SENTENCE SIMILARITY

Using Siamese Networks

NLP Project

Introduction

- **What is Sentence Similarity?**
  - Sentence similarity models convert input texts into vectors (embeddings) that capture semantic information and calculate how close (similar) they are between them.
  - Useful for information retrieval and clustering/grouping.
  - Sentence similarity refers to the measurement of how alike or related two sentences are in terms of their meaning, structure, or content. It's a fundamental concept in natural language processing (NLP) and has numerous applications, including information retrieval, question answering, text summarization, and machine translation.

How Do Sentence Similarity Models Work?

- **Embedding Generation:**
  - Obtain sentence embeddings using pretrained language models (e.g., BERT, MiniLM).
  - Example: Use Sentence Transformers library to encode sentences into embeddings.
- **Cosine Similarity Calculation:**
  - Compute cosine similarity between sentence embeddings.
  - Higher cosine similarity values indicate greater similarity.
- **Use Cases:**
  - **Information Retrieval:**
    - Rank documents based on relevance to a query.
    - Search for similar documents.
  - **Semantic Textual Similarity:**
    - Evaluate how similar two texts are.
    - Useful for paraphrase detection and question answering.

What Are Siamese Networks

- Siamese Networks are a type of neural network architecture used for tasks involving similarity measurement, such as image or text comparison. They consist of two identical subnetworks, often referred to as "twins" or "branches", which share the same weights and architecture. These twin networks process two different inputs (e.g., images, sentences) in parallel, aiming to learn representations that can effectively capture the similarity or dissimilarity between them.
- The key advantage of Siamese Networks lies in their ability to learn robust representations that can capture subtle similarities or differences between input pairs, even with limited labeled data. By sharing parameters between the twin networks, they can effectively leverage the data to learn meaningful features and improve generalization.

Steps

- Firstly we install the needed libraries
- After importing from the needed Libraries, We download the dataset and save it so it can be read whenever

```
!pip install tensorflow
!pip install numpy
!pip install matplotlib
!pip install arabic-reshaper
!pip install python-bidi
```

```python
# Download the dataset
url = 'https://raw.githubusercontent.com/SamirMoustafa/nmt-with-attention-for-ar-to-en/master/ara_.txt'
response = requests.get(url)

# Save the data to a temporary file
with open('ara_.txt', 'wb') as f:
    f.write(response.content)

# Read the dataset
with open('ara_.txt', 'r', encoding='utf-8') as file:
    lines = file.readlines()
```

Next We Combine the dataset(English & Arabic)

We access the dataset using for loop and assign labels to both Arabic and English Samples before shuffling

```
# Preprocess the data and assign labels
english_sentences = []
arabic_sentences = []

for line in lines:
    parts = line.strip().split('\t')
    english_sentences.append(parts[0])
    arabic_sentences.append(parts[1])

# Assign labels
english_labels = ['English'] * len(english_sentences)
arabic_labels = ['Arabic'] * len(arabic_sentences)

# Combine English and Arabic data
data = pd.DataFrame({'english': english_sentences + arabic_sentences,
                     'arabic': arabic_sentences + english_sentences,
                     'label': english_labels + arabic_labels})

# Shuffle the data
data = data.sample(frac=1).reset_index(drop=True)
```

Preparing Labels

- Tokenizing both English & Arabic sentences
- Padding to insure uniform length
- Convert to Numerical values

```
# Tokenize English sentences
tokenizer_en = Tokenizer()
tokenizer_en.fit_on_texts(data['english'])
train_en_seq = tokenizer_en.texts_to_sequences(data['english'])

# Tokenize Arabic sentences
tokenizer_ar = Tokenizer()
tokenizer_ar.fit_on_texts(data['arabic'])
train_ar_seq = tokenizer_ar.texts_to_sequences(data['arabic'])

# Pad sequences to ensure uniform length
max_seq_length = 20
train_en_pad = pad_sequences(train_en_seq, maxlen=max_seq_length, padding='post')
train_ar_pad = pad_sequences(train_ar_seq, maxlen=max_seq_length, padding='post')

# Convert labels to numerical values
train_labels = (data['label'] == 'English').astype(int)
```

SIAMESE NETWORK ARCHITECTURE

```
# Define Siamese Network
def siamese_model(input_shape):
    input_1 = layers.Input(shape=input_shape)
    input_2 = layers.Input(shape=input_shape)

    # Base network
    base_network = tf.keras.Sequential([
        layers.Embedding(input_dim=num_words, output_dim=100, input_length=max_seq_length),
        layers.LSTM(100, return_sequences=True),
        layers.LSTM(100),
        layers.Dense(100, activation='relu')
    ])

    # Process the inputs through the base network
    processed_1 = base_network(input_1)
    processed_2 = base_network(input_2)

    # Calculate L1 distance between the processed vectors
    distance = tf.abs(tf.subtract(processed_1, processed_2))

    # Output layer
    output = layers.Dense(1, activation='sigmoid')(distance)

    model = Model(inputs=[input_1, input_2], outputs=output)

    return model
```

- **Siamese Architecture**: Utilizes two identical branches sharing weights for efficient comparison.
- **Text Processing Layers**: Employs LSTM layers for sequential data processing to capture long-term dependencies.
- **Representation Learning**: Learns dense representations of input sequences via embedding layers.
- **Distance Calculation**: Computes the L1 distance between processed representations to measure similarity.
- **Sequential Processing**: LSTM layers capture sequential patterns and dependencies within input sequences.
- **Feature Extraction**: Dense layers with ReLU activation extract higher-level features from LSTM outputs.
- **Output Layer**: Employs a dense layer with sigmoid activation to output similarity scores between input sequences.
- **Training Objective**: Optimizes model parameters to minimize the distance between similar pairs and maximize it for dissimilar pairs.

Prepare and Training Phase

- We prepare and split the dataset with a ratio of 80% to 20% (Train to Test)
- We split the dataset again into train and validation with a ratio of 90% to 10% (Train To Test)
- Next is the Training Phase

```
# Prepare dataset
num_samples = len(train_en_pad)
num_words = 10000

# Split the data into 80% training and 20% test
train_en, test_en, train_ar, test_ar, train_labels, test_labels = train_test_split(train_en_pad, train_ar_pad, train_labels, test_size=0.2, random_state=42)
```

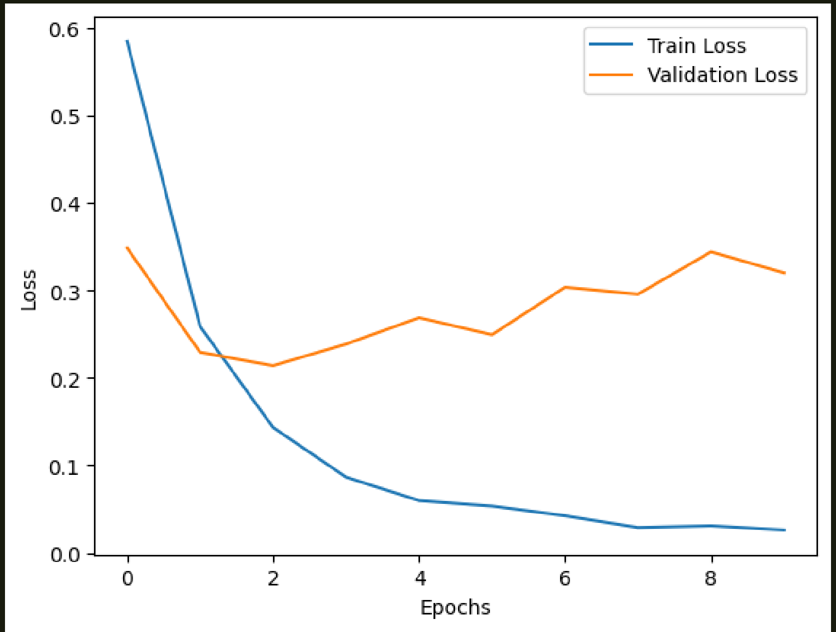
```
# Train the Siamese Network
input_shape = (max_seq_length,)
model = siamese_model(input_shape)
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
# Train the model
history = model.fit([train_en, train_ar], train_labels, epochs=10, batch_size=32, validation_split=0.1)
```

