ECE-60146 HW10

Ian Noronha

April 2025

1 Loss Graphs

I ran the two scripts seq2seq with transformerFG.py seq2seq with transformerPreLN.py with the default hyperparameters for the specified 40 epochs. I added the function to calculate the Leveshtein distance in the run_code_for_evaluating_Transformer function of DLStudio to print the outputs of the Leveshtein distance on the cleaned gt and predicted outputs

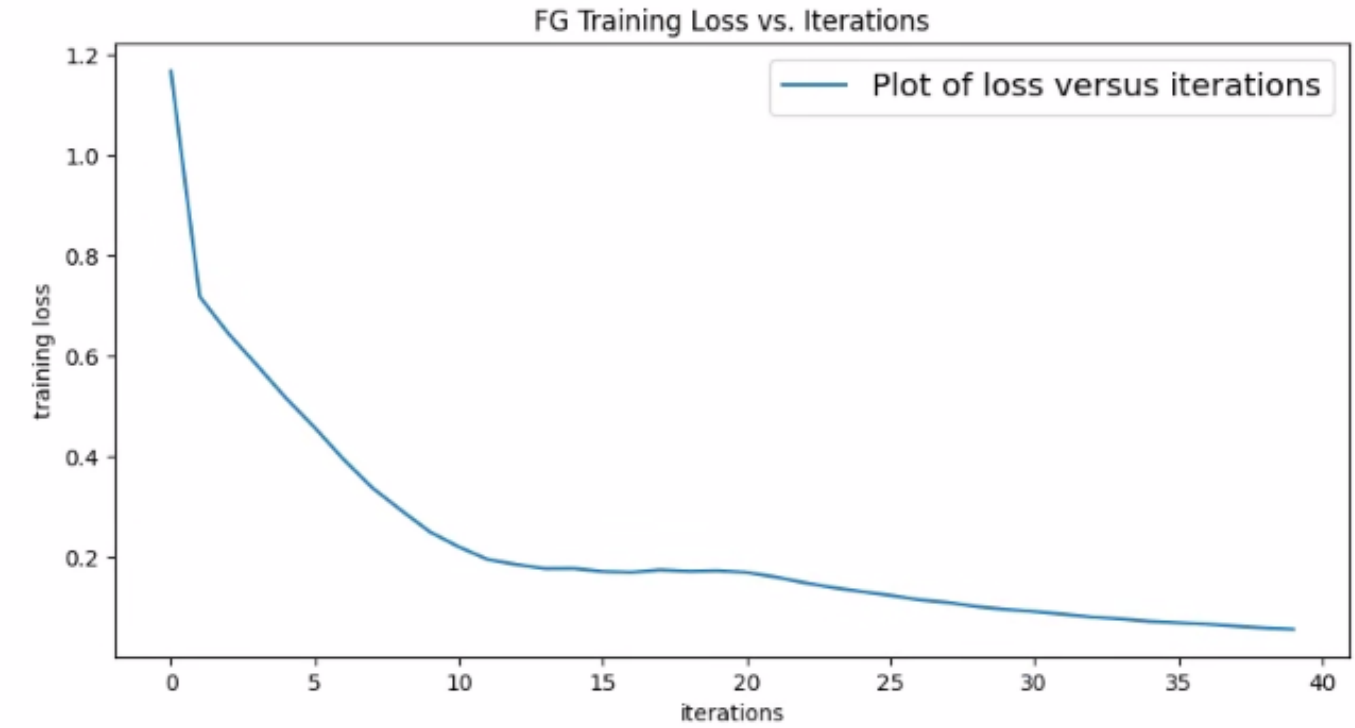

Figure 1: FG training loss

Figure 2: PreLN training loss

2 Output and Levenshtein distance

```
The input sentence pair:  ['SOS why is this room locked EOS'] ['SOS por qué está con llave esta habitación EOS']
The translation produced by TransformerFG:  EOS por qué está con con esta habitación EOS EOS
The ground truth clean:  por qué está con con esta habitación
The translation produced by TransformerFG clean:  por qué está con con esta habitación
Levenshtein distance:  5
```

