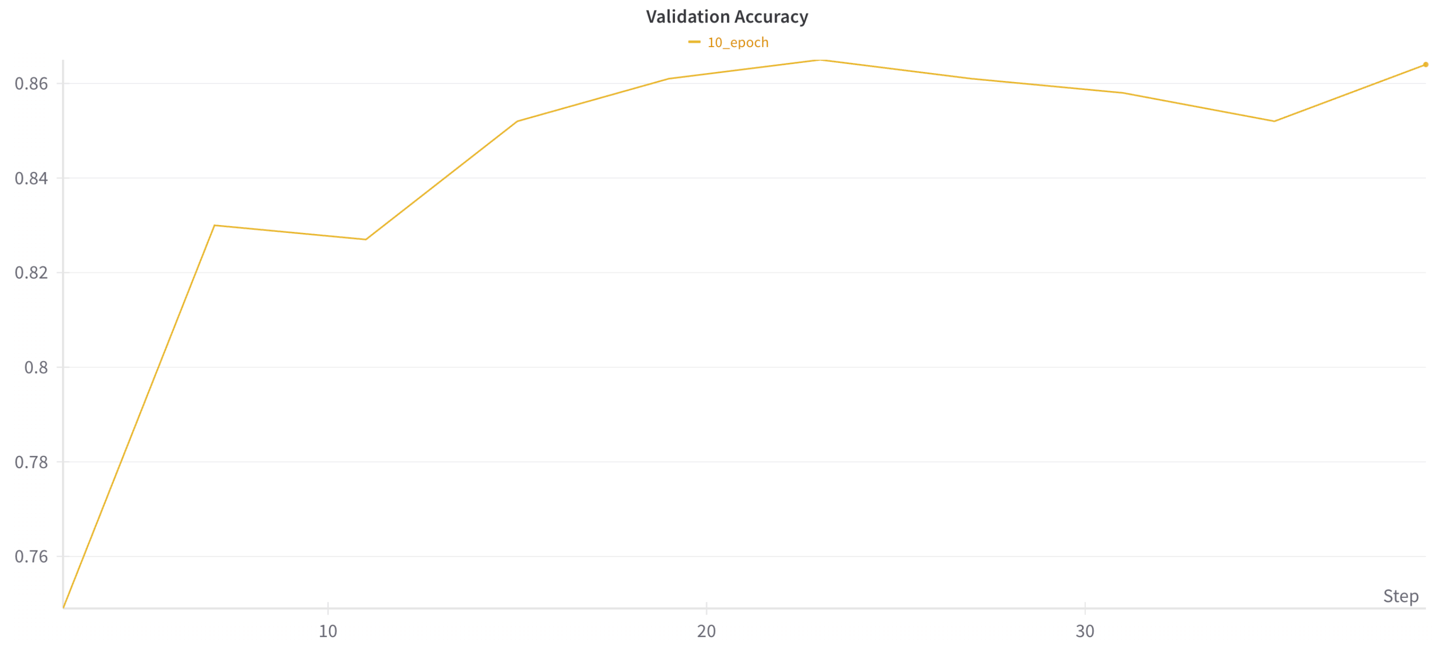
- Dropout Layer: To mitigate overfitting, a dropout layer with a rate of 0.5 randomly omits units during training.

Model Parameters

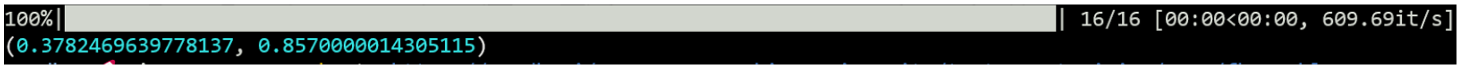
- Epochs: 30 epochs.
- Batch Size: 64.
- optimizer: Adam
- Learning Rate: 0.001.
- Dropout Probability: 0.5.
- Early Stopping: Stops training if there is no improvement on the validation loss for 5 consecutive epochs with a minimum delta change set to 0.

Performance

the validation accuracy is as below



The final performance of the model is 85.7%



```
100%|████████████████████████████████████████████████████████████| 16/16 [00:00<00:00, 609.69it/s]
(0.3782469639778137, 0.8570000014305115)
```

3. TASK3:

Task Description

Train a LSTM with attention，which can translate Japaness to English.

