Day 18 of Training at Ansh Info Tech

Topics Covered

- **RNN**
    - **LSTM**
- **NLP**
- **Sequence Modeling**
- **Time Series Analysis**
- **Text Preprocessing and Word Embeddings**
- **Language Modeling**
- **Sentiment Analysis**
- **Named Entity Recognition**
- **Part of Speech Recognition**
- **Machine Translation**
- **Tokenization**
- **Stop Words**
- **Lemmatization**
- **Stemming**

Summary

RNN

Recurrent Neural Networks (RNNs) are a type of neural network designed for sequence data. They use loops to allow information to persist across time steps, making them suitable for tasks involving sequential data.

LSTM

Long Short-Term Memory (LSTM) is a type of RNN architecture that addresses the vanishing gradient problem. It is capable of learning long-term dependencies and is widely used in various sequential tasks.

NLP

Natural Language Processing (NLP) involves the interaction between computers and human language. It encompasses a variety of tasks such as text preprocessing, tokenization, and understanding linguistic structures.

Sequence Modeling

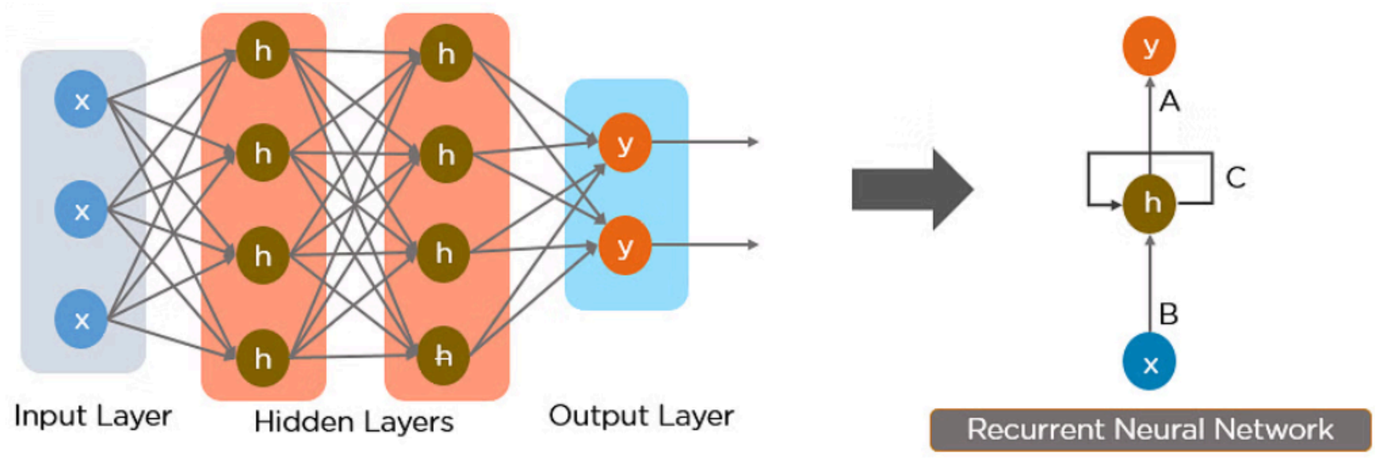
Sequence modeling involves predicting the next item in a sequence, which can be applied to various fields such as time series

