**Sentiment Analysis in a Network Graph Using RNN**

Sentiment analysis involves determining the sentiment or emotion behind a piece of text, such as whether it is positive, negative, or neutral. When combined with a network graph, it is a way to analyze how sentiments are distributed or propagated across nodes (representing entities like users, products, or topics) in a graph structure.

**Key Concepts**

**1. Network Graph**

* A network graph consists of nodes and edges, where nodes represent entities (e.g., people, products, tweets, etc.), and edges represent relationships between these entities (e.g., interactions, friendships, communications, etc.).
* In sentiment analysis, each node or edge can have associated sentiment data, representing the emotional tone of interactions.

**2. Recurrent Neural Networks (RNN)**

* RNNs are a type of neural network used for sequential data processing. Unlike traditional neural networks, RNNs have loops that allow information to persist, making them ideal for handling time-series data or sequences, such as text or communication between nodes in a network.
* In the context of sentiment analysis, RNNs are used to process text data and understand the sentiment expressed in the text.

**Applying Sentiment Analysis to Network Graphs Using RNN**

**Step 1: Data Representation**

* Each node in the network represents an entity (e.g., a user or a product).
* Each edge represents a relationship between two entities (e.g., communication, interaction).
* Each node/edge has text-based data (such as tweets, reviews, or messages) that can be processed for sentiment analysis.
* The network graph structure helps to visualize how sentiment spreads or connects between entities.

**Step 2: Text Preprocessing**

* Text data in the graph (e.g., posts, comments) is preprocessed by cleaning, tokenizing, removing stop words, and normalizing (lowercasing, stemming, or lemmatizing).
* This cleaned text is then ready to be fed into an RNN model for sentiment analysis.

**Step 3: Sentiment Analysis Using RNN**

* Use an RNN (such as an LSTM or GRU) to classify the sentiment of each node’s text data.
* The RNN processes the sequential text data, outputting a sentiment classification for each text input (positive, negative, or neutral).
* Optionally, sentiment scores (e.g., a scale from -1 to 1) can be generated instead of discrete labels.

**Step 4: Sentiment Propagation in the Graph**

* **Propagation across edges**: In a network graph, sentiment may propagate across edges. For instance, if two connected users (nodes) exchange messages, the sentiment of one user’s message could influence the other’s sentiment.
* **Sentiment dynamics**: By analyzing how sentiments propagate across edges in the network, you can determine sentiment trends or understand how certain sentiments spread through the community.
* A few approaches for sentiment propagation:
  + **Diffusion models**: Sentiment from one node can "diffuse" to neighboring nodes based on edge weights (e.g., frequency or strength of interactions).
  + **Message passing**: Nodes pass sentiment data to their neighbors iteratively, adjusting based on the sentiments received.

**Step 5: Analyzing Sentiment Flow**

* After applying the RNN and sentiment analysis, the network graph will have nodes and edges labeled with sentiment data.
* You can visualize how sentiment flows across the network, such as which nodes (users or topics) tend to have positive or negative sentiments.
* **Community sentiment analysis**: By analyzing connected components (subgraphs) of the network, you can observe how specific communities or groups within the network share similar sentiments.

**Example Workflow**

1. **Network Construction**:
   * Nodes: Users in a social network.
   * Edges: Interactions (e.g., comments, messages).
   * Sentiment: Each comment or message between users is analyzed for sentiment.
2. **Text Sentiment Analysis**:
   * Apply an RNN model (like LSTM) to analyze the sentiment of each message/comment between users. Classify them as positive, negative, or neutral.
3. **Propagation**:
   * Model how sentiment spreads between users through their interactions. For instance, if one user expresses a negative sentiment, it might influence connected users (e.g., mutual friends) over time.
4. **Graph Analysis**:
   * Once sentiment has been classified for each node/edge, analyze the graph:
     + Which users or communities have a positive/negative sentiment?
     + How does sentiment change or spread over time?
     + Are there any clusters of users with similar sentiments?

