6CS012 - Artificial Intelligence and Machine Learning. Sequential Modeling for Text Classification Using RNN.

Prepared By: Siman Giri {Module Leader - 6CS012}

April 21, 2025

Tutorial - 9.

1 Instructions

This sheet contains hands on exercises for various operations performed during Text Pre - processing, followed by Text Representations.

- Please Complete all the exercise with pen and on paper.

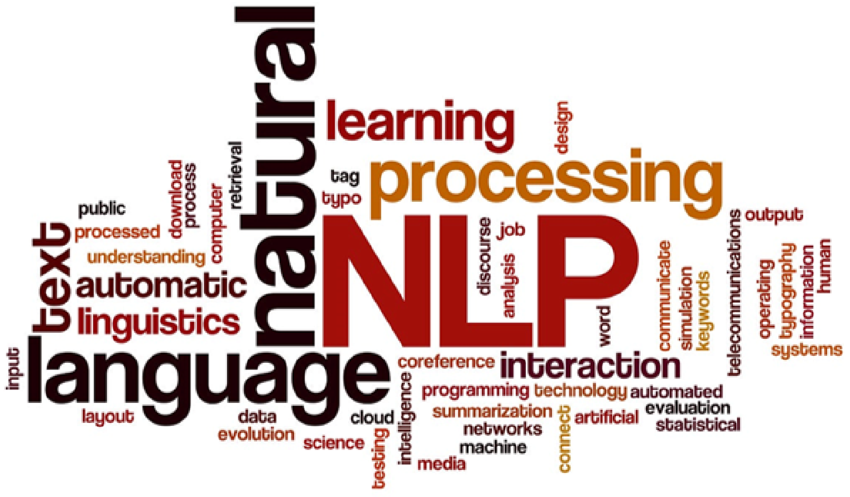


Figure 1: Text and Natural Language Processing.

2 Introduction.

Sentiment analysis, a sub-task of **text classification** within Natural Language Processing (NLP), involves the identification and categorization of the sentiment or opinion expressed within a piece of text. This technique is employed to determine whether a given text conveys a positive, neutral, or negative tone. Sentiment analysis is widely applied in various domains, including customer feedback analysis, social media monitoring, and brand reputation management.

This study focuses on sentiment analysis specifically applied to Twitter reviews, which serve as a rich and dynamic source of real-time opinions and feedback. Due to the brevity and informality of tweets, combined with the frequent use of slang, emojis, hashtags, and abbreviations, sentiment classification in this domain poses significant challenges. These unique characteristics make Twitter data both complex and interesting to analyze, requiring specialized techniques to accurately interpret sentiment.

In this work, we will explore the development of a sentiment analysis pipeline using deep learning approaches, particularly Recurrent Neural Networks (RNN) and Long Short-Term Memory (LSTM) models. These models are well-suited for processing sequential data due to their ability to capture temporal dependencies within text. This makes them particularly effective for understanding the context and flow of words in a tweet, which is crucial for accurate sentiment classification.

In addition to the deep learning-based approaches, this study will also address the development of a heuristic approach for sentiment analysis. This rule-based method can be beneficial for quick insights or when computational resources or data availability are limited. While the heuristic approach is less sophisticated than deep learning models, it offers a useful alternative for preliminary sentiment analysis tasks.

The objective of this tutorial is to provide a comprehensive guide on how to build an effective sentiment analysis system, covering the following key steps:

- Data Preprocessing and Cleaning: This step involves preparing the raw text data by removing noise, handling missing values, and transforming the data into a format suitable for modeling.
- Tokenization and Padding: Text data is converted into numerical representations using tokenization. Padding is then applied to ensure uniform sequence lengths, which is essential for processing with neural networks.
- Model Creation: We will design and implement both an RNN and an LSTM model for sentiment classification. These models are designed to learn sequential dependencies in the data, making them suitable for understanding the context in which sentiment is expressed.
- Training and Evaluation: The models will be trained on labeled Twitter review data, and various performance metrics such as accuracy, precision, recall, and F1 score will be used to evaluate their effectiveness.
- Visualization of Results: The training and validation losses will be visualized to track model performance over time, and insights into model behavior will be derived.
- Development of a Simple GUI for Real-Time Sentiment Prediction: A graphical user interface (GUI) will be created to allow for the real-time prediction of sentiment in new, unseen Twitter reviews.

By the conclusion of this tutorial, students will have gained a detailed understanding of the end-to-end process of sentiment analysis, from data preprocessing to model evaluation, and will be equipped with the knowledge to build and deploy sentiment classifiers using deep learning techniques.

3 System Design.

The system is designed to perform sentiment analysis on Twitter review data using Recurrent Neural Network. It follows a modular architecture with distinct components for data handling, model training, prediction, and user interaction through a GUI. The following subsections describe each part of the system:

3.1 Data Pre - processing and Analysis:

Import Necessary Library

```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Embedding, SimpleRNN, LSTM, Dense, Dropout
import re
import nltk
from nltk.corpus import stopwords
from nltk.stem import WordNetLemmatizer

nltk.download('stopwords')
nltk.download('wordnet')
```

1. Data Collection and Input:

- The system uses a pre-collected dataset of tweets.
- Each entry contains two key fields:
  1. `text`: the tweet content
  2. `sentiment`: the corresponding sentiment label:
     - **1** → `Positive`.
     - **0** → `Negative`.

1. Data Cleaning:

- Responsible for cleaning and preparing the text for modeling. Steps include:
  1. Lowercasing the text
  2. Removing URLs, mentions, hashtags, numbers, and special characters
  3. Handling contractions and whitespace
  4. Removing stop words and handling emojis
  5. Any methods seems fit and required as per your data requirement.
- Cleaned data is passed to the tokenization module.

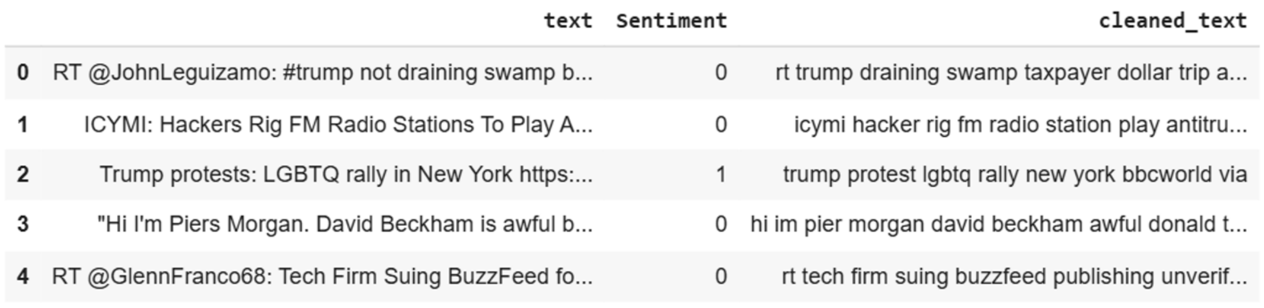
Sample Code - Data Cleaning.

```
# Preprocessing function
def preprocess_text(text):
    text = text.lower() # Lowercase
    text = re.sub(r’http\S+|www\S+|https\S+’, ’’, text, flags=re.MULTILINE) # Remove URLs
    text = re.sub(r’\@\w+|\#’, ’’, text) # Remove mentions/hashtags
    text = re.sub(r’[^\w\s]’, ’’, text) # Remove punctuation
    text = re.sub(r’\d+’, ’’, text) # Remove numbers

    # Remove stopwords and lemmatize
    stop_words = set(stopwords.words(’english’))
    lemmatizer = WordNetLemmatizer()
    tokens = text.split()
    tokens = [lemmatizer.lemmatize(word) for word in tokens if word not in stop_words]

    return ’ ’.join(tokens)
```

