EIE4122 Deep Learning and Deep Neural Networks

Lab 2: RNN and LSTM for Text Analysis

HAO Jiadong (20084595d)

Recurrent Neural Network (RNN)

1. Data Preprocessing

a. Dataset used

The dataset, Large Movie Review Dataset, is for binary sentiment classification. In our experiment, to save training time, we only used 2780 records for both training and testing. We further split the 2780 training samples into a 70% training set and a 30% validation set to select the model with the best performance. Finally, Figure 1 shows the dataset splitting results.

```
Number of training examples: 1946
Number of validation examples: 834
Number of testing examples: 2780
```

Figure1. Dataset Splitting

b. Tokenization using spaCy

To enable our RNN to learn the sentences at word-level, we tokenize the sentences into smaller units called tokens using a tokenizer, en_core_web_sm, in spaCy library.

c. Build the vocabulary

To enable our model to take in the tokens, we give each token a unique one-hot vector.

For TEXT INPUT,

To reduce the number of one-hot-vectors, we only keep the most frequent 25,000 words and replace other encoding vectors with an unknown token denoted by <unk>. Additionally, to keep all sentences in a batch to have the same size in order to feed to the model in one run, we use an additional token <pad> for padding. Hence, the total number of tokens for text input is 25,002.

For LABEL,

Since we only have two labels, positive and negative, we use 0 to represent negative labels and 1 to represent positive labels.

d. BucketIterator

We create BucketIterators to hold a batch of samples (batch size =64) to enable parallel processing. BucketIterator ensures sentences of similar length are grouped into the same batch, significantly reducing the padding efforts.

2. Model Architecture

Layer	**Function**
Embedding layer	Turn sparse one-hot vectors into dense vectors for dimensionality reduction. Words of similar meanings are closer in the dense vector space.
RNN layer	Based on the previous hidden state h(t-1) and the current word's dense vector, generate the current hidden state h(t).
Fully-connect linear layer	Take in the final hidden state and output the predicted possibility of positive class.

Table 1. Three Layers and Corresponding Functions

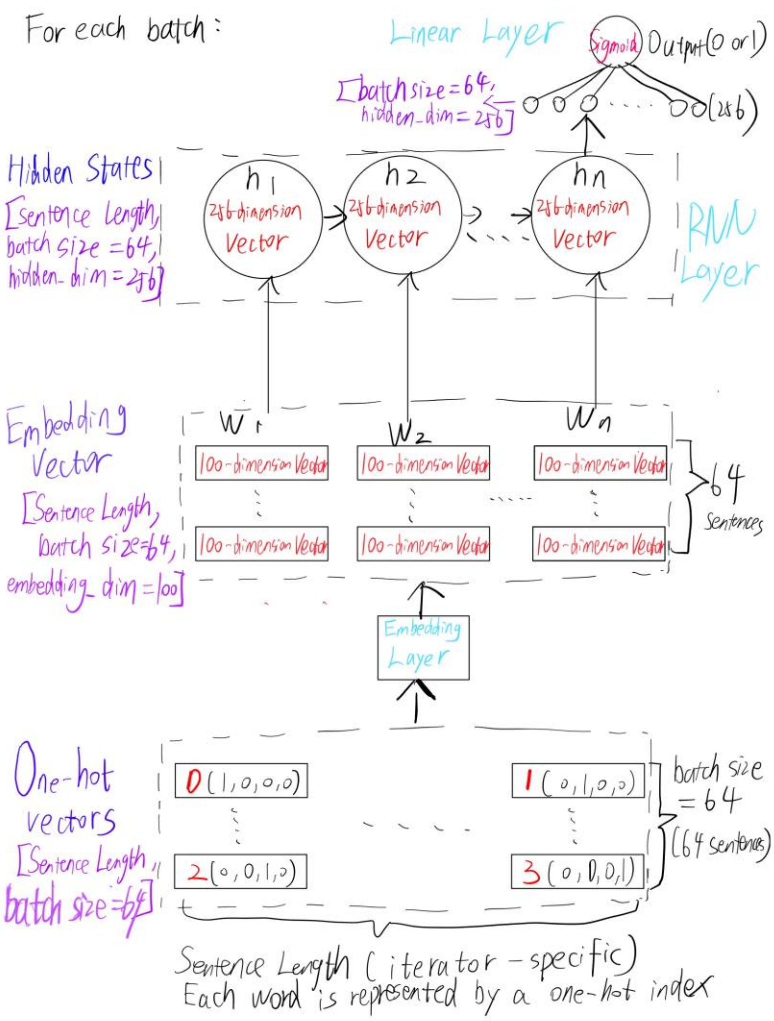

Figure 2. Overall Architecture of RNN

3. Model Training

a. SGD optimizer

stochastic gradient descent (SGD) optimizer with a learning rate of 0.001 is used to update all the parameters based on the gradients of the loss function with respect to the parameters.

b. Binary cross entropy with logits

Binary cross entropy with logits is adopted as the loss function. “nn.BCEWithLogitsLoss()” first applies a sigmoid function to the raw output of RNN and then calculates the loss using binary cross entropy.

The objective of the sigmoid function is to convert the unbound last hidden state of RNN to a value between 0 and 1 (referring to Figure 3) as the predicted possibility of the positive class.

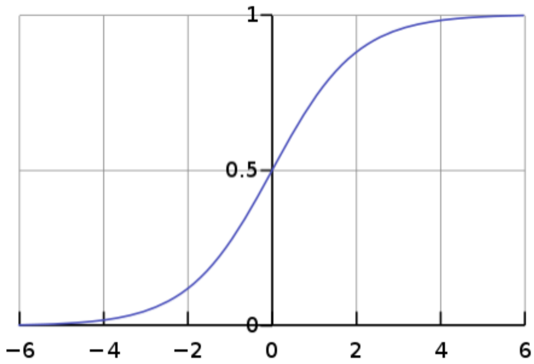

Figure 3. Curve for Sigmoid

The formula for cross-entropy loss is shown in Figure 4, where y represents the actual label (0 or 1), and p(y) represents the predicted possibility of the positive class.

$$H_p(q) = -\frac{1}{N}\sum_{i=1}^{N} y_i \cdot log(p(y_i)) + (1 - y_i) \cdot log(1 - p(y_i))$$

Figure 4. BCE Loss Formula

The main idea of cross-entropy loss is to penalize the predicted probability that deviates from the actual label. When the true label y = 1, the loss term is -log(p(y)) as shown on the left of Figure 5. The model will impose a high penalty on those “wrong predictions” with p(y) close to 0 and a small penalty on those “correct predictions” with p(y) close to 1. When the true label y = 0, the loss term is -log(1-p(y)) as shown on the right of Figure 5. The model will impose a high penalty on those “wrong predictions” with p(y) close to 1 and a small penalty on those “correct predictions” with p(y) close to 0.

