Assignment 4

chadha aouina, mail: chadhausi.ch

Deadline: 17 January 2024 - 10.00am

Conversational model with Transformers

1 Data (40 pts)

1. **movie_lines.txt:**

   - This file contains the actual lines of dialogue from various movies.
   - Each line is uniquely identified by a code (e.g., L1045).
   - The format includes a series of identifiers separated by "+++$$$+++":
     - User ID (e.g., u0, u2)
     - Movie ID (e.g., m0)
     - The name of the character speaking the line (e.g., BIANCA, CAMERON)
     - The dialogue text itself.
   - The structured format delineates these different elements clearly, facilitating parsing and analysis.

   movie_conversations.txt:

   - This file documents the structure of conversations, referencing the lines from movie_lines.txt.
   - Each entry starts with user IDs (e.g., u0, u2) and a movie ID (m0), similar to the movie_lines.txt format.
   - Following these identifiers, there's a list of line codes enclosed in square brackets.
   - These codes correspond to the lines in movie_lines.txt, indicating the flow of conversation between characters.

2. - **Function:** `create-sentence-pairs`
   - **Purpose:** This function takes the dialogue lines and their corresponding conversations from the Cornell Movie Dialogues Corpus and creates pairs of sentences.
   - **Implementation:**
     - It iterates through each conversation (a list of line IDs).

  - For each conversation, it pairs sequential lