SEP 775 : Introduction to Computational Natural Language Processing

Final Project

Text Style Transfer

Submitted by

Rushabh Kakadiya (400546152)

Jaimis Miyani (400551743)

Param Mehta (400547035)

Supervisor

Prof. Hamidreza Mahyar

mahyarh@mcmaster.ca

W Booth School of Engineering Practice and Technology

Objective:

Develop and refine Natural Language Processing (NLP) models for style transfer by analyzing text styles, gathering appropriate datasets, and employing state-of-the-art techniques to separate content from style. Utilize evaluation metrics to assess the effectiveness of style transfer and iteratively refine the models based on feedback.

Dataset Information:

The NUS Social Media Text Normalization and Translation Corpus provides a solution to the scarcity of publicly available normalized datasets. It consists of 2,000 messages randomly selected from the NUS English SMS corpus. These messages were normalized into formal English, making it a valuable resource for training and evaluating models for text normalization and style transfer tasks.

The NUS Social Media Text Normalization and Translation Corpus, tailored for text normalization and translation tasks, is structured with two distinct columns:

- "**informal text**": Contains messages in their original informal form.
- "**formal text**": Presents the corresponding normalized messages in formal English.

1. Dataset Preprocessing

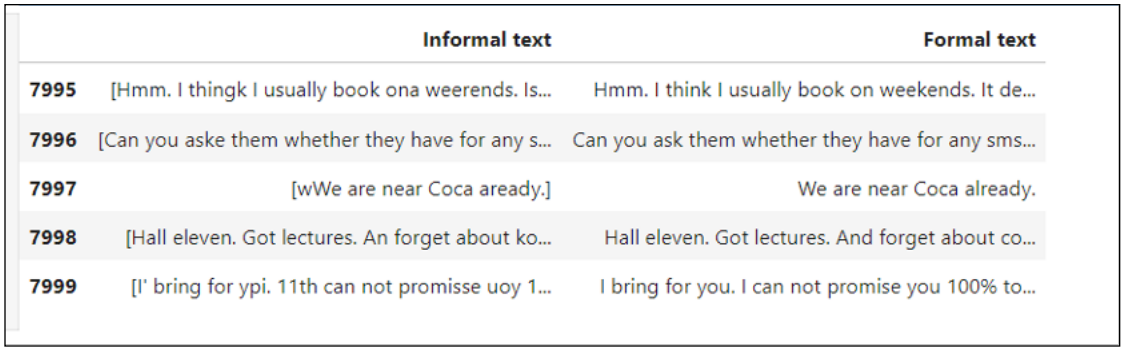
- **Data loading:**

  The provided code segment facilitates the extraction and structuring of textual data stored in a file named "en2cn-2k.en2nen2cn". Upon reading the file's contents, the text is split by newline characters ("\n"), with the last instance discarded as it represents an empty string. The extracted text data is then organized into a pandas DataFrame termed "df", featuring two columns: "Informal text" and "Formal text". Each row within the DataFrame corresponds to a pair of entries, where "Informal text" represents unstructured language, and "Formal text" denotes its corresponding formalized version. This structured DataFrame serves as a foundational dataset for subsequent data processing and analysis tasks.

- **Data Augmenting:**

  The code applies data augmentation techniques to enhance the textual data stored in a DataFrame. It first utilizes synonym augmentation, replacing words with their synonyms sourced from WordNet. Then, spelling augmentation is employed to introduce variations by

altering the spelling of words. Each augmentation step generates new entries in the DataFrame pairing the augmented text with the original. This process expands the DataFrame to include the augmented data, enriching the dataset with diverse textual variations for improved model training or analysis.



	Informal text	Formal text
7995	[Hmm. I thingk I usually book ona weerends. Is...	Hmm. I think I usually book on weekends. It de...
7996	[Can you aske them whether they have for any s...	Can you ask them whether they have for any sms...
7997	[wWe are near Coca aready.]	We are near Coca already.
7998	[Hall eleven. Got lectures. An forget about ko...	Hall eleven. Got lectures. And forget about co...
7999	[I' bring for ypi. 11th can not promisse uoy 1...	I bring for you. I can not promise you 100% to...

Figure 1: After augmenting Data becomes 4x larger.

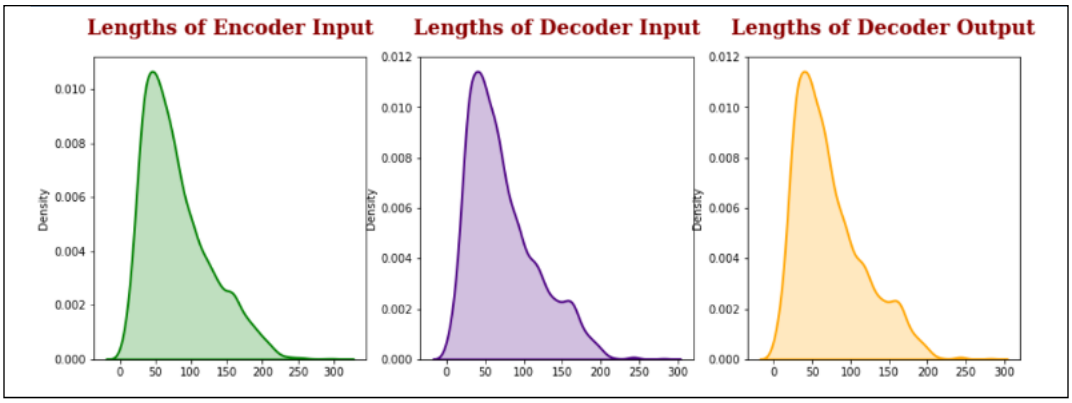
- **Preprocessing for Seq2Seq Model:**

The code segment preprocesses textual data for training a Seq2Seq model. It involves creating encoder input, decoder input, and decoder output sequences from the original data. Each input sequence is enclosed within angle brackets ("<"), with the informal text appended for the encoder input and prepended for the decoder input. The decoder output is formed by appending the formal text with a closing angle bracket (">"). Finally, a DataFrame is constructed to organize the preprocessed data, with columns representing the encoder input, decoder input, and decoder output sequences.

	encoder_inp	decoder_inp	decoder_out
0	<U wan me to "chop" seat 4 u nt?>	<Do you want me to reserve seat for you or not?	Do you want me to reserve seat for you or not?>
1	<Yup. U reaching. We order some durian pastry ...	<Yeap. You reaching? We ordered some Durian pa...	Yeap. You reaching? We ordered some Durian pas...
2	<They become more ex oredi... Mine is like 25....	<They become more expensive already. Mine is l...	They become more expensive already. Mine is li...
3	<I'm thai. what do u do?>	<I'm Thai. What do you do?	I'm Thai. What do you do?>
4	<Hi! How did your week go? Haven heard from yo...	<Hi! How did your week go? Haven't heard from ...	Hi! How did your week go? Haven't heard from y...

Figure 2: Output after performing preprocessing

- **Splitting the data into training and validation:**
  - Sequence Length Distribution Visualization



As we can see, most of the sentences are of length around 50 and almost all the sentences have lengths less than 40. Hence, we can filter out the sentences which are of length more than 200.

- Filtering out sentences of length more than 200
- Splitting Data into train, validation, and test
- The above screenshot shows the final dimension of the data after dividing.

- **Tokenizing Data:**
  - Tokenization is applied to informal and formal data using Keras' Tokenizer class.
  - The tokenizers preserve case and exclude common punctuations like periods, commas, question marks, exclamation marks, colons, and semicolons.
  - For informal data, the tokenizer is fitted on the "encoder_inp" column of the training dataset.
  - The resulting tokenizer object for informal data is saved as "tknizer_informal.pkl".
  - For formal data, the tokenizer is fitted on the "decoder_inp" column after appending a "<end>" token to the first sentence.
  - This "<end>" token ensures that the vocabulary learns this special token.
  - The resulting tokenizer object for formal data is saved as "tknizer_formal.pkl".
  - Finally, the sizes of the vocabularies for both informal and formal text are printed, indicating the number of unique tokens in each vocabulary.

- **Padding the Data:**
  - It encodes the sentences into numerical sequences using the tokenizers created previously for informal and formal texts.

- For the encoder input (informal text), it applies the "texts_to_sequences" method of the informal tokenizer (tknizer_informal) to convert the sentences into numerical sequences.
- For the decoder input and output (formal text), it applies the "texts_to_sequences" method of the formal tokenizer (tknizer_formal) to convert the sentences into numerical sequences.
- It pads the sequences to ensure uniform length. This is done using the "pad_sequences" function from Keras, which pads sequences with zeros to a maximum length of 40, ensuring that all sequences have the same length.
- The padded sequences are stored in variables named encoder_seq, decoder_inp_seq, and decoder_out_seq, representing the encoded sequences for the encoder input, decoder input, and decoder output, respectively.

2. Simple Encoder Decoder Network:

• Encoder

Initialization:

1. Parameters: The Encoder is initialized with parameters such as input vocabulary size (inp_vocab_size), LSTM size (lstm_size), and input sequence length (input_length).
2. Layers: It consists of an Embedding layer (Embedding) and two LSTM layers (LSTM).

Embedding Layer:

1. Functionality: The input sequences are passed through an Embedding layer to convert them into dense vectors.
2. Parameters: The Embedding layer is configured with the input vocabulary size and input sequence length. Additionally, it uses a Random Normal initializer to initialize the embeddings.

LSTM Layers (Bidirectional):

1. Functionality: The embedded sequences are then processed by two LSTM layers in a bidirectional manner.
2. Parameters: Each LSTM layer returns both the hidden and current states (return_state=True) and sequences (return_sequences=True). The LSTM layers are initialized using Glorot uniform and orthogonal initializers for the kernel and recurrent weights, respectively.

Call Method:

1. Inputs: It takes a sequence input and the initial states of the Encoder.
2. Processing: The input sequence is passed through the Embedding layer, and the resulting embeddings are fed into the LSTM layers. The final hidden and current states are returned as outputs.

State Initialization Method:

1. Functionality: This method initializes the initial hidden and current states based on the specified batch size.
2. Output: It returns tensors representing the initial hidden and current states.

- **Decoder**

Initialization:

1. Instantiated with the output vocabulary size and LSTM size.
2. Comprises an Embedding layer and two LSTM layers.

Processing:

1. Embeds decoder input sequences using the Embedding layer.
2. Takes encoder final hidden and current states as initial states.
3. Processes embedded sequences through LSTM layers to generate output sequences.

Output Layer:

1. Functionality: A Dense layer with softmax activation that converts decoder outputs into normalized probabilities of tokens in the target vocabulary.

Call Method:

1. Input: Takes input data, which includes encoder and decoder inputs.
2. Processing:
   a) Passes encoder input through the Encoder model to obtain encoder outputs.

b) Passes decoder input and encoder outputs through the Decoder model to generate decoder outputs.

c) Applies the output layer to the decoder outputs to obtain normalized probabilities of tokens in the target vocabulary.

3. Training the Encode Decoder Model

- **Creating Model Callbacks:**

  The create_tensorboard_cb function generates a TensorBoard callback instance for logging training metrics and model performance during training. It takes a string representing the log directory path as input and returns a TensorBoard callback initialized with that path. By leveraging the current directory and timestamp, it creates a unique log directory for each training run. The function ensures easy integration of TensorBoard logging into TensorFlow models, facilitating comprehensive monitoring and analysis of the training process.

- **Training the model:**

  Model Parameters:

  1. UNITS: Specifies the dimensionality of the LSTM units in the model, set to 256.
  2. EPOCHS: Defines the number of training epochs, set to 50.
  3. TRAIN_STEPS and VALID_STEPS: Calculate the number of training and validation steps per epoch based on the batch size and dataset size.

  Model Initialization:

  1. An instance of the Encoder_Decoder class is created with the following parameters:
     a. inp_vocab_size: Size of the informal text vocabulary.
     b. out_vocab_size: Size of the formal text vocabulary.
     c. lstm_size: Dimensionality of the LSTM units, set to UNITS.
     d. input_length: Maximum length of input sequences.
     e. batch_size: Batch size used for training.
  2. The model is initialized with an **Adam** optimizer with a **learning rate of 0.01.**

Callbacks:

1. Several callbacks are defined to control the training process:
    a. ReduceLROnPlateau: Reduces the learning rate when a monitored metric has stopped improving.
    b. create_tensorboard_cb: Creates a TensorBoard callback for logging training metrics.
    c. EarlyStopping: Stops training when a monitored metric has stopped improving.
    d. ModelCheckpoint: Saves the model after every epoch if it's the best so far.

4. The Attention based Encoder Decoder Model

Encoder

It is exactly same as the previous model.

Attention Model

Class Definition:

- The Attention class inherits from tf.keras.Model and represents an Attention mechanism used in sequence-to-sequence models.
- It takes two inputs: the decoder hidden state and all the encoder outputs.

Initialization:

1. In the __init__ method, the parameters are initialized:
    a. lstm_size: Dimensionality of the LSTM units.
    b. scoring_function: Specifies the type of scoring function to be used for attention calculation.
2. Weights are initialized based on the scoring function:
    a. For 'dot' scoring function, a Dense layer (V) is initialized.
    b. For 'general' scoring function, a Dense layer (W) is initialized.
    c. For 'concat' scoring function, Dense layers (W1, W2) and a Dense layer (V) are initialized.

Call Method:

1. The call method defines the forward pass of the Attention mechanism.

2. Depending on the specified scoring function:
    a. For 'dot' scoring function, the dot product between encoder outputs and the decoder hidden state is computed using a Dense layer (V).
    b. For 'general' scoring function, the decoder hidden state is transformed using a Dense layer (W), and then the dot product is computed.
    c. For 'concat' scoring function, both the decoder hidden state and encoder outputs are transformed using Dense layers (W1, W2) before computing the dot product.
4. Softmax is applied to the scores to obtain attention weights.
5. The context vector is calculated by multiplying attention weights with encoder outputs and summing along the time axis.

Return:

The method returns the context vector and attention weights.

Timestep Decoder

Class Definition:

1. The Timestep_Decoder class inherits from tf.keras.Model.
2. It represents a model that processes one input token at a time and predicts the final hidden state of the LSTM unit.

Initialization:

1. The __init__ method initializes the parameters:
    a. out_vocab_size: Size of the output vocabulary.
    b. embedding_dim: Dimensionality of the embedding space.
    c. input_length: Length of input sequences.
    d. lstm_size: Dimensionality of the LSTM units.
    e. scoring_function: Specifies the type of scoring function used for attention calculation.
    f. embedding_matrix: Optional pre-trained embedding matrix.
2. An Attention mechanism is initialized based on the LSTM size and scoring function.
3. An Embedding layer is initialized based on the availability of the embedding matrix.
4. Two LSTM layers (lstm1, lstm2) are initialized.

Call Method:

1. The call method defines the forward pass of the Timestep Decoder model.
2. It takes the input token, encoder outputs, and encoder states as input.
3. The input token is passed through the Embedding layer to obtain the embedded token.
4. The Attention mechanism computes the context vector using encoder hidden states and outputs.
5. The context vector is concatenated with the embedded token.
6. The concatenated vector is passed through the LSTM layers to generate the final LSTM hidden state.
7. The final hidden state is passed through a Dense layer without activation, producing the output prediction.
8. The method returns the output prediction along with updated encoder hidden and current states.

Decoder

Class Definition:

1. The Decoder class inherits from tf.keras.Model.
2. It represents a model responsible for generating final output tokens based on all encoder states and the decoder input sequence.

Initialization:

1. The __init__ method initializes the parameters:
   a. out_vocab_size: Size of the output vocabulary.
   b. embedding_dim: Dimensionality of the embedding space.
   c. input_length: Length of input sequences.
   d. lstm_size: Dimensionality of the LSTM units.
   e. scoring_function: Specifies the type of scoring function used for attention calculation.
   f. embedding_matrix: Optional pre-trained embedding matrix.
2. An instance of the Timestep Decoder model is initialized, which handles the generation of output tokens at each timestep.

Call Method:

1. The call method defines the forward pass of the Decoder model.
2. It takes the decoder input sequence, encoder outputs, and encoder states as input.
3. Inside a TensorFlow graph (@tf.function decorator), it iterates over each timestep of the decoder input sequence.
4. At each timestep, it calls the Timestep Decoder model to generate the output token.
5. The output tokens at each timestep are stored in a tensor array.
6. Finally, the tensor array is reshaped and returned as the output tensor containing the sequence of decoded tokens.

Final Model Architecture

Class Definition:

1. It represents a model responsible for both encoding and decoding sequences while incorporating attention mechanisms.

Call Method:

1. The call method defines the forward pass of the model.
2. It takes data from the data pipeline in tuples, where the first element is the encoder input and the second element is the decoder input.
3. The encoder input is fed to the Encoder model along with initial states, and the resulting encoder states are obtained.
4. These encoder states, along with the decoder input, are passed to the Decoder model.
5. The Decoder model generates normalized output probabilities of tokens in the target vocabulary.
6. The final output is returned.

5. Training the Model

Creating Custom Loss Function:

1. This code snippet defines a custom loss function, loss_function, tailored for training sequence-to-sequence neural network models, such as those used in machine translation tasks. It initializes a Sparse Categorical Crossentropy loss object, configured to handle logits directly (from_logits=True) and compute loss for each sequence element independently (reduction='none').
2. The loss_function itself takes two arguments: the true labels (real) and the predicted logits (pred). It masks out the loss contribution from padded zeros in the sequences to prevent them from affecting the training process. By creating a mask based on non-padding tokens in the true labels, the function ensures that only relevant tokens contribute to the loss calculation.
3. Finally, it computes the mean of the masked loss values to obtain the overall loss for the batch. This custom loss function is crucial for effectively training sequence-to-sequence models, as it allows the model to focus solely on learning from meaningful tokens while disregarding padding tokens.

Training the Mode using Dot scoring function:

```
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 encoder (Encoder)           multiple                  566332

 decoder (Decoder)           multiple                  742451

=================================================================
Total params: 1308783 (4.99 MB)
Trainable params: 1308783 (4.99 MB)
Non-trainable params: 0 (0.00 Byte)
```

Definition:

The dot scoring function computes the similarity between the decoder hidden state and encoder outputs by taking the dot product between them. It's used in attention mechanisms to determine the relevance of each encoder output to the current decoding step. This function facilitates focusing the model's attention on pertinent parts of the input sequence during sequence generation.

Training:

1. To ensure reproducibility, we begin by setting a random seed using TensorFlow's tf.random.set_seed() function. This ensures that the results of our experiments remain consistent across different runs.
2. Next, we define various parameters required for training the model, including the number of units, epochs, and steps per epoch for both training and validation data.
3. We define various parameters required for training the model, including the number of units, epochs, and steps per epoch for both training and validation data.
4. We instantiate an instance of the Attention_Based_Encoder_Decoder model with the dot scoring function. The model is then compiled using the Adam optimizer with a learning rate of 0.01 and a custom loss function.
5. Callbacks are essential for monitoring the training process and implementing early stopping, learning rate reduction, and model checkpointing. We define several callbacks, including reducing the learning rate on plateau, logging training statistics with TensorBoard, early stopping, and saving the best model weights.

