ENGLISH TO FRENCH: A HANDS-ON GUIDE TO SEQ2SEQ LANGUAGE TRANSLATION

Imagine this: you're traveling in a foreign country, holding a conversation with a local, but you don't know their language. Yet, with just a small device or app, their words seamlessly transform into your native tongue. How does this magic happen? The answer lies in the powerful world of **Seq2Seq Neural Networks**.

In this tutorial, you won't just learn *about* this technology—you'll build it yourself! Together, we'll unlock the secrets of Seq2Seq models, the cornerstone of translation tools like Google Translate and multilingual chatbots. Ready to dive into the cutting-edge of artificial intelligence? Let's go!

What are Seq2Seq Models?

The **Seq2Seq (Sequence-to-Sequence)** model is a powerful architecture used for tasks where both the input and output are sequences of data. It's particularly useful for applications in **machine translation**, where one language is converted into another, and **text summarization**, where long pieces of text are condensed into shorter summaries. Seq2Seq models have also been successfully applied in building **chatbots**, which can hold conversations with users, and in **question answering systems**, where the model provides answers based on given text. As you explore these applications, you'll see how Seq2Seq models can be adapted to solve many complex problems, making them a fundamental tool in the field of natural language processing.

Seq2seq Model Architecture

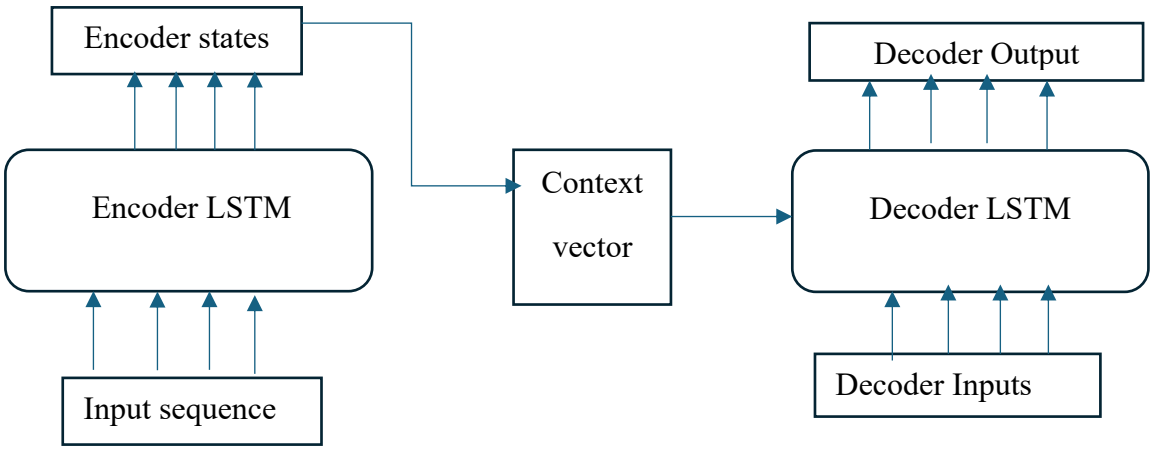

Fig1: Basic Architecture of seq2seq neural network

The Seq2Seq model is a powerful architecture used to translate one sequence into another, such as translating an English sentence into a French sentence. The architecture consists of two main parts: the encoder and the decoder, which work together to process the input and generate the output sequence.

Encoder LSTM

The encoder acts as a "master note-taker" for the input sequence. It reads and processes the input, one word at a time, transforming each word into a numerical format that the model can understand. The encoder then builds an internal understanding of the sentence, creating a context vector, which is essentially a compressed summary of the entire sentence. This context vector is the key that the decoder will use to generate the output.

To understand the working of the encoder, it helps to think of it as a sentence detective. The encoder processes the input sequence one word at a time, updating its internal hidden state to keep track of the meaning of the sentence as it progresses. At each timestep, the encoder's hidden state is updated based on the current word and the previous hidden state. This iterative process ensures that the final hidden state contains all the important details about the entire input sentence. The final hidden state becomes the context vector, which the decoder uses to begin generating the output.