The Cleaning pipeline applied to `text` of dataset.



	text	Sentiment	cleaned_text
0	RT @JohnLeguizamo: #trump not draining swamp b...	0	rt trump draining swamp taxpayer dollar trip a...
1	ICYMI: Hackers Rig FM Radio Stations To Play A...	0	icymi hacker rig fm radio station play antitru...
2	Trump protests: LGBTQ rally in New York https:...	1	trump protest lgbtq rally new york bbcworld via
3	"Hi I'm Piers Morgan. David Beckham is awful b...	0	hi im pier morgan david beckham awful donald t...
4	RT @GlennFranco68: Tech Firm Suing BuzzFeed fo...	0	rt tech firm suing buzzfeed publishing unverif...

Figure 2: A sample data before and after cleaning.

1.1 Some Basic Data Visualizations:

A wordcloud visualizations.

```
# --- Visualization: Word Cloud (Top 100 Words) ---
all_words = ’ ’.join(data[’cleaned_text’])
# Generate WordCloud with only top 100 words
wordcloud = WordCloud(
    width=300,
    height=100,
    background_color=’white’,
    max_words=100 # Limit to top 100 words
).generate(all_words)
# Plot settings
plt.figure(figsize=(10, 6))
plt.imshow(wordcloud, interpolation=’bilinear’)
plt.axis(’off’)
plt.title(’Top 100 Most Frequent Words’)
plt.show()
```

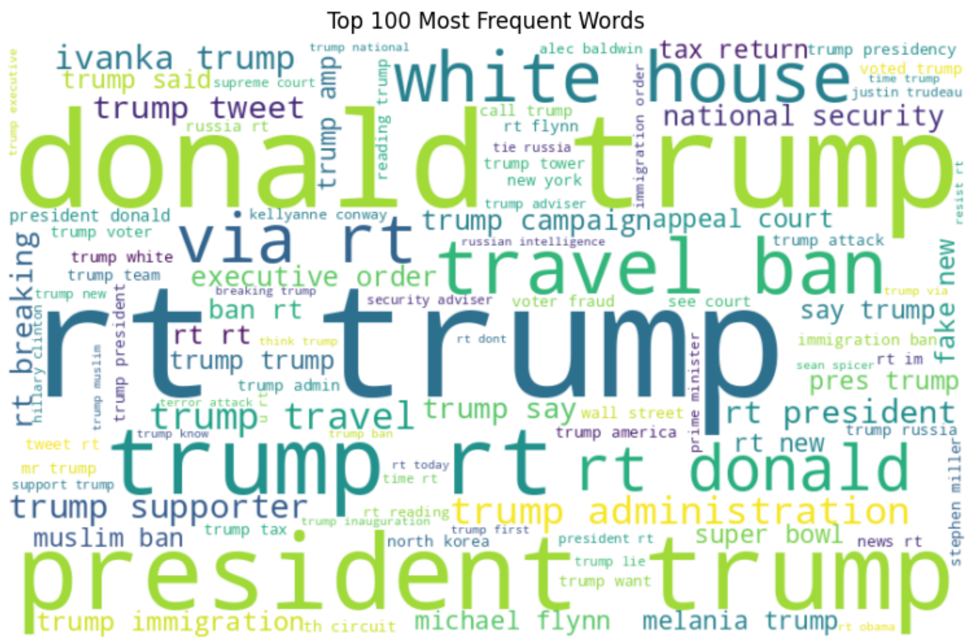


Figure 3: A word cloud.

3.2 Tokenization and Padding:

Tokenization is the process of converting text into individual units (tokens), such as words or subwords, which can then be numerically encoded for model training.

Tokenization is performed after splitting the dataset to prevent information leakage ensuring that the tokenizer only learns vocabulary from the training data and not from unseen test data.

Correct Workflow → Split → Tokenize → Apply Padding

1. Split the Data:

Split the Dataset.

```python
from sklearn.model_selection import train_test_split
# Split FIRST (before tokenization)
X_train, X_test, y_train, y_test = train_test_split(
    data['cleaned_text'], # Features (text)
    data['sentiment_encoded'], # Labels
    test_size=0.2,
    random_state=42
)
```

2. Basic Tokenization workflow with keras:

Tokenization Syntax with keras:

```python
from keras.preprocessing.text import Tokenizer
from keras.preprocessing.sequence import pad_sequences
# Sample Twitter reviews
```

```
texts = [
    "I love this product!",
    "Worst purchase ever.",
    "Happy with the experience.",
    "Not great, not terrible.",
    "Absolutely fantastic!"
]
# Initialize the tokenizer
tokenizer = Tokenizer(num_words=10000, oov_token="<OOV>") # 'num_words' limits vocab size; '
    oov_token' handles unknown words
tokenizer.fit_on_texts(texts) # Learn word index from the training texts
# Convert text to sequences of integers
sequences = tokenizer.texts_to_sequences(texts)
# Print results
print("Word Index:", tokenizer.word_index)
print("Sequences:", sequences)
```

- **Explanation of key Steps:**
  1. `Tokenizer()`: Creates a tokenizer object that learns word - to - index mappings.
  2. `.fit_on_texts(texts)`: Builds the vocabulary by assigning a unique index to each word based on frequency.
  3. `.texts_to_sequences(texts)`: Converts words in each sentence to their corresponding integer index.

2.1 Tokenization Using Training Data:

How to Tokenize Training Data?

```
from keras.preprocessing.text import Tokenizer
from keras.preprocessing.sequence import pad_sequences
tokenizer = Tokenizer(num_words=10000, oov_token='<OOV>')
tokenizer.fit_on_texts(X_train) # Vocabulary based ONLY on training data
```

- The above code:
  1. **Purpose:** Initializes a Keras `Tokenizer` with a maximum vocabulary size of $10,000$ words.
  2. `oov_token = "<OOV>"`: Ensures that any word not seen during training is replaced with a special "out - of - vocabulary" token during inference.
  3. **Why only on `X_train` ?** Prevents information from test data leaking into the training process, ensuring fair and generalizable evaluation.

Why only on `X_train`?

We tokenize the test data using the tokenizer fitted on the training data to ensure consistent preprocessing without leaking test information. So we apply `.texts_to_sequences()` on the test data using the same tokenizer fitted or build on train vocabulary.

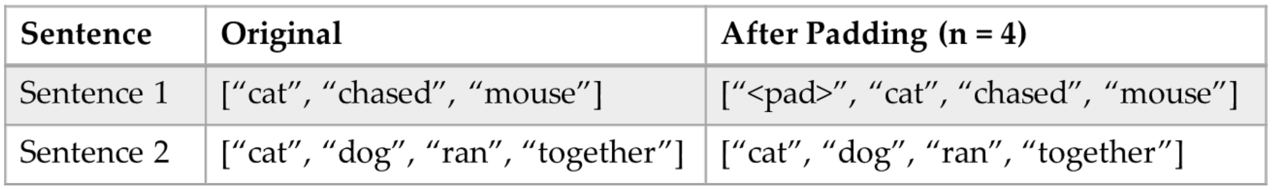
Cautions: `Don't run tokenizer.fit_on_texts(X_test) - That would update the tokenizer's vocabulary and leak information from the test set into the model.`

How to Tokenize Training and Test Data?

```
from keras.preprocessing.text import Tokenizer
from keras.preprocessing.sequence import pad_sequences
tokenizer = Tokenizer(num_words=10000, oov_token='<OOV>')
tokenizer.fit_on_texts(X_train) # Vocabulary based ONLY on training data
# Use the same tokenizer to transform both sets
X_train_seq = tokenizer.texts_to_sequences(X_train)
X_test_seq = tokenizer.texts_to_sequences(X_test)
```

3. Padding:

Padding is a technique used in Natural Language Processing (NLP) to ensure all input sequences (sentences, tweets, etc.) have the same length before feeding them into a machine learning model (like RNNs, LSTMs, or Transformers). There is two option in applying maximum length padding, let's look



Sentence	Original	After Padding (n = 4)
Sentence 1	[“cat”, “chased”, “mouse”]	[“<pad>”, “cat”, “chased”, “mouse”]
Sentence 2	[“cat”, “dog”, “ran”, “together”]	[“cat”, “dog”, “ran”, “together”]

Figure 4: Max Length Padding.

one after another:

Option 1: Fixed Padding (Using Maximum Length)

Max Len Padding:

```
max_len = max(len(seq) for seq in X_train_seq) # Longest sequence in training data
```

- **How it Works?**
  - Pads all sequences to match the length of the longest sentence in your dataset.
- **Pros:**
  1. No information loss (no truncation)
  2. Simple to implement
- **Cons:**
  1. Can create very long vectors (memory-heavy)
  2. Wastes space if 99% of sentences are shorter
  3. Example: If longest tweet has 200 words but 95% have ≤ 50 words, you're adding 150 zeros to most sequences.
- **When to Use?**
  1. Small datasets or all sentences are roughly similar in length.

Option 2: Percentile-Based Padding

Percentile - Based Padding.

```python
import numpy as np
seq_lengths = [len(seq) for seq in X_train_seq]
max_len = int(np.percentile(seq_lengths, 95)) # Covers 95% of sentences
```

- **How it Works?**
  - Uses a percentile (e.g., 95th) to determine padding length.
  - Only 5% of sentences will be truncated.
- **Pros:**
  1. Balances memory usage and information retention
  2. Removes outlier-length sentences
- **Cons:**
  1. May truncate a small portion of long sentences
- **When to Use?**
  1. Large datasets with varied sentence lengths
  2. When memory efficiency is important.

A Sample Implementation in Practice with Keras:

Padding with Keras.

```python
# Calculate max_len using percentile
seq_lengths = [len(seq) for seq in X_train_seq]
max_len = int(np.percentile(seq_lengths, 95)) # Adjust percentile as needed
# Pad sequences
X_train_pad = pad_sequences(X_train_seq, maxlen=max_len, padding='post', truncating='post')
X_test_pad = pad_sequences(X_test_seq, maxlen=max_len, padding='post', truncating='post')
```

1. padding='post': Adds zeros at the end of sequences (better for RNNs).
2. truncating='post': Cuts off excess words from the end if exceeding max_len.

Why 95% percentile?

Choosing the 95th percentile ensures that 95% of your sequences are preserved without truncation, while keeping the model input size manageable. But the appropriate percentile can be figured out from simple analysis of tokens, for example:

"Why 95th percentile ?"

```
X_seq = tokenizer.texts_to_sequences(data['cleaned_text'])
seq_lengths = [len(seq) for seq in X_seq]
plt.figure(figsize=(8, 5))
plt.hist(seq_lengths, bins=50, color='skyblue')
plt.title('Text Length Distribution')
plt.xlabel('Sequence Length')
plt.ylabel('Frequency')
plt.axvline(np.percentile(seq_lengths, 95), color='red', linestyle='dashed', label='95th Percentile'
    )
plt.legend()
plt.show()
```

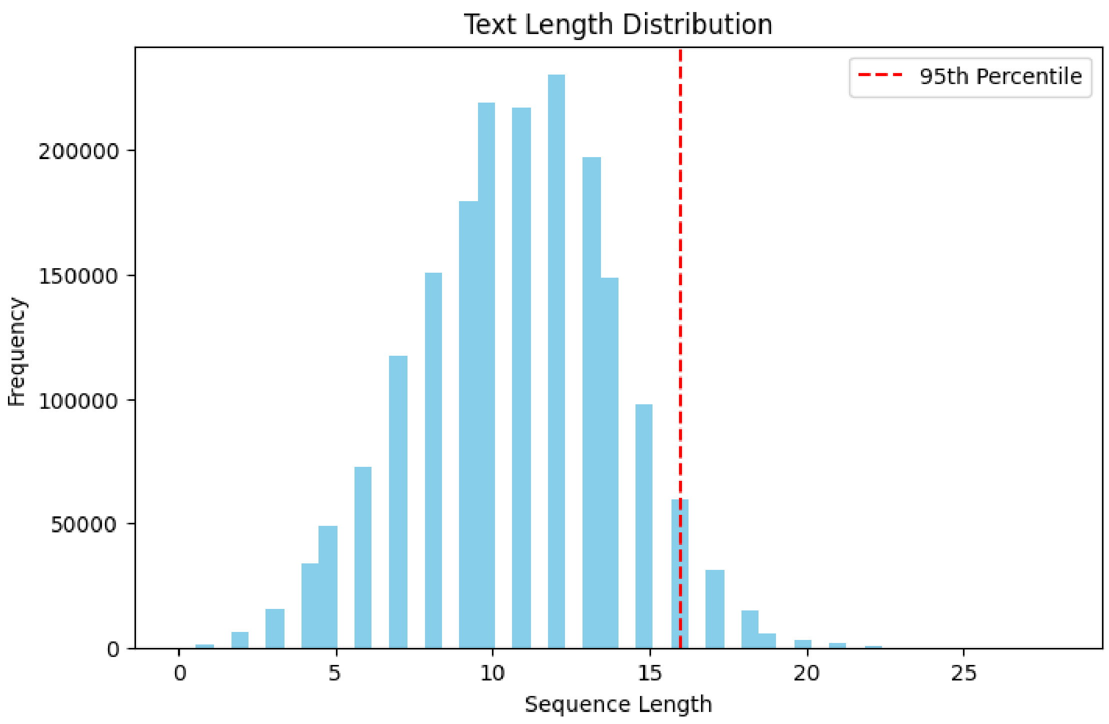


Figure 5: A Sample visualization of Tokens - This is Optional.