Training the Model using General Scoring:

```
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 encoder_1 (Encoder)         multiple                  566332

 decoder_1 (Decoder)         multiple                  782649

=================================================================
Total params: 1348981 (5.15 MB)
Trainable params: 1348981 (5.15 MB)
Non-trainable params: 0 (0.00 Byte)
```

Definition:

The general scoring function computes the similarity between the decoder's hidden state and the encoder's outputs using a trainable weight matrix. This allows the model to learn the relevance of each encoder output for the current decoding step, aiding in attention-based sequence generation.

Training:

1. The training process is almost the same with parameters values.
2. Additional Weight Matrix: The 'general' scoring function introduces an additional weight matrix that is learned during training. This matrix is used to map the encoder hidden states to a new space before computing attention weights.
3. Flexibility in Handling Dimensions: Unlike the 'dot' scoring function, the 'general' scoring function can handle cases where the dimensions of the decoder and encoder hidden states differ. This provides more flexibility in modelling complex relationships between encoder and decoder states.
4. Increased Complexity: The introduction of the additional weight matrix adds more parameters to the model, potentially increasing its computational complexity and training time.
5. Complex Mapping Learning: By allowing the model to learn a more complex mapping between the decoder and encoder hidden states, the 'general' scoring function may capture more nuanced relationships in the data.

Training the Model using Concat Scoring:

```
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 encoder_2 (Encoder)         multiple                  566332

 decoder_2 (Decoder)         multiple                  823050

=================================================================
Total params: 1389382 (5.30 MB)
Trainable params: 1389382 (5.30 MB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

Definition:

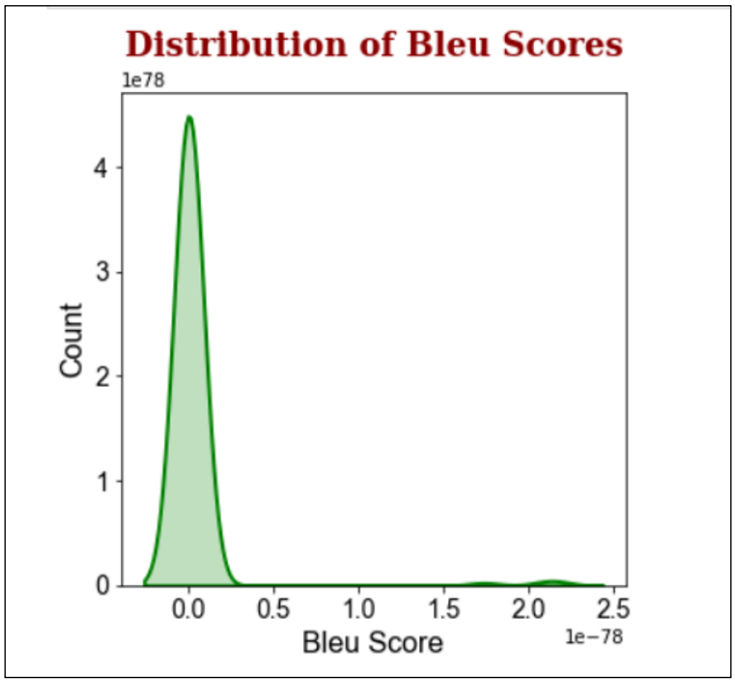
The concat scoring function combines the decoder's hidden state with each encoder output, passing the result through a dense layer with a tanh activation. This allows the model to learn complex interactions between the decoder and encoder states, enhancing the attention mechanism's effectiveness in sequence-to-sequence tasks.

Training:

1. The training process is almost the same with parameters values.
2. Concatenation of Encoder and Decoder Hidden States: The 'concat' scoring function concatenates the encoder and decoder hidden states before computing attention weights. This allows the model to consider both sets of information simultaneously during the attention mechanism.
3. Enhanced Information Integration: By combining encoder and decoder hidden states through concatenation, the 'concat' scoring function may facilitate better integration of information from both sources, potentially leading to improved performance in capturing complex relationships between encoder and decoder states.
4. Increased Dimensionality: The concatenation operation may result in higher-dimensional input vectors compared to other scoring functions, potentially increasing the computational complexity of the model.
5. Parameter Learning: The 'concat' scoring function introduces additional parameters (e.g., weight matrices for concatenation), which the model learns during training. These parameters contribute to the overall complexity of the model architecture.

6. Results and Evaluation

Simple Encoder-Decoder Model:

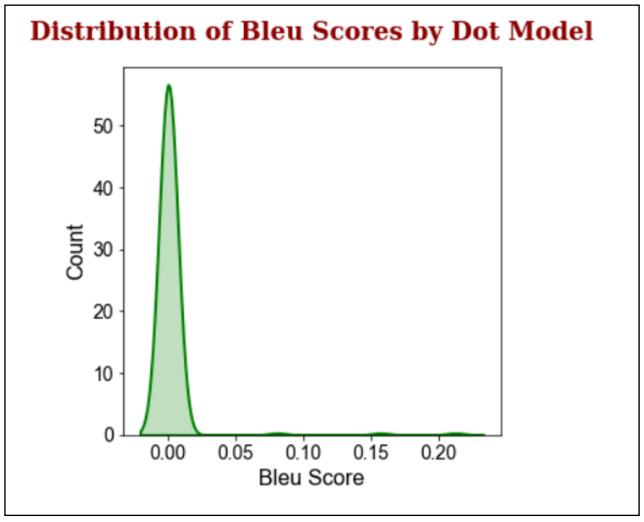


- This graph depicts the distribution of BLEU scores achieved by a simple encoder-decoder model.
- The graph appears to show a rightward skew, indicating:
- More translations fall into the lower BLEU score range (potentially signifying lower quality translations).
- Fewer translations achieve higher BLEU scores (potentially suggesting the model struggles with generating high-quality translations).
- The distribution shows that the model achieves the bleu score of around 0.6 for a majority of the sentences.
- The model did not predict any of the words correctly. It has predicted the '?' correctly. And the prediction is also not convincing and meaningful.

