Natural Language Processing: Python, RNN & CNN / project #1

[Boyuan Chen] [2200017816]

Project Code: `https://github.com/cby-pku/pku_nlp_2024/tree/main/hw1`

1 Task 1: Python Basics

Question: Solving the two simple tasks. (1) Flatten a list of lists. (2) Character count (only for lower case).

Answer: Code details can be found in `https://github.com/cby-pku/pku_nlp_2024/tree/main/hw1`.

Question: Then you need to design a large-scale experiment by yourself, i.e., increase the scale of the inputs. For example, you can prepare some very large lists that contain $> 10^7$ elements after flatting. Gradually increase the input scale (from 10^3, 10^4, 10^5, ..., 10^7) and see how the time increase with your algorithm.

Answer: We designed a task to scale the input size, gradually increasing the input from 10^3, 10^4, 10^5, ..., 10^7, and tested the speed of the implemented code. The results are shown in Table 1.

From the experimental data, we observe that for the task `Flatten a list of lists`, the algorithm achieves a time complexity of $O(n)$. However, for the `Character count` task, the current implementation exhibits a time complexity of $O(n^2)$. This insight suggests potential opportunities for further algorithmic optimization.

Table 1: Scale-Time.

Task & Scale	10^3	10^4	10^5	10^6	10^7
Flatten	0.000020 (s)	0.000170	0.001932	0.016332	0.18324
Count	0.002302	0.048451	4.048843	400.83545	-

Question: Can you come up with multiple ways of solving the above two problems (at least 2 different ways. e.g., using for-loop and using Python list comprehension are two different ways)?

Answer: We implemented three different approaches to solve the problems, corresponding to the code in `submission.py`, `submission_dev0.py`, and `submission_dev1.py`. The specific implementations are as follows:

- **Task 1: Flatten a list of lists**
  - **Approach A: Using List Comprehension.** This approach iterates through each sublist, then through each item in the sublist, flattening the list into a single list.
  - **Approach B: Using a `for` loop.** This method iterates over the sublists and appends each item to a result list.
  - **Approach C: Recursion.** We recursively flatten the list, which can handle deeply nested structures.
- **Task 2: Character count (only for lowercase).**
  - **Approach A: Using `str.count`.** This method directly calls 'str.count' inside a dictionary comprehension to count occurrences of each character.
  - **Approach B: Using a `for` loop and `Counter`.** We utilize Python's 'Counter' to count characters, then filter out the non-lowercase ones.
  - **Approach C: Only using a `for` loop.** A manual counting method that iterates through the string and counts each character without using any external libraries.

Table 2: Scale-Time-Methods.

Task & Scale	10^3	10^4	10^5	10^6	10^7
Flatten-A	0.000020 (s)	0.000170	0.001932	0.016332	0.18324
Flatten-B	0.000107	0.000978	0.008983	0.078537	0.808842
Flatten-C	0.000031	0.000273	0.002579	0.023635	0.224532
Count-A	0.002302	0.048451	4.048843	400.83545	-
Count-B	0.016337	0.000459	0.004446	0.044471	0.449641
Count-C	0.001495	0.000897	0.009264	0.101486	0.913374

The results are shown in Table 2.

From the experimental data, we observe the following: For `Task 1: Flatten a list of lists`, all three implemented methods exhibit a time complexity of $O(n)$. Among these, the approach using List Comprehension proves to be more efficient compared to both the `for` loop and Recursion, and scales better with large inputs.

For `Task 2: Character count (only for lowercase)`: List Comprehension with `str.count`: This method is inefficient with a time complexity of $O(n^2)$, making it unsuitable for large strings. Using `collections.Counter` and Manual Count: Both approaches

have a time complexity of $O(n)$ and are highly efficient, with `Counter` being the more concise and optimized solution.

Specifically, in Task 2, Approach B & C, we observed anomalous timing results for input sizes of 10^3 and 10^4. This could be related to the characteristics of our input data or optimizations in the underlying system affecting Python's execution time.Since in our code, we controlled the sequence of time measurement to minimize any extraneous time costs introduced by other operations, as shown below:

```
start_time = time.time()
result = function(input_data)
end_time = time.time()
```

2 Task2: Text Classification with CNN

Question: Read this paper, and implement a CNN-based neural network for sentence classification (Chinese). The datasets are already processed as follows (each line is a datapoint with text + label), and you need to construct the vocabulary by yourself (you may need `jieba` to tokenize sentences and construct a vocabulary).