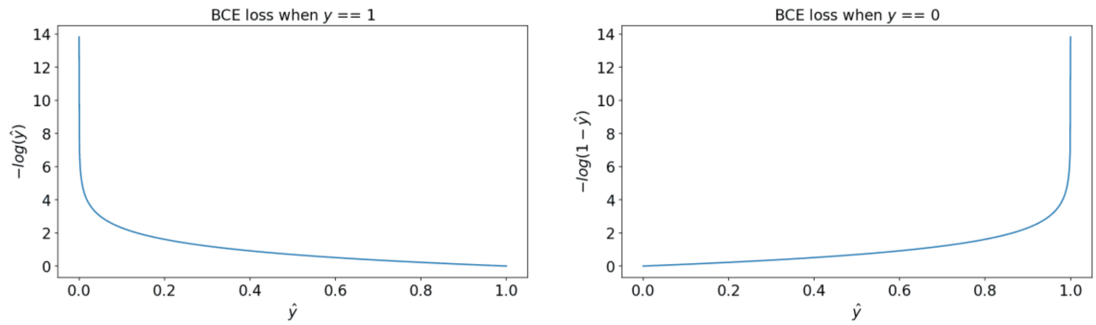

Figure 5. BCE Loss for y = 1 and y = 0

4. Model Evaluation and Modification

a. Original architecture

Observation: From Figure 6 and 7, the performance of the original model is very poor, with training, validation, and testing accuracy around 50%, like random guessing.

```
Epoch: 01 | Epoch Time: 0m 2s
        Train Loss: 0.697 | Train Acc: 50.07%
         Val. Loss: 0.696 |  Val. Acc: 50.22%
Epoch: 02 | Epoch Time: 0m 1s
        Train Loss: 0.694 | Train Acc: 49.90%
         Val. Loss: 0.696 |  Val. Acc: 50.22%
Epoch: 03 | Epoch Time: 0m 1s
        Train Loss: 0.693 | Train Acc: 49.83%
         Val. Loss: 0.696 |  Val. Acc: 50.45%
Epoch: 04 | Epoch Time: 0m 1s
        Train Loss: 0.693 | Train Acc: 49.32%
         Val. Loss: 0.697 |  Val. Acc: 50.33%
Epoch: 05 | Epoch Time: 0m 1s
        Train Loss: 0.693 | Train Acc: 47.61%
         Val. Loss: 0.697 |  Val. Acc: 50.67%
Epoch: 06 | Epoch Time: 0m 1s
        Train Loss: 0.693 | Train Acc: 49.35%
         Val. Loss: 0.697 |  Val. Acc: 46.09%
Epoch: 07 | Epoch Time: 0m 1s
        Train Loss: 0.693 | Train Acc: 50.41%
         Val. Loss: 0.697 |  Val. Acc: 45.65%
Epoch: 08 | Epoch Time: 0m 1s
        Train Loss: 0.693 | Train Acc: 47.69%
         Val. Loss: 0.697 |  Val. Acc: 46.21%
Epoch: 09 | Epoch Time: 0m 1s
        Train Loss: 0.693 | Train Acc: 50.35%
         Val. Loss: 0.697 |  Val. Acc: 44.87%
Epoch: 10 | Epoch Time: 0m 1s
        Train Loss: 0.693 | Train Acc: 50.17%
         Val. Loss: 0.697 |  Val. Acc: 45.65%
```

Figure 6. Training Results for the Original RNN Model

```
Test Loss: 0.697 | Test Acc: 49.68%
```

Figure 7. Testing Results for the Original RNN Model

b. Add one more RNN layer

As shown in Figure 8, we add one more RNN layer.

```
import torch.nn as nn

class RNN(nn.Module):
    def __init__(self, input_dim, embedding_dim, hidden_dim, output_dim):
        super().__init__()
        self.embedding = nn.Embedding(input_dim, embedding_dim)
        self.rnn1 = nn.RNN(embedding_dim, hidden_dim)
        self.rnn2 = nn.RNN(hidden_dim, hidden_dim)
        self.fc = nn.Linear(hidden_dim, output_dim)

    def forward(self, text):
        embedded = self.embedding(text)
        output1, h1 = self.rnn1(embedded)
        output2, h2 = self.rnn2(output1)
        return self.fc(h2.squeeze(0))
```

Figure 8. Add one more RNN Layer

Observation: From Figures 9 and 10, the performance of the revised model with two RNN layers is still unacceptable. Theoretically, an additional RNN layer will contribute to generating a more abstract representation of the sentences to capture the long-range dependencies and extract more nuanced features. The reason why the performance doesn't improve is discussed in section e (conclusion).

```
Epoch: 01 | Epoch Time: 0m 2s
        Train Loss: 0.698 | Train Acc: 50.15%
         Val. Loss: 0.702 |  Val. Acc: 45.65%
Epoch: 02 | Epoch Time: 0m 1s
        Train Loss: 0.695 | Train Acc: 50.62%
         Val. Loss: 0.700 |  Val. Acc: 45.54%
Epoch: 03 | Epoch Time: 0m 1s
        Train Loss: 0.694 | Train Acc: 50.30%
         Val. Loss: 0.699 |  Val. Acc: 45.54%
Epoch: 04 | Epoch Time: 0m 1s
        Train Loss: 0.694 | Train Acc: 50.62%
         Val. Loss: 0.698 |  Val. Acc: 45.42%
Epoch: 05 | Epoch Time: 0m 1s
        Train Loss: 0.693 | Train Acc: 50.80%
         Val. Loss: 0.698 |  Val. Acc: 44.98%
Epoch: 06 | Epoch Time: 0m 1s
        Train Loss: 0.693 | Train Acc: 50.43%
         Val. Loss: 0.697 |  Val. Acc: 45.09%
Epoch: 07 | Epoch Time: 0m 1s
        Train Loss: 0.693 | Train Acc: 50.33%
         Val. Loss: 0.697 |  Val. Acc: 44.98%
Epoch: 08 | Epoch Time: 0m 1s
        Train Loss: 0.693 | Train Acc: 50.55%
         Val. Loss: 0.697 |  Val. Acc: 45.20%
Epoch: 09 | Epoch Time: 0m 2s
        Train Loss: 0.693 | Train Acc: 50.31%
         Val. Loss: 0.697 |  Val. Acc: 44.98%
Epoch: 10 | Epoch Time: 0m 2s
        Train Loss: 0.693 | Train Acc: 50.40%
         Val. Loss: 0.697 |  Val. Acc: 45.09%
```