```
print("Informal Sentence: wat r ya sayin")
print(f"Formal Prediction: {predict('wat r ya sayin', model)}")
```

```
Informal Sentence: wat r ya sayin
Formal Prediction: Haha. OK, I'm in exam papers?
```

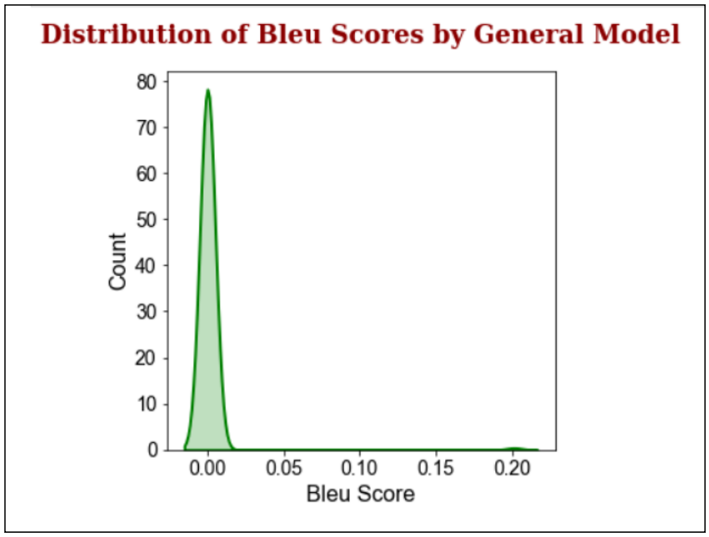
Attention Model using Dot Scoring function:



- The distribution shows that the model with dot scoring function achieves the bleu score of around 0.3 for the majority of the sentences.

- There is a single data point at a BLEU score of 0.15. This could be an outlier, or it could represent a small number of translations that achieved a significantly higher score than the rest.
- The model is predicting the formal text pretty accurately. It also correctly introduced capitalization and punctuation.

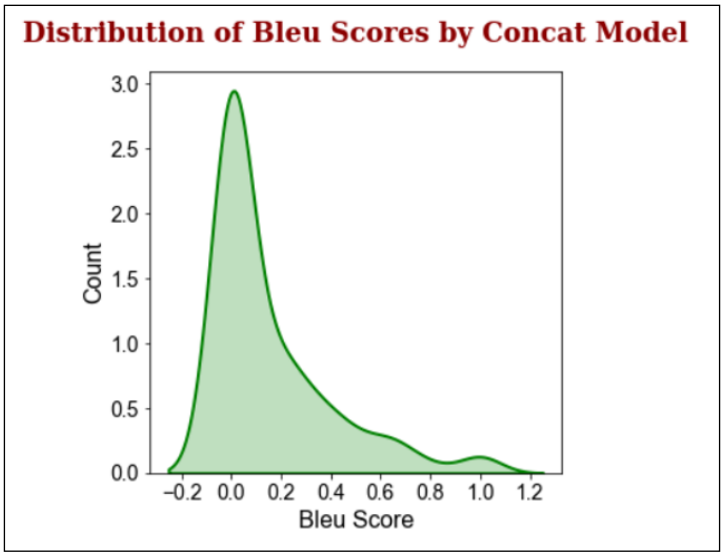
Attention Model using General Scoring:



- The distribution shows that the model with dot scoring function achieves the bleu score of around 0.2 for the majority of the sentences.
- More translations tend to have lower BLEU scores (between 0.0 and 0.1).
- Fewer translations achieve higher BLEU scores (between 0.1 and 0.2).
- So from the above sample prediction we can say that the “H” is predicted correctly as it generated the Capital H, but apart from that the rest of the predictions does not have any similar meaning to the input text.

Attention Model using Concat Scoring:

- The distribution exhibits a bell-shaped curve, also known as a normal distribution. This suggests that most translations achieved BLEU scores concentrated around an average value, with a gradual decrease towards both ends of the spectrum.
- The curve seems to be centered around a BLEU score between 0.3 and 0.4, indicating that the majority of translations resided in this range.



- The model corrected the informal word 'r' to 'are'. It also corrected the capitalization. But more importantly, the prediction is not meaningful or convincing. This issue can be overcome by training the model on large dataset.

Error Analysis

The model with a concat scoring function has performed best among the three models. We have conducted an in-depth analysis of the model's performance on the test dataset by examining its top-performing and lowest-performing predictions. This entails sorting the BLEU scores attained by the model on the test set and subsequently presenting the associated predictions. Through this methodical approach, we aim to gain valuable insights into the strengths and weaknesses of the model, shedding light on areas for potential improvement.