**Applications**

1. **Social Media Monitoring**:
   * Detect sentiment patterns in user conversations (e.g., positive/negative sentiment around a particular event).
   * Track how sentiment propagates in a network (e.g., does a celebrity's tweet affect fans' sentiments?).
2. **Product Feedback Analysis**:
   * Analyze sentiment from product reviews or customer support interactions. Identify how customer sentiment about a product spreads within a community.
3. **Political Sentiment**:
   * Track how political opinions spread through social networks or between politicians and their followers.
4. **Influencer Identification**:
   * Identify nodes (users) that have a significant influence on the sentiment of others in the network. These might be key individuals who can steer the overall sentiment in a network.

**Conclusion**

Sentiment analysis in network graphs using RNNs provides a powerful approach to understanding the flow of emotions, opinions, or sentiments through interconnected entities. RNNs allow for processing sequential data (such as text) efficiently, while the network graph structure helps analyze the relationships and interactions between entities. Combining these techniques can reveal deeper insights into how sentiment propagates and influences different nodes in the system.

**CODE EXPLANATION**

**1. Importing Libraries**

python

CopyEdit

import tensorflow as tf

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Embedding, SimpleRNN, Dense

from tensorflow.keras.datasets import imdb

from tensorflow.keras.preprocessing import sequence

from tensorflow.keras.utils import plot\_model

import matplotlib.pyplot as plt

* tensorflow: This is the core library for building and training neural networks.
* Sequential: A linear stack of layers in the neural network.
* Embedding, SimpleRNN, Dense: These are different layers used in the model:
  + Embedding: Used for turning positive integers into dense vectors of fixed size (usually used in NLP tasks).
  + SimpleRNN: A type of Recurrent Neural Network (RNN) that works well with sequential data.
  + Dense: A fully connected layer used for the final output.
* imdb: A dataset of movie reviews used for sentiment analysis. This is a binary classification dataset (positive/negative sentiment).
* sequence: A utility to handle padding and preprocessing sequences of text (like reviews).
* plot\_model: Used to visualize the structure of the model.
* matplotlib.pyplot: A plotting library used for visualizing the training and validation metrics.

**2. Setting Parameters**

python

CopyEdit

vocab\_size = 5000 # only consider top 5000 words

maxlen = 100 # cut reviews after 100 words

embedding\_size = 32

* vocab\_size: The maximum number of words to consider for the model. This means the model will only use the top 5000 most frequent words from the IMDB dataset.
* maxlen: The maximum number of words in each review. If a review has more than 100 words, it will be truncated; if less, it will be padded.
* embedding\_size: The size of the embedding vectors for words in the dataset.

**3. Loading and Preprocessing Data**

python

CopyEdit

(X\_train, y\_train), (X\_test, y\_test) = imdb.load\_data(num\_words=vocab\_size)

X\_train = sequence.pad\_sequences(X\_train, maxlen=maxlen)

X\_test = sequence.pad\_sequences(X\_test, maxlen=maxlen)

* The imdb.load\_data() function loads the dataset. The dataset consists of 25,000 movie reviews, split into training and testing data.
  + X\_train, X\_test: These are the sequences of words in the reviews (represented as integers based on the word index).
  + y\_train, y\_test: These are the labels for the reviews, where 1 represents positive sentiment and 0 represents negative sentiment.
* The sequence.pad\_sequences() function ensures that all reviews are of the same length (maxlen = 100). Short reviews are padded with zeros at the beginning, and longer reviews are truncated.

**4. Building the RNN Model**

python

CopyEdit

model = Sequential()

model.add(Embedding(vocab\_size, embedding\_size, input\_length=maxlen))

model.add(SimpleRNN(64))

model.add(Dense(1, activation='sigmoid')) # Binary classification (positive/negative)

* Sequential(): Initializes the model as a linear stack of layers.
* Embedding(vocab\_size, embedding\_size, input\_length=maxlen): This layer converts the integer-encoded words into dense vectors of fixed size (embedding\_size). It maps each word index (from 0 to 5000) into a 32-dimensional vector. This layer is essential for handling text data.
* SimpleRNN(64): This is the recurrent layer. It has 64 units and processes the input sequences in a sequential manner, capturing temporal dependencies in the data.
* Dense(1, activation='sigmoid'): This is the final fully connected layer. Since it's a binary classification task (positive or negative sentiment), the output is a single neuron with a sigmoid activation function, producing a probability value between 0 and 1.