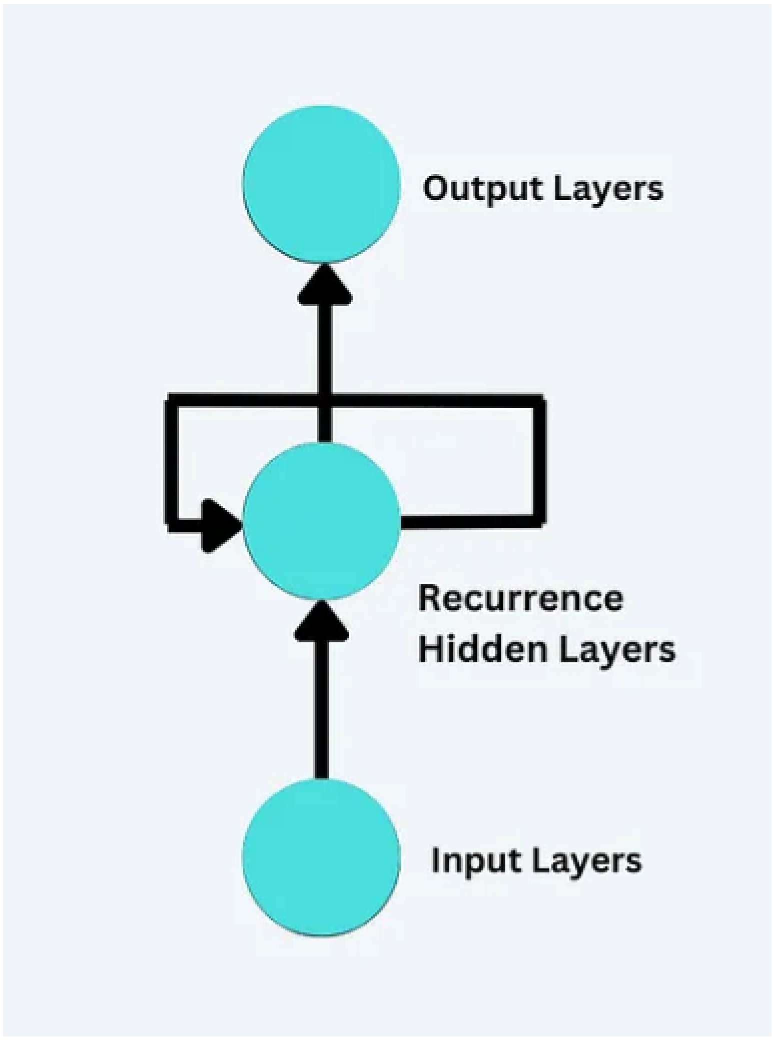
```
Start coding or generate with AI.
```

Introduction to Recurrent Neural Network

Recurrent Neural Network also known as (RNN) that works better than a simple neural network when data is sequential like Time-Series data and text data.

RNNs are called recurrent because they perform the same task for every element of a sequence, with the output being depended on the previous computations. Another way to think about RNNs is that they have a "memory" which captures information about what has been calculated so far.





Basic RNN Architecture A basic RNN consists of the following components:

1. Input Layer The input layer takes the sequential data (e.g., a series of words or characters) and feeds it into the network.

Example: If you are processing a sentence, each word or character in the sentence will be fed into the RNN one at a time.

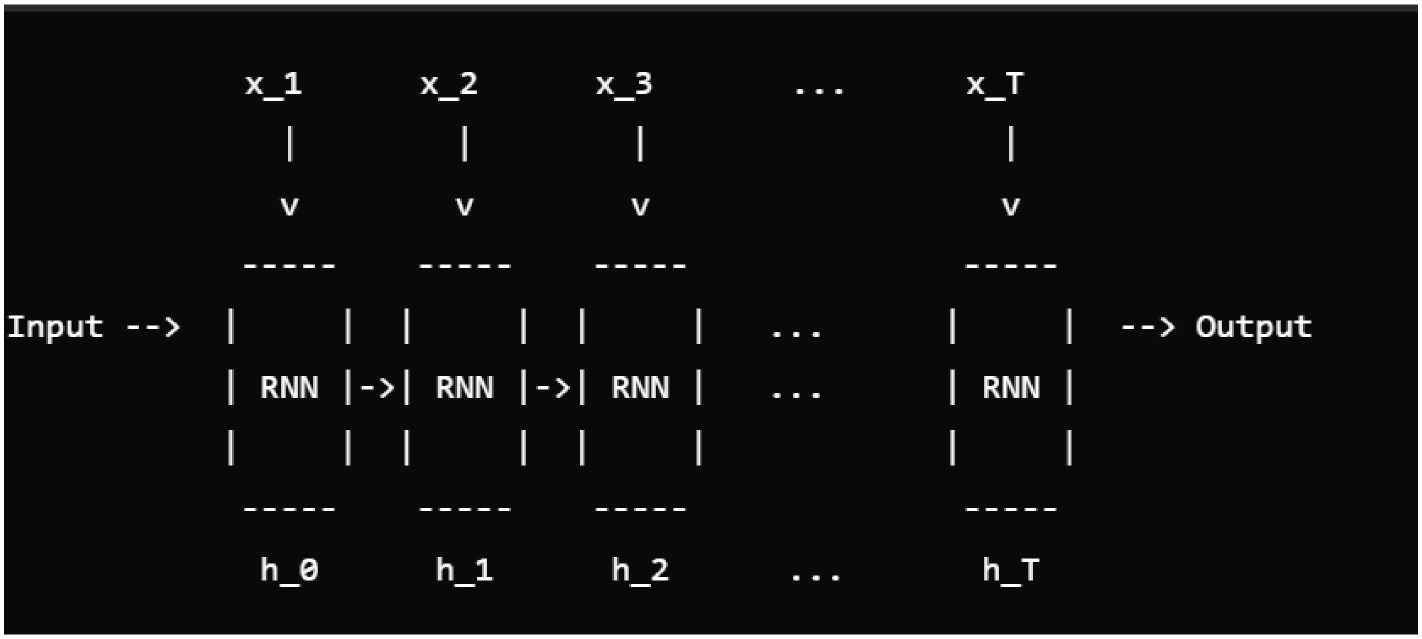
2. Hidden Layer The hidden layer processes each input in the sequence and maintains a hidden state, which is updated at each time step.