Figure 3: FG example with cleaned string and Levenshtein distance

```
The input sentence pair:  ['SOS no one lives in this building EOS'] ['SOS nadie vive en este edificio EOS']
The translation produced by TransformerFG:  EOS nadie vive en este edificio EOS EOS EOS EOS
The ground truth clean:  nadie vive en este edificio
The translation produced by TransformerFG clean:  nadie vive en este edificio
Levenshtein distance:  0
```

Figure 4: FG example with cleaned string and Levenshtein distance

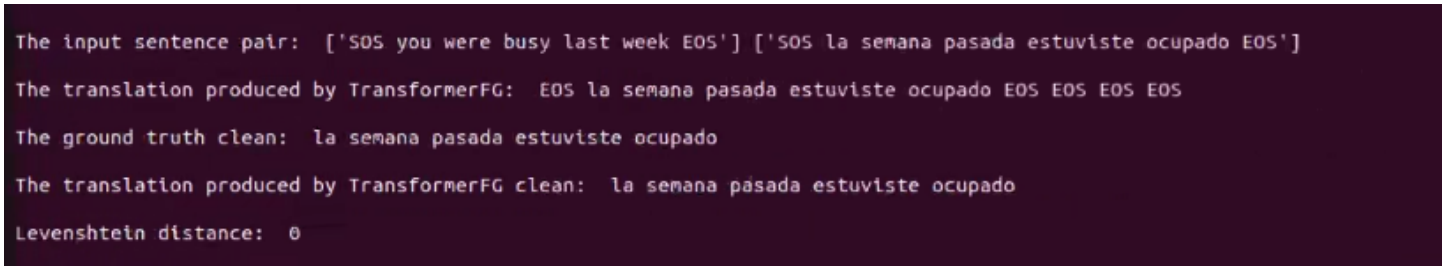

Figure 5: FG example with cleaned string and Levenshtein distance

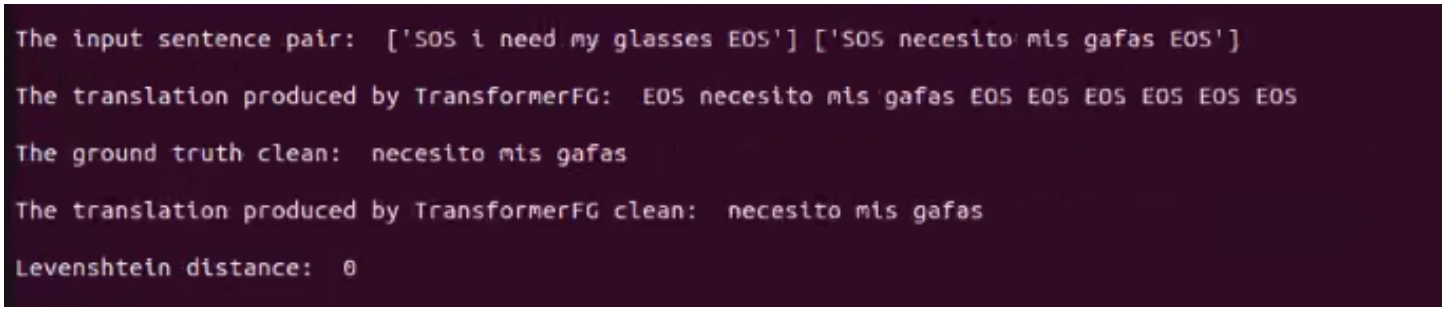

Figure 6: FG example with cleaned string and Levenshtein distance

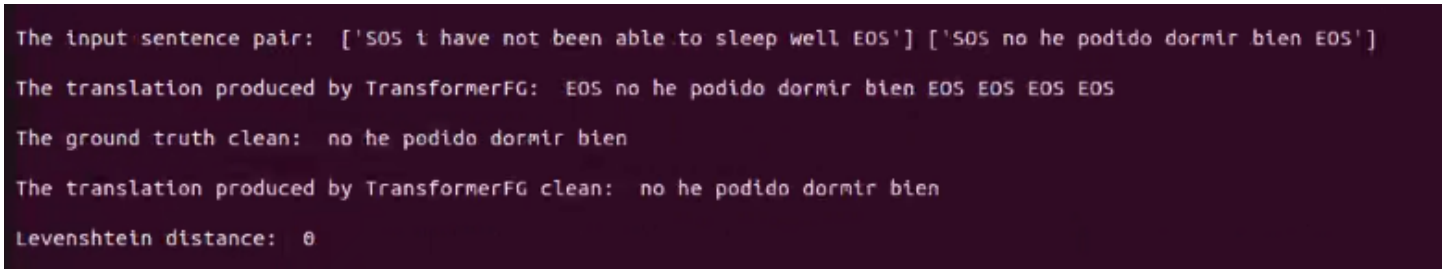

Figure 7: FG example with cleaned string and Levenshtein distance

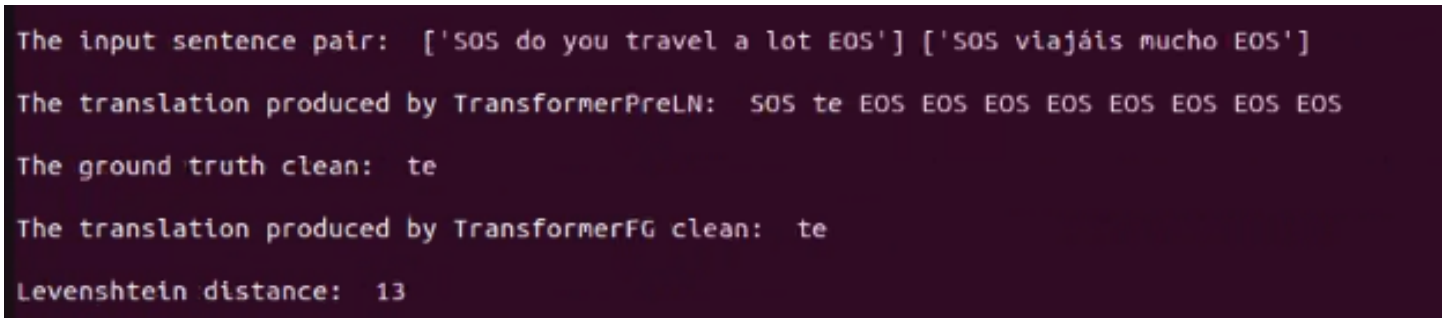

Figure 8: PreLN example with cleaned string and Levenshtein distance

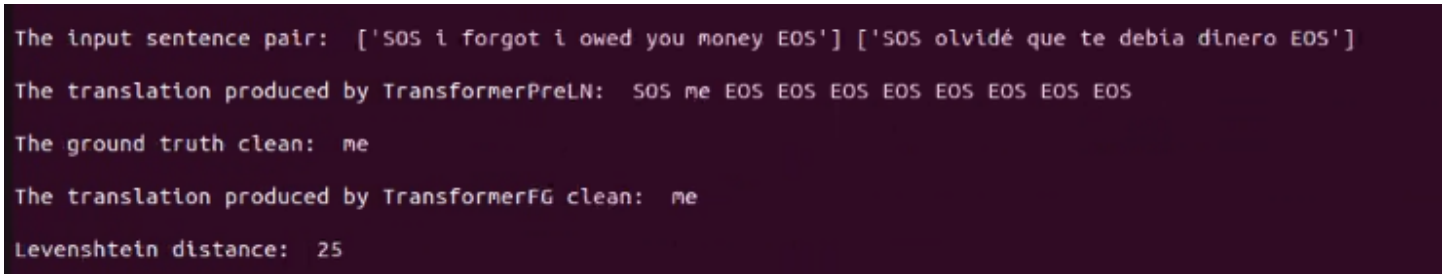

Figure 9: PreLN example with cleaned string and Levenshtein distance

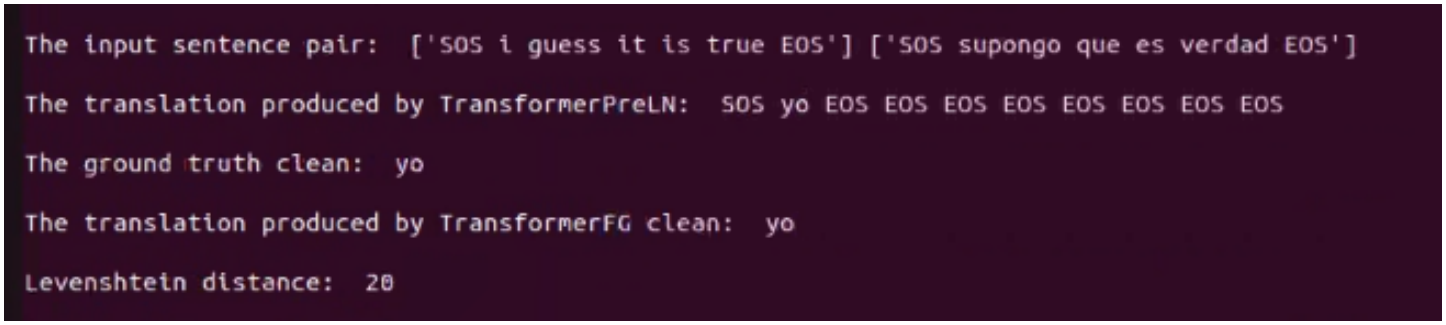

Figure 10: PreLN example with cleaned string and Levenshtein distance

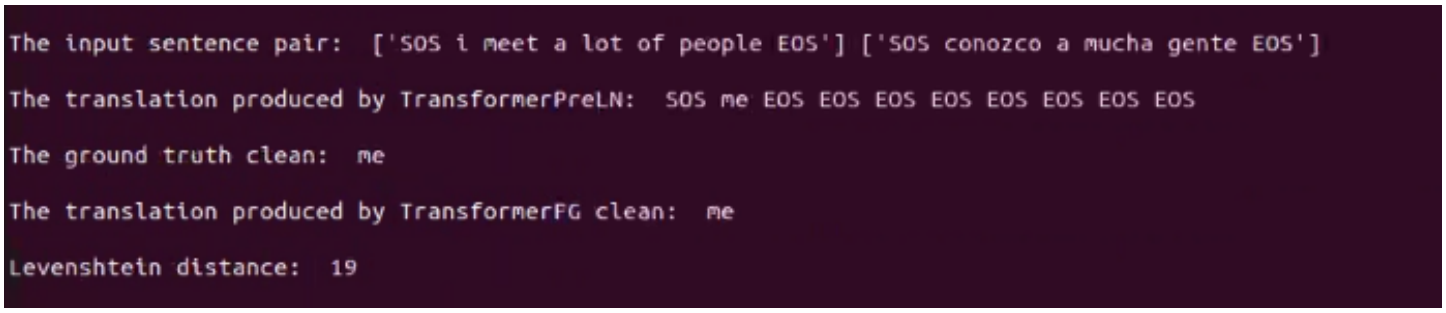

Figure 11: PreLN example with cleaned string and Levenshtein distance

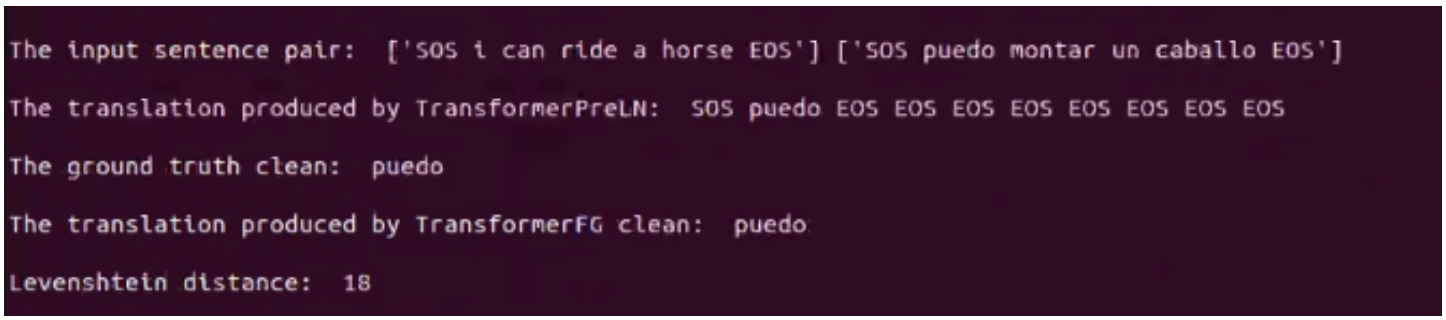

Figure 12: PreLN example with cleaned string and Levenshtein distance

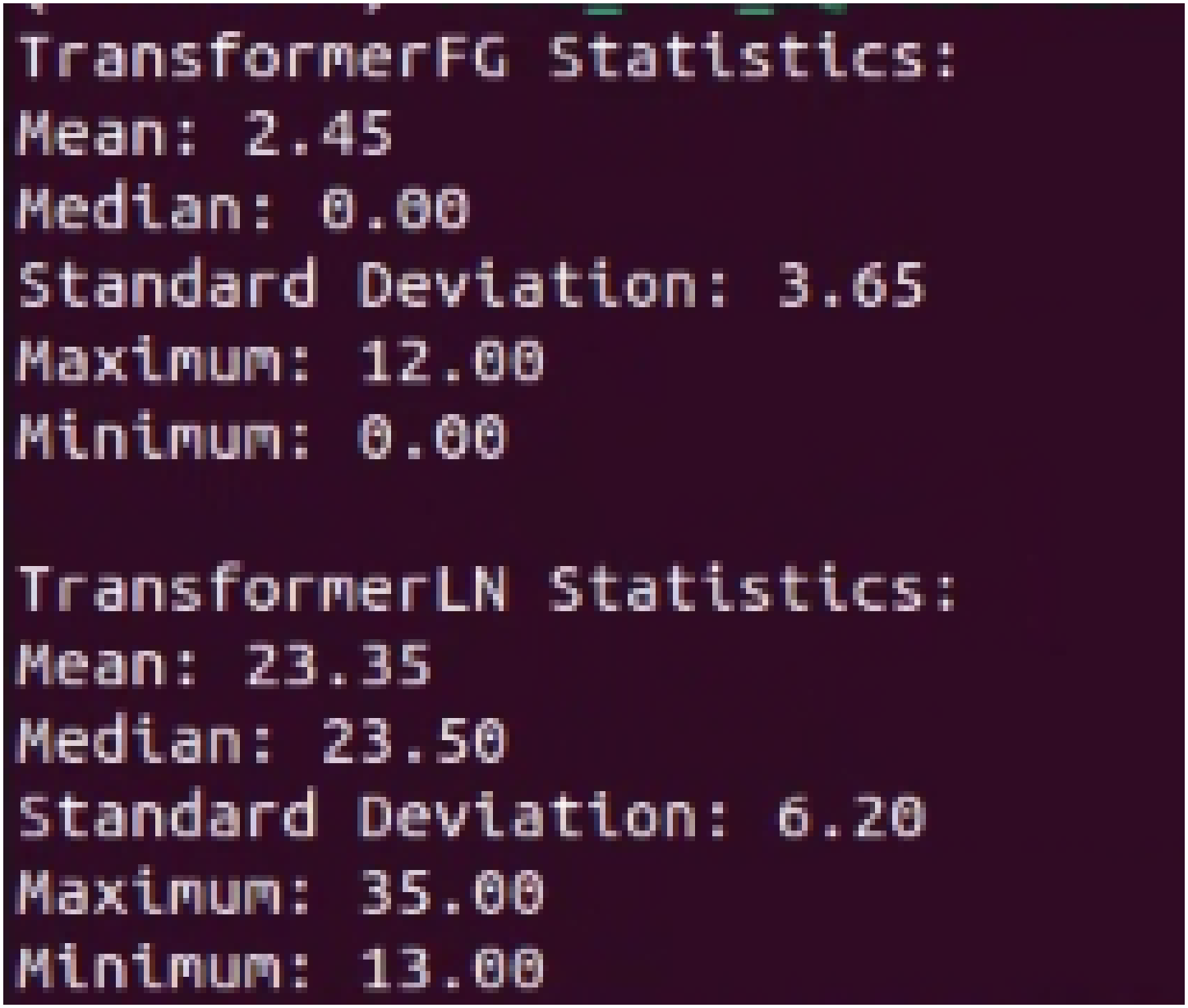

Figure 13: 2x5 Table of metrics

3 Discussion

The mean, and standard deviation of the transformerFG is lower indicating that the model is more accurate and that its consistency is higher than the PreLN model. The median of the model is 0 indicating at least half the translations were completely accurate. PreLN Normalizes the input before it enters the selfattention layer so it will take longer to converge. In thoery it is supposed to perform better if the training is allowed to go on for a longer duration of time.

```
    def levenshtein_distance(self, str1, str2):
        if len(str1) < len(str2):
            str1, str2 = str2, str1  # Ensure str1 is the longer string
        len_str1, len_str2 = len(str1), len(str2)
        # Initialize two rows for dynamic programming
        previous_row = list(range(len_str2 + 1))
        current_row = [0] * (len_str2 + 1)

        for i in range(1, len_str1 + 1):
            current_row[0] = i
            for j in range(1, len_str2 + 1):
                cost = 0 if str1[i - 1] == str2[j - 1] else 1
                current_row[j] = min(
                    previous_row[j] + 1,      # Deletion
                    current_row[j - 1] + 1,    # Insertion
                    previous_row[j - 1] + cost # Substitution
```

```
                )
            previous_row, current_row = current_row, previous_row

        return previous_row[-1]


    def run_code_for_evaluating_TransformerFG(self, master_encoder, master_decoder, result_file=None):
        """
        The main difference between the training code shown in the previous function and the
        evaluation code shown here is with regard to the input to MasterDecoder and how we
        process its output. As shown in the previous function, for the training loop, the
        input to MasterDecoder consists of the both the target sentence and the output of
        the MasterEncoder for the source sentence.  However, at inference time (that is, in
        the evaluation loop shown below), the target sentence at the input to the MasterDecoder
        is replaced by an encoding of a "starter stub" output sentence as defined in line (B).
        The main message conveyed by the stub in line (B) is that we want to start the
        translation with the first word of the output as being the token "SOS".  The encoding
        for the stub is generated in lines (F) and (G).

        The second significant difference between the training and the testing code is
        with regard to how we process the output of the MasterDecoder.  As you will recall
        from the docstring associated with MasterDecoder, it returns two things: (1) the
        predicted log probabilities (logprob) over the target vocabulary for every word
        position in the target language; and (2) for each target-language word position,
        the word_vocab_index at which the logprob is maximum.  The loss calculation in