Pipeline

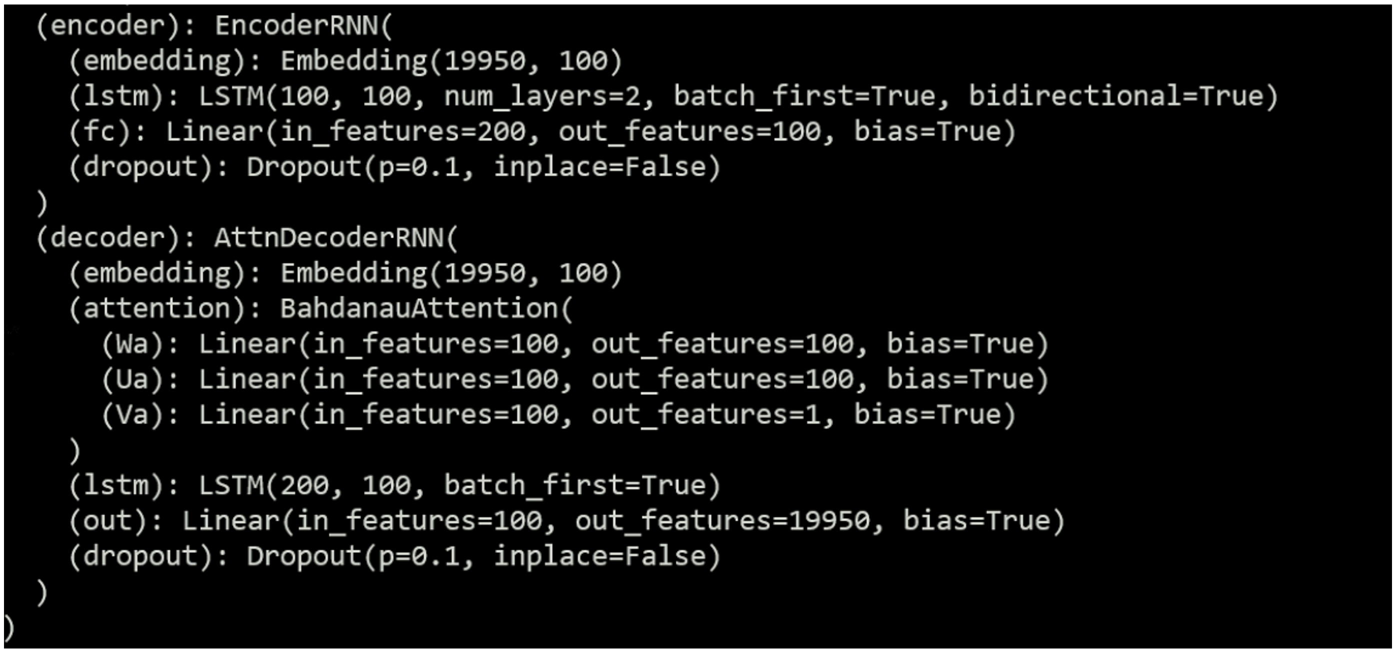
1. First, I fashioned a vocabulary by instituting a BPE-based tokenizer. I commenced by transmuting all uppercase letters into lowercase, followed by appending an underscore “_” as a terminus to each word and punctuation mark. Subsequently, I segregated each sentence, partitioning Japanese by sentences and English by words. Thereafter, adhering to the standard BPE algorithm, I dissected each unit and gradually augmented the lexicon based on frequency, ultimately constructing a vocabulary with 9,998 Japanese subtokens, 9,998 English subtokens, totaling 19,946— due t o the presence of numerals and punctuation marks common to both languages within the bilingual corpus.
2. To train a word2vec model, I opted for the skip-gram model, given the generally diminutive sentence lengths within the dataset. Upon convergence, I preserved the embeddings to serve as pre-trained word embeddings for an LSTM.
3. I orchestrated an LSTM with an Attention mechanism, largely adhering to the architecture delineated at https://pytorch.org/tutorials/intermediate/seq2seq_translation_tutorial.html. I modified the model by substituting the GRU with a bidirectional, two-layer LSTM.

The Structure of LSTM with attention

My RNN is a seq2seq model that contains an encoder and a decoder.

The encoder utilizes an embedding layer followed by a 10% dropout layer. After embedding, the inputs are processed through a bidirectional two-layer LSTM. The output vector of the LSTM is reduced from 200 dimensions to 100, while the four types of hidden layers are combined.