3.3 Model Creation:

Two Models are implemented:

- **Embedding Layer.**
- **Simple RNN.**
- **LSTM.**

1. Understanding Embedding Layer:

This layer is used to convert the input text (words) into dense vectors of fixed size (e.g., 128 dimensions). It learns a continuous representation of words based on their semantic meanings.

Embedding Layer.

```
model.add(Embedding(input_dim=10000, output_dim=128, input_length=50))
```

This line defines an Embedding Layer in a neural network. Here's a breakdown of the parameters:

1. `input_dim = 10000`:
   - This specifies the size of the vocabulary. In other words, it defines how many unique words your model can handle.
   - For example, if you have a vocabulary size of 10,000 then the model will be able to work with the top 10,000 most frequent words in your dataset.
   - Typically, this is set to the number of unique tokens in your dataset + 1 (to account for padding).
   - In context: If your dataset contains 10,000 unique words, setting input_dim=10000 means you will have an index range from 0 to 9999, where each unique word is mapped to a number in this range.
2. `output_dim = 128`:
   - This defines the size of the dense vector (embedding) that will represent each word.
   - A higher value for `output_dim` creates a larger vector, which allows the model to capture more detailed relationships between words, but it also requires more computational resources.
   - In this case, each word will be represented by a 128-dimensional vector.
   - In context: The 128-dimensional vector will be initialized randomly, but during training, it will adjust based on the model's learning (backpropagation) to represent the words in a way that is useful for your task.
3. input_length = 50:
   - This specifies the length of the input sequences.
   - Here, 50 means that the model expects each input (e.g., a sentence or tweet) to have a fixed length of 50 tokens (words or subwords).
   - If a sequence is shorter than 50 tokens, it will be padded (typically with zeros or other padding tokens). If it is longer, it will be truncated to fit this length.
   - In context: This ensures that each input sequence has the same length, which is important for efficient batch processing in neural networks. In practice, shorter sequences will be padded with 0, while longer sequences will be truncated.

1.1 How Does the Embedding Work?

The Embedding Layer in Keras works by creating an embedding matrix that is essentially a lookup table where each row corresponds to a word in the vocabulary and each column corresponds to a dimension in the embedding vector.

Steps in Detail:

1. **Initialization of the Embedding Matrix:**
   - Initially, the embedding matrix is randomly initialized. It has a shape of (input_dim, output_dim), i.e., (10000, 128) in this case.
     - This means it has 10,000 rows (one for each word in the vocabulary) and 128 columns (one for each dimension of the embedding vector).
2. **Word Indexing:**
   - When text data is provided as input (such as a sentence or tweet), it is first tokenized and converted into a sequence of integers, where each integer corresponds to the index of a word in the vocabulary.
3. **Lookup** :
   - The Embedding Layer then uses these indices to look up the corresponding embedding vectors from the embedding matrix.
4. **Training:**
   - During training, the embedding vectors are updated based on the model's learning (backpropagation). The network will adjust the embeddings so that words that are used in similar contexts are represented by similar vectors in the embedding space. This allows the model to learn semantic relationships between words during the training process.

Example:

If the embedding matrix looks like this (just a very simplified example for illustration):

Index	**Word**	**Embedding Vector**
4	”I”	[0.12, -0.03, 0.27, . . . , 0.05]
2	”love”	[0.35, 0.05, -0.27, . . . , -0.09]
8	”this”	[-0.11, 0.45, 0.33, . . . , -0.23]
15	”movie”	[0.09, -0.12, 0.15, . . . , 0.11]

Table 1: Word embeddings table

The embedding layer will output a matrix like this for your input:

$$\begin{bmatrix} 0.12 & -0.03 & 0.27 & \cdots & 0.05 \\ 0.35 & 0.05 & -0.27 & \cdots & -0.09 \\ -0.11 & 0.45 & 0.33 & \cdots & -0.23 \\ 0.09 & -0.12 & 0.15 & \cdots & 0.11 \end{bmatrix}$$

This matrix is the embedded representation of the input text ”I love this movie”, where each word is represented by a 128-dimensional vector.

A model with Embedding Layer and RNN.

```
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Embedding, SimpleRNN, Dense
# Example parameters
vocab_size = 10000 # Size of vocabulary
embedding_dim = 128 # Dimension of embedding vectors
max_length = 100 # Maximum length of input sequences
rnn_units = 64 # Number of RNN units
# Build the model
model = Sequential([
    # Embedding layer converts word indices to dense vectors
    Embedding(input_dim=vocab_size,
              output_dim=embedding_dim,
              input_length=max_length),

    # RNN layer processes the sequence
    SimpleRNN(units=rnn_units, return_sequences=False),
    # Final dense layer for classification
    Dense(1, activation='sigmoid')
])
```

1.2 Using word2vec with the Embedding Layer:

You can combine Word2Vec and the Embedding Layer in your deep learning models to leverage pre-trained embeddings for better results.

Here is How it Works?

1. **Train or Obtain Pre-trained Word2Vec Embeddings:**
   - You can train your own Word2Vec model using a library like Gensim, or you can use pre-trained embeddings like Google's Word2Vec (trained on Google News data).
   - The output will be a set of word vectors where each word is represented by a dense vector.
2. **Create an Embedding Matrix**: Once you have the Word2Vec embeddings (either trained or pre-trained), you need to create an embedding matrix that maps each word in your dataset's vocabulary to its corresponding Word2Vec vector.
   - Here is an example of how to create the embedding matrix:

Create an Embedding Matrix for word2vec.

```
    import numpy as np
from gensim.models import KeyedVectors
# Load the pre-trained Word2Vec model (e.g., Google's Word2Vec)
word2vec_model = KeyedVectors.load_word2vec_format('GoogleNews-vectors-negative300.bin',
    binary=True)
# Initialize the embedding matrix with zeros (vocab_size, embedding_dim)
vocab_size = len(tokenizer.word_index) + 1 # Add 1 for padding
embedding_dim = 300 # Word2Vec uses 300-dimensional vectors
embedding_matrix = np.zeros((vocab_size, embedding_dim))
# Map each word in the tokenizer's word index to its Word2Vec vector
for word, index in tokenizer.word_index.items():
    if word in word2vec_model:
        embedding_matrix[index] = word2vec_model[word]
```

   - **Use the Embedding Matrix in the Embedding Layer**: After creating the embedding_matrix, you can pass it to the Embedding Layer as weights. This will initialize the Embedding Layer with pre-trained Word2Vec vectors, and you can either freeze these weights (i.e., don't update them during training) or fine-tune them.

A model with word2vec and LSTM.

```
from keras.models import Sequential
from keras.layers import Embedding, LSTM, Dense
model = Sequential()
# Use the pre-trained Word2Vec embeddings in the Embedding Layer
model.add(Embedding(input_dim=vocab_size,
                    output_dim=embedding_dim,
                    weights=[embedding_matrix], # Load the pre-trained Word2Vec embeddings
                    input_length=50,
                    trainable=False)) # Set trainable=True if you want to fine-tune the
                        embeddings
model.add(LSTM(64)) # Add an LSTM layer
model.add(Dense(3, activation='softmax')) # Output layer for sentiment classification
```

Summary: Embedding Layer vs. Word2vec:

- **Embedding Layer**: A layer in a neural network that learns word embeddings as part of the model's training. It is used to transform words into dense vector representations that capture semantic meaning.
- **Word2Vec**: A pre-trained method of learning word embeddings that can be used to initialize the Embedding Layer. Word2Vec embeddings capture semantic relationships between words based on their context in large corpora.