        the training code was based on the former.  ON the other hand, as shown in line (H)
        below, it is the latter that lets us do the the translations in the target words
        in line (I).
        """
        master_encoder.load_state_dict(torch.load(self.dl_studio.path_saved_model['encoder_FG']))
        master_decoder.load_state_dict(torch.load(self.dl_studio.path_saved_model['decoder_FG']))
        embeddings_generator_en = self.EmbeddingsGenerator(self, 'en', self.embedding_size)
        embeddings_generator_es = self.EmbeddingsGenerator(self, 'es', self.embedding_size)

        ↪  embeddings_generator_en.load_state_dict(torch.load(self.dl_studio.path_saved_model['embeddings_generator_en_

        ↪  embeddings_generator_es.load_state_dict(torch.load(self.dl_studio.path_saved_model['embeddings_generator_es_
        master_encoder.to(self.dl_studio.device)
        master_decoder.to(self.dl_studio.device)
        embeddings_generator_en.to(self.dl_studio.device)
        embeddings_generator_es.to(self.dl_studio.device)
        debug = False
        FILE = open("translations_with_FG_" + str(self.dl_studio.epochs) + ".txt", 'w')
        with torch.no_grad():
            for iter in range(20):
                starter_stub_for_translation = ['SOS', 'EOS', 'EOS', 'EOS', 'EOS', 'EOS', 'EOS', 'EOS', 'EOS',
                ↪  'EOS']      ## (A)
                batched_pairs = random.sample(self.training_corpus, 1)
                ↪  ## (B)
                source_sentences = [pair[0] for pair in batched_pairs]
                target_sentences = [pair[1] for pair in batched_pairs]
                if debug:
                    print("\n\nsource sentences: ", source_sentences)
                    print("\ntarget sentences: ", target_sentences)
                en_sent_ints =  self.sentence_with_words_to_ints(source_sentences,
                ↪  'en').to(self.dl_studio.device)         ## (C)
                if debug:
                    es_sent_ints =  self.sentence_with_words_to_ints(target_sentences, 'es')
                    print("\n\nsource sentence tensor: ", en_sent_ints)
                    print("\n\ntarget sentence tensor: ", es_sent_ints)
