Assignment 03

Implementing English-to-French Machine Translation Using RNN or Transformer

Batool Ali Akbar
ba07612@st.habib.edu.pk

I. Chosen Model

A translation task is a common use case of a sequence-to-sequence model. An encoder-decoder model implemented via RNN is one of the earliest sequence-to-sequence models. However, the model is quite slow and cannot preserve long dependencies. Therefore, I have chosen a transformer based model for this assignment. Unlike RNN, a transformer can consider all parts of a sentence simultaneously, capturing both past and future context which improves the overall accuracy of the translation.

II. Implementation

This entire section focuses on how the model was implemented and evaluated. The provided dataset was quite large with about 175,622 data entries. This kind of data processing needed huge computation power which my system could not support and hence threw an error entailing resource exhaustion. Therefore, I had to cut down the file to 60,000 entries.

A. Data Preprocessing

The following block of code preprocesses the data read from the file using the built-in pandas library. Line 4 reads only the first 60,000 entries from the entire data file. Lines 6 and 7 turn the entire dataset into lower case. Lines 9 to 16 tokenize each word, assigning them an integer and converting each sentence into an array of integers, each denoting the corresponding word of the sentence. Lines 18 to 22 use padding to ensure equal length for each sentence. Lines 24 and 25 split the data into training sample (80%), validation sample (10%) and testing sample (10%).

```
1
2  def preprocess_data(file_path):
3
4      data = pd.read_csv(file_path)[:60000]
5
6      eng_text = data['English words/sentences'].apply
       (lambda x: x.lower().strip())
7      french_text = data['French words/sentences'].
       apply(lambda x: x.lower().strip())
8
9      eng_tokenizer = Tokenizer()
10     french_tokenizer = Tokenizer()
11
12     eng_tokenizer.fit_on_texts(eng_text)
13     french_tokenizer.fit_on_texts(french_text)
14
15     eng_sequences = eng_tokenizer.texts_to_sequences
       (eng_text)
16     french_sequences = french_tokenizer.
       texts_to_sequences(french_text)
17
18     max_eng_len = max(len(seq) for seq in
       eng_sequences)
19     max_french_len = max(len(seq) for seq in
       french_sequences)
20
21     eng_padded = pad_sequences(eng_sequences, maxlen
       =max_eng_len, padding='post')
22     french_padded = pad_sequences(french_sequences,
       maxlen=max_french_len, padding='post')
23
24     eng_train, eng_temp, french_train, french_temp =
        train_test_split(eng_padded, french_padded,
       test_size=0.2, random_state=42)
25     eng_val, eng_test, french_val, french_test =
       train_test_split(eng_temp, french_temp,
       test_size=0.5, random_state=42)
26
27     return eng_train, eng_val, eng_test,
       french_train, french_val, french_test,
       eng_tokenizer, french_tokenizer, max_eng_len,
       max_french_len
28
```

Listing 1: Data Preprocessing

B. Model Implementation

Code listing 2 provides an implementation of the transformer model. The model has been built on the architecture illustrated in Figure 1.