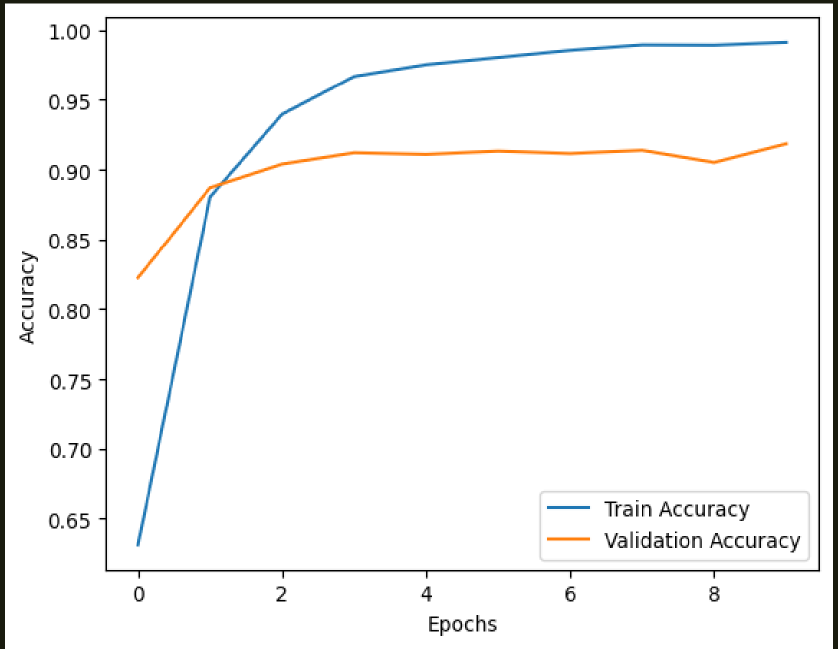
```
# Plotting training history
plt.plot(history.history['loss'], label='Train Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()

plt.plot(history.history['accuracy'], label='Train Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()
plt.show()


# Print final train loss and accuracy
train_loss, train_accuracy = model.evaluate([train_en, train_ar], train_labels)
print("Final Train Loss:", train_loss)
print("Final Train Accuracy:", train_accuracy*100,"%")

# Evaluate the model on test data
test_loss, test_accuracy = model.evaluate([test_en, test_ar], test_labels)
print("Test Loss:", test_loss)
print("Test Accuracy:", test_accuracy*100,"%")
```

RUNNING THIS SNAPSHOT DISPLAYS THE ACCURACY OF THE MODEL

- Final Train Loss: 0.043146368116140366
- Final Train Accuracy: 98.83050918579102 %
- Test Loss: 0.3846551179885864
- Test Accuracy: 90.22573828697205 %

THE RESULT GRAPHS (LOSS & ACCURACY)





Similarity Function

The Defined function bellow used for splitting sentences into words to compute the similarity among them

```
def preprocess_sentence(sentence, max_seq_length):
    words = sentence.split()
    if len(words) > max_seq_length:
        words = words[:max_seq_length]
    else:
        words += [''] * (max_seq_length - len(words))
    return words

def compute_similarity(sentence1, sentence2, model, tokenizer_en, tokenizer_ar, max_seq_length):
    processed_sentence1 = preprocess_sentence(sentence1, max_seq_length)
    processed_sentence2 = preprocess_sentence(sentence2, max_seq_length)

    # Tokenize and pad the sequences
    input_en = pad_sequences(tokenizer_en.texts_to_sequences([processed_sentence1]), maxlen=max_seq_length, padding='post')
    input_ar = pad_sequences(tokenizer_ar.texts_to_sequences([processed_sentence2]), maxlen=max_seq_length, padding='post')

    similarity_score = model.predict([input_en, input_ar])[0][0]

    return similarity_score
```

Examples

```
# Example usage
user_sentence1 = "Why Me "
user_sentence2 = " لماذا أنا "

similarity_score = compute_similarity(user_sentence1, user_sentence2, model, tokenizer_en, tokenizer_ar, max_seq_length)
print("Similarity Score:", similarity_score*100)
```

```
1/1 [==============================] - 0s 32ms/step
Similarity Score: 79.81667518615723
```

```
# Example usage
user_sentence1 = "Programming Is Fun "
user_sentence2 = " البرمجة ممتعة "

similarity_score = compute_similarity(user_sentence1, user_sentence2, model, tokenizer_en, tokenizer_ar, max_seq_length)
print("Similarity Score:", similarity_score*100)
```

```
1/1 [==============================] - 0s 20ms/step
Similarity Score: 99.89726543426514
```

```
# Example usage
user_sentence1 = "Is the cat on the chair or under the chair?"
user_sentence2 = "هل القطة فوق الكرسي أم أسفله ؟"

similarity_score = compute_similarity(user_sentence1, user_sentence2, model, tokenizer_en, tokenizer_ar, max_seq_length)
print("Similarity Score:", similarity_score*100)
```

```
1/1 [==============================] - 0s 20ms/step
Similarity Score: 100.0
```