                en_sentence_tensor = embeddings_generator_en(en_sent_ints).float()
                master_encoder_output = master_encoder( en_sentence_tensor )
                ↪  ## (E)
                starter_stub_as_ints = self.sentence_with_words_to_ints(
                                       [" ".join(starter_stub_for_translation)], 'es').to(self.dl_studio.device)
                                       ↪  ## (F)
                starter_stub_tensor = embeddings_generator_es(starter_stub_as_ints).float()
                ↪  ## (G)
```

```
                _, predicted_word_index_values = master_decoder(starter_stub_tensor, master_encoder_output)
                ↪  ## (H)
                predicted_word_index_values = predicted_word_index_values[0].unsqueeze(1)
                decoded_words = [self.es_index_2_word[predicted_word_index_values[di].item()]
                                                                    for di in range(self.max_seq_length)]
                                                                    ↪  ## (I)
                output_sentence = " ".join(decoded_words)
                print("\n\n\nThe input sentence pair: ", source_sentences, target_sentences)
                print("\nThe translation produced by TransformerFG: ", output_sentence)
                FILE.write("\n\n\nThe input sentence pair: %s    %s" % (source_sentences, target_sentences))
                FILE.write("\nThe translation produced by TransformerFG:  %s" % output_sentence)
                decoded_words = [word for word in decoded_words if word not in ['SOS', 'EOS']]
                target_tokens = target_sentences[0].split()
                output_sentence = " ".join(decoded_words)
                target_tokens = [word for word in target_tokens if word not in ['SOS', 'EOS']]
                ground_truth = " ".join(target_tokens)
                lev_distance = self.levenshtein_distance(output_sentence, ground_truth)
                print("\nThe ground truth clean: ", output_sentence)
                print("\nThe translation produced by TransformerFG clean: ", output_sentence)
                print("\nLevenshtein distance: ", lev_distance)