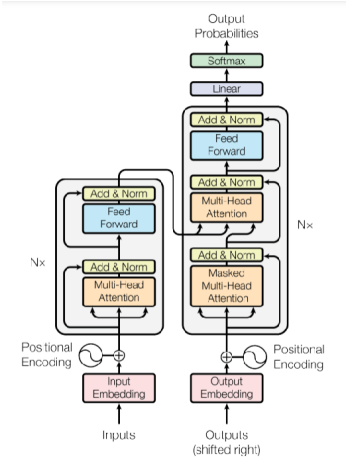

Fig. 1: Tranformers Architecture

```
def transformer_model(input_vocab_size, target_vocab_size, input_max_len, target_max_len, embedding_dim=256, num_heads=4, feed_forward_dim=512):

    # Encoder
    inputs = Input(shape=(input_max_len,))
    enc_emb = Embedding(input_vocab_size, embedding_dim)(inputs)
    positional_encoding_input = positional_encoding(input_max_len, embedding_dim)
    enc_emb = enc_emb + positional_encoding_input

    for _ in range(2):
        # Multi-Head Attention
        attn_output = MultiHeadAttention(num_heads=num_heads, key_dim=embedding_dim)(enc_emb, enc_emb, enc_emb)
        # Add & Norm
        norm_1 = LayerNormalization()(attn_output)
        add_1 = norm_1 + enc_emb
        # Feed Forward Network
        ffn_output = Dense(feed_forward_dim, activation='relu')(add_1)
        ffn_output = Dense(embedding_dim)(ffn_output)
        # Add & Norm
        norm_2 = LayerNormalization()(ffn_output)
        add_2 = norm_2 + add_1
        enc_emb = add_2
    enc_output = enc_emb

    # Decoder
    dec_inputs = Input(shape=(target_max_len,))
    dec_emb = Embedding(target_vocab_size, embedding_dim)(dec_inputs)
    positional_encoding_target = positional_encoding(target_max_len, embedding_dim)
    dec_emb = dec_emb + positional_encoding_target

    for _ in range(2):
        # Masked Multi-Head Attention (Self-Attention)
        attn_output = MultiHeadAttention(num_heads=num_heads, key_dim=embedding_dim)(dec_emb, dec_emb, dec_emb, attention_mask=create_causal_mask(target_max_len))
        # Add & Norm
        norm_1 = LayerNormalization()(attn_output)
        add_1 = dec_emb + norm_1
        # Encoder-Decoder Attention
        enc_dec_attn_output = MultiHeadAttention(num_heads=num_heads, key_dim=embedding_dim)(add_1, enc_output, enc_output)
        # Add & Norm
        norm_2 = LayerNormalization()(enc_dec_attn_output)
        add_2 = norm_1 + add_1
        # Feed Forward Network
        ffn_output = Dense(feed_forward_dim, activation='relu')(add_2)
        ffn_output = Dense(embedding_dim)(ffn_output)
        # Add & Norm
        norm_3 = LayerNormalization()(ffn_output)
        add_3 = ffn_output + norm_3
        dec_emb = add_3
    dec_output = dec_emb

    # Final Dense Layer
    output = Dense(target_vocab_size, activation='softmax')(dec_output)
    model = Model(inputs=[inputs, dec_inputs], outputs=output)
    return model
```

Listing 2: Model Implementation

The current implementation uses 2 layers of encoder and 2 layers of decoder with an embedding dimension of 256, 4 attention heads for each attention mechanism and RELU as an activation function. It then uses built-in functions provided by the tensorflow library to create a transformer model with an architecture similar to the one mentioned in Figure 1. Lines 4 to 23 are responsible for encoding the input sequence while the lines beyond 25 are the decoder part of the model. Line 52 applies the softmax function to the output of the decoder to normalize the values and line 53 then compiles the entire model.

C. Model Training

Python's builtin library 'tensorflow' is used to train the model. The code below illustrates how the model has been trained.

```
model = transformer_model(input_vocab_size, target_vocab_size, max_eng_len, max_french_len)
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])

history = model.fit(
    [eng_train, french_train],
    np.expand_dims(french_train, -1),
    epochs=10,
    batch_size=64,
    validation_data=([eng_val, french_val], np.expand_dims(french_val, -1))
)

```

Listing 3: Model Training

The model is compiled using the Adam optimizer along with sparse categorical cross-entropy loss. The model is then trained over 10 epochs with a batch size of 64. During training, the input English sentences are paired with their corresponding French translations. The training also includes validation data to monitor the model's performance on unseen data. The model.fit() function performs the training, with both training and validation accuracy being tracked throughout the process.