Combining Word2Vec with the Embedding Layer:

You can use pre-trained Word2Vec embeddings to initialize the Embedding Layer in your model, and either freeze the embeddings or fine-tune them during model training to improve performance on the task at hand (like sentiment analysis).

2. A Simple RNN:

An RNN (Recurrent Neural Network) is a class of neural networks designed for sequential data. Unlike traditional neural networks, which assume that the input data is independent of each other (i.e., feedforward networks), RNNs take into account the sequential nature of the data. They have feedback loops, meaning that the output from the previous step is fed back into the network, allowing the model to maintain memory of previous inputs.

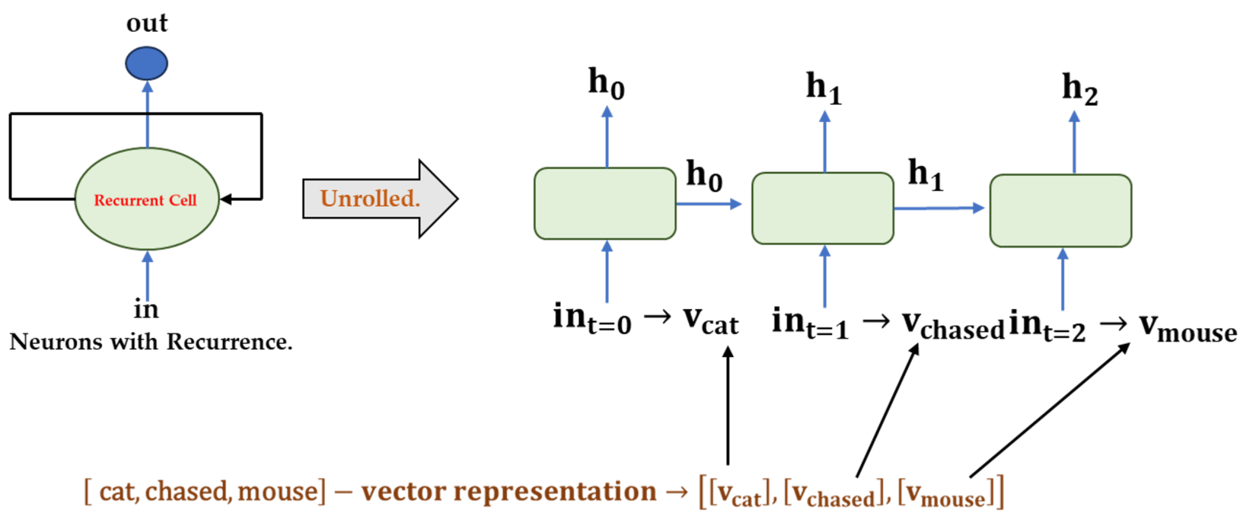


Figure 6: Unrolled Recurrence Unit.

2.1 Architecture of "simple" or "vanilla" RNN:

At the heart of an RNN is the idea that each output depends on both the current input and the previous hidden state. Here's how it works in more detail:

- **Input Layer**: Each input is fed into the network one at a time. For example, a sentence is split into words (tokens), and each word is fed as an input vector to the RNN.

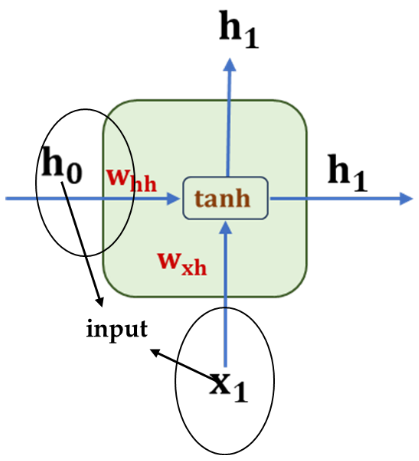


Figure 7: Input

- **Recurrent Hidden State**: The hidden state is updated at each time step based on the current input and the previous hidden state. This enables the network to "remember" information from earlier time steps in the sequence.

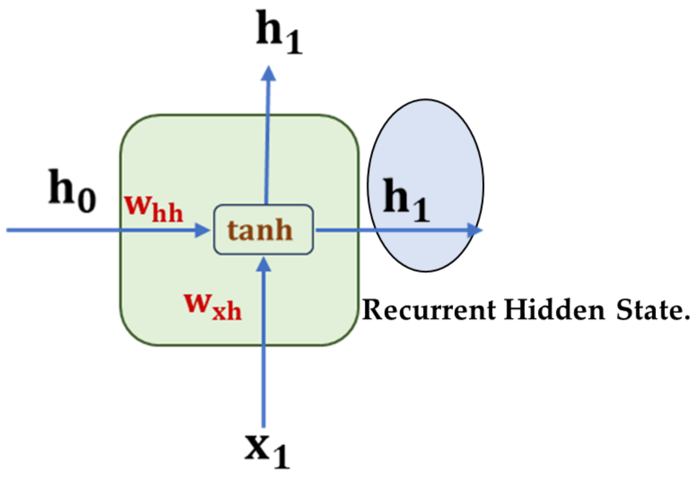


Figure 8: Hidden State

- **Output Layer** : The final output is typically computed from the last hidden state (Many-to-One architecture), but for sequence generation tasks, an output could be produced at each time step (Many-to-Many).

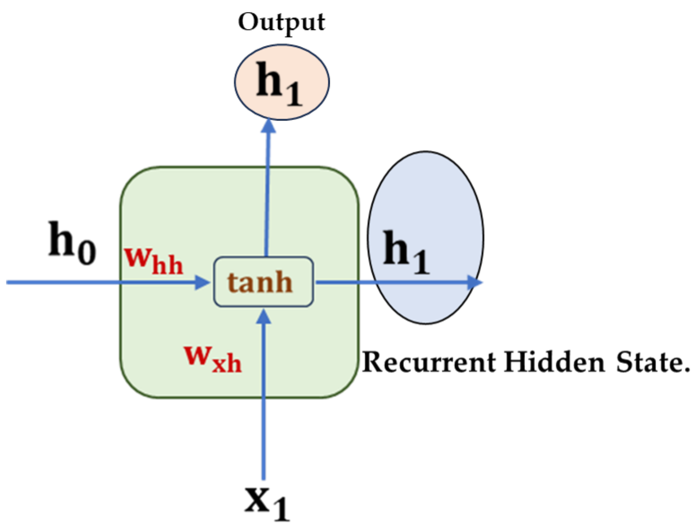


Figure 9: Output of Recurrent Unit.