Hidden State: A set of neurons that store information about previous inputs in the sequence. The hidden state is updated using the current input and the previous hidden state.

Activation Function: Usually, a non-linear function such as tanh or ReLU is applied to compute the hidden state.

3. Output Layer The output layer produces the output at each time step, based on the current hidden state.

Output: The output can be a probability distribution over the next possible elements in the sequence, or a predicted value.



```
           x_1       x_2       x_3        ...       x_T
            |         |         |                    |
            v         v         v                    v
          -----     -----     -----                -----
Input -->  |     |  |     |  |     |    ...      |     |   --> Output
           | RNN |->| RNN |->| RNN |    ...      | RNN |
           |     |  |     |  |     |             |     |
          -----     -----     -----                -----
            h_0       h_1       h_2       ...       h_T
```

- $x_1, x_2, x_3, ..., x_T$ are the inputs at each time step.
- $h_0, h_1, h_2, ..., h_T$ are the hidden states.
- Each RNN block processes the current input and the previous hidden state to produce the next hidden state and output.

TYPE OF RNN

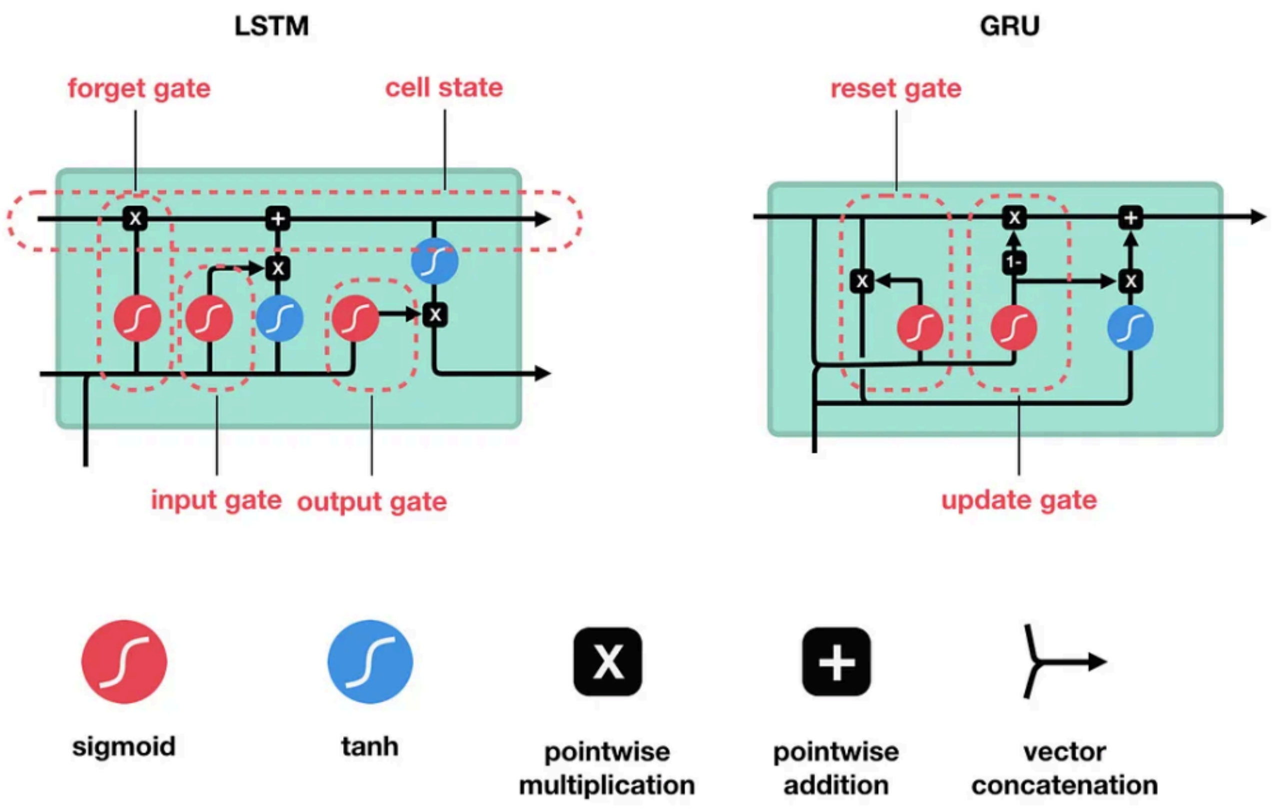
1.Long Short-Term Memory (LSTM) Networks:

LSTMs are a type of RNN designed to handle long-term dependencies. They introduce a memory cell that can maintain its state over time, and gates that control the flow of information.

input gate decides which information to store in the memory cell. It is trained to open when the input is important and close when it is not.

forget gate decides which information to discard from the memory cell. It is trained to open when the information is no longer important and close when it is.

output gate is responsible for deciding which information to use for the output of the LSTM. It is trained to open when the information is important and close when it is not.



2.GRU

What is a Gated Recurrent Unit (GRU)?

A. GRU is a type of recurrent neural network (RNN) that uses gating mechanisms to process sequential data. It's to remember long-term dependencies and forget irrelevant information

GRUs are similar to Long Short-Term Memory (LSTM) networks but have a simpler structure.

GRUs require less memory.

1. Update Gate

Controls how much of the previous state needs to be passed to the current state. Helps the model to retain the useful information from the past.

2. Reset Gate

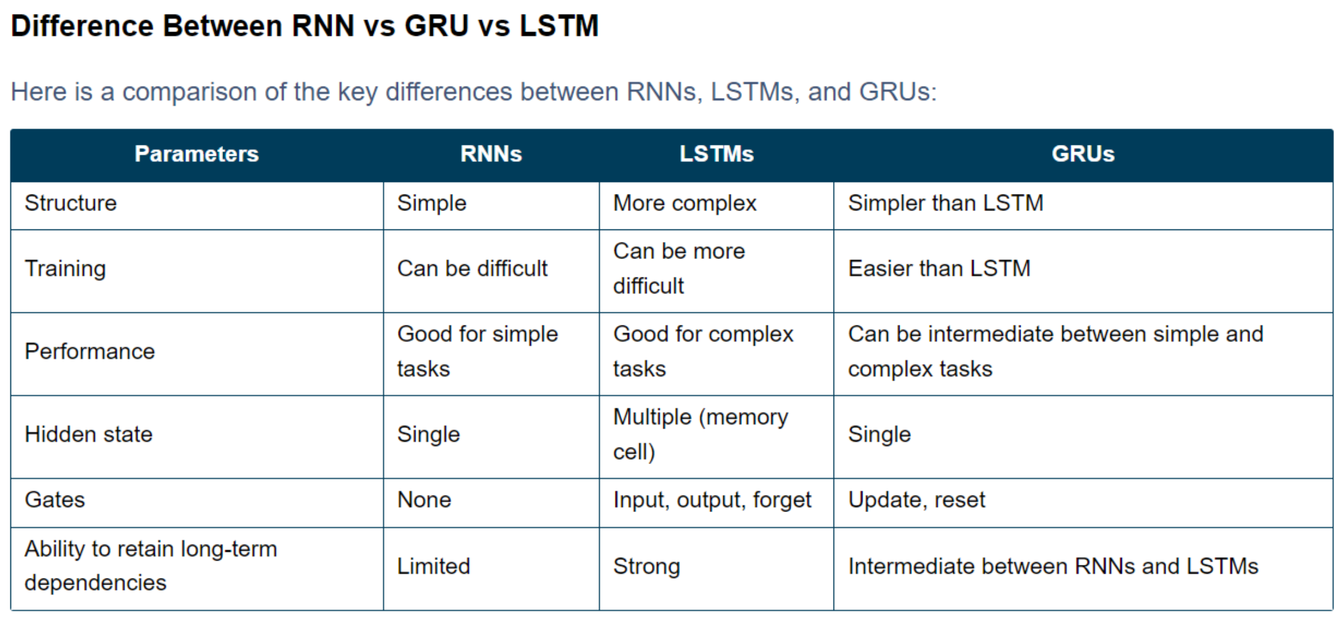
Determines how much of the past information to forget. Helps the model to reset the state when necessary.