**5. Compiling the Model**

python

CopyEdit

model.compile(loss='binary\_crossentropy', optimizer='adam', metrics=['accuracy'])

* loss='binary\_crossentropy': This is the loss function used for binary classification.
* optimizer='adam': The Adam optimizer is used, which is an efficient optimization algorithm for training neural networks.
* metrics=['accuracy']: We track the accuracy metric during training.

**6. Training the Model**

python

CopyEdit

history = model.fit(X\_train, y\_train, epochs=5, batch\_size=64, validation\_data=(X\_test, y\_test))

* The model is trained on the training data (X\_train and y\_train) for 5 epochs with a batch size of 64. This means that the model will look at 64 samples at a time during training and adjust weights accordingly.
* validation\_data=(X\_test, y\_test): The validation data is used to evaluate the model’s performance during training. After each epoch, the model's performance on the test set (X\_test and y\_test) is evaluated.

**7. Visualizing the Network Graph**

python

CopyEdit

plot\_model(model, show\_shapes=True, show\_layer\_names=True)

* This line generates a visual representation of the model architecture. It shows each layer of the network, including the input and output shapes, and the type of layers used (Embedding, SimpleRNN, Dense).

**8. Plotting Accuracy and Loss**

python

CopyEdit

plt.figure(figsize=(12,5))

plt.subplot(1,2,1)

plt.plot(history.history['accuracy'], label='Train Accuracy')

plt.plot(history.history['val\_accuracy'], label='Validation Accuracy')

plt.legend()

plt.title('Accuracy')

plt.subplot(1,2,2)

plt.plot(history.history['loss'], label='Train Loss')

plt.plot(history.history['val\_loss'], label='Validation Loss')

plt.legend()

plt.title('Loss')

plt.tight\_layout()

plt.show()

* This block plots the training and validation accuracy as well as the training and validation loss over the epochs.
  + **Accuracy Plot**: Shows how well the model is performing in terms of classification.
  + **Loss Plot**: Shows how the model’s loss is decreasing over time.
* plt.subplot(1,2,1): This creates two subplots side by side.
* plt.tight\_layout(): Ensures that the plots are spaced correctly and don’t overlap.

**9. Evaluating the Model on Test Data**

python

CopyEdit

test\_loss, test\_acc = model.evaluate(X\_test, y\_test)

print(f"Test Accuracy: {test\_acc:.4f}")

* This evaluates the final trained model on the test set and prints the test accuracy.

**10. Decoding Reviews and Making Predictions**

python

CopyEdit

word\_index = imdb.get\_word\_index()

reverse\_word\_index = {value: key for (key, value) in word\_index.items()}

def decode\_review(encoded\_review):

return ' '.join([reverse\_word\_index.get(i - 3, '?') for i in encoded\_review])

* word\_index: A dictionary mapping words to integers in the dataset.
* reverse\_word\_index: A reverse mapping to get the word back from its corresponding integer index.
* decode\_review(encoded\_review): A function that decodes a review (encoded as integers) back into its original text.

python

CopyEdit

sample\_indices = np.random.randint(0, len(X\_test), 5) # Randomly pick 5 samples

for i in sample\_indices:

sample\_review = X\_test[i].reshape(1, -1)

prediction = model.predict(sample\_review, verbose=0)

sentiment = "Positive " if prediction[0][0] > 0.5 else "Negative"

print("\nReview:")

print(decode\_review(X\_test[i]))

print(f"Predicted Sentiment: {sentiment}")

print(f"Actual Sentiment: {'Positive' if y\_test[i]==1 else 'Negative'}")

* Randomly selects 5 samples from the test set and decodes them.
* The model predicts the sentiment for each review, and it compares the predicted sentiment with the actual sentiment.
* The sentiment is displayed as either "Positive" or "Negative" based on the model’s output.

**Summary of the Graphs**

1. **Accuracy Plot**: This shows how the model’s accuracy improves over training epochs for both training and validation datasets.
2. **Loss Plot**: This shows how the model’s loss decreases over training epochs. A lower loss indicates better performance in classification tasks.

These visualizations help to understand the model's performance throughout the training process and ensure that the model is not overfitting or underfitting.