- **Mathematically:**

$$\mathbf{h_t} = \mathbf{tanh}(\mathbf{w_h} \cdot \mathbf{h_{t-1}} + \mathbf{w_x} \cdot \mathbf{x_t} + \mathbf{b_h})$$

  - $\mathbf{h_t} \rightarrow$ hidden state at time t,
  - $\mathbf{x_t} \rightarrow$ input at time t and
  - **tanh** $\rightarrow$ activation function

  This Requires two matrix product, thus following is what we do in practice:

  - Concatenate previous hidden state with $\mathbf{h_{t-1}}$ and current input $\mathbf{x_t}$. i.e.

$$\mathbf{z_t} = \begin{bmatrix} h_{t-1} \\ x_t \end{bmatrix}$$

  - Then, we apply a single weight matrix:

$$\mathbf{h_t} = \mathbf{tanh}(\mathbf{W} \cdot \mathbf{z_t})$$

    * **W** is a single weight matrix with dimension:

$$\mathbf{W} \in \mathbb{R}^{\texttt{hidden\_dim} \times (\texttt{hidden\_dim + input\_dim})}$$

    * `hidden_dim` $\rightarrow$ number of neurons in RNN Layer.
    * `input_dim` $\rightarrow$ embedded vector representation of input data.

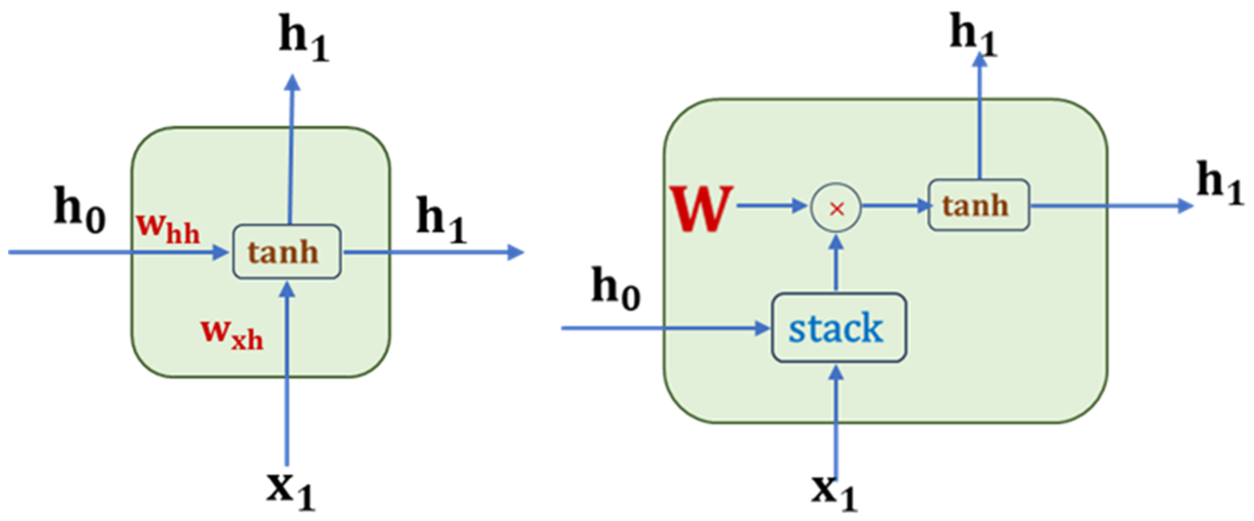


Figure 10: An "vanilla" RNN.

2.2 RNN Challenges:

While RNNs are powerful, they come with certain challenges:

- **Vanishing Gradient Problem:** As the length of the sequences increases, gradients may become very small during backpropagation, making it hard for the model to learn long-term dependencies.
- **Exploding Gradient Problem:**Conversely, gradients may become too large, leading to instability during training.

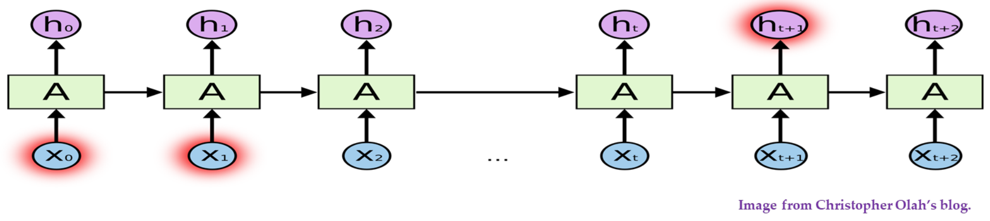


Figure 11: Caption

3. An LSTM {Long Short Term Memory}.

An LSTM is a special type of RNN designed to address the vanishing gradient problem and improve the ability to capture long-range dependencies in sequences. It was introduced by Sepp Hochreiter and Jürgen Schmidhuber in 1997. LSTMs solve the problem of long-term dependencies by using gates to

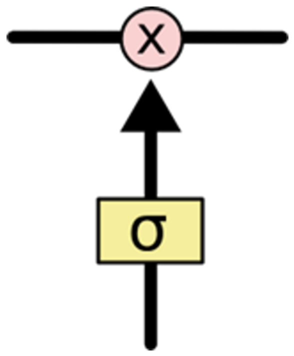


Figure 12: A Multiplicative Gate Mechanism.

control the flow of information. These gates regulate which information gets retained, updated, or forgotten, allowing the LSTM to "remember" information over long sequences without being overwhelmed by irrelevant data.

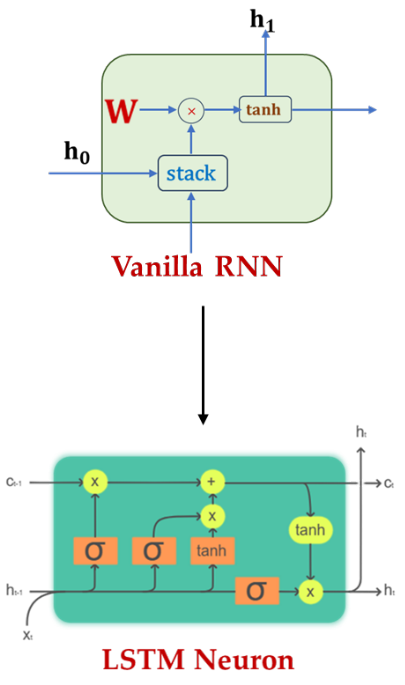


Figure 13: Caption

3.1 LSTM Architecture:

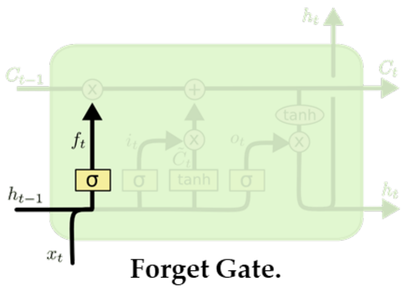
An LSTM unit consists of the following components:

Forget Gate

Decides which information from the previous cell state should be forgotten. This is done by applying a sigmoid function to the previous hidden state and the current input.

$$\mathbf{f_t} = \sigma\left(\mathbf{W_f} \cdot \begin{bmatrix} h_{t-1} \\ x_t \end{bmatrix} + \mathbf{b_f}\right)$$

The output of this gate is a number between 0 and 1, which multiplies the previous cell state to retain or discard the information.



Forget Gate.