3. Current Memory Content

Creates a new candidate state that could replace the previous state. Combines the reset gate and the input.

4. Final Memory at Current Time Step

The final hidden state for the current time step, which is a combination of the previous hidden state and the candidate state, weighted by the update gate.



Difference Between RNN vs GRU vs LSTM

Here is a comparison of the key differences between RNNs, LSTMs, and GRUs:

Parameters	RNNs	LSTMs	GRUs
Structure	Simple	More complex	Simpler than LSTM
Training	Can be difficult	Can be more difficult	Easier than LSTM
Performance	Good for simple tasks	Good for complex tasks	Can be intermediate between simple and complex tasks
Hidden state	Single	Multiple (memory cell)	Single
Gates	None	Input, output, forget	Update, reset
Ability to retain long-term dependencies	Limited	Strong	Intermediate between RNNs and LSTMs

RNNs: Used for simple sequence tasks but limited by vanishing gradients.

LSTMs: Preferred for tasks requiring long-term dependencies.

GRUs: Used for efficient training while maintaining good performance.

Sequence Modeling and Time Series Analysis

Sequence modeling involves predicting future elements of a sequence based on past elements. Common applications include language modeling, machine translation, and time series forecasting.

Time Series Forecasting:

Predicting future values based on past data points. Applications: Stock price prediction, weather forecasting, demand forecasting.

Application of RNN, LSTM, GRU

1. Natural Language Processing (NLP)

Text Generation

Machine Translation

2. Time Series Forecasting

Stock Price Prediction

Weather Forecasting

Demand Forecasting

3. Speech Recognition

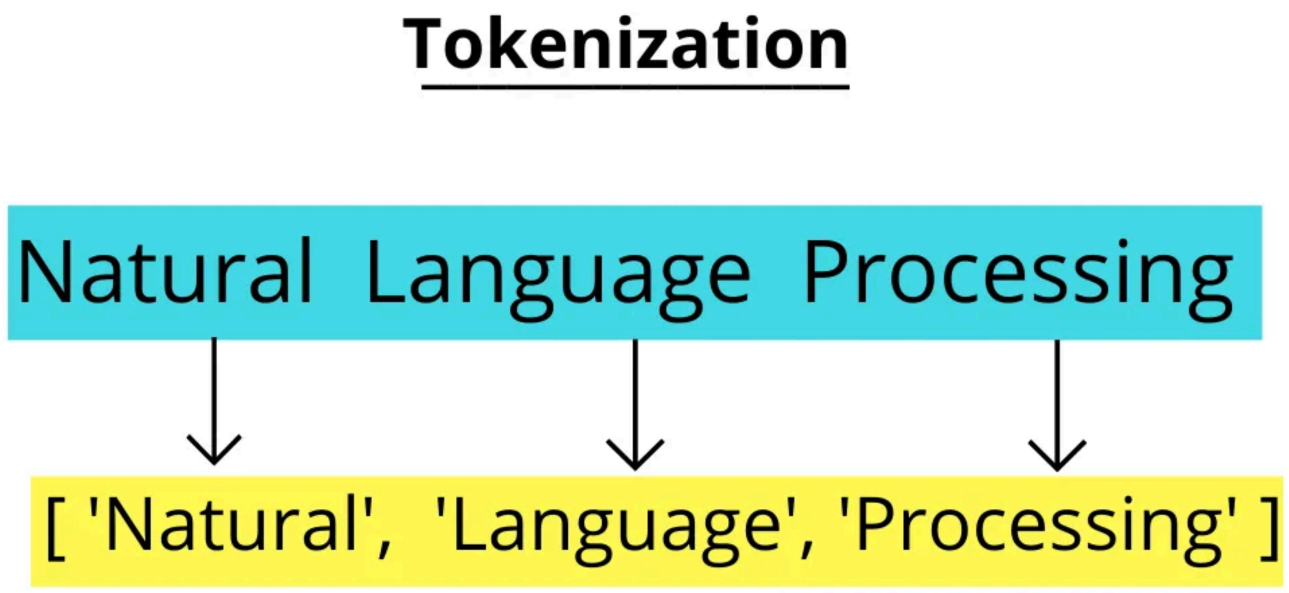
Audio Transcription

Text Preprocessing and Word Embeddings

Text Preprocessing:

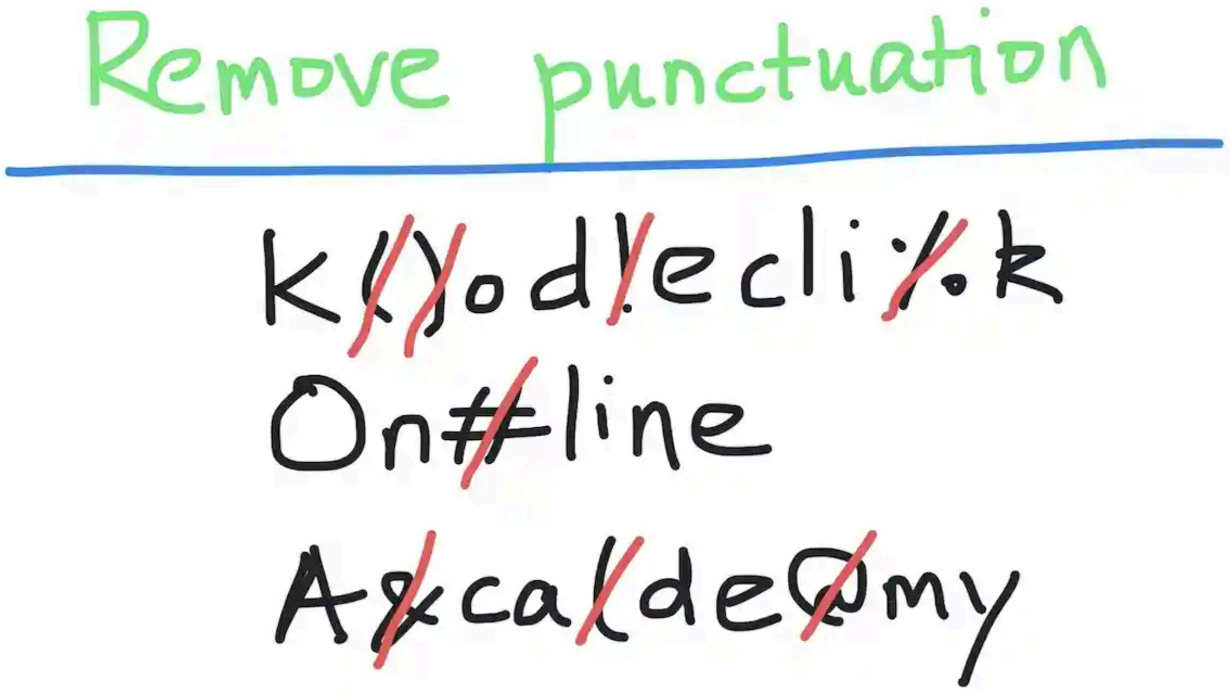
It involves several steps to clean and prepare raw text data for analysis.

1.Tokenization: Splitting text into individual words or tokens.


Tokenization
Natural Language Processing
['Natural', 'Language', 'Processing']

2.Lowercasing: Converting all text to lowercase to ensure uniformity.

3. Removing Punctuation: Eliminating punctuation marks.



4.Removing Stop Words: Removing common words that may not contribute significantly to the meaning (e.g., "and," "the").

5.Stemming and Lemmatization: Reducing words to their root form (e.g., "running" to "run").

6.Removing Numbers: Depending on the context, numbers may be removed.

7.Handling Special Characters: Removing or transforming special characters (e.g., emojis, HTML tags).

Word Embeddings:

Word embeddings words that capture their meanings and relationships.

1.Word2vec is a technique in natural language processing (NLP) for obtaining vector representations of words.

2. FastText Extends Word2Vec by considering subword information

FastText represents each word as a bag of character n-grams in addition to the whole word itself. This means that the word “apple” is represented by the word itself and its constituent n-grams like “ap”, “pp”, “pl”, “le”, etc. This approach helps capture the meanings of shorter words and affords a better understanding of suffixes and prefixes.

Language Modeling and Sentiment Analysis

Language Modeling: Language models predict the next word in a sequence, based on the previous words. Key applications include:

1.Autocomplete: Predicting the next word as the user types.

2.Text Generation: Generating coherent text based on a given prompt.

Popular models:

RNNs: Basic sequence models.

LSTMs and **GRUs**: Handle long-term dependencies.

Transformers: State-of-the-art models like GPT-3/4 and BERT for powerful language understanding and generation.