Figure 9. Training Results for RNN with Two Layers

```
Test Loss: 0.693 | Test Acc: 49.96%
```

Figure 10. Testing Results for RNN with Two Layers

c. Change the embedding dimension of the RNN

As shown in Figures 11 and 12, we increase the embedding dimension (the dimension of the dense vectors to be fed into the RNN) from 100 to 200 and 1000.

```
INPUT_DIM = len(TEXT.vocab)
#EMBEDDING_DIM = 100
EMBEDDING_DIM = 200
HIDDEN_DIM = 256
OUTPUT_DIM = 1

model = RNN(INPUT_DIM, EMBEDDING_DIM, HIDDEN_DIM, OUTPUT_DIM)
```

Figure 11. Increase the Embedding Dimension to 200

```
INPUT_DIM = len(TEXT.vocab)
#EMBEDDING_DIM = 100
EMBEDDING_DIM = 1000
HIDDEN_DIM = 256
OUTPUT_DIM = 1

model = RNN(INPUT_DIM, EMBEDDING_DIM, HIDDEN_DIM, OUTPUT_DIM)
```

Figure 12. Increase the Embedding Dimension to 1000

Observation: From Figure 13, 14, 15, 16, the performance of the revised models with increased embedding dimension is still unacceptable. Theoretically, more embedding dimension will provide richer semantic features to increase the performance. The reason why the performance doesn't improve is discussed in section e (conclusion).

```
Epoch: 01 | Epoch Time: 0m 3s
        Train Loss: 0.696 | Train Acc: 49.77%
         Val. Loss: 0.689 |  Val. Acc: 54.13%
Epoch: 02 | Epoch Time: 0m 1s
        Train Loss: 0.694 | Train Acc: 49.75%
         Val. Loss: 0.690 |  Val. Acc: 55.13%
Epoch: 03 | Epoch Time: 0m 1s
        Train Loss: 0.693 | Train Acc: 49.72%
         Val. Loss: 0.691 |  Val. Acc: 49.89%
Epoch: 04 | Epoch Time: 0m 1s
        Train Loss: 0.694 | Train Acc: 49.35%
         Val. Loss: 0.690 |  Val. Acc: 52.12%
Epoch: 05 | Epoch Time: 0m 1s
        Train Loss: 0.693 | Train Acc: 49.12%
         Val. Loss: 0.690 |  Val. Acc: 50.11%
Epoch: 06 | Epoch Time: 0m 1s
        Train Loss: 0.693 | Train Acc: 50.43%
         Val. Loss: 0.691 |  Val. Acc: 49.11%
Epoch: 07 | Epoch Time: 0m 1s
        Train Loss: 0.693 | Train Acc: 49.78%
         Val. Loss: 0.691 |  Val. Acc: 49.33%
Epoch: 08 | Epoch Time: 0m 1s
        Train Loss: 0.693 | Train Acc: 47.44%
         Val. Loss: 0.690 |  Val. Acc: 49.78%
Epoch: 09 | Epoch Time: 0m 1s
        Train Loss: 0.693 | Train Acc: 50.23%
         Val. Loss: 0.690 |  Val. Acc: 54.24%
Epoch: 10 | Epoch Time: 0m 1s
        Train Loss: 0.693 | Train Acc: 49.97%
         Val. Loss: 0.691 |  Val. Acc: 49.44%
```

Figure 13. Training Results for RNN with Embedding Dimension = 200

```
Test Loss: 0.694 | Test Acc: 49.07%
```

Figure 14. Testing Results for RNN with Embedding Dimension = 200

```
Epoch: 01 | Epoch Time: 0m 2s
	Train Loss: 0.694 | Train Acc: 49.75%
	 Val. Loss: 0.706 |  Val. Acc: 50.00%
Epoch: 02 | Epoch Time: 0m 1s
	Train Loss: 0.694 | Train Acc: 49.47%
	 Val. Loss: 0.707 |  Val. Acc: 50.11%
Epoch: 03 | Epoch Time: 0m 1s
	Train Loss: 0.695 | Train Acc: 50.05%
	 Val. Loss: 0.707 |  Val. Acc: 44.20%
Epoch: 04 | Epoch Time: 0m 1s
	Train Loss: 0.694 | Train Acc: 48.22%
	 Val. Loss: 0.708 |  Val. Acc: 44.75%
Epoch: 05 | Epoch Time: 0m 1s
	Train Loss: 0.694 | Train Acc: 50.25%
	 Val. Loss: 0.705 |  Val. Acc: 49.89%
Epoch: 06 | Epoch Time: 0m 1s
	Train Loss: 0.695 | Train Acc: 48.69%
	 Val. Loss: 0.705 |  Val. Acc: 44.20%
Epoch: 07 | Epoch Time: 0m 2s
	Train Loss: 0.694 | Train Acc: 48.41%
	 Val. Loss: 0.705 |  Val. Acc: 44.20%
Epoch: 08 | Epoch Time: 0m 2s
	Train Loss: 0.694 | Train Acc: 49.99%
	 Val. Loss: 0.707 |  Val. Acc: 44.75%
Epoch: 09 | Epoch Time: 0m 1s
	Train Loss: 0.694 | Train Acc: 49.34%
	 Val. Loss: 0.706 |  Val. Acc: 44.75%
Epoch: 10 | Epoch Time: 0m 1s
	Train Loss: 0.694 | Train Acc: 50.16%
	 Val. Loss: 0.704 |  Val. Acc: 44.42%
```