The decoder comprises an embedding layer with a 10% dropout, an additive attention mechanism, and a straightforward LSTM. Following this, a fully connected (FC) layer and a softmax layer are employed to transform the feature vector into a distribution.

- 

```
  (encoder): EncoderRNN(
    (embedding): Embedding(19950, 100)
    (lstm): LSTM(100, 100, num_layers=2, batch_first=True, bidirectional=True)
    (fc): Linear(in_features=200, out_features=100, bias=True)
    (dropout): Dropout(p=0.1, inplace=False)
  )
  (decoder): AttnDecoderRNN(
    (embedding): Embedding(19950, 100)
    (attention): BahdanauAttention(
      (Wa): Linear(in_features=100, out_features=100, bias=True)
      (Ua): Linear(in_features=100, out_features=100, bias=True)
      (Va): Linear(in_features=100, out_features=1, bias=True)
    )
    (lstm): LSTM(200, 100, batch_first=True)
    (out): Linear(in_features=100, out_features=19950, bias=True)
    (dropout): Dropout(p=0.1, inplace=False)
  )
)
```

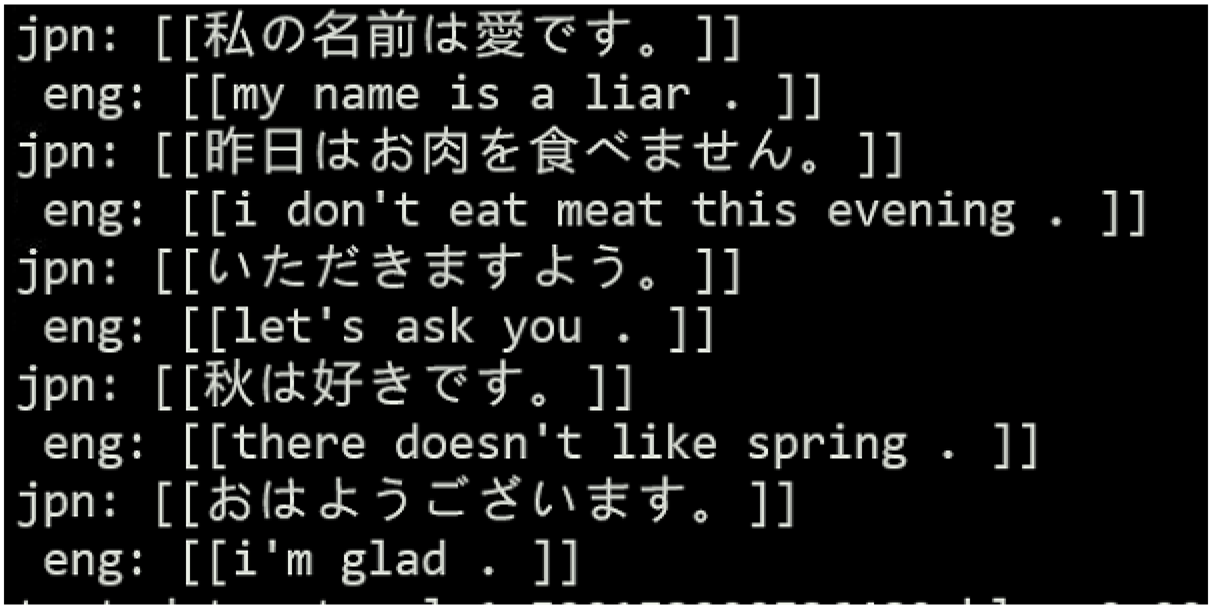
Model Parameters

- word2vet:
  - context window: 2
  - negative sampling number: 5
  - negative sampling temperature: 0.75
  - learning Rate: 0.025
  - epochs: 30 epochs
  - batch Size: 150
  - optimizer: Adam
  - early Stopping:
    - patience: 5 epoch
    - delta: 0
- LSTM:
  - epoches: 100
  - learning rate: 0.001
  - batch size:128
  - optimizer: Adam
  - sequence length: 50
  - early Stopping:
    - patience: 5 epoch
    - delta: 0

Performance

- BLEU(4-gram) and PPL score after training for 50 epochs.

split	PPL	BLEU
train	1.154	0.521
valid	1.568	0.226
test	1.590	0.220

- the translation of the following sentences in Japaness.
  - 

```
jpn: [[私の名前は愛です。]]
 eng: [[my name is a liar . ]]
jpn: [[昨日はお肉を食べません。]]
 eng: [[i don't eat meat this evening . ]]
jpn: [[いただきますよう。]]
 eng: [[let's ask you . ]]
jpn: [[秋は好きです。]]
 eng: [[there doesn't like spring . ]]
jpn: [[おはようございます。]]
 eng: [[i'm glad . ]]
```

Discussion

- When constructing the vocabulary, the construction speed for English is much faster than for Japanese because Japanese is constructed using sentences.
- English should ideally be aggregated into whole words as much as possible because the dataset is too small, subwords can hardly convey any information.