# RNN and LSTM for Text Analysis

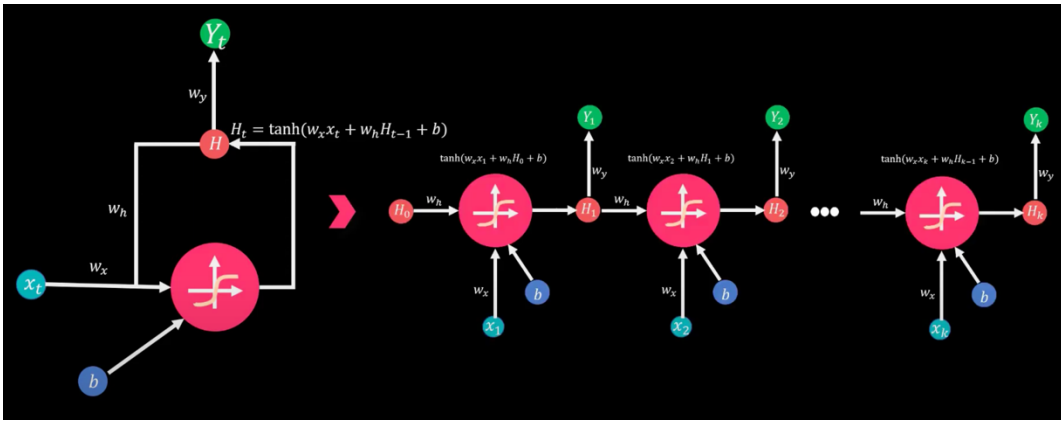
## Objectives and Outcomes

Use the reduced-size IMDb dataset, PyTorch, and TorchText to build a machine-learning model to detect sentiment (Detecting if a sentence is positive or negative). We can increase the number of RNN layers, change the embedding dimension of RNN, and reduce the dimension of the word embeddings to report the type of loss function used by the RNN. We can also increase the number of LSTM layers, use a uni-directional LSTM, and change the embedding dimension in the LSTM model to gain the change results. This means the training is based on the RNN and LSTM models to modify the code and obtain meaningful results. As a result, we can understand the architecture of RNNs, LSTMs, bidirectional-LSTMs, and multi-layer LSTMs.

## RNN Introduction

The Recurrent Neural Networks (RNNs) are neural networks designed specifically for sequence data, which can retain information about previous inputs through internal memory. By doing this, RNNs can capture the complex semantics and syntax rules for prediction, such as language modeling and sentiment analysis.

## Structure of RNN




The RNN architecture structure consists of input layer $(x_1, x_2, \ldots, x_t)$, hidden

layer $(H_0, H_1, H_2, \ldots, H_k)$, and output layer $(Y_1, Y_2, \ldots, Y_t)$. The hidden layer contains the activation function to obtain the $Y_t$. There are examples to show the calculation structure of output $Y_t$.

Using $w_y$ to convert $H_1$ into the output $Y_1$:

$$Y_1 = w_y \cdot H_1$$

The $H_1$ calculation (Including the activation function of tanh):

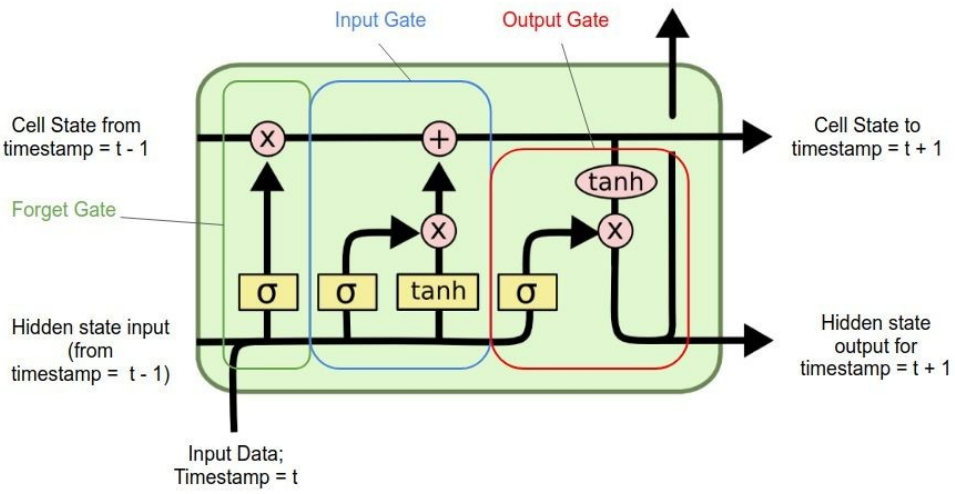
$$H_1 = \tanh\left(w_x \cdot x_1 + w_h \cdot H_0 + b\right)$$

The $w_x$ and $w_h$ are input $x_1$ weight and previous hidden state $H_0$ weight respectively while b represents the bias term.

## LSTM Introduction

The Long Short-Term Memory (LSTM) LSTM builds based on the RNN structure, which is designed to solve the RNN vanishing and exploding gradients problems. Moreover, LSTM contains feedback connections and three gates (Input, output, and forget) to retain and manage information over long sequences for handling long-term dependencies.

## Structure of LSTM




The LSTM architecture structure consists of a forget gate, input gate, output gate,

and cell state. There are examples of different gate calculation structures.

Forget Gate:

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)$$

$f_t$ is forget gate's output (Value between 0 and 1)

$\sigma$ is sigmoid function (Activation function), it constrains output between 0 and 1

$W_f$ is weight matrix for forget gate

$h_{t-1}$ is previous hidden state

$x_t$ is current input

$b_f$ is bias term

Input Gate:

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$$

$$\widetilde{C_t} = \tanh(W_c \cdot [ht - 1, x_t] + b_c)$$

$i_t$ is output of input gate, it determines new function impact

$W_i$ and $W_c$ are weight matrices

$b_i$ and $b_c$ are bias

$\widetilde{C_t}$ represents candidate cell state values

Update the old cell state $C_{t-1}$ to new cell state $C_t$:

$$C_t = f_t \odot C_{t-1} + i_t \odot \widetilde{C_t}$$

$C_t$ is updated cell state

$C_{t-1}$ is previous cell state

Output Gate:

$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o)$$

$$h_t = o_t \odot \tanh(C_t)$$

$o_t$ is output of output gate, it regulates final output

$W_o$ is weight matrix for output gate

$b_o$ is bias term

$h_t$ is current hidden state (Final output)

## Methodology

### (a) RNN Part

**RNN_SentimentAnalysis initial code (Unchanged):**

**Model Definition (RNN Class)**

```python
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.rnn = nn.RNN(embedding_dim, hidden_dim)
        self.fc = nn.Linear(hidden_dim, output_dim)

    def forward(self, text):

        embedded = self.embedding(text)
        output, hidden = self.rnn(embedded)
        assert torch.equal(output[-1,:,:], hidden.squeeze(0))
        return self.fc(hidden.squeeze(0))  # Remove the first dim in hidden to return [batch_size, hid_dim]
```

1. Class Definition

```python
class RNN(nn.Module):
```

Defines a class named RNN that inherits from nn.Module.

**Model Structure Definition**

```
def __init__(self, input_dim, embedding_dim, hidden_dim, output_dim):

    super().__init__()
    self.embedding = nn.Embedding(input_dim, embedding_dim)
    self.rnn = nn.RNN(embedding_dim, hidden_dim)
    self.fc = nn.Linear(hidden_dim, output_dim)
```

2. Initialization Definition

```
def __init__(self, input_dim, embedding_dim, hidden_dim, output_dim):
```

Defines the model's initialization. It contains input dimension, embedding dimension, hidden dimension, and output dimension as parameters.

3. Cells the parent class

```
super().__init__()
```

Calls the parent class nn.Module's initialization method to ensure the model is properly initialized.

4. Embedding Layer

```
self.embedding = nn.Embedding(input_dim, embedding_dim)
```

Defines an embedding layer that maps each word index in the vocabulary to an embedding vector.

input_dim: Vocabulary size (Number of unique words)

embedding_dim: Dimension of the embedding vectors (Length of the vector each word is converted into)

5. Recurrent Neural Network Layer

```
self.rnn = nn.RNN(embedding_dim, hidden_dim)
```

Defines a recurrent neural network layer.

embedding_dim: Input size for the RNN layer

hidden_dim: Hidden layer size (The dimension of the hidden state used in the RNN calculations)

6. Fully Connected Layer

```
self.fc = nn.Linear(hidden_dim, output_dim)
```

Defines a fully connected layer that maps the RNN's hidden state to the output dimension.

hidden_dim: Input dimension (Dimension of the output from the RNN layer.)

output_dim: Output dimension

**Model Calculation Definition**

```
def forward(self, text):

    embedded = self.embedding(text)
    output, hidden = self.rnn(embedded)
    assert torch.equal(output[-1,:,:], hidden.squeeze(0))
    return self.fc(hidden.squeeze(0))
```

7. Method Definition

```
def forward(self, text):
```

Defines the forward propagation method.

text: Input data

8. Embedding Layer Processing

```
embedded = self.embedding(text)
```

Passes the input text data by the embedding layer to obtain embedding vectors.

9. RNN Layer Processing

```
output, hidden = self.rnn(embedded)
```

Feeds the embeddings into the RNN layer, obtaining outputs and hidden states.

10. Verify Output and Hidden State

```
assert torch.equal(output[-1,:,:], hidden.squeeze(0))
```

Asserts that the output from the last time step equals the hidden state.

output[-1,:,:]: This selects the output from the last time step of the RNN. The output shape is [seq_len, batch_size, hidden_dim], but output[-1,:,:] means extract the last sequence element, resulting in a tensor of shape [batch_size, hidden_dim].

hidden.squeeze(0): Applying .squeeze(0) removes the first dimension, resulting in a tensor of shape [batch_size, hidden_dim]. The original shape is [1 (number of layers), batch_size, hidden_dim].

11. Output Layer Computation

```
return self.fc(hidden.squeeze(0))
```

Passes the hidden state through the fully connected layer to get the final output.

**Hyperparameter Settings**

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

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

INPUT_DIM: The size of the vocabulary, for the embedding layer.

len(TEXT.vocab): Calculates number of unique tokens (words) in vocabulary.

EMBEDDING_DIM: The size of the word embeddings.

HIDDEN_DIM: The size of the hidden state in the RNN.

OUTPUT_DIM: The size of the output layer (1 for binary classification).

model = RNN(INPUT_DIM, EMBEDDING_DIM, HIDDEN_DIM, OUTPUT_DIM): Builds an instance of an RNN model using the specified hyperparameters.

**Initial Code Result (Unchanged)**

```
Epoch: 01 | Epoch Time: 0m 2s
        Train Loss: 0.697 | Train Acc: 49.82%
         Val. Loss: 0.694 |  Val. Acc: 50.45%
Epoch: 02 | Epoch Time: 0m 1s
        Train Loss: 0.695 | Train Acc: 49.70%
         Val. Loss: 0.695 |  Val. Acc: 50.45%
Epoch: 03 | Epoch Time: 0m 1s
        Train Loss: 0.694 | Train Acc: 49.88%
         Val. Loss: 0.695 |  Val. Acc: 50.45%
Epoch: 04 | Epoch Time: 0m 1s
        Train Loss: 0.694 | Train Acc: 49.02%
         Val. Loss: 0.695 |  Val. Acc: 50.45%
Epoch: 05 | Epoch Time: 0m 1s
        Train Loss: 0.694 | Train Acc: 49.62%
         Val. Loss: 0.695 |  Val. Acc: 50.67%
Epoch: 06 | Epoch Time: 0m 1s
        Train Loss: 0.694 | Train Acc: 49.98%
         Val. Loss: 0.696 |  Val. Acc: 44.20%
Epoch: 07 | Epoch Time: 0m 1s
        Train Loss: 0.693 | Train Acc: 50.05%
         Val. Loss: 0.696 |  Val. Acc: 44.31%
Epoch: 08 | Epoch Time: 0m 1s
        Train Loss: 0.693 | Train Acc: 47.59%
         Val. Loss: 0.696 |  Val. Acc: 44.08%
Epoch: 09 | Epoch Time: 0m 1s
        Train Loss: 0.693 | Train Acc: 50.50%
         Val. Loss: 0.696 |  Val. Acc: 44.20%
Epoch: 10 | Epoch Time: 0m 1s
        Train Loss: 0.693 | Train Acc: 48.71%
         Val. Loss: 0.696 |  Val. Acc: 44.53%
```

| Test Loss | Test Acc |
|---|---|
| 0.697 | 49.89% |

**Further Investigations**

## 1. Increasing the Number of RNN Layers

### Increasing 2 Layers

```python
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.rnn = nn.RNN(embedding_dim, hidden_dim, num_layers = 2)
        self.fc = nn.Linear(hidden_dim, output_dim)

    def forward(self, text):
        embedded = self.embedding(text)
        output, hidden = self.rnn(embedded)
        hidden = hidden[-1,:,:]
        return self.fc(hidden)
```

### Result (2 Layers)

```
Epoch: 01 | Epoch Time: 0m 2s
        Train Loss: 0.693 | Train Acc: 50.58%
         Val. Loss: 0.694 |  Val. Acc: 49.33%
Epoch: 02 | Epoch Time: 0m 1s
        Train Loss: 0.693 | Train Acc: 50.15%
         Val. Loss: 0.694 |  Val. Acc: 49.44%
Epoch: 03 | Epoch Time: 0m 1s
        Train Loss: 0.693 | Train Acc: 49.50%
         Val. Loss: 0.694 |  Val. Acc: 49.44%
Epoch: 04 | Epoch Time: 0m 1s
        Train Loss: 0.693 | Train Acc: 50.18%
         Val. Loss: 0.694 |  Val. Acc: 50.00%
Epoch: 05 | Epoch Time: 0m 1s
        Train Loss: 0.693 | Train Acc: 49.45%
         Val. Loss: 0.694 |  Val. Acc: 49.89%
Epoch: 06 | Epoch Time: 0m 1s
        Train Loss: 0.693 | Train Acc: 49.63%
         Val. Loss: 0.694 |  Val. Acc: 49.44%
Epoch: 07 | Epoch Time: 0m 1s
        Train Loss: 0.693 | Train Acc: 50.43%
         Val. Loss: 0.694 |  Val. Acc: 49.33%
Epoch: 08 | Epoch Time: 0m 2s
        Train Loss: 0.693 | Train Acc: 49.28%
         Val. Loss: 0.694 |  Val. Acc: 50.00%
Epoch: 09 | Epoch Time: 0m 1s
        Train Loss: 0.693 | Train Acc: 50.28%
         Val. Loss: 0.694 |  Val. Acc: 49.67%
Epoch: 10 | Epoch Time: 0m 1s
        Train Loss: 0.693 | Train Acc: 49.67%
         Val. Loss: 0.694 |  Val. Acc: 49.33%
```

| Test Loss | Test Acc |
|---|---|
| 0.693 | 50.89% |

### Increasing 4 Layers

```python
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.rnn = nn.RNN(embedding_dim, hidden_dim, num_layers = 4)
        self.fc = nn.Linear(hidden_dim, output_dim)

    def forward(self, text):
        embedded = self.embedding(text)
        output, hidden = self.rnn(embedded)
        hidden = hidden[-1,:,:]
        return self.fc(hidden)
```

**Result (4 Layers)**

```
Epoch: 01 | Epoch Time: 0m 3s
        Train Loss: 0.698 | Train Acc: 49.55%
         Val. Loss: 0.696 |  Val. Acc: 51.00%
Epoch: 02 | Epoch Time: 0m 3s
        Train Loss: 0.696 | Train Acc: 49.87%
         Val. Loss: 0.694 |  Val. Acc: 51.23%
Epoch: 03 | Epoch Time: 0m 3s
        Train Loss: 0.695 | Train Acc: 49.85%
         Val. Loss: 0.694 |  Val. Acc: 51.12%
Epoch: 04 | Epoch Time: 0m 3s
        Train Loss: 0.694 | Train Acc: 49.82%
         Val. Loss: 0.693 |  Val. Acc: 51.12%
Epoch: 05 | Epoch Time: 0m 3s
        Train Loss: 0.694 | Train Acc: 49.50%
         Val. Loss: 0.693 |  Val. Acc: 50.78%
Epoch: 06 | Epoch Time: 0m 3s
        Train Loss: 0.694 | Train Acc: 49.75%
         Val. Loss: 0.692 |  Val. Acc: 50.67%
Epoch: 07 | Epoch Time: 0m 3s
        Train Loss: 0.693 | Train Acc: 50.12%
         Val. Loss: 0.692 |  Val. Acc: 50.67%
Epoch: 08 | Epoch Time: 0m 3s
        Train Loss: 0.694 | Train Acc: 49.55%
         Val. Loss: 0.692 |  Val. Acc: 50.78%
Epoch: 09 | Epoch Time: 0m 3s
        Train Loss: 0.693 | Train Acc: 49.85%
         Val. Loss: 0.692 |  Val. Acc: 50.67%
Epoch: 10 | Epoch Time: 0m 3s
        Train Loss: 0.693 | Train Acc: 49.72%
         Val. Loss: 0.692 |  Val. Acc: 51.12%
```

| Test Loss | Test Acc |
|---|---|
| 0.693 | 50.61% |

**Increasing 8 Layers**

```python
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.rnn = nn.RNN(embedding_dim, hidden_dim, num_layers = 8)
        self.fc = nn.Linear(hidden_dim, output_dim)

    def forward(self, text):
        embedded = self.embedding(text)
        output, hidden = self.rnn(embedded)
        hidden = hidden[-1,:,:]
        return self.fc(hidden)
```

**Result (8 Layers)**

```
Epoch: 01 | Epoch Time: 0m 6s
        Train Loss: 0.694 | Train Acc: 49.93%
         Val. Loss: 0.694 |  Val. Acc: 49.67%
Epoch: 02 | Epoch Time: 0m 6s
        Train Loss: 0.694 | Train Acc: 50.08%
         Val. Loss: 0.694 |  Val. Acc: 49.67%
Epoch: 03 | Epoch Time: 0m 6s
        Train Loss: 0.694 | Train Acc: 50.00%
         Val. Loss: 0.694 |  Val. Acc: 49.67%
Epoch: 04 | Epoch Time: 0m 6s
        Train Loss: 0.693 | Train Acc: 50.00%
         Val. Loss: 0.694 |  Val. Acc: 49.67%
Epoch: 05 | Epoch Time: 0m 6s
        Train Loss: 0.693 | Train Acc: 50.00%
         Val. Loss: 0.694 |  Val. Acc: 49.67%
Epoch: 06 | Epoch Time: 0m 6s
        Train Loss: 0.693 | Train Acc: 50.30%
         Val. Loss: 0.694 |  Val. Acc: 49.67%
Epoch: 07 | Epoch Time: 0m 6s
        Train Loss: 0.693 | Train Acc: 50.15%
         Val. Loss: 0.694 |  Val. Acc: 49.67%
Epoch: 08 | Epoch Time: 0m 6s
        Train Loss: 0.693 | Train Acc: 50.22%
         Val. Loss: 0.694 |  Val. Acc: 49.67%
Epoch: 09 | Epoch Time: 0m 6s
        Train Loss: 0.693 | Train Acc: 50.08%
         Val. Loss: 0.694 |  Val. Acc: 49.67%
Epoch: 10 | Epoch Time: 0m 6s
        Train Loss: 0.693 | Train Acc: 49.93%
         Val. Loss: 0.694 |  Val. Acc: 49.67%
```

| Test Loss | Test Acc |
|---|---|
| 0.694 | 49.86% |

**Summary Test Result (Increasing 2, 4, and 8 Layers)**

| | Initial | 2 Layers | 4 Layers | 8 Layers |
|---|---|---|---|---|
| Test Loss | 0.697 | 0.693 | 0.693 | 0.694 |
| Test Acc | 49.89% | 50.89% | 50.61% | 49.86% |

## 2. Changing the Embedding Dimension of RNN

### Changing the Embedding Dimension of RNN to 200

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

### Result (Embedding Dimension of RNN to 200)

```
Epoch: 01 | Epoch Time: 0m 1s
        Train Loss: 0.694 | Train Acc: 49.84%
         Val. Loss: 0.705 |  Val. Acc: 50.45%
Epoch: 02 | Epoch Time: 0m 1s
        Train Loss: 0.693 | Train Acc: 48.66%
         Val. Loss: 0.706 |  Val. Acc: 44.64%
Epoch: 03 | Epoch Time: 0m 1s
        Train Loss: 0.693 | Train Acc: 49.42%
         Val. Loss: 0.705 |  Val. Acc: 50.56%
Epoch: 04 | Epoch Time: 0m 1s
        Train Loss: 0.694 | Train Acc: 49.49%
         Val. Loss: 0.705 |  Val. Acc: 50.67%
Epoch: 05 | Epoch Time: 0m 1s
        Train Loss: 0.694 | Train Acc: 48.82%
         Val. Loss: 0.706 |  Val. Acc: 44.31%
Epoch: 06 | Epoch Time: 0m 1s
        Train Loss: 0.693 | Train Acc: 48.49%
         Val. Loss: 0.706 |  Val. Acc: 45.09%
Epoch: 07 | Epoch Time: 0m 1s
        Train Loss: 0.693 | Train Acc: 50.55%
         Val. Loss: 0.706 |  Val. Acc: 44.87%
Epoch: 08 | Epoch Time: 0m 1s
        Train Loss: 0.694 | Train Acc: 50.30%
         Val. Loss: 0.706 |  Val. Acc: 44.53%
Epoch: 09 | Epoch Time: 0m 1s
        Train Loss: 0.693 | Train Acc: 48.74%
         Val. Loss: 0.706 |  Val. Acc: 44.31%
Epoch: 10 | Epoch Time: 0m 1s
        Train Loss: 0.693 | Train Acc: 48.91%
         Val. Loss: 0.706 |  Val. Acc: 45.31%
```

| Test Loss | Test Acc |
|---|---|
| 0.692 | 50.31% |

### Changing the Embedding Dimension of RNN to 300

```
INPUT_DIM = len(TEXT.vocab)
EMBEDDING_DIM = 300
HIDDEN_DIM = 256
OUTPUT_DIM = 1
```