```

```
def levenshtein_distance(self, str1, str2):
        if len(str1) < len(str2):
            str1, str2 = str2, str1  # Ensure str1 is the longer string
        len_str1, len_str2 = len(str1), len(str2)
        # Initialize two rows for dynamic programming
        previous_row = list(range(len_str2 + 1))
        current_row = [0] * (len_str2 + 1)

        for i in range(1, len_str1 + 1):
            current_row[0] = i
            for j in range(1, len_str2 + 1):
                cost = 0 if str1[i - 1] == str2[j - 1] else 1
                current_row[j] = min(
                    previous_row[j] + 1,      # Deletion
                    current_row[j - 1] + 1,    # Insertion
                    previous_row[j - 1] + cost # Substitution
                )
            previous_row, current_row = current_row, previous_row

        return previous_row[-1]

    def run_code_for_evaluating_TransformerPreLN(self, master_encoder, master_decoder):
        """
        The main difference between the training code shown in the previous function and the
        evaluation code shown here is with regard to the input to MasterDecoder and how we process
        its output. As shown in the previous function, for the training loop, the input to
        MasterDecoder consists of the both the target sentence and the output of the MasterEncoder
        for the source sentence.  However, at inference time (that is, in the evaluation loop shown
        below), the target sentence at the input to the MasterDecoder is replaced by an encoding of
        a "starter stub" output sentence as defined in line (B).  The main message conveyed by the
        stub in line (B) is that we want to start the translation with the first word of the output
        as being the token "SOS".  The encoding for the stub is generated in lines (F) and (G).

        The second significant difference between the training and the testing code is with regard
        to how we process the output of the MasterDecoder.  As you will recall from the docstring
        associated with MasterDecoder, it returns two things: (1) the predicted log probabilities