Please make sure that:

- You cannot use any pre-trained model or pre-trained word vectors. The training should be from scratch.
- The accuracy should be higher than 75% on the test set.
- You cannot use the test set to tune the hyper-parameters. You should implement **early stopping** with the dev set.
  - If you are not familiar with early stopping (which is a popular regularization method in machine learning), just google it.
- You need to submit a tiny report that contains:
  1. Configurations of your CNN model.
  2. Classification accuracy on the test set.

Answer: We implemented a CNN-based architecture for text classification, utilizing a combination of embedding, convolutional, and fully connected layers. The model's input is passed through an embedding layer, which transforms word indices into dense vectors of a predefined size, `embed_size`. This embedding layer is followed by multiple 1D convolutional layers with varying kernel sizes. Specifically, we used filter sizes of 3, 4, and

5 to capture different n-gram features, and each filter size was assigned 100 convolutional filters. This allows the model to extract local features across different contexts.

After applying the convolution operations, the model applies a ReLU activation function to introduce non-linearity. Max-pooling is then performed on the output of each convolution to retain only the most salient features for each filter. The outputs from the different filters are concatenated along the feature dimension.

To reduce the risk of overfitting, we applied a dropout layer with a dropout rate of 0.5 after the concatenation step. Finally, the pooled features are passed through a fully connected layer, which maps the combined features to the desired number of output classes.

This architecture allowed us to effectively capture important textual patterns while maintaining regularization through dropout.

The Table 3 presents the details and configuration of the CNN model used for training.

Table 3: Training Details and Configuration of CNN Model.

Parameter	**Value**
Number of Epochs	10
Learning Rate	0.001
Batch Size	32
Embedding Size	128

Table 4 summarizes the final training results, including accuracy and loss metrics. And we report results by `wandb.ai`, related loss and accuracy loss can be referred to Figure 1.

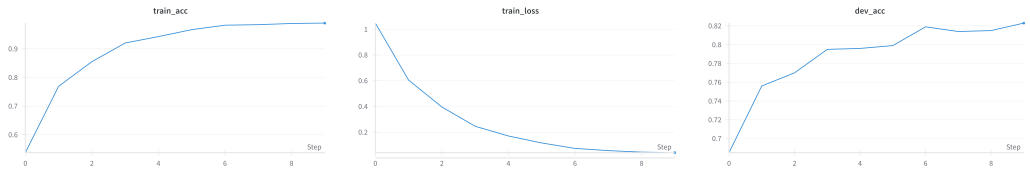

Figure 1: Training & Validation Accuracy and Training Loss.

In conclusion, we successfully implemented a CNN-based neural network for sentence classification in Chinese. The model was trained from scratch without using any pre-trained vectors. After training for 10 epochs, we achieved an accuracy of 85.4% on the test set, surpassing our target accuracy of 75%.

Table 4: Final Training Results.

Metric	Value
Training Accuracy	0.990
Training Loss	0.0408
Development Set Accuracy	0.823
Test Set Accuracy	0.854
Number of Epochs	10

3 Task 3: Machine Translation with RNN

Question: In this part, you have to solve Japanese to English machine translation task with recurrent neural network. In your submission, you should show how to run your model clearly. (Wrapping in a .sh file is advised). You also need to include requirements.txt.

- Construct the vocabulary by yourself. You need to choose an appropriate way to tokenize Japanese and English. You need to randomly split the corpus into training set, validation set, and test set (the proportion = 8:1:1).
- Train your word embedding using training set with any word embedding algorithm you like (e.g., CBOW, Skip-gram). Evaluate your trained word embedding with appropriate methods.
- Implement an LSTM with attention mechanism, and use the pre-trained word embedding as the embedding layer. Train your neural network on the training set.
- Evaluate your LSTM with BLEU score and Perplexity. Try to improve the BLEU score and Perplexity via some tricks (e.g., larger network. . .) we don't require a very good BLEU but at least the model should output decent English words.
- You need to submit a tiny report that contains (1) configurations of your RNN model; (2) BLEU score and perplexity on training, validation, and test set (test set can not be seen during the entire training phase). (3) Show the prediction of your model on the following test case. (4) some analysis.