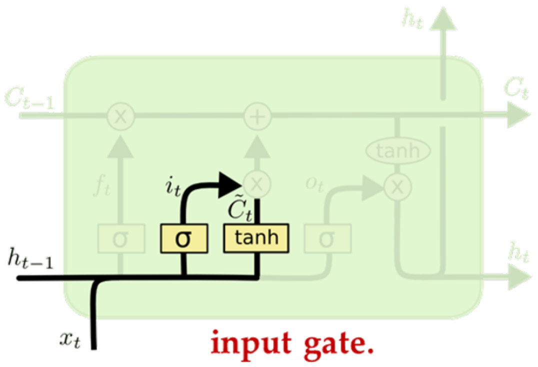
Input Gate

Decides which values from the current input will update the cell state. This is also done using a sigmoid activation function.

$$\mathbf{i_t} = \sigma\left(\mathbf{W_i} \cdot \begin{bmatrix} h_{t-1} \\ x_t \end{bmatrix} + \mathbf{b_i}\right)$$

Along with this, a tanh function generates new candidate values for the cell state:

$$\mathbf{\tilde{C}_t} = \tanh\left(\mathbf{W_C} \cdot \begin{bmatrix} h_{t-1} \\ x_t \end{bmatrix} + \mathbf{b_C}\right)$$



input gate.

Cell State Update

The cell state is updated by combining the old state (selectively forgotten) with new candidate values:

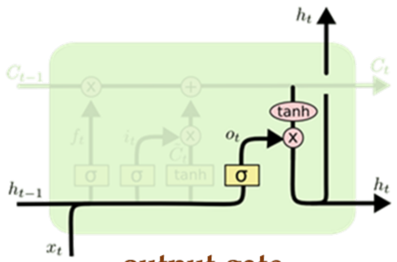
$$\mathbf{C_t} = \mathbf{f_t} * \mathbf{C_{t-1}} + \mathbf{i_t} * \tilde{\mathbf{C}}_\mathbf{t}$$

Output Gate

Decides which parts of the cell state should be output. The final output is based on the updated cell state:

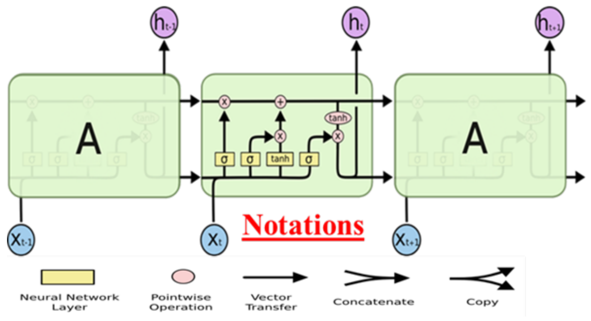
$$\mathbf{o_t} = \sigma\left(\mathbf{W_o} \cdot \begin{bmatrix} h_{t-1} \\ x_t \end{bmatrix} + \mathbf{b_o}\right)$$

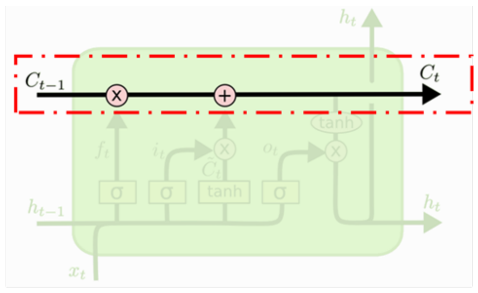
$$\mathbf{h_t} = \mathbf{o_t} * \tanh(\mathbf{C_t})$$



output gate.

To conclude, The core idea behind LSTM is the Cell State $\mathbf{C_t}$. This allows LSTM to retain long - term dependencies over sequences.





Why Use LSTM in Sentiment Analysis?

LSTMs are especially useful in sentiment analysis because they can capture long-range dependencies in text. Sentiment is often spread across different parts of a text and may depend on context. For example, the sentiment of a tweet could be influenced by a sentiment word earlier in the text, and LSTMs are capable of learning these long-range dependencies.

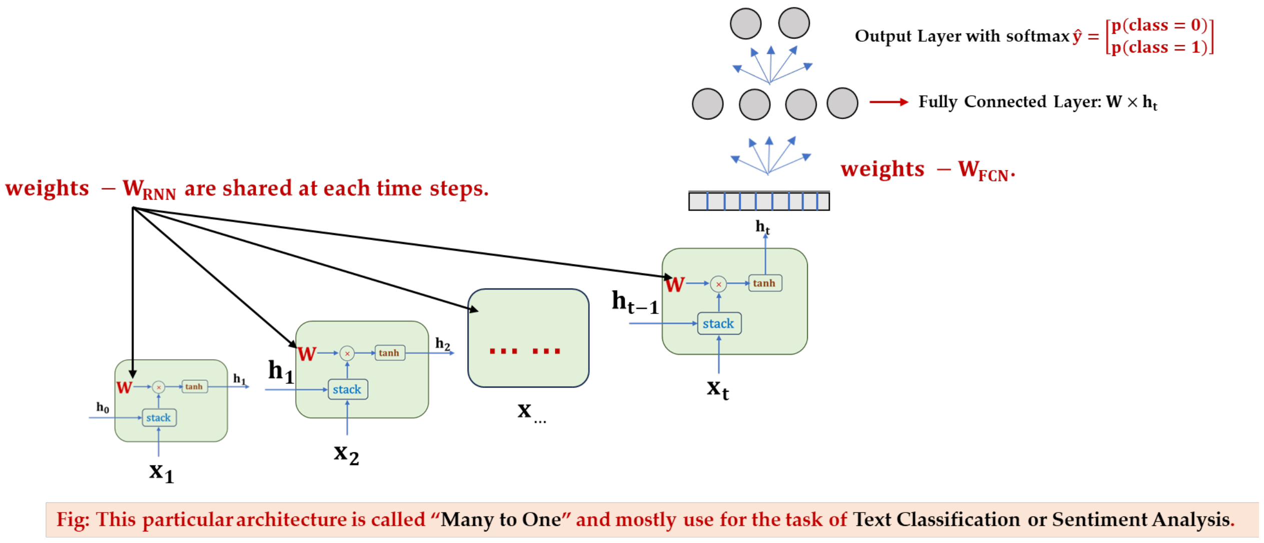

Output Layer with softmax $\hat{y} = \begin{bmatrix} p(class = 0) \\ p(class = 1) \end{bmatrix}$
Fully Connected Layer: $W \times h_t$
weights – W_{FCN}.
weights – W_{RNN} are shared at each time steps.
h_t
h_{t-1}
x_t
h_1
h_2
x_2
$x_{...}$
h_0
x_1
W
stack
tanh
... ...
Fig: This particular architecture is called "Many to One" and mostly use for the task of Text Classification or Sentiment Analysis.

Figure 14: Many-to-One Architecture for Sentiment Classification.

Here's a sample code to build and compare models using Vanilla RNN and LSTM using Keras. Both models use the same input preprocessing and differ only in the recurrent layer.