        (logprob) over the target vocabulary for every word position in the target language; and
        (2) for each target-language word position, the word_vocab_index at which the logprob is
        maximum.  The loss calculation in the training code was based on the former.  ON the other
        hand, as shown in line (H) below, it is the latter that lets us do the the translations in
        the target words in line (I).
        """
        master_encoder.load_state_dict(torch.load(self.dl_studio.path_saved_model['encoder_PreLN']))
        master_decoder.load_state_dict(torch.load(self.dl_studio.path_saved_model['decoder_PreLN']))
        embeddings_generator_en = self.EmbeddingsGenerator(self, 'en', self.embedding_size)
```

```
        embeddings_generator_es = self.EmbeddingsGenerator(self, 'es', self.embedding_size)

        embeddings_generator_en.load_state_dict(torch.load(self.dl_studio.path_saved_model['embeddings_generator_en_

        embeddings_generator_es.load_state_dict(torch.load(self.dl_studio.path_saved_model['embeddings_generator_es_
    master_encoder.to(self.dl_studio.device)
    master_decoder.to(self.dl_studio.device)
    embeddings_generator_en.to(self.dl_studio.device)
    embeddings_generator_es.to(self.dl_studio.device)
    debug = False
    FILE = open("translations_with_PreLN_" + str(self.dl_studio.epochs) + ".txt", 'w')
    with torch.no_grad():
        for iter in range(20):
            starter_stub_for_translation = ['SOS', 'EOS', 'EOS', 'EOS', 'EOS', 'EOS', 'EOS', 'EOS', 'EOS',
            'EOS']       ## (A)
            batched_pairs = random.sample(self.training_corpus, 1)
            ## (B)
            source_sentences = [pair[0] for pair in batched_pairs]
            target_sentences = [pair[1] for pair in batched_pairs]
            if debug:
                print("\n\nsource sentences: ", source_sentences)
                print("\ntarget sentences: ", target_sentences)
            en_sent_ints =  self.sentence_with_words_to_ints(source_sentences, 'en')
            ## (C)
            if debug:
                es_sent_ints =  self.sentence_with_words_to_ints(target_sentences, 'es')