D. Evaluation

The model has been evaluated using the BLEU score. The following code depicts how the BLEU score was calculated.

```
def evaluate_model(model, eng_test, french_test, eng_tokenizer, french_tokenizer):

    # Predicting the French sentences
    french_pred = model.predict([eng_test, french_test])
    french_pred_text = [
        french_tokenizer.sequences_to_texts([np.argmax(pred, axis=-1)])[0]
        for pred in french_pred
    ]
```

```
    # Decoding the French sentences
    french_test_decoded = [
        ' '.join([french_tokenizer.index_word[token]
     for token in sentence if token != 0])
        for sentence in french_test
    ]

    # Prepare references and hypotheses for BLEU
    score calculation
    references = [[sent.split()] for sent in
    french_test_decoded]
    hypotheses = [sent.split() for sent in
    french_pred_text]

    # BLEU score calculation
    smoothing = SmoothingFunction().method1
    bleu_score = corpus_bleu(references, hypotheses,
     smoothing_function=smoothing)
    print(f'BLEU Score: {bleu_score}')
```

Listing 4: BLEU Score Calculation

The code uses the transformer model defined above to make predictions for the test set. It then converts the predictions into proper texts and then compares the predicted output with the expected output to calculate the BLEU score using the 'nltk' library.

III. Performance

The BLEU score for the model was calculated to be 0.87. It suggests that the model's translations are close to the reference translations. This high score shows that the model's output is generally fluent and accurate in terms of word choice, syntax, and meaning. This can also be seen in the following 3 samples that were printed out while testing.

Expected Translation	Predicted Translation
je vais en ville	je vais en ville
les chaussures se vendent par paires	les chaussures se dames par réussirait
nous étions chanceux	nous étions chanceux

Table 1: Translation Samples in Test Set

Furthermore, the accuracy and loss of the model for the training and the validation set over the 10 epochs was also recorded. The values are mentioned in the table below.

Epoch	Training Accuracy	Training Loss	Validation Accuracy	Validation Loss
1	0.7822	1.6869	0.9533	0.3502
2	0.9334	0.4752	0.9647	0.2900
3	0.9608	0.2602	0.9720	0.2403
4	0.9763	0.1465	0.9739	0.2199
5	0.9771	0.1233	0.9757	0.2097
6	0.9815	0.0930	0.9773	0.2100
7	0.9815	0.0852	0.9662	0.2408
8	0.9827	0.0744	0.9812	0.1919
9	0.9885	0.0476	0.9756	0.2161
10	0.9836	0.0713	0.9841	0.1842

Table 2: Training and Validation Accuracy and Loss per Epoch

This trend can also be seen in the following graphs.

Fig. 2: Loss and Accuracy over 10 Epochs

The graph on the left depicts the training loss (blue line) and the validation loss (orange line). Both of them decrease initially, indicating that the model is learning and improving its predictions. The training is initially quite high but drops rapidly in the first few epochs, then gradually levels out, suggesting that the model is fitting well on the training data. The validation loss starts from quite a small value and continues to decrease slowly. However, the loss for both training and validation set fluctuates slightly in later epochs, suggesting some generalization challenges but overall the trend is consistent.

The graph on the right depicts the training accuracy (blue line) and the validation accuracy (orange line). The training accuracy starts off with a low value but then increases quickly in the first few epochs and then stabilizes near 0.98, indicating that the model performs well on the training data. The validation accuracy starts at around 0.95 and follows a similar trend, achieving a high accuracy close to that of the training set. The slight fluctuation in the validation accuracy suggests some variation in performance on the validation set, but the overall trend shows good model generalization with high accuracy on both the training and validation sets.

Overall, the model is performing well, with a high BLEU score, low loss and high accuracy on both training and validation datasets. The small gap between training and validation metrics suggests minimal overfitting, indicating that the model is likely generalizing well to unseen data.

IV. Reflection

Although the implementation of the model has been done entirely using the libraries, I still faced some challenges. I initially planned to implement an RNN model but the model was taking much longer than expected to predict an output. Therefore, I had to switch to a transformers based model. Moreover, the size of the dataset was quite large. The normal VS code compiler could not process the entire data. I had to move to Google Colab and switch from CPU to GPU to run the model in a reasonable amount of time. However, this had its own challenges. Google Colab had a limited time GPU

access which once again slowed down the entire pace of the assignment.