Figure 15. Training Results for RNN with Embedding Dimension = 1000

```
Test Loss: 0.698 | Test Acc: 49.33%
```

Figure 16. Testing Results for RNN with Embedding Dimension = 1000

d. Reduce the dimension of the word embeddings (vocabulary size)

As shown in Figures 17 and 18, we reduce the dimension of the word embeddings (vocabulary size) from 25,000 to 20,000 and 10,000.

```
# MAX_VOCAB_SIZE = 25_000      # In Python 25_000 means 25000
MAX_VOCAB_SIZE = 20_000
TEXT.build_vocab(train_data, max_size = MAX_VOCAB_SIZE)
LABEL.build_vocab(train_data)
```

Figure 17. Reduce the Dimension of Word Embeddings to 20,000

```
# MAX_VOCAB_SIZE = 25_000      # In Python 25_000 means 25000
MAX_VOCAB_SIZE = 10_000
TEXT.build_vocab(train_data, max_size = MAX_VOCAB_SIZE)
LABEL.build_vocab(train_data)
```

Figure 18. Reduce the dimension of Word Embeddings to 10,000

Observation: From Figure 19, 20, 21, 22, the performance of the revised models with reduced dimension of word embeddings are unacceptable. The reason why the performance is poor is discussed in section e (conclusion).

```
Epoch: 01 | Epoch Time: 0m 1s
        Train Loss: 0.696 | Train Acc: 50.28%
         Val. Loss: 0.696 |  Val. Acc: 48.55%
Epoch: 02 | Epoch Time: 0m 1s
        Train Loss: 0.694 | Train Acc: 50.53%
         Val. Loss: 0.694 |  Val. Acc: 48.44%
Epoch: 03 | Epoch Time: 0m 1s
        Train Loss: 0.693 | Train Acc: 50.70%
         Val. Loss: 0.693 |  Val. Acc: 48.77%
Epoch: 04 | Epoch Time: 0m 1s
        Train Loss: 0.693 | Train Acc: 49.98%
         Val. Loss: 0.692 |  Val. Acc: 48.44%
Epoch: 05 | Epoch Time: 0m 1s
        Train Loss: 0.693 | Train Acc: 50.28%
         Val. Loss: 0.692 |  Val. Acc: 48.77%
Epoch: 06 | Epoch Time: 0m 1s
        Train Loss: 0.693 | Train Acc: 48.26%
         Val. Loss: 0.692 |  Val. Acc: 48.55%
Epoch: 07 | Epoch Time: 0m 1s
        Train Loss: 0.693 | Train Acc: 50.85%
         Val. Loss: 0.692 |  Val. Acc: 48.55%
Epoch: 08 | Epoch Time: 0m 1s
        Train Loss: 0.694 | Train Acc: 49.81%
         Val. Loss: 0.691 |  Val. Acc: 48.44%
Epoch: 09 | Epoch Time: 0m 1s
        Train Loss: 0.693 | Train Acc: 50.55%
         Val. Loss: 0.691 |  Val. Acc: 48.55%
Epoch: 10 | Epoch Time: 0m 1s
        Train Loss: 0.693 | Train Acc: 49.67%
         Val. Loss: 0.692 |  Val. Acc: 48.66%
```

Figure 19. Training Results for RNN with Reduced Vocabulary Size = 20000

```
Test Loss: 0.693 | Test Acc: 50.78%
```

Figure 20. Testing Results for RNN with Reduced Vocabulary Size = 20000

```
Epoch: 01 | Epoch Time: 0m 1s
        Train Loss: 0.693 | Train Acc: 50.33%
         Val. Loss: 0.697 |  Val. Acc: 44.98%
Epoch: 02 | Epoch Time: 0m 1s
        Train Loss: 0.693 | Train Acc: 50.25%
         Val. Loss: 0.696 |  Val. Acc: 45.31%
Epoch: 03 | Epoch Time: 0m 2s
        Train Loss: 0.693 | Train Acc: 50.10%
         Val. Loss: 0.696 |  Val. Acc: 45.20%
Epoch: 04 | Epoch Time: 0m 1s
        Train Loss: 0.693 | Train Acc: 50.43%
         Val. Loss: 0.696 |  Val. Acc: 44.64%
Epoch: 05 | Epoch Time: 0m 1s
        Train Loss: 0.693 | Train Acc: 50.43%
         Val. Loss: 0.696 |  Val. Acc: 44.87%
Epoch: 06 | Epoch Time: 0m 1s
        Train Loss: 0.693 | Train Acc: 49.40%
         Val. Loss: 0.696 |  Val. Acc: 45.09%
Epoch: 07 | Epoch Time: 0m 1s
        Train Loss: 0.693 | Train Acc: 50.68%
         Val. Loss: 0.696 |  Val. Acc: 45.31%
Epoch: 08 | Epoch Time: 0m 1s
        Train Loss: 0.693 | Train Acc: 50.14%
         Val. Loss: 0.696 |  Val. Acc: 44.75%
Epoch: 09 | Epoch Time: 0m 1s
        Train Loss: 0.693 | Train Acc: 50.38%
         Val. Loss: 0.696 |  Val. Acc: 45.20%
Epoch: 10 | Epoch Time: 0m 1s
        Train Loss: 0.693 | Train Acc: 49.49%
         Val. Loss: 0.696 |  Val. Acc: 44.87%
```

Figure 21. Training Results for RNN with Reduced Vocabulary Size = 10000

```
Test Loss: 0.693 | Test Acc: 51.13%
```

Figure 22. Testing Results for RNN with Reduced Vocabulary Size = 10000

e. Conclusion
From the above three modifications and results, the observation is that no matter how we fine-tune the model, the performance remains poor, with an accuracy of around 50%. It seems that the model learns nothing but makes random guesses.