                print("\n\nsource sentence tensor: ", en_sent_ints)
                print("\n\ntarget sentence tensor: ", es_sent_ints)
            en_sentence_tensor = embeddings_generator_en(en_sent_ints).float()
            master_encoder_output = master_encoder( en_sentence_tensor )
            ## (E)
            starter_stub_as_ints = self.sentence_with_words_to_ints(
                                        [" ".join(starter_stub_for_translation)],
                                        'es').to(self.dl_studio.device)    ## (F)
            starter_stub_tensor = embeddings_generator_es(starter_stub_as_ints).float()
            ## (G)
            _, predicted_word_index_values = master_decoder(starter_stub_tensor, master_encoder_output)
            ## (H)
            predicted_word_index_values = predicted_word_index_values[0].unsqueeze(1)
            decoded_words = [self.es_index_2_word[predicted_word_index_values[di].item()]
                                                                    for di in range(self.max_seq_length)]
                                                                    ## (I)
            output_sentence = " ".join(decoded_words)
            print("\n\n\nThe input sentence pair: ", source_sentences, target_sentences)
            print("\nThe translation produced by TransformerPreLN: ", output_sentence)
            FILE.write("\n\n\nThe input sentence pair: %s    %s" % (source_sentences, target_sentences))
            FILE.write("\nThe translation produced by TransformerPreLN:  %s" % output_sentence)
            decoded_words = [word for word in decoded_words if word not in ['SOS', 'EOS']]
            output_sentence = " ".join(decoded_words)
            target_tokens = target_sentences[0].split()
            target_tokens = [word for word in target_tokens if word not in ['SOS', 'EOS']]
            ground_truth = " ".join(target_tokens)
            lev_distance = self.levenshtein_distance(output_sentence, ground_truth)
            print("\nThe ground truth clean: ", output_sentence)
            print("\nThe translation produced by TransformerPreLN clean: ", output_sentence)
            print("\nLevenshtein distance: ", lev_distance)
```