Vanilla RNN Model (Simple RNN)

Vanilla RNN

```python
from keras.models import Sequential
from keras.layers import Embedding, SimpleRNN, Dense
# RNN model
rnn_model = Sequential()
rnn_model.add(Embedding(input_dim=10000, output_dim=128, input_length=50))
rnn_model.add(SimpleRNN(64)) # Vanilla RNN layer
rnn_model.add(Dense(3, activation='softmax')) # For 3 sentiment classes
rnn_model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
rnn_model.summary()
```

LSTM Model

LSTM

```python
from keras.models import Sequential
from keras.layers import Embedding, LSTM, Dense
# LSTM model
lstm_model = Sequential()
lstm_model.add(Embedding(input_dim=10000, output_dim=128, input_length=50))
lstm_model.add(LSTM(64)) # LSTM layer
lstm_model.add(Dense(3, activation='softmax'))
lstm_model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
lstm_model.summary()
```

The above code:

- input_dim=10000: Size of the vocabulary.

- output_dim=128: Size of the word embedding vector.
- input_length=50: Length of each input sequence (padded).
- SimpleRNN vs LSTM: Both return a fixed-size output for many-to-one classification.
- Dense(2, activation='softmax'): Used for multi-class sentiment (positive, neutral, negative).

3.4 Model Training and Validation:

- The dataset is split into training and validation sets.
- Models are trained using the training set and evaluated on the validation set.
- Training metrics such as loss and accuracy are recorded for visualization.
- **Visualization of Training Behavior:**
  - Plots the training and validation loss curves to help diagnose overfitting and underfitting.
  - Uses Matplotlib for plotting performance metrics over epochs.

Training the Model.

```
# Common for both RNN and LSTM
from keras.callbacks import ModelCheckpoint, EarlyStopping
model.compile(
    loss='categorical_crossentropy', # or 'sparse_categorical_crossentropy' if labels are integers
    optimizer='adam',
    metrics=['accuracy']
)
# Save the best model
checkpoint = ModelCheckpoint('best_model.h5', monitor='val_accuracy', save_best_only=True, mode='max
    ', verbose=1)

# Stop training early if validation loss doesn't improve
early_stop = EarlyStopping(monitor='val_loss', patience=3, restore_best_weights=True, verbose=1)
history = model.fit(
    X_train_pad, y_train,
    epochs=10,
    batch_size=64,
    validation_data=(X_val_pad, y_val),
    callbacks=[checkpoint, early_stop]
)
```

3.5 Results and Evaluation:

Now that the model is trained, we should evaluate the performance. This includes examining the accuracy, confusion matrix, and other relevant metrics.

Final Evaluation.

```
from sklearn.metrics import accuracy_score, confusion_matrix, classification_report
# Predictions on the validation set
y_pred = model.predict(X_val)
y_pred_classes = np.argmax(y_pred, axis=1) # Convert probabilities to class labels
# Accuracy score
accuracy = accuracy_score(y_val, y_pred_classes)
print(f"Accuracy: {accuracy:.4f}")
# Confusion Matrix
cm = confusion_matrix(y_val, y_pred_classes)
print("Confusion Matrix:")
print(cm)
# Classification Report
```

```
cr = classification_report(y_val, y_pred_classes, target_names=label_encoder.classes_)
print("Classification Report:")
print(cr)
```

4 A Simple GUI for your Application:

We will use Tkinter to build a simple user interface that allows users to input a tweet, and the system will predict the sentiment based on the trained model. This is a sample implementation and might be different as per your requirements also the use of tkinter is not mandatory, feel free to use any api like gradio, streamlit etc. or build with flask.

GUI Code (Tkinter)

```
import tkinter as tk
from tkinter import messagebox
# Function to predict sentiment based on user input
def predict_sentiment_gui():
    user_input = entry.get() # Get the text input from the user
    if user_input.strip() == "":
        messagebox.showwarning("Input Error", "Please enter a tweet for prediction!")
        return

    # Predict sentiment using the trained model
    prediction = predict_sentiment(user_input)

    # Display the result in the label
    result_label.config(text=f"Predicted Sentiment: {prediction}")
# Create the main window
root = tk.Tk()
root.title("Twitter Sentiment Predictor")
# Label for instructions
instruction_label = tk.Label(root, text="Enter a tweet to predict its sentiment:", font=("Arial",
    14))
instruction_label.pack(pady=10)
# Entry box for user to input a tweet
entry = tk.Entry(root, width=50, font=("Arial", 12))
entry.pack(pady=10)
# Predict button to trigger sentiment prediction
predict_button = tk.Button(root, text="Predict Sentiment", font=("Arial", 12), command=
    predict_sentiment_gui)
predict_button.pack(pady=10)
# Label to display the predicted sentiment
result_label = tk.Label(root, text="", font=("Arial", 14), fg="blue")
result_label.pack(pady=10)
# Run the Tkinter main loop
root.mainloop()
```

How It Works:

- User Input: The user enters a tweet into the input field.
- Prediction: Once the user clicks the "Predict Sentiment" button, the predict_sentiment_gui function is called, which:

  - Cleans and tokenizes the text.
  - Passes the tokenized text to the trained model
  - Displays the predicted sentiment (positive, neutral, negative) in the label below the button.

- Feedback: If the input is empty, the system will show a warning message asking the user to input a tweet.

5 Tasks:

1. Data Preprocessing & Cleaning

- Load the dataset using Pandas.
- Clean the text by:
  - Lowercasing
  - Removing URLs, mentions (`@user`), hashtags (`#`), numbers, and special characters
  - Handling contractions (e.g., "don't" → "do not")
  - Removing stopwords and lemmatizing words
- Visualize the cleaned data (e.g., word cloud, most frequent words).

2. Tokenization & Padding

- Split the dataset into **80% training** and **20% testing** using `train_test_split`.
- Use **Keras Tokenizer** to convert text into sequences.
- Apply **padding** to ensure uniform sequence lengths (use **percentile-based padding**).

3. Model Building

Implement **two models**:

1. **Simple RNN**
2. **LSTM**

Each model should include:

- **Embedding Layer** (`input_dim=vocab_size, output_dim=128`)
- **Recurrent Layer** (`SimpleRNN` or `LSTM` with 64 units)
- **Dense Layer** (`sigmoid` activation for binary classification)

4. Training & Evaluation

- Compile models with:
  - **Loss:** `binary_crossentropy`
  - **Optimizer:** `adam`
  - **Metrics:** `accuracy`
- Train for **10 epochs** with early stopping.
- Evaluate using:

  - **Accuracy**
  - **Confusion Matrix**
  - **Classification Report (Precision, Recall, F1-Score)**

5. Visualization

- Plot **training vs. validation loss & accuracy** over epochs.
- Compare RNN vs. LSTM performance.

6. (Optional) GUI for Real-Time Prediction

- Use **Tkinter, Gradio, or Streamlit** to create a simple interface.
- Allow users to input a tweet and get a sentiment prediction.

Expected Deliverables

1. **Jupyter Notebook / Python Script** with complete implementation.
2. **Model Performance Report** (Accuracy, Confusion Matrix, F1-Score).
3. **Visualizations** (Training Curves, Word Cloud).
4. **(Optional) GUI Demo** (Screenshot/Video).

Good Luck! ★