One potential reason is that the basic RNN's capacity to do sentiment analysis is limited. For example, RNNs suffer from gradient vanishing problems and thus make it challenging to capture the long-term dependencies. We should try other more powerful models like LSTM to gain better performance.

Long Short-term Memory (LSTM)

Only the key improvements over the RNN model are highlighted in the following part.

1. Model Architecture

a. Pack the sequences
To enable the LSTM to process only the non-padded elements, we pack the padded sequences at the output of the embedding model before forwarding them to the LSTM, thus optimizing the memory and computation resources.

b. Bidirectional Multi-Layer LSTM

Layer	**Function**
Embedding layer	Turn sparse one-hot vectors into dense vectors for dimensionality reduction. Further, pack the padded sequences to ignore the paddings.
Bidirectional 2-layer LSTM	Take in packed embeddings and output the concatenation of the last forward and backward hidden states.
Fully-connect linear layer	Take in the concatenated hidden state and output the predicted probability of positive class.

Table 2. Three Layers and Corresponding Functions for LSTM

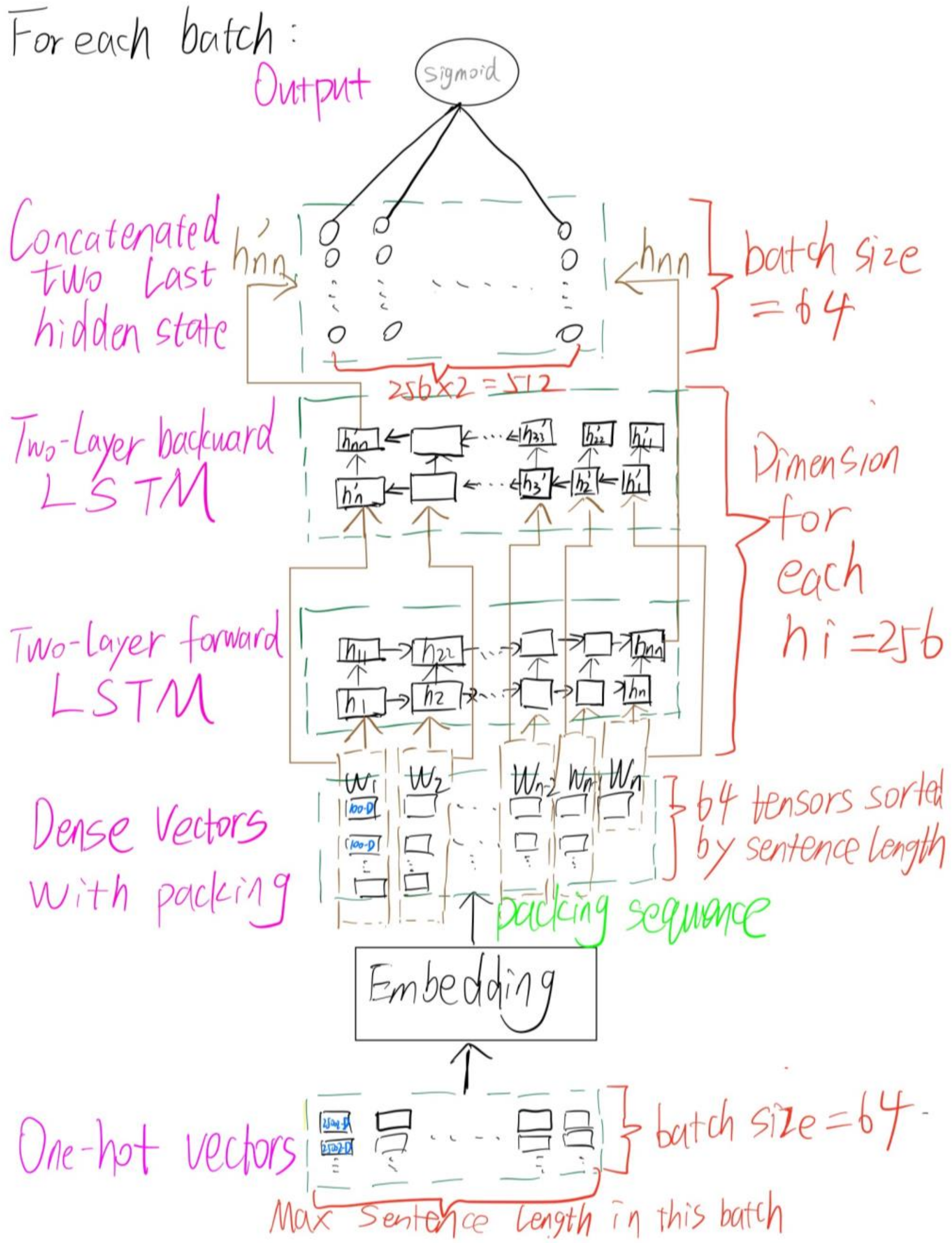

Figure 23. Bidirectional 2-Layer LSTM

c. Dropout

To prevent overfitting, we adopt a regularization technique called dropout. The main idea is to set the outputs of some units to zeros with a probability (0.5 in our experiment). By doing this, the network will not rely on a particular group of units too heavily and thus gain stronger generalization ability.

In our experiment, we use three dropouts, respectively, at the output of the embedding model, between two LSTM layers, and at the concatenated last hidden state of the bidirectional LSTM.

2. Model Training

a. Adam optimizer

We use an Adam optimizer to update the parameters rather than SGD. Unlike SGD, which updates all parameters with a fixed learning rate, Adam gives a lower learning rate for frequently-updated parameters and a higher for infrequently-updated parameters. By introducing this adaptive learning rate, Adam is more robust to the hyperparameter and more efficient than SGD.