Answer: In this task, we implemented Japanese to English machine translation. For Japanese tokenization, we used MeCab, and for English, we used NLTK's word tokenizer. All code can be found at `https://github.com/cby-pku/pku_nlp_2024/tree/main/hw1/task3`, and instructions for reproduction are provided in the README at `https://github.com/cby-pku/pku_nlp_2024/blob/main/hw1/README.md`.

Pretraining of CBOW. In this experiment, we trained a CBOW model to learn word embeddings for both Japanese and English corpora. The model consists of an embedding layer that transforms each word into a dense vector, followed by averaging the context vectors and passing them through a linear layer to predict the target word.

We set the embedding dimension to 256, balancing complexity and the ability to capture word semantics. The context window was 2 words on each side of the target word, allowing the model to capture local dependencies effectively.

We used the Adam optimizer with a learning rate of 0.001 for efficient convergence and `NLLLoss` for prediction. The model was trained for 5 epochs with a batch size of 64, and its performance was evaluated after each epoch on the validation set. These embeddings formed the foundation for subsequent sequence-to-sequence translation tasks.

The final test accuracy can be found in Table 5.

Table 5: CBOW models' accuracy.

CBOW model	Test Accuracy
Japanese	53.28%
English	49.03%

Implementation details of LSTM. In our experiment, we implemented an LSTM-based sequence-to-sequence model with attention for Japanese to English translation. We used pre-trained CBOW embeddings for both Japanese and English vocabularies.

The encoder is a bidirectional LSTM with an embedding layer initialized using Japanese CBOW embeddings. We set the hidden dimension to 512 for capturing sufficient context while maintaining efficiency.

The attention mechanism computes alignment scores between the decoder's hidden state and encoder outputs. These scores are used to create a weighted context vector for each time step.

The decoder consists of an embedding layer initialized with English CBOW embeddings, followed by an LSTM and a fully connected layer. The attention context vector is concatenated with the current word embedding at each time step.

We used an embedding size of 256, a hidden size of 512, and trained with the Adam optimizer at a learning rate of 0.001. The model was trained for up to 10 epochs with a batch size of 64, using gradient clipping (threshold of 1) and early stopping (patience of 5 epochs) to prevent overfitting and ensure stable training.

Summary of implementation details can be found in Table 6.

Table 6: RNN Model Configurations.

Configuration Parameter	**Value**
Embedding Dimension	256
Hidden Dimension (LSTM)	512
Learning Rate	0.001
Optimizer	Adam
Batch Size	64
Gradient Clipping	1.0
Early Stopping	Patience 5, Min Delta 0.001
Teacher Forcing Ratio	0.5
Padding Token	`<PAD>`
Tokenization (Japanese)	MeCab
Tokenization (English)	NLTK
Pre-trained Embeddings	CBOW (256 dim)

Experiment Results. We reported the final training, validation, and test dataset perplexity, loss, and BLEU score, as shown in Table 7.

Table 7: Experiment Results.

Dataset	Loss	BLEU Score	Perplexity
Training Split	1.864	6.06	6.45
Test Split	2.425	4.37	20.73
Validation Split	2.409	4.32	20.24

We also evaluated the model on sample cases, with the output translations provided below as shown in Table 8:

We also found that introducing special tokens such as `<SOS>`, `<EOS>`, `<UNK>`, and `<PAD>` during vocabulary construction contributed to greater training stability and more reliable inference, reducing unintended behaviors like repetitive outputs.

Case	English Translation
Case 1	my name is love. love... ” love.” love.” love.” love.” love.” love.” love.” love.” love.” love.” love.” love.” love.” love.”
Case 2	i does n’t meat loaf meat yesterday. meat, meat.” meat meat.” meat.” meat.” meat.” meat.” meat.” meat.” meat.” meat.”
Case 3	a thank you do... ” .”
Case 4	i are fond of likes science... ” .” .” .” .” .” .” .” .” .” .” .” .” .” .” .” .”
Case 5	you have it for you... ”

Table 8: Case Study: Translation Outputs