Decoder LSTM

The decoder functions as the "translator" in this model. It uses the context vector from the encoder to start generating the output sequence, word by word. For example, when translating "I am a student" into "Je suis un étudiant", the decoder starts by using the context vector to predict the first word, "Je", and continues generating each subsequent word: "suis", "un", and finally "étudiant".

The decoder uses an LSTM (Long Short-Term Memory) network to generate each word in the sequence. LSTMs are a special type of Recurrent Neural Network (RNN) that are designed to handle long-term dependencies. They are crucial for managing information over longer sequences, ensuring that important context from the encoder is maintained throughout the decoding process.

LSTM: The Memory Master

The use of LSTM cells in the encoder and decoder is a key part of the Seq2Seq model. LSTMs are designed to overcome the vanishing gradient problem, which can occur when processing long sequences in standard RNNs. LSTMs can remember important information over long sequences and forget irrelevant details. This makes them highly effective for tasks like translation, where maintaining the context over long sentences is crucial.

Inside an LSTM cell, there are three key components or gates that manage the flow of information: the forget gate, the input gate, and the output gate. The forget gate decides what information to discard from the past, the input gate decides what new information to add to the memory, and the output gate determines which part of the memory should be used for the current task. This system of remembering, forgetting, and focusing allows the LSTM to maintain the context of a sentence over long sequences, making it ideal for tasks like Seq2Seq translation.

The Seq2Seq model works by encoding the input sequence into a compact context vector and then decoding it into an output sequence, one word at a time. The use of LSTMs allows the model to effectively manage long-term dependencies in the data, making it highly suited for

natural language processing tasks such as machine translation, text summarization, and question answering.

Unveiling the Magic of Seq2Seq: Building Your Own Language Translator

Step 1: Download the Dataset

For this tutorial, we'll use the English French language translation dataset from Kaggle, which contains over 100,000 sentence pairs. To ensure efficient training on limited resources, we'll only use a subset of 10,000 sentence pairs. This allows everyone to run and explore the model without needing extensive GPU or memory.

Our focus is on building and understanding the model, rather than achieving perfect results, so this smaller dataset will suffice. You can download the dataset and inspecting its columns using the following code:

```
# Step 1: Download the dataset
path = kagglehub.dataset_download("devicharith/language-translation-englishfrench")

# Step 2: List the files in the dataset directory
print("\nFiles in the dataset directory:")
for root, dirs, files in os.walk(path):
    for file in files:
        print(file)

# Step 3: If the dataset contains a CSV file, load it and check the column names
# Assume the file is named 'english_french.csv' or similar (you can adjust based on the actual file name)
csv_files = [file for file in os.listdir(path) if file.endswith('.csv')]

for csv_file in csv_files:
    file_path = os.path.join(path, csv_file)
    df = pd.read_csv(file_path)

    # Step 4: Print the first few rows and column names
    print(f"\nInspecting {csv_file}...")
    print("Columns:", df.columns)
    print(df.head())
```

Step2: Data Preprocessing.

The dataset contains over 100,000 sentence pairs, but we sample 10,000 to ensure the code can be reproduced on any system, regardless of CPU or memory limitations. Text is cleaned by converting to lowercase and removing extra spaces to maintain consistency. The cleaned data is then split into 80% for training and 20% for testing, preparing it for efficient model training and evaluation.

```
def clean_text(text):
    text = text.lower()  # Convert to lowercase
    text = text.strip()  # Remove leading/trailing whitespaces
    return text
df = df.sample(n=10000, random_state=42)
# Apply cleaning function to both English and French sentences
df['English words/sentences'] = df['English words/sentences'].apply(clean_text)
df['French words/sentences'] = df['French words/sentences'].apply(clean_text)

# Split the data into training and testing sets
X = df['English words/sentences'].values
y = df['French words/sentences'].values

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

Step3: Tokenization and Padding

In this step, we dive into the exciting process of **tokenization and padding**, transforming raw text data into a format the Seq2Seq model can understand. This is where we bridge the gap between human language and the model's numerical world!

First, we use tokenizers to break down sentences into their numerical equivalents. Imagine the input and output sentences as puzzles, where each word is assigned a unique number to fit perfectly into the model's structure. By fitting the tokenizers on the training data, we create a custom dictionary for both languages, mapping each unique word to an integer. This transforms sentences into sequences of numbers, ready for the model to process.