3. Model Evaluation and Modification

a. Original architecture

Observation: From Figure 24 and 25, the bidirectional 2-layer LSTM significantly outperforms the previous RNN, with testing accuracy increasing from 49.68% (RNN) to 67.72% (LSTM).

```
Epoch: 01 | Epoch Time: 0m 4s
        Train Loss: 0.691 | Train Acc: 53.08%
         Val. Loss: 0.672 |  Val. Acc: 61.83%
Epoch: 02 | Epoch Time: 0m 3s
        Train Loss: 0.653 | Train Acc: 62.90%
         Val. Loss: 0.688 |  Val. Acc: 59.04%
Epoch: 03 | Epoch Time: 0m 3s
        Train Loss: 0.622 | Train Acc: 66.19%
         Val. Loss: 0.648 |  Val. Acc: 63.84%
Epoch: 04 | Epoch Time: 0m 3s
        Train Loss: 0.582 | Train Acc: 69.64%
         Val. Loss: 0.667 |  Val. Acc: 64.51%
Epoch: 05 | Epoch Time: 0m 4s
        Train Loss: 0.533 | Train Acc: 73.55%
         Val. Loss: 0.630 |  Val. Acc: 67.97%
Epoch: 06 | Epoch Time: 0m 3s
        Train Loss: 0.517 | Train Acc: 74.35%
         Val. Loss: 0.587 |  Val. Acc: 68.42%
Epoch: 07 | Epoch Time: 0m 3s
        Train Loss: 0.461 | Train Acc: 77.73%
         Val. Loss: 0.623 |  Val. Acc: 72.66%
Epoch: 08 | Epoch Time: 0m 4s
        Train Loss: 0.485 | Train Acc: 76.60%
         Val. Loss: 0.550 |  Val. Acc: 74.44%
Epoch: 09 | Epoch Time: 0m 4s
        Train Loss: 0.405 | Train Acc: 81.01%
         Val. Loss: 0.688 |  Val. Acc: 65.07%
Epoch: 10 | Epoch Time: 0m 4s
        Train Loss: 0.374 | Train Acc: 83.49%
         Val. Loss: 0.708 |  Val. Acc: 65.07%
```

Figure 24. Training Results for the Original LSTM

```
Test Loss: 0.682 | Test Acc: 67.72%
```

Figure 25. Testing Results for the Original LSTM

b. Increase the number of LSTM layers

As shown in Figure 26, we increase the number of LSTM layers to three.

```
INPUT_DIM = len(TEXT.vocab)
EMBEDDING_DIM = 100
HIDDEN_DIM = 256
OUTPUT_DIM = 1
N_LAYERS = 3
#N_LAYERS = 2
BIDIRECTIONAL = True
DROPOUT = 0.5
PAD_IDX = TEXT.vocab.stoi[TEXT.pad_token]

model = RNN(INPUT_DIM,
            EMBEDDING_DIM,
            HIDDEN_DIM,
            OUTPUT_DIM,
            N_LAYERS,
            BIDIRECTIONAL,
            DROPOUT,
            PAD_IDX)
```

Figure 26. Increase LSTM Layers to 3

Observation: From Figure 27 and 28, the performance for the 3-layer LSTM decreases compared to the 2-layer LSTM, with testing accuracy dropping from 67.72% to 64.03%. It may indicate that 2 LSTM layers are enough to capture the semantic meanings behind the sentences. 3 LSTM layers may give a representation so abstract that some relevant features are omitted.

```
Epoch: 01 | Epoch Time: 0m 6s
        Train Loss: 0.692 | Train Acc: 52.99%
         Val. Loss: 0.685 |  Val. Acc: 53.35%
Epoch: 02 | Epoch Time: 0m 6s
        Train Loss: 0.671 | Train Acc: 59.42%
         Val. Loss: 0.664 |  Val. Acc: 60.04%
Epoch: 03 | Epoch Time: 0m 6s
        Train Loss: 0.640 | Train Acc: 62.49%
         Val. Loss: 0.640 |  Val. Acc: 63.84%
Epoch: 04 | Epoch Time: 0m 6s
        Train Loss: 0.620 | Train Acc: 66.97%
         Val. Loss: 0.626 |  Val. Acc: 66.63%
Epoch: 05 | Epoch Time: 0m 6s
        Train Loss: 0.693 | Train Acc: 55.24%
         Val. Loss: 0.691 |  Val. Acc: 49.67%
Epoch: 06 | Epoch Time: 0m 6s
        Train Loss: 0.702 | Train Acc: 48.91%
         Val. Loss: 0.706 |  Val. Acc: 49.44%
Epoch: 07 | Epoch Time: 0m 6s
        Train Loss: 0.697 | Train Acc: 51.46%
         Val. Loss: 0.690 |  Val. Acc: 52.34%
Epoch: 08 | Epoch Time: 0m 6s
        Train Loss: 0.688 | Train Acc: 52.39%
         Val. Loss: 0.674 |  Val. Acc: 55.36%
Epoch: 09 | Epoch Time: 0m 6s
        Train Loss: 0.689 | Train Acc: 53.02%
         Val. Loss: 0.683 |  Val. Acc: 56.25%
Epoch: 10 | Epoch Time: 0m 6s
        Train Loss: 0.667 | Train Acc: 58.54%
         Val. Loss: 0.670 |  Val. Acc: 59.15%
```

Figure 27. Training Results for the 3-layer LSTM

```
Test Loss: 0.636 | Test Acc: 64.03%
```

Figure 28. Testing Results for the 3-layer LSTM

c. Use a uni-directional LSTM

As shown in Figure 29, we replace the bi-directional LSTM with uni-directional LSTM.

```
INPUT_DIM = len(TEXT.vocab)
EMBEDDING_DIM = 100
HIDDEN_DIM = 256
OUTPUT_DIM = 1
N_LAYERS = 2
# BIDIRECTIONAL = True
BIDIRECTIONAL = False
DROPOUT = 0.5
PAD_IDX = TEXT.vocab.stoi[TEXT.pad_token]

model = RNN(INPUT_DIM,
            EMBEDDING_DIM,
            HIDDEN_DIM,
            OUTPUT_DIM,
            N_LAYERS,
            BIDIRECTIONAL,
            DROPOUT,
            PAD_IDX)
```