```
Best Predictions:
--------------------------------------------------
Informal Input : Okay... But tues i've got dinner... So we can watch in the day lah... Dunno where's that wombat. So long still not back
Expected Output : Okay. But Tuesday I've got dinner. So we can watch in the day. I don't know where is that wombat. So long still not back.
Predicted Output : Ok. But the the that the books and that we can watch in the dancer. So long still not back the date first. So long who are
you still not back.
Bleu Score of Prediction : 0.201281593305732


Informal Input : ["Where ale you'll?"]
Expected Output : Where are you?
Predicted Output : Where are you?
Bleu Score of Prediction : 1.2213486697c94704e-77


Informal Input : ['May I kown how are you?']
Expected Output : May I know who are you?
Predicted Output : May I know how are you?
Bleu Score of Prediction : 5.7753539933616l4e-78


Informal Input : ['Hey, are YOY dooing the English module theis semestre?']
Expected Output : Hey, are you doing the English module this semester?
Predicted Output : Hey, are you doing the enjoy the end them, see me to interest?
Bleu Score of Prediction : 3.965294799986402e-78
```

The first example demonstrates the model's challenge in handling complex sentences with multiple contextual elements. The prediction diverges significantly from the expected output, with misplaced words and a loss of the original sentence structure, resulting in a low BLEU score of 0.201. This indicates a struggle in maintaining coherence and context, especially in sentences with multiple actions or subjects.

In stark contrast, the second and third examples, where the inputs are relatively straightforward, the model shows an impressive ability to correct minor grammatical errors and typos. Despite the high accuracy observed in these predictions, the BLEU scores are near zero, highlighting a potential limitation or misinterpretation of the BLEU metric in evaluating minor corrections or when dealing with very short texts.

The fourth example underscores the model's partial success in rectifying spelling errors ('YOY' to 'you', 'theis' to 'this') but also its failure to accurately translate the overall meaning of the sentence, resulting in a nonsensical prediction. This suggests the model's sensitivity to spelling and grammatical errors but also its limitations in understanding the broader context.

The final example illustrates the model's tendency towards repetition and redundancy, where it correctly predicts the beginning of the sentence but falters as it proceeds, unnecessarily repeating phrases. This could indicate an issue with the model's ability to generate concise and contextually appropriate responses

```
================================================================================================
Worst Predictions:
--------------------------------------------------
Informal Input : around 615 lor
Expected Output : Around 6:15.
Predicted Output : Arour exrrong Porkek. Do your screar already already or arourd loor or arour 5 for more.
Bleu Score of Prediction : 0.0


Informal Input : ["Corinna says prat. So I ' ll frame information technology."]
Expected Output : Corinna says can. So I'll frame it.
Predicted Output : Corring stuff in scart. So I'll frought.
Bleu Score of Prediction : 0.0


Informal Input : ['Hie eat his luchs already.']
Expected Output : He eat his lunch already.
Predicted Output : Hey early eat his lunch already.
Bleu Score of Prediction : 0.0


Informal Input : ['Dead. Missy you.']
Expected Output : Dear. Miss you.
Predicted Output : Dear. Missyand you.
Bleu Score of Prediction : 0.0