```
# Initialize tokenizers for English and French
english_tokenizer = Tokenizer()
french_tokenizer = Tokenizer()

# Fit the tokenizers on the training data
english_tokenizer.fit_on_texts(X_train)
french_tokenizer.fit_on_texts(y_train)

# Convert sentences to sequences
X_train_seq = english_tokenizer.texts_to_sequences(X_train)
y_train_seq = french_tokenizer.texts_to_sequences(y_train)

X_test_seq = english_tokenizer.texts_to_sequences(X_test)
y_test_seq = french_tokenizer.texts_to_sequences(y_test)
```

But here's the challenge: sentences come in all shapes and sizes. Some are long, others short, and this variability can confuse the model. To solve this, we introduce **padding**. Padding ensures that all sequences are the same length by adding zeros to the shorter ones. Think of it as levelling the playing field for sentences, so every input and output sequence has equal footing when it enters the model.

```
# Pad the sequences to ensure uniform length
max_input_len = max([len(seq) for seq in X_train_seq])
max_output_len = max([len(seq) for seq in y_train_seq])

X_train_pad = pad_sequences(X_train_seq, maxlen=max_input_len, padding='post')
y_train_pad = pad_sequences(y_train_seq, maxlen=max_output_len, padding='post')

X_test_pad = pad_sequences(X_test_seq, maxlen=max_input_len, padding='post')
y_test_pad = pad_sequences(y_test_seq, maxlen=max_output_len, padding='post')
```

Once the tokenization and padding are complete, we have clean, uniform data sequences that are ready to flow seamlessly into the Seq2Seq model. This step is more than just preparation—it's the foundation that ensures the model can learn effectively. Now that your text is translated into a language the model understands, you're ready to unlock the magic of machine translation!

Step4: Building the Seq2Seq Model

Here's where the magic begins! We're constructing the Seq2Seq model, the powerhouse that transforms one language into another by piecing together the encoder and decoder into a unified system.

The encoder serves as the model's interpreter, meticulously processing the input sentence—think of it as translating each word into a numerical language the model can understand. Through an embedding layer, the encoder creates meaningful representations of each word. These word embeddings are then passed through an LSTM layer, which processes the sentence step by step, remembering important details while summarizing its essence. By the end of this process, the encoder produces a context vector—a compact yet powerful summary of the entire sentence, ready to be handed off to the decoder.

```
# Define the encoder
encoder_input = Input(shape=(max_input_len,))
encoder_embedding = Embedding(input_dim=len(english_tokenizer.word_index) + 1, output_dim=embedding_dim)(encoder_input)
encoder_lstm = LSTM(latent_dim, return_state=True)
encoder_output, state_h, state_c = encoder_lstm(encoder_embedding)
```

The decoder takes this context vector and begins crafting the output sentence. It starts with an initial target sequence and, step by step, predicts the next word, drawing on the information provided by the encoder. With each step, the decoder refines its understanding of what to predict next, generating probabilities for every word in the target vocabulary. A dense layer with a softmax activation selects the most likely word, bringing us closer to a fluent translation.

```
# Define the decoder
decoder_input = Input(shape=(max_output_len,))
decoder_embedding = Embedding(input_dim=len(french_tokenizer.word_index) + 1, output_dim=embedding_dim)(decoder_input)
decoder_lstm = LSTM(latent_dim, return_sequences=True, return_state=True)
decoder_output, _, _ = decoder_lstm(decoder_embedding, initial_state=[state_h, state_c])
decoder_dense = TimeDistributed(Dense(len(french_tokenizer.word_index) + 1, activation='softmax'))
decoder_output = decoder_dense(decoder_output)
```

Finally, the encoder and decoder are connected to form the complete Seq2Seq model