Figure 29. Convert to a Uni-directional LSTM

Observation: From Figure 30 and 31, the performance for the uni-directional LSTM decreases compared to the bi-directional LSTM, with testing accuracy dropping from 67.72% to 65.92%. It verifies that bi-directional LSTM may be more powerful in capturing the semantic meanings of the sentence because it considers both past and future words when processing the current word.

```
Epoch: 01 | Epoch Time: 0m 2s
	Train Loss: 0.693 | Train Acc: 51.19%
	 Val. Loss: 0.688 |  Val. Acc: 53.24%
Epoch: 02 | Epoch Time: 0m 1s
	Train Loss: 0.685 | Train Acc: 55.17%
	 Val. Loss: 0.684 |  Val. Acc: 55.36%
Epoch: 03 | Epoch Time: 0m 1s
	Train Loss: 0.674 | Train Acc: 58.84%
	 Val. Loss: 0.681 |  Val. Acc: 57.48%
Epoch: 04 | Epoch Time: 0m 1s
	Train Loss: 0.650 | Train Acc: 62.72%
	 Val. Loss: 0.661 |  Val. Acc: 60.49%
Epoch: 05 | Epoch Time: 0m 2s
	Train Loss: 0.630 | Train Acc: 64.10%
	 Val. Loss: 0.635 |  Val. Acc: 66.85%
Epoch: 06 | Epoch Time: 0m 2s
	Train Loss: 0.592 | Train Acc: 67.73%
	 Val. Loss: 0.628 |  Val. Acc: 63.39%
Epoch: 07 | Epoch Time: 0m 2s
	Train Loss: 0.572 | Train Acc: 70.92%
	 Val. Loss: 0.624 |  Val. Acc: 65.40%
Epoch: 08 | Epoch Time: 0m 1s
	Train Loss: 0.555 | Train Acc: 72.49%
	 Val. Loss: 0.628 |  Val. Acc: 66.63%
Epoch: 09 | Epoch Time: 0m 1s
	Train Loss: 0.496 | Train Acc: 76.27%
	 Val. Loss: 0.659 |  Val. Acc: 68.97%
Epoch: 10 | Epoch Time: 0m 1s
	Train Loss: 0.460 | Train Acc: 79.66%
	 Val. Loss: 0.753 |  Val. Acc: 65.40%
```

Figure 30. Training Results for the Uni-directional LSTM

```
Test Loss: 0.640 | Test Acc: 65.92%
```

Figure 31. Testing Results for the Uni-directional LSTM

d. Change the embedding dimension

As shown in Figure 32 and 33, we try a decreased embedding dimension of 50 and an increased embedding dimension of 200.

```
INPUT_DIM = len(TEXT.vocab)
# EMBEDDING_DIM = 100
EMBEDDING_DIM = 50
HIDDEN_DIM = 256
OUTPUT_DIM = 1
N_LAYERS = 2
BIDIRECTIONAL = True
DROPOUT = 0.5
PAD_IDX = TEXT.vocab.stoi[TEXT.pad_token]

model = RNN(INPUT_DIM,
            EMBEDDING_DIM,
            HIDDEN_DIM,
            OUTPUT_DIM,
            N_LAYERS,
            BIDIRECTIONAL,
            DROPOUT,
            PAD_IDX)
```

Figure 32. Decrease Embedding Dimension to 50

```
INPUT_DIM = len(TEXT.vocab)
# EMBEDDING_DIM = 100
EMBEDDING_DIM = 200
HIDDEN_DIM = 256
OUTPUT_DIM = 1
N_LAYERS = 2
BIDIRECTIONAL = True
DROPOUT = 0.5
PAD_IDX = TEXT.vocab.stoi[TEXT.pad_token]

model = RNN(INPUT_DIM,
            EMBEDDING_DIM,
            HIDDEN_DIM,
            OUTPUT_DIM,
            N_LAYERS,
            BIDIRECTIONAL,
            DROPOUT,
            PAD_IDX)
```

Figure 33. Increase Embedding Dimension to 200

Observation: From Figure 34 and 35, decreasing the embedding dimension to 50 may be detrimental to the model performance, with testing accuracy dropping from 67.72% to 59.35%. It indicates that 50 embeddings may not be sufficient to represent a word's semantic meanings accurately.

From Figure 36 and 37, the training accuracies of the LSTM with an embedding dimension of 200 are much greater than the validation and testing accuracies. It may indicate that the model is suffering from overfitting.

```
Epoch: 01 | Epoch Time: 0m 3s
	Train Loss: 0.691 | Train Acc: 53.12%
	 Val. Loss: 0.686 |  Val. Acc: 57.25%
Epoch: 02 | Epoch Time: 0m 3s
	Train Loss: 0.677 | Train Acc: 58.15%
	 Val. Loss: 0.675 |  Val. Acc: 54.46%
Epoch: 03 | Epoch Time: 0m 3s
	Train Loss: 0.658 | Train Acc: 60.45%
	 Val. Loss: 0.657 |  Val. Acc: 61.94%
Epoch: 04 | Epoch Time: 0m 4s
	Train Loss: 0.659 | Train Acc: 61.00%
	 Val. Loss: 0.684 |  Val. Acc: 58.71%
Epoch: 05 | Epoch Time: 0m 3s
	Train Loss: 0.671 | Train Acc: 58.44%
	 Val. Loss: 0.676 |  Val. Acc: 56.36%
Epoch: 06 | Epoch Time: 0m 3s
	Train Loss: 0.652 | Train Acc: 62.53%
	 Val. Loss: 0.672 |  Val. Acc: 59.26%
Epoch: 07 | Epoch Time: 0m 3s
	Train Loss: 0.648 | Train Acc: 63.22%
	 Val. Loss: 0.696 |  Val. Acc: 53.24%
Epoch: 08 | Epoch Time: 0m 4s
	Train Loss: 0.624 | Train Acc: 64.75%
	 Val. Loss: 0.672 |  Val. Acc: 61.50%
Epoch: 09 | Epoch Time: 0m 3s
	Train Loss: 0.603 | Train Acc: 68.71%
	 Val. Loss: 0.734 |  Val. Acc: 54.58%
Epoch: 10 | Epoch Time: 0m 3s
	Train Loss: 0.582 | Train Acc: 69.01%
	 Val. Loss: 0.677 |  Val. Acc: 59.60%
```

Figure 34. Training Results for the LSTM with Embedding Dimension = 50

```
Test Loss: 0.660 | Test Acc: 59.35%
```