Sentiment Analysis:

Sentiment analysis determines the sentiment (positive, negative, neutral) expressed in text. Techniques include:

Traditional Machine Learning: Using algorithms like Naive Bayes, SVM, with features such as TF-IDF vectors.

Deep Learning: Using RNNs, LSTMs, GRUs, or transformers (e.g., BERT) to capture contextual information and sentiment.

Named Entity Recognition (NER) and Part-of-Speech (POS) Tagging

NER helps us to identify and classify named entities in text, such as people, organizations, locations, and dates

POS that involves assigning a grammatical category or part-of-speech label (such as noun, verb, adjective, etc.) to each word in a sentence.

Sequence-to-Sequence Models and Machine Translation

Sequence-to-Sequence Models: Seq2Seq models convert an input sequence into an output sequence, commonly used in translation, summarization, and chatbots.

Machine Translation: Machine translation translates text from one language to another.

Sentiment Analysis with an LSTM

```
import numpy as np
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, LSTM, Dense
```

```
# Sample data (text and corresponding labels)
texts = ["I love this movie", "I hate this movie", "This movie is amazing", "This movie is t
labels = [1, 0, 1, 0]  # 1 = Positive, 0 = Negative


# Text preprocessing
tokenizer = Tokenizer(num_words=100)  # Consider the top 100 words
tokenizer.fit_on_texts(texts)
sequences = tokenizer.texts_to_sequences(texts)


# Pad sequences to ensure equal length
data = pad_sequences(sequences, maxlen=5)


# Convert labels to numpy array
labels = np.array(labels) #labels is an array of 0s and 1s indicating negative and positive


# Build a simple LSTM model
model = Sequential()
model.add(Embedding(input_dim=100, output_dim=8, input_length=5))  # Embedding layer
# input_dim=100: This specifies the size of the vocabulary. We set it to 100, meaning it wil
# output_dim=8: This specifies the size of the dense vectors (embeddings). Each word will be
# input_length=5: This specifies the length of the input sequences.
model.add(LSTM(8))  # LSTM layer with 8 units
model.add(Dense(1, activation='sigmoid'))  # Output layer
#Dense Layer: Outputs a single value between 0 and 1
#1: This specifies that there is a single output unit


# Compile the model
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
#adam Adjusts model parameters to minimize the loss.
#Loss Function ('binary_crossentropy'): Measures the difference between actual labels and pr



# Train the model
model.fit(data, labels, epochs=5, batch_size=2)
#epochs: An epoch is one complete pass through the entire training dataset
#batch_size=2: The model will update its weights after every 2 samples.
```

```
Epoch 1/5
15/15 [==============================] - 2s 5ms/step - loss: 0.6931 - accuracy: 0.5517
Epoch 2/5
15/15 [==============================] - 0s 5ms/step - loss: 0.6913 - accuracy: 0.5862
Epoch 3/5
15/15 [==============================] - 0s 5ms/step - loss: 0.6897 - accuracy: 0.6897
Epoch 4/5
15/15 [==============================] - 0s 5ms/step - loss: 0.6881 - accuracy: 0.6897
Epoch 5/5
```

```
15/15 [==============================] - 0s 5ms/step - loss: 0.6855 - accuracy: 0.8966
<keras.src.callbacks.History at 0x7a233b7b7850>
```

```
# Evaluate the model
loss, accuracy = model.evaluate(data, labels)
print(f'Loss: {loss}')
print(f'Accuracy: {accuracy}')
```

```
1/1 [==============================] - 1s 542ms/step - loss: 0.6928 - accuracy: 0.5000
Loss: 0.6927748918533325
Accuracy: 0.5
```

```
# Make a prediction
sample_text = ["This movie is amazing"]
sample_seq = tokenizer.texts_to_sequences(sample_text)
sample_pad = pad_sequences(sample_seq, maxlen=5)
prediction = model.predict(sample_pad)
print(f'Prediction: {"Positive" if prediction[0] > 0.5 else "Negative"}')
```

```
1/1 [==============================] - 0s 21ms/step
Prediction: Positive
```

Start coding or generate with AI.

1. Tokenization

Tokenization is the process of breaking text into individual words or sentences.

Explanation:

Downloading 'punkt': nltk.download('punkt') downloads the Punkt tokenizer models, which are pre-trained models for tokenizing text.