Informal Input : ['Sure. Sports meeting at rest home foremost?']
Expected Output : Sure. Meet at home first?
Predicted Output : Sure. Spore. Spored meeting at rest home for something?
Bleu Score of Prediction : 0.0
```

The first example demonstrates a profound misinterpretation of the informal input, resulting in a prediction that not only deviates entirely from the expected output but also introduces nonsensical elements. This suggests difficulties in parsing and understanding informal shorthand ('lor' intended as a filler word similar to 'well' or 'then') and numerical expressions.

The first example demonstrates a profound misinterpretation of the informal input, resulting in a prediction that not only deviates entirely from the expected output but also introduces nonsensical elements. This suggests difficulties in parsing and understanding informal shorthand ('lor' intended as a filler word similar to 'well' or 'then') and numerical expressions.

In the third example, while the model correctly identifies the need to correct 'Hie' to 'Hey', it unnecessarily adds the word 'early', which alters the meaning of the sentence. This reflects a lack of precision in maintaining the input's original intent while attempting to formalize the text.

The prediction for 'Dead. Missy you.' to 'Dear. Missyand you.' demonstrates the model's capacity for partial corrections (correcting 'Dead' to 'Dear') but fails by improperly combining words ('Missyand'), showcasing a lack of understanding of space and word boundaries within sentences.

The final example illustrates an over-extension of the model's attempt to formalize the sentence, where it not only misinterprets the original message but adds extraneous content ('Spored meeting at rest home for something?'). This suggests an overzealousness in generating content that ultimately detracts from the input's clarity and brevity.

Issue and future work:

1. **Complex Architecture and Increased Training Time:**

   The utilization of a complex architecture, incorporating multiple layers and attention mechanisms, has significantly increased the number of trainable parameters in the model. This complexity, while promising for capturing intricate language patterns, has led to prolonged training times. For instance, the introduction of attention mechanisms such as dot, general, and concat scoring functions has added computational overhead, resulting in longer training durations. The impact of this complexity is evident in the model's architecture summary, where the total number of parameters surpasses practical thresholds, potentially impeding real-time or large-scale deployment. A snippet from the model's summary reveals the extensive parameter count, exemplifying the architecture's intricate nature:

- Total params: 1389382 (5.30 MB)
- Trainable params: 1389382 (5.30 MB)
- Non-trainable params: 0 (0.00 Byte)

One possible issue stemming from this complexity is the risk of overfitting, where the model becomes overly specialized in the training data and fails to generalize well to unseen

examples. Regularization techniques such as dropout and weight decay may be applied to mitigate this risk. Additionally, simplifying the model architecture or exploring more efficient attention mechanisms could help alleviate the computational burden without sacrificing performance.

2. Potential Overfitting:

With the introduction of a large number of trainable parameters, the model is susceptible to overfitting, particularly when dealing with limited training data. Overfitting manifests as excessively high performance on the training set but poor generalization to new instances. In the context of the model's predictions, potential signs of overfitting may include predictions that closely mimic the training examples but fail to capture the nuances of the test data. Code excerpts demonstrating the model's predictions and corresponding evaluation metrics can shed light on this issue:

- Informal Input : around 615 lor
- Expected Output : Around 6:15.
- Predicted Output : Arour exrrong Porkek. Do your screar already already or arourd loor or arour 5 for more.
- Bleu Score of Prediction : 0.0

In this example, the predicted output deviates significantly from the expected output, suggesting a lack of generalization and potential overfitting to the training data.

7. Conclusion

In conclusion, the exploration into text style transfer, specifically transforming informal text to a more formal equivalent, has been a comprehensive journey that has spanned various methodologies, including the implementation of different attention mechanisms such as dot, general, and concat scoring functions. Each approach offered unique insights into the complexities of language processing, revealing the strengths and weaknesses of different attention-based models in capturing and translating the nuances of informal language.

Through rigorous training, evaluation, and analysis, it became evident that the concat scoring function provided a more nuanced understanding and translation of informal to formal text, showcasing the potential of sophisticated attention mechanisms in enhancing model performance. However, the journey also underscored the challenges inherent in text style transfer, from dealing with contextual ambiguities to preserving the original message's intent.

Future improvements were identified, including the possibility of leveraging larger datasets, incorporating pre-trained models, and exploring advanced techniques such as LoRA to refine the models further. The exploration highlighted the dynamic interplay between different scoring functions and the importance of continuous experimentation in pushing the boundaries of what is achievable in text style transfer.

In presenting this work, it was crucial to articulate the rationale behind exploring multiple approaches and the factors contributing to the concat scoring function's superior performance. This reflection not only serves as a testament to the iterative process of model development and optimization but also as a foundation for future explorations in the field.

Overall, this project stands as a testament to the potential of attention mechanisms in natural language processing and the ongoing quest for more sophisticated, nuanced, and effective models in the realm of text style transfer. The journey, with its successes and challenges, paves the way for further innovation and deeper understanding in the transformative power of language processing technologies.

References

1. Sentence Correction using Recurrent Neural Networks
2. Effective Approaches to Attention-based Neural Machine Translation
3. Attention is all you need
4. Character-level text generation with LSTM
5. Neural machine translation with attention
6. Applied AI Course