Figure 35. Testing Results for the LSTM with Embedding Dimension = 50

```
Epoch: 01 | Epoch Time: 0m 4s
	Train Loss: 0.523 | Train Acc: 73.90%
	 Val. Loss: 0.625 |  Val. Acc: 70.42%
Epoch: 02 | Epoch Time: 0m 4s
	Train Loss: 0.475 | Train Acc: 77.22%
	 Val. Loss: 0.643 |  Val. Acc: 71.32%
Epoch: 03 | Epoch Time: 0m 4s
	Train Loss: 0.415 | Train Acc: 81.13%
	 Val. Loss: 0.670 |  Val. Acc: 71.88%
Epoch: 04 | Epoch Time: 0m 4s
	Train Loss: 0.378 | Train Acc: 83.14%
	 Val. Loss: 0.773 |  Val. Acc: 70.20%
Epoch: 05 | Epoch Time: 0m 4s
	Train Loss: 0.339 | Train Acc: 85.09%
	 Val. Loss: 0.746 |  Val. Acc: 66.63%
Epoch: 06 | Epoch Time: 0m 4s
	Train Loss: 0.313 | Train Acc: 87.15%
	 Val. Loss: 0.877 |  Val. Acc: 73.21%
Epoch: 07 | Epoch Time: 0m 4s
	Train Loss: 0.277 | Train Acc: 88.00%
	 Val. Loss: 0.753 |  Val. Acc: 71.21%
Epoch: 08 | Epoch Time: 0m 4s
	Train Loss: 0.222 | Train Acc: 90.73%
	 Val. Loss: 1.053 |  Val. Acc: 73.44%
Epoch: 09 | Epoch Time: 0m 4s
	Train Loss: 0.176 | Train Acc: 93.40%
	 Val. Loss: 0.853 |  Val. Acc: 76.45%
Epoch: 10 | Epoch Time: 0m 4s
	Train Loss: 0.188 | Train Acc: 93.15%
	 Val. Loss: 0.837 |  Val. Acc: 73.77%
```

Figure 36. Training Results for the LSTM with Embedding Dimension = 200

```
Test Loss: 0.711 | Test Acc: 65.08%
```

Figure 37. Testing Results for the LSTM with Embedding Dimension = 200

e. Disable the dropout during training

As shown in Figure 38, we disable the three dropouts in the original model.

```python
class RNN(nn.Module):
    def __init__(self, vocab_size, embedding_dim, hidden_dim, output_dim, n_layers,
                 bidirectional, dropout, pad_idx):
        super().__init__()
        self.embedding = nn.Embedding(vocab_size, embedding_dim, padding_idx = pad_idx)
        self.rnn = nn.LSTM(embedding_dim,
                           hidden_dim,
                           num_layers=n_layers,
                           bidirectional=bidirectional,
                           dropout=0)
        self.fc = nn.Linear(hidden_dim * 2, output_dim)
        self.dropout = nn.Dropout(dropout)

    def forward(self, text, text_lengths):

        # embedded = self.dropout(self.embedding(text))
        embedded = self.embedding(text)

        packed_embedded = nn.utils.rnn.pack_padded_sequence(embedded, text_lengths.to('cpu'))

        packed_output, (hidden, cell) = self.rnn(packed_embedded)

        # hidden = self.dropout(torch.cat((hidden[-2,:,:], hidden[-1,:,:]), dim = 1))

        hidden = torch.cat((hidden[-2,:,:], hidden[-1,:,:]), dim = 1)

        return self.fc(hidden)
```

Figure 38. Disable All Dropouts

From Figure 39 and 40, there is a huge gap between the training accuracies and validation/testing accuracies when we disable the dropouts, suggesting our model is overfitting. It verifies that dropouts can efficiently alleviate the overfitting problems for complex models.

```
Epoch: 01 | Epoch Time: 0m 4s
	Train Loss: 0.691 | Train Acc: 52.57%
	 Val. Loss: 0.682 |  Val. Acc: 54.02%
Epoch: 02 | Epoch Time: 0m 4s
	Train Loss: 0.620 | Train Acc: 66.77%
	 Val. Loss: 0.696 |  Val. Acc: 62.95%
Epoch: 03 | Epoch Time: 0m 4s
	Train Loss: 0.507 | Train Acc: 76.24%
	 Val. Loss: 0.746 |  Val. Acc: 62.50%
Epoch: 04 | Epoch Time: 0m 3s
	Train Loss: 0.336 | Train Acc: 85.86%
	 Val. Loss: 0.770 |  Val. Acc: 66.63%
Epoch: 05 | Epoch Time: 0m 3s
	Train Loss: 0.177 | Train Acc: 93.53%
	 Val. Loss: 1.041 |  Val. Acc: 67.52%
Epoch: 06 | Epoch Time: 0m 4s
	Train Loss: 0.088 | Train Acc: 97.08%
	 Val. Loss: 1.304 |  Val. Acc: 63.62%
Epoch: 07 | Epoch Time: 0m 3s
	Train Loss: 0.037 | Train Acc: 98.89%
	 Val. Loss: 1.488 |  Val. Acc: 61.16%
Epoch: 08 | Epoch Time: 0m 3s
	Train Loss: 0.016 | Train Acc: 99.65%
	 Val. Loss: 1.878 |  Val. Acc: 60.71%
Epoch: 09 | Epoch Time: 0m 4s
	Train Loss: 0.016 | Train Acc: 99.50%
	 Val. Loss: 1.921 |  Val. Acc: 65.18%
Epoch: 10 | Epoch Time: 0m 3s
	Train Loss: 0.022 | Train Acc: 99.29%
	 Val. Loss: 2.071 |  Val. Acc: 60.83%
```

Figure 39. Training Results for the LSTM with Dropout Disabled

```
Test Loss: 0.684 | Test Acc: 54.68%
```

Figure 40. Testing Results for the LSTM with Dropout Disabled

f. Conclusion

By comparison, LSTM outperforms RNN greatly in sentiment analysis tasks. The best model in our experiment is a bi-directional 2-layer LSTM with embedding dimension = 100 and dropout enabled, which achieved an accuracy of 67.72% in the testing set.