Tokenizing Text: word_tokenize(text) splits the text into individual words and punctuation marks.

```
import nltk
nltk.download('punkt')
from nltk.tokenize import word_tokenize

text = "Hello world! Welcome to NLP."
tokens = word_tokenize(text)
print(tokens)
```

```
['Hello', 'world', '!', 'Welcome', 'to', 'NLP', '.']
[nltk_data] Downloading package punkt to /root/nltk_data...
[nltk_data]   Package punkt is already up-to-date!
```

2. Removing Stop Words

Stop words are common words that are often removed from text data.

```
import nltk
from nltk.corpus import stopwords
nltk.download('stopwords')

stop_words = set(stopwords.words('english'))
words = word_tokenize("This is a simple NLP example.")
filtered_words = [word for word in words if word.lower() not in stop_words]
print(filtered_words)
```

```
['simple', 'NLP', 'example', '.']
[nltk_data] Downloading package stopwords to /root/nltk_data...
[nltk_data]   Unzipping corpora/stopwords.zip.
```

3. Stemming

Stemming is the process of reducing words to their base or root form.

```
from nltk.stem import PorterStemmer

stemmer = PorterStemmer()
words = ["running", "jumps", "easily", "fairly"]
stemmed_words = [stemmer.stem(word) for word in words]
print(stemmed_words)
```

```
['run', 'jump', 'easili', 'fairli']
```

4. Lemmatization

Lemmatization is the process of reducing words to their base or dictionary form.

```
from nltk.stem import WordNetLemmatizer
nltk.download('wordnet')

lemmatizer = WordNetLemmatizer()
words = ["running", "jumps", "easily", "fairly"]
lemmatized_words = [lemmatizer.lemmatize(word) for word in words]
print(lemmatized_words)
```

```
[nltk_data] Downloading package wordnet to /root/nltk_data...
['running', 'jump', 'easily', 'fairly']
```

5. Part-of-Speech Tagging

POS tagging assigns parts of speech to each word in a sentence.

DT: Determiner (e.g., "the", "a") JJ: Adjective (e.g., "quick", "lazy") NN: Noun, singular or mass (e.g., "fox", "dog") VBZ: Verb, 3rd person singular present (e.g., "jumps") IN: Preposition or subordinating conjunction (e.g., "over") .: Punctuation mark (e.g., ".")

```
nltk.download('averaged_perceptron_tagger')

sentence = "The quick brown fox jumps over the lazy dog."
pos_tags = nltk.pos_tag(word_tokenize(sentence))
print(pos_tags)
```

```
[nltk_data] Downloading package averaged_perceptron_tagger to
[nltk_data]     /root/nltk_data...
[('The', 'DT'), ('quick', 'JJ'), ('brown', 'NN'), ('fox', 'NN'), ('jumps', 'VBZ'), ('ove
[nltk_data]   Unzipping taggers/averaged_perceptron_tagger.zip.
```

6. Named Entity Recognition

NER identifies named entities in text.

```
#This imports the spaCy library and loads the English language model ("en_core_web_sm").

import spacy
nlp = spacy.load("en_core_web_sm")

text = "Apple is looking at buying U.K. startup for $1 billion"
doc = nlp(text)
#Iterates over the entities recognized in the processed document (doc) and prints each entit
for ent in doc.ents:
    print(ent.text, ent.label_)
```

```
Apple ORG
U.K. GPE
$1 billion MONEY
```

Output Explanation The output shows the recognized entities and their corresponding labels:

Apple: Recognized as an organization (ORG). U.K.: Recognized as a geopolitical entity (GPE). $1 billion: Recognized as a monetary value (MONEY).

```
# 7. Sentence Tokenization
# Sentence tokenization splits text into sentences.

from nltk.tokenize import sent_tokenize

text = "Hello world! How are you today? Welcome to NLP."
sentences = sent_tokenize(text)
print(sentences)
```

```
['Hello world!', 'How are you today?', 'Welcome to NLP.']
```

```
# 8. Text Normalization
# Text normalization converts text to a standard format.

import re

text = "This is an example text with punctuation, numbers 123 and UPPERCASE letters."
normalized_text = re.sub(r'\d+', '', text).lower()
print(normalized_text)
```

```
this is an example text with punctuation, numbers  and uppercase letters.
```

re.sub(r'\d+', '', text): Uses the re.sub() function to substitute (replace) all sequences of digits (\d+) in the text with an empty string '', effectively removing the digits. .lower(): Converts the resulting text to lowercase.

Text normalization is an important preprocessing step in natural language processing tasks. While this example focuses on removing digits and converting text to lowercase

```
#9. Spell Checking
#Spell checking corrects spelling errors in text.

from textblob import TextBlob
```