```
# Build the model
model = Model([encoder_input, decoder_input], decoder_output)
# Compile the model
model.compile(optimizer=Adam(), loss='sparse_categorical_crossentropy', metrics=['accuracy'])
# Display the model architecture
model.summary()
```

Step 5: Training seq2seq model

In this step, we prepare the decoder input and start training the Seq2Seq model to master the art of translation. Let's explore how this is achieved step by step.

To train the decoder effectively, we need to guide it during training. This is where the decoder input comes into play. During training, the decoder needs to predict the next word in the output sequence based on the context and the previous words. To make this work, we shift the target sequence by one position. This means the decoder learns to predict the next word at each step, using the previous word as input. Imagine showing the decoder part of a sentence and asking it, "What comes next?" This step-by-step guidance is key to teaching the model how to construct fluent output sentences.

After preparing the decoder input, the model is ready for training. We feed it the padded input sentences and the prepared decoder input, along with the original target sequences as the desired output. The target sequences are expanded to match the shape the model expects for its predictions. The model processes this data over multiple epochs, which are cycles through the entire training dataset. In each cycle, the model adjusts its internal weights to improve its predictions, reducing the gap between its outputs and the actual target sequences.

During training, the data is divided into smaller batches, which helps the model process efficiently and adjust weights more frequently. A portion of the data is set aside for validation, allowing us to monitor how well the model generalizes to unseen data as it trains.

```
[ ] # Prepare decoder input for training
    y_train_decoder_input = np.zeros_like(y_train_pad)
    y_train_decoder_input[:, 1:] = y_train_pad[:, :-1]

    y_train_decoder_input = y_train_decoder_input.reshape((y_train_decoder_input.shape[0], y_train_decoder_input.shape[1], 1))

    # Train the model
    history = model.fit(
        [X_train_pad, y_train_decoder_input],
        np.expand_dims(y_train_pad, -1),  # Add an extra dimension for the target sequence
        epochs=10,
        batch_size=64,
        validation_split=0.2
    )
```

we visualize the model's performance during training by plotting the training and validation loss and training and validation accuracy over the epochs.

The graphs show that the model's training and validation loss steadily decrease, indicating improved predictions and good generalization to unseen data without overfitting. Training and validation accuracy also rise, reflecting consistent improvements. With only 10,000 data points used, the model demonstrates strong performance, and training on more data would likely enhance its ability to generalize and converge more effectively. These results highlight that the model is learning well and has potential for even better outcomes with additional data.

Step 6: Inference/Prediction

This section evaluates the model by generating predictions for the test data, converting both the predictions and actual target sequences back into readable text, and comparing them. The results, including the input sentences, actual translations, and predicted translations, are compiled into a DataFrame for easy assessment

```
# Evaluate the model's predictions on the test data
test_predictions = model.predict([X_test_pad, y_test_decoder_input])

# Convert predicted sequences to words
pred_fra_text = [convert_idx_to_sent(np.argmax(pred, axis=1), french_tokenizer) for pred in test_predictions]

# Convert actual French sequences to words (use tokenized data, not raw text)
orig_fra_text = [convert_idx_to_sent(seq, french_tokenizer) for seq in y_test_pad]

# Show the results
predictions = pd.DataFrame({
    'Original English Sentence': [convert_idx_to_sent(seq, english_tokenizer) for seq in X_test_pad],
    'Original French Sentence': orig_fra_text,
    'Predicted French Sentence': pred_fra_text
})

# Display the first few predictions
print(predictions.head())
```

Conclusion

In this session, we explored the fascinating journey of building a Seq2Seq model, transforming raw text into meaningful translations step by step. From understanding the architecture to preparing the data, training the model, and evaluating its performance, we've uncovered the magic behind language translation systems. This is more than just a model; it's a gateway to solving real-world problems in natural language processing. Remember, each challenge you tackle in AI brings you closer to mastering this field and unlocking its limitless potential. Keep experimenting, stay curious, and never stop exploring—your creativity and persistence will shape the innovations of tomorrow.

Github Link: https://github.com/aswanyshaji/Machine-_learning-assignment/tree/main