**Result (Embedding Dimension of RNN to 300)**

```
Epoch: 01 | Epoch Time: 0m 1s
        Train Loss: 0.696 | Train Acc: 50.21%
         Val. Loss: 0.697 |  Val. Acc: 45.76%
Epoch: 02 | Epoch Time: 0m 1s
        Train Loss: 0.694 | Train Acc: 50.05%
         Val. Loss: 0.695 |  Val. Acc: 45.76%
Epoch: 03 | Epoch Time: 0m 1s
        Train Loss: 0.694 | Train Acc: 48.52%
         Val. Loss: 0.696 |  Val. Acc: 45.87%
Epoch: 04 | Epoch Time: 0m 1s
        Train Loss: 0.694 | Train Acc: 49.62%
         Val. Loss: 0.695 |  Val. Acc: 45.65%
Epoch: 05 | Epoch Time: 0m 1s
        Train Loss: 0.694 | Train Acc: 48.37%
         Val. Loss: 0.697 |  Val. Acc: 45.54%
Epoch: 06 | Epoch Time: 0m 1s
        Train Loss: 0.694 | Train Acc: 49.85%
         Val. Loss: 0.696 |  Val. Acc: 45.54%
Epoch: 07 | Epoch Time: 0m 1s
        Train Loss: 0.694 | Train Acc: 48.32%
         Val. Loss: 0.696 |  Val. Acc: 45.54%
Epoch: 08 | Epoch Time: 0m 1s
        Train Loss: 0.694 | Train Acc: 49.34%
         Val. Loss: 0.697 |  Val. Acc: 45.54%
Epoch: 09 | Epoch Time: 0m 1s
        Train Loss: 0.693 | Train Acc: 49.92%
         Val. Loss: 0.697 |  Val. Acc: 45.65%
Epoch: 10 | Epoch Time: 0m 1s
        Train Loss: 0.694 | Train Acc: 50.12%
         Val. Loss: 0.697 |  Val. Acc: 45.65%
```

| Test Loss | Test Acc |
|---|---|
| 0.696 | 50.50% |

**Summary Test Result (200 and 300)**

| | Initial (100) | 200 dimensions | 300 dimensions |
|---|---|---|---|
| Test Loss | 0.697 | 0.692 | 0.696 |
| Test Acc | 49.89% | 50.31% | 50.50% |

### 3. Reducing the Dimension of the Word Embeddings

**Reducing the dimension of the word embeddings to 50**

```
INPUT_DIM = len(TEXT.vocab)
EMBEDDING_DIM = 50
HIDDEN_DIM = 256
OUTPUT_DIM = 1
```

**Result (Dimension of the word embeddings to 50)**

```
Epoch: 01 | Epoch Time: 0m 1s
        Train Loss: 0.693 | Train Acc: 50.19%
         Val. Loss: 0.692 |  Val. Acc: 53.57%
Epoch: 02 | Epoch Time: 0m 1s
        Train Loss: 0.693 | Train Acc: 49.95%
         Val. Loss: 0.693 |  Val. Acc: 54.13%
Epoch: 03 | Epoch Time: 0m 1s
        Train Loss: 0.693 | Train Acc: 49.37%
         Val. Loss: 0.693 |  Val. Acc: 54.58%
Epoch: 04 | Epoch Time: 0m 1s
        Train Loss: 0.694 | Train Acc: 48.66%
         Val. Loss: 0.693 |  Val. Acc: 54.69%
Epoch: 05 | Epoch Time: 0m 1s
        Train Loss: 0.694 | Train Acc: 49.42%
         Val. Loss: 0.693 |  Val. Acc: 55.13%
Epoch: 06 | Epoch Time: 0m 1s
        Train Loss: 0.693 | Train Acc: 50.46%
         Val. Loss: 0.693 |  Val. Acc: 48.55%
Epoch: 07 | Epoch Time: 0m 1s
        Train Loss: 0.693 | Train Acc: 50.30%
         Val. Loss: 0.693 |  Val. Acc: 48.44%
Epoch: 08 | Epoch Time: 0m 1s
        Train Loss: 0.693 | Train Acc: 50.40%
         Val. Loss: 0.693 |  Val. Acc: 48.55%
Epoch: 09 | Epoch Time: 0m 1s
        Train Loss: 0.693 | Train Acc: 49.67%
         Val. Loss: 0.693 |  Val. Acc: 48.55%
Epoch: 10 | Epoch Time: 0m 1s
        Train Loss: 0.694 | Train Acc: 48.48%
         Val. Loss: 0.693 |  Val. Acc: 48.55%
```

| Test Loss | Test Acc |
|---|---|
| 0.694 | 48.12% |

**Reducing the dimension of the word embeddings to 25**

```
INPUT_DIM = len(TEXT.vocab)
EMBEDDING_DIM = 25
HIDDEN_DIM = 256
OUTPUT_DIM = 1
```

**Result (Dimension of the word embeddings to 25)**

```
Epoch: 01 | Epoch Time: 0m 1s
        Train Loss: 0.694 | Train Acc: 50.05%
         Val. Loss: 0.692 |  Val. Acc: 54.35%
Epoch: 02 | Epoch Time: 0m 1s
        Train Loss: 0.694 | Train Acc: 49.82%
         Val. Loss: 0.692 |  Val. Acc: 54.46%
Epoch: 03 | Epoch Time: 0m 1s
        Train Loss: 0.694 | Train Acc: 49.85%
         Val. Loss: 0.692 |  Val. Acc: 54.58%
Epoch: 04 | Epoch Time: 0m 1s
        Train Loss: 0.693 | Train Acc: 50.10%
         Val. Loss: 0.692 |  Val. Acc: 54.46%
Epoch: 05 | Epoch Time: 0m 1s
        Train Loss: 0.693 | Train Acc: 49.92%
         Val. Loss: 0.692 |  Val. Acc: 54.69%
Epoch: 06 | Epoch Time: 0m 1s
        Train Loss: 0.693 | Train Acc: 49.50%
         Val. Loss: 0.693 |  Val. Acc: 54.69%
Epoch: 07 | Epoch Time: 0m 1s
        Train Loss: 0.693 | Train Acc: 48.99%
         Val. Loss: 0.693 |  Val. Acc: 54.13%
Epoch: 08 | Epoch Time: 0m 1s
        Train Loss: 0.693 | Train Acc: 48.72%
         Val. Loss: 0.693 |  Val. Acc: 54.24%
Epoch: 09 | Epoch Time: 0m 1s
        Train Loss: 0.693 | Train Acc: 49.72%
         Val. Loss: 0.693 |  Val. Acc: 46.88%
Epoch: 10 | Epoch Time: 0m 1s
        Train Loss: 0.693 | Train Acc: 49.39%
         Val. Loss: 0.693 |  Val. Acc: 48.44%
```

| Test Loss | Test Acc |
|---|---|
| 0.695 | 48.04% |

**Summary Test Result (200 and 300)**

| | Initial (100) | 50 dimensions | 25 dimensions |
|---|---|---|---|
| Test Loss | 0.697 | 0.694 | 0.695 |
| Test Acc | 49.89% | 48.12% | 48.04% |

**The Type of Loss Function Used by the RNN**

RNN model's output is an unbounded real number, and the BCEWithLogitsLoss function applies a sigmoid transformation to restrict the output between 0 and 1 before calculating the loss. Therefore, we need to use the sigmoid function to the output and compute the binary cross-entropy loss based on the transformed values.

Sigmoid Transformation:

The output $\hat{y}$ of the model (Which is any real number) is first passed through a sigmoid function to squash it into the range [0, 1], representing a probability.

$$\sigma(\hat{y}) = \frac{1}{1 + e^{-\hat{y}}}$$

Binary Cross-Entropy:

The binary cross-entropy loss calculates the difference between the predicted probability and the actual label.

$$BCE(y, p) = -(y \cdot \log(p) + (1 - y) \cdot \log(1 - p))$$

For example:

The model outputs a raw prediction score $\hat{y}$ of 0.4 (Before sigmoid).

$$P = \sigma(\hat{y}) = \frac{1}{1 + e^{-\hat{y}}}$$

$$P = \sigma(0.4) = \frac{1}{1 + e^{-0.4}}$$

$$P = 0.5987$$

Binary Cross-Entropy is:

$$BCE(y, p) = -(y \cdot \log(p) + (1 - y) \cdot \log(1 - p))$$

$y = 1$ (It means the actual label is positive), $P = 0.5987$

$$BCE(1,0.5987) = -(1 \cdot \log(0.5987) + (1-1) \cdot \log(1-0.5987))$$

$$= 0.5123$$

This loss value of 0.5123 represents the degree of error between the model's prediction and the true label. A smaller value means the model's prediction is closer to the actual label, indicating better performance. Conversely, a larger loss value indicates a greater difference between the prediction and the true label, showing worse performance.

### (b) LSTM Part

**LSTM_SentimentAnalysis initial code (Unchanged):**

```python
import torch.nn as nn

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=dropout)
        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))
        packed_embedded = nn.utils.rnn.pack_padded_sequence(embedded, text_lengths.to('cpu'))
        packed_output, (hidden, cell) = self.rnn(packed_embedded)
        output, output_lengths = nn.utils.rnn.pad_packed_sequence(packed_output)
        hidden = self.dropout(torch.cat((hidden[-2,:,:], hidden[-1,:,:]), dim = 1))
        return self.fc(hidden)
```

1. Class Definition

```python
class RNN(nn.Module):
```

Defines a class named RNN that inherits from nn.Module.

**Model Structure Definition**

```
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=dropout)
    self.fc = nn.Linear(hidden_dim * 2, output_dim)
    self.dropout = nn.Dropout(dropout)
```

2. Initialization Definition

```
def __init__(self, vocab_size, embedding_dim, hidden_dim, output_dim, n_layers,
             bidirectional, dropout, pad_idx):
```

The initializer for the RNN class. It sets up layers and configurations for RNN.

vocab_size: Vocabulary size, used to create the embedding layer.

embedding_dim: Dimension of the word embeddings.

hidden_dim: Number of features in the hidden state of the LSTM.

output_dim: Output layer size.

n_layers: Number of LSTM layers in the network.

bidirectional: Determines whether the LSTM should be bidirectional. The 'bidirectional = True' means the LSTM processes inputs in both forward and backward directions, allowing it to capture information from past and future steps in the sequence, While the 'bidirectional = False' means that it is uni-directional, the LSTM will only process the sequence in the forward direction, considering only past data at each step without looking ahead in the sequence.

dropout: The dropout probability, a value between 0 and 1, is used to prevent overfitting by randomly setting some of the layer outputs to zero during training.

pad_idx: Index for padding, which allows the embedding layer to ignore padding tokens.

3. Cells the parent class

```
super().__init__()
```

Calls the parent class nn.Module's initialization method to ensure the model is properly initialized.

4. Embedding Layer

```
self.embedding = nn.Embedding(vocab_size, embedding_dim, padding_idx = pad_idx)
```

The embedding layer converts each word in a sequence into a dense vector of embedding_dim size.

vocab_size: Vocabulary size.

embedding_dim: Dimension of the embedding vectors, defining the length of each word's vector representation.

padding_idx = pad_idx: Specifies the padding index to ignore any padding tokens when processing sequences of different lengths.

5. LSTM Layer

```
self.rnn = nn.LSTM(embedding_dim,
                   hidden_dim,
                   num_layers=n_layers,
                   bidirectional=bidirectional,
                   dropout=dropout)
```

The LSTM layer processes the embedded input and outputs the hidden states.

embedding_dim: Input dimension for the LSTM, equal to the embedding vector size.

hidden_dim: Hidden layer size, controlling how much context information the LSTM can store.

num_layers=n_layers: Number of LSTM layers.

bidirectional=bidirectional: Toggle between a unidirectional and bidirectional LSTM by changing the bidirectional parameter when creating an instance of the class.

dropout=dropout: Applies dropout between layers in multi-layer LSTMs to prevent overfitting.

6. Fully Connected Layer

```
self.fc = nn.Linear(hidden_dim * 2, output_dim)
```

Fully connected (linear) layer that converts the hidden state of the LSTM into the desired output size (output_dim).

hidden_dim: If the LSTM is bidirectional, the hidden layer output size will be doubled

output_dim: Final output dimension

7. Dropout Layer

```
self.dropout = nn.Dropout(dropout)
```

Dropout layer is used to randomly set some of the input units to zero with probability dropout, preventing overfitting during training.

**Model Calculation Definition**

```
def forward(self, text, text_lengths):
    embedded = self.dropout(self.embedding(text))
    packed_embedded = nn.utils.rnn.pack_padded_sequence(embedded, text_lengths.to('cpu'))
    packed_output, (hidden, cell) = self.rnn(packed_embedded)
    output, output_lengths = nn.utils.rnn.pad_packed_sequence(packed_output)
    hidden = self.dropout(torch.cat((hidden[-2,:,:], hidden[-1,:,:]), dim = 1))
    return self.fc(hidden)
```

8. Method Definition

```
def forward(self, text, text_lengths):
```

Defines the forward propagation method.

text: The input text sequences represented as token indices.

text_lengths: Actual lengths of each sequence in the batch.

9. Embedding the Input and Applying Dropout

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

self.embedding(text): The embedding layer takes the input text, which consists of token indices, and converts each token into a dense vector of fixed size.

self.droput(): Dropout randomly zeros out parts of the embedding vectors, which helps prevent overfitting.

10. Packing the Embedded Sequence

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

nn.utils.rnn.pack_padded_sequence(): Converts embedded into a packed sequence, which removes padding elements to ensure the LSTM only processes actual data.

text_lengths.to('cpu'): Moves text_lengths to the CPU, which is necessary if the model is on a GPU because pack_padded_sequence expects the lengths on the CPU.

11. Passing Through the LSTM Layer

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

This runs the packed sequence through the LSTM.

self.rnn(packed_embedded): The LSTM layer processes the packed sequence and returns two things.

packed_output: Contains the output features from each timestep of the LSTM, in a packed format.

(hidden, cell): The final hidden and cell states of the LSTM.

12. Unpacking the Output

```
output, output_lengths = nn.utils.rnn.pad_packed_sequence(packed_output)
```

nn.utils.rnn.pad_packed_sequence(packed_output): This restores packed_output to its original shape, adding padding back to any sequences that were compressed.

output: Contains the output for each timestep after padding.

output_lengths: The lengths of each sequence in the batch.

13. Concatenating the Final Hidden States

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

Since the LSTM is bidirectional, it has two hidden states for the final layer (One for each direction).

hidden[-2,:,:]: Final hidden state from the last layer in the forward direction.

hidden[-1,:,:]: Final hidden state from the last layer in the backward direction.

torch.cat( , dim = 1): Concatenates the forward and backward hidden states along the feature dimension (dim=1).

14. Passing through the Fully Connected Layer

```
return self.fc(hidden)
```

This passes the concatenated hidden states through the fully connected layer to produce the final output of the model.

self.fc(hidden): Fully connected layer transforms the concatenated hidden states into a tensor.

**Hyperparameter Settings**

```
INPUT_DIM = len(TEXT.vocab)
EMBEDDING_DIM = 100
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)
```

INPUT_DIM: Input vocabulary size.

EMBEDDING_DIM: Embedding layer size, which determines the size of the vector representing each token.

HIDDEN_DIM: Number of units in the hidden layer.

OUTPUT_DIM: Defines the size of the output layer, which is typically 1 for binary classification, such as positive or negative sentiment.

N_LAYERS: Number of layers in the RNN.

BIDIRECTIONAL: Indicates (Boolean) the RNN is bidirectional (True) or uni-directional (False).

DROPOUT: The dropout rate, which helps prevent overfitting by randomly dropping units during training.

PAD_IDX: The index of the padding token, which is used to ignore padding in calculations.

model = RNN(INPUT_DIM, EMBEDDING_DIM, HIDDEN_DIM, OUTPUT_DIM, N_LAYERS, BIDIRECTIONAL, DROPOUT, PAD_IDX): Initializes with those parameters, which configure various aspects of the model.

**Initial Code Result (Unchanged)**

```
Epoch: 01 | Epoch Time: 0m 4s
        Train Loss: 0.692 | Train Acc: 51.06%
         Val. Loss: 0.691 |  Val. Acc: 53.79%
Epoch: 02 | Epoch Time: 0m 4s
        Train Loss: 0.675 | Train Acc: 58.18%
         Val. Loss: 0.684 |  Val. Acc: 53.57%
Epoch: 03 | Epoch Time: 0m 4s
        Train Loss: 0.639 | Train Acc: 63.36%
         Val. Loss: 0.663 |  Val. Acc: 59.93%
Epoch: 04 | Epoch Time: 0m 4s
        Train Loss: 0.597 | Train Acc: 68.56%
         Val. Loss: 0.649 |  Val. Acc: 61.50%
Epoch: 05 | Epoch Time: 0m 4s
        Train Loss: 0.572 | Train Acc: 70.93%
         Val. Loss: 0.633 |  Val. Acc: 62.28%
Epoch: 06 | Epoch Time: 0m 4s
        Train Loss: 0.528 | Train Acc: 73.15%
         Val. Loss: 0.681 |  Val. Acc: 63.84%
Epoch: 07 | Epoch Time: 0m 4s
        Train Loss: 0.472 | Train Acc: 78.08%
         Val. Loss: 0.621 |  Val. Acc: 69.98%
Epoch: 08 | Epoch Time: 0m 4s
        Train Loss: 0.445 | Train Acc: 79.11%
         Val. Loss: 0.652 |  Val. Acc: 68.30%
Epoch: 09 | Epoch Time: 0m 4s
        Train Loss: 0.477 | Train Acc: 76.69%
         Val. Loss: 0.741 |  Val. Acc: 66.18%
Epoch: 10 | Epoch Time: 0m 4s
        Train Loss: 0.407 | Train Acc: 81.86%
         Val. Loss: 0.698 |  Val. Acc: 69.20%
```

| Test Loss | Test Acc |
|---|---|
| 0.732 | 64.52% |

| | |
|---|---|
| predict_sentiment(model, "This film is bad.") | 0.0672 |
| predict_sentiment(model, "This film is good.") | 0.3861 |

1. **Increasing the number of LSTM layers**

## 4 Layers

```
INPUT_DIM = len(TEXT.vocab)
EMBEDDING_DIM = 100
HIDDEN_DIM = 256
OUTPUT_DIM = 1
N_LAYERS = 4
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)
```

## Result (4 Layers)

```
Epoch: 01 | Epoch Time: 0m 8s
        Train Loss: 0.696 | Train Acc: 49.02%
         Val. Loss: 0.692 |  Val. Acc: 55.02%
Epoch: 02 | Epoch Time: 0m 7s
        Train Loss: 0.687 | Train Acc: 55.97%
         Val. Loss: 0.692 |  Val. Acc: 52.79%
Epoch: 03 | Epoch Time: 0m 8s
        Train Loss: 0.656 | Train Acc: 61.34%
         Val. Loss: 0.656 |  Val. Acc: 61.61%
Epoch: 04 | Epoch Time: 0m 8s
        Train Loss: 0.628 | Train Acc: 65.84%
         Val. Loss: 0.641 |  Val. Acc: 63.28%
Epoch: 05 | Epoch Time: 0m 8s
        Train Loss: 0.585 | Train Acc: 69.29%
         Val. Loss: 0.649 |  Val. Acc: 61.38%
Epoch: 06 | Epoch Time: 0m 8s
        Train Loss: 0.566 | Train Acc: 69.70%
         Val. Loss: 0.633 |  Val. Acc: 63.06%
Epoch: 07 | Epoch Time: 0m 8s
        Train Loss: 0.544 | Train Acc: 73.12%
         Val. Loss: 0.670 |  Val. Acc: 65.51%
Epoch: 08 | Epoch Time: 0m 8s
        Train Loss: 0.485 | Train Acc: 77.17%
         Val. Loss: 0.627 |  Val. Acc: 69.98%
Epoch: 09 | Epoch Time: 0m 8s
        Train Loss: 0.452 | Train Acc: 78.83%
         Val. Loss: 0.636 |  Val. Acc: 72.88%
Epoch: 10 | Epoch Time: 0m 8s
        Train Loss: 0.423 | Train Acc: 81.35%
         Val. Loss: 0.767 |  Val. Acc: 67.63%
```

| Test Loss | Test Acc |
|---|---|

| | |
|---|---|
| 0.691 | 66.45% |

| | |
|---|---|
| predict_sentiment(model, "This film is bad.") | 0.051 |
| predict_sentiment(model, "This film is good.") | 0.3179 |

**8 Layers**

```
INPUT_DIM = len(TEXT.vocab)
EMBEDDING_DIM = 100
HIDDEN_DIM = 256
OUTPUT_DIM = 1
N_LAYERS = 8
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)
```

**Result (8 Layers)**

```
Epoch: 01 | Epoch Time: 0m 16s
        Train Loss: 0.696 | Train Acc: 49.22%
         Val. Loss: 0.694 |  Val. Acc: 46.21%
Epoch: 02 | Epoch Time: 0m 16s
        Train Loss: 0.693 | Train Acc: 51.11%
         Val. Loss: 0.704 |  Val. Acc: 50.11%
Epoch: 03 | Epoch Time: 0m 17s
        Train Loss: 0.692 | Train Acc: 52.65%
         Val. Loss: 0.682 |  Val. Acc: 57.14%
Epoch: 04 | Epoch Time: 0m 17s
        Train Loss: 0.699 | Train Acc: 54.28%
         Val. Loss: 0.700 |  Val. Acc: 46.76%
Epoch: 05 | Epoch Time: 0m 17s
        Train Loss: 0.693 | Train Acc: 51.19%
         Val. Loss: 0.705 |  Val. Acc: 46.21%
Epoch: 06 | Epoch Time: 0m 17s
        Train Loss: 0.695 | Train Acc: 50.50%
         Val. Loss: 0.691 |  Val. Acc: 53.79%
Epoch: 07 | Epoch Time: 0m 17s
        Train Loss: 0.694 | Train Acc: 50.56%
         Val. Loss: 0.694 |  Val. Acc: 51.79%
Epoch: 08 | Epoch Time: 0m 17s
        Train Loss: 0.692 | Train Acc: 53.73%
         Val. Loss: 0.694 |  Val. Acc: 52.68%
Epoch: 09 | Epoch Time: 0m 17s
        Train Loss: 0.693 | Train Acc: 53.12%
         Val. Loss: 0.695 |  Val. Acc: 52.34%
Epoch: 10 | Epoch Time: 0m 17s
        Train Loss: 0.694 | Train Acc: 49.75%
         Val. Loss: 0.694 |  Val. Acc: 46.21%
```

| Test Loss | Test Acc |
|---|---|
| 0.681 | 57.70% |

| | |
|---|---|
| predict_sentiment(model, "This film is bad.") | 0.4973 |
| predict_sentiment(model, "This film is good.") | 0.5277 |

**Summary Test Result (4 and 8 Layers)**

| | Initial (2 Layers) | 4 Layers | 8 Layers |
|---|---|---|---|
| Test Loss | 0.732 | 0.691 | 0.681 |
| Test Acc | 64.52% | 66.45% | 57.70% |
| This film is bad. | 0.0672 | 0.051 | 0.4973 |
| This film is good. | 0.3861 | 0.3179 | 0.5277 |

## 2. Using a uni-directional LSTM

```python
import torch.nn as nn

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=dropout)
        self.fc = nn.Linear(hidden_dim, output_dim)
        self.dropout = nn.Dropout(dropout)

    def forward(self, text, text_lengths):
        embedded = self.dropout(self.embedding(text))
        packed_embedded = nn.utils.rnn.pack_padded_sequence(embedded, text_lengths.to('cpu'))
        packed_output, (hidden, cell) = self.rnn(packed_embedded)
        output, output_lengths = nn.utils.rnn.pad_packed_sequence(packed_output)
        hidden = self.dropout(hidden[-1,:,:])
        return self.fc(hidden)

INPUT_DIM = len(TEXT.vocab)
EMBEDDING_DIM = 100
HIDDEN_DIM = 256
OUTPUT_DIM = 1
N_LAYERS = 2
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)
```

**Result (Uni-directional LSTM)**

```
Epoch: 01 | Epoch Time: 0m 2s
        Train Loss: 0.694 | Train Acc: 49.80%
         Val. Loss: 0.689 |  Val. Acc: 52.01%
Epoch: 02 | Epoch Time: 0m 2s
        Train Loss: 0.684 | Train Acc: 56.60%
         Val. Loss: 0.682 |  Val. Acc: 55.80%
Epoch: 03 | Epoch Time: 0m 1s
        Train Loss: 0.667 | Train Acc: 59.20%
         Val. Loss: 0.664 |  Val. Acc: 62.17%
Epoch: 04 | Epoch Time: 0m 1s
        Train Loss: 0.641 | Train Acc: 63.05%
         Val. Loss: 0.638 |  Val. Acc: 62.95%
Epoch: 05 | Epoch Time: 0m 1s
        Train Loss: 0.614 | Train Acc: 65.19%
         Val. Loss: 0.704 |  Val. Acc: 61.61%
Epoch: 06 | Epoch Time: 0m 1s
        Train Loss: 0.591 | Train Acc: 68.69%
         Val. Loss: 0.646 |  Val. Acc: 63.62%
Epoch: 07 | Epoch Time: 0m 1s
        Train Loss: 0.565 | Train Acc: 71.38%
         Val. Loss: 0.640 |  Val. Acc: 64.40%
Epoch: 08 | Epoch Time: 0m 2s
        Train Loss: 0.532 | Train Acc: 72.99%
         Val. Loss: 0.687 |  Val. Acc: 66.63%
Epoch: 09 | Epoch Time: 0m 2s
        Train Loss: 0.481 | Train Acc: 78.25%
         Val. Loss: 1.100 |  Val. Acc: 63.17%
Epoch: 10 | Epoch Time: 0m 2s
        Train Loss: 0.444 | Train Acc: 78.74%
         Val. Loss: 0.633 |  Val. Acc: 70.98%
```

| Test Loss | Test Acc |
|---|---|
| 0.701 | 67.98% |

| | |
|---|---|
| predict_sentiment(model, "This film is bad.") | 0.1952 |
| predict_sentiment(model, "This film is good.") | 0.5726 |

**Summary Test Result (Uni-directional LSTM)**

| | Initial (Bidirectional) | Uni-directional |
|---|---|---|
| Test Loss | 0.732 | 0.701 |
| Test Acc | 64.52% | 67.98% |
| This film is bad. | 0.0672 | 0.1952 |
| This film is good. | 0.3861 | 0.5726 |

### 3. Changing the embedding dimension

## Changing the Embedding Dimension to 200

```
INPUT_DIM = len(TEXT.vocab)
EMBEDDING_DIM = 200
HIDDEN_DIM = 256
OUTPUT_DIM = 1
N_LAYERS = 2
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)
```

## Result (Embedding Dimension to 200)

```
Epoch: 01 | Epoch Time: 0m 4s
        Train Loss: 0.696 | Train Acc: 50.65%
         Val. Loss: 0.683 |  Val. Acc: 61.61%
Epoch: 02 | Epoch Time: 0m 4s
        Train Loss: 0.667 | Train Acc: 59.77%
         Val. Loss: 0.686 |  Val. Acc: 57.70%
Epoch: 03 | Epoch Time: 0m 4s
        Train Loss: 0.611 | Train Acc: 66.64%
         Val. Loss: 0.812 |  Val. Acc: 60.83%
Epoch: 04 | Epoch Time: 0m 4s
        Train Loss: 0.564 | Train Acc: 69.93%
         Val. Loss: 0.631 |  Val. Acc: 69.20%
Epoch: 05 | Epoch Time: 0m 4s
        Train Loss: 0.459 | Train Acc: 79.01%
         Val. Loss: 0.618 |  Val. Acc: 71.99%
Epoch: 06 | Epoch Time: 0m 4s
        Train Loss: 0.403 | Train Acc: 80.65%
         Val. Loss: 0.790 |  Val. Acc: 69.64%
Epoch: 07 | Epoch Time: 0m 4s
        Train Loss: 0.338 | Train Acc: 84.78%
         Val. Loss: 0.997 |  Val. Acc: 66.74%
Epoch: 08 | Epoch Time: 0m 4s
        Train Loss: 0.299 | Train Acc: 87.50%
         Val. Loss: 0.724 |  Val. Acc: 71.76%
Epoch: 09 | Epoch Time: 0m 4s
        Train Loss: 0.247 | Train Acc: 90.05%
         Val. Loss: 0.833 |  Val. Acc: 74.44%
Epoch: 10 | Epoch Time: 0m 4s
        Train Loss: 0.227 | Train Acc: 91.38%
         Val. Loss: 1.213 |  Val. Acc: 69.64%
```

| Test Loss | Test Acc |
| --- | --- |

| | |
|---|---|
| 0.747 | 66.30% |

| | |
|---|---|
| predict_sentiment(model, "This film is bad.") | 0.0745 |
| predict_sentiment(model, "This film is good.") | 0.5731 |

**Changing the Embedding Dimension to 300**

```
INPUT_DIM = len(TEXT.vocab)
EMBEDDING_DIM = 300
HIDDEN_DIM = 256
OUTPUT_DIM = 1
N_LAYERS = 2
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)
```

**Result (Embedding Dimension to 300)**

```
Epoch: 01 | Epoch Time: 0m 4s
        Train Loss: 0.694 | Train Acc: 50.76%
         Val. Loss: 0.685 |  Val. Acc: 53.24%
Epoch: 02 | Epoch Time: 0m 4s
        Train Loss: 0.648 | Train Acc: 63.20%
         Val. Loss: 0.659 |  Val. Acc: 62.95%
Epoch: 03 | Epoch Time: 0m 4s
        Train Loss: 0.573 | Train Acc: 71.01%
         Val. Loss: 0.634 |  Val. Acc: 67.41%
Epoch: 04 | Epoch Time: 0m 4s
        Train Loss: 0.496 | Train Acc: 74.84%
         Val. Loss: 0.664 |  Val. Acc: 68.75%
Epoch: 05 | Epoch Time: 0m 4s
        Train Loss: 0.412 | Train Acc: 81.30%
         Val. Loss: 0.633 |  Val. Acc: 71.32%
Epoch: 06 | Epoch Time: 0m 4s
        Train Loss: 0.332 | Train Acc: 85.19%
         Val. Loss: 0.730 |  Val. Acc: 70.42%
Epoch: 07 | Epoch Time: 0m 4s
        Train Loss: 0.289 | Train Acc: 88.11%
         Val. Loss: 1.183 |  Val. Acc: 65.29%
Epoch: 08 | Epoch Time: 0m 4s
        Train Loss: 0.213 | Train Acc: 91.26%
         Val. Loss: 0.878 |  Val. Acc: 70.42%
Epoch: 09 | Epoch Time: 0m 4s
        Train Loss: 0.205 | Train Acc: 91.99%
         Val. Loss: 0.975 |  Val. Acc: 70.98%
Epoch: 10 | Epoch Time: 0m 4s
        Train Loss: 0.144 | Train Acc: 94.53%
         Val. Loss: 0.846 |  Val. Acc: 76.56%
```

| Test Loss | Test Acc |
|---|---|
| 0.733 | 64.69% |

| | |
|---|---|
| predict_sentiment(model, "This film is bad.") | 0.0251 |
| predict_sentiment(model, "This film is good.") | 0.2672 |

**Summary Test Result (200 and 300)**

| | Initial (100) | 200 dimensions | 300 dimensions |
|---|---|---|---|
| Test Loss | 0.732 | 0.747 | 0.733 |
| Test Acc | 64.52% | 66.30% | 64.69% |
| This film is bad. | 0.0672 | 0.0745 | 0.0251 |
| This film is good. | 0.3861 | 0.5731 | 0.2672 |

## 4. Do Not Use Dropout During Training

```
INPUT_DIM = len(TEXT.vocab)
EMBEDDING_DIM = 100
HIDDEN_DIM = 256
OUTPUT_DIM = 1
N_LAYERS = 2
BIDIRECTIONAL = False
DROPOUT = 0
PAD_IDX = TEXT.vocab.stoi[TEXT.pad_token]

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

### Result (Dropout is not used)

```
Epoch: 01 | Epoch Time: 0m 4s
        Train Loss: 0.694 | Train Acc: 51.59%
         Val. Loss: 0.679 |  Val. Acc: 61.05%
Epoch: 02 | Epoch Time: 0m 4s
        Train Loss: 0.641 | Train Acc: 63.88%
         Val. Loss: 0.663 |  Val. Acc: 60.49%
Epoch: 03 | Epoch Time: 0m 4s
        Train Loss: 0.525 | Train Acc: 74.56%
         Val. Loss: 0.665 |  Val. Acc: 64.73%
Epoch: 04 | Epoch Time: 0m 4s
        Train Loss: 0.363 | Train Acc: 84.32%
         Val. Loss: 0.704 |  Val. Acc: 66.18%
Epoch: 05 | Epoch Time: 0m 4s
        Train Loss: 0.195 | Train Acc: 92.59%
         Val. Loss: 0.937 |  Val. Acc: 67.41%
Epoch: 06 | Epoch Time: 0m 4s
        Train Loss: 0.098 | Train Acc: 96.57%
         Val. Loss: 0.958 |  Val. Acc: 66.85%
Epoch: 07 | Epoch Time: 0m 4s
        Train Loss: 0.066 | Train Acc: 97.88%
         Val. Loss: 1.188 |  Val. Acc: 66.96%
Epoch: 08 | Epoch Time: 0m 4s
        Train Loss: 0.043 | Train Acc: 98.79%
         Val. Loss: 1.398 |  Val. Acc: 65.96%
Epoch: 09 | Epoch Time: 0m 4s
        Train Loss: 0.019 | Train Acc: 99.45%
         Val. Loss: 1.503 |  Val. Acc: 65.40%
Epoch: 10 | Epoch Time: 0m 4s
        Train Loss: 0.016 | Train Acc: 99.60%
         Val. Loss: 1.439 |  Val. Acc: 68.19%
```

| Test Loss | Test Acc |
|---|---|
| 0.683 | 55.99% |

| | |
|---|---|
| predict_sentiment(model, "This film is bad.") | 0.3213 |
| predict_sentiment(model, "This film is good.") | 0.3335 |

**Summary Test Result (Dropout)**

| | Initial (Dropout used) | Dropout not used |
|---|---|---|
| Test Loss | 0.732 | 0.683 |
| Test Acc | 64.52% | 55.99% |
| This film is bad. | 0.0672 | 0.3213 |
| This film is good. | 0.3861 | 0.3335 |

## Conclusion

**RNN:**

**Summary the All Results**

Based on the increasing number of RNN layers results, increasing to 2 layers can slightly boost the test accuracy from 49.89% to 50.89% while the test loss changed from 0.697 to 0.693. The test accuracy of the 4 and 8 layers is 50.61% and 49.86%, respectively, while the test loss is 0.693 and 0.694. However, the RNN layers increase to 4 and 8 layers do not continue to boost accuracy significantly and even slightly reduce. This means adding more layers may lead to overfitting. Also, RNNs are susceptible to the vanishing gradient problem. These factors will affect test accuracy.

Moreover, changing the embedding dimension of RNN also impacts the model performance. The embedding dimension of RNN increase to 200 can slightly boost the test accuracy from 49.89% to 50.31% and test loss decrease from 0.697 to 0.692. With the embedding dimension of RNN increased to 300, the test accuracy reaches 50.50%, and the test loss is 0.696. In contrast, reducing the embedding dimension of RNN to 50 will decrease the test accuracy from 49.89% to 48.12%, and the test loss decrease from 0.697 to 0.694 while the embedding dimension of RNN decreases to 25, the test accuracy and test loss is 48.04% and 0.695 respectively. In view of this, increasing the embedding dimension from the

initial setup (100) to 200 and 300 provided slight improvements in accuracy, while decreasing it below 100 led to poorer performance. This means increasing the embedding dimension gives each word a richer, more detailed representation to help capture more nuances in the text.

**LSTM:**

**Summary the All Results**

When increasing the LSTM layers from 2 to 4, the test loss decreases from 0.732 to 0.691, and test accuracy improves from 64.52% to 66.45%. Adding layers enhances the model's ability to learn complex patterns. However, increasing to 8 layers causes the test loss to decrease to 0.681, but test accuracy reduces to 57.70%. This means overfitting or vanishing gradients in networks, impairing generalization. Regarding sentiment predictions, with 4 layers, the model assigned a low score (0.051) to "This film is bad." and a higher score (0.3179) to "This film is good." At 8 layers, the scores converge around 0.5 for both sentences, showing reduced discriminative ability. This means that additional layers hinder the model's capacity to predict sentiments correctly beyond a certain depth.

Furthermore, changing from a bidirectional to a unidirectional LSTM reduces test loss from 0.732 to 0.701 and increases test accuracy from 64.52% to 67.98%. This improvement suggests that the unidirectional model generalizes better. In sentiment predictions, the unidirectional LSTM assigns a higher score to "This film is good." (0.5726) and a moderate score to "This film is bad." (0.1952), effectively distinguishing between positive and negative sentiments. The bidirectional model assigns lower scores to both sentences (0.3861 and 0.0672), indicating less discriminative power. The unidirectional model's simplicity may prevent overfitting and allow it to enhance prediction accuracy.

In addition, increasing the embedding dimensions from 100 to 200 slightly raises test loss from 0.732 to 0.747 but improves test accuracy from 64.52% to 66.30%.

This suggests that a 200-dimensional embedding captures more semantic nuances, aiding classification. However, increasing dimensions to 300 does not enhance performance; test accuracy decreases slightly to 64.69%, and test loss returns to 0.733. In sentiment predictions, the 200-dimensional model assigns a higher score to "This film is good." (0.5731) and keeps a low score for "This film is bad." (0.0745), improving discriminative ability. The 300-dimensional model needs to show further improvement, possibly due to increased complexity leading to overfitting. Therefore, the balance of the 200 dimensions captures meaningful information and maintains generalization in sentiment prediction.

Also, when dropout is not used, test loss decreases from 0.732 to 0.683, indicating the model fits the training data more closely. However, test accuracy drops significantly from 64.52% to 55.99%, showing poor generalization. The sentiment scores for "This film is bad." (0.3213) and "This film is good." (0.3335) become similar, suggesting the model cannot effectively distinguish between negative and positive sentiments. In view of this, dropout can prevent overfitting and maintain its ability to discriminate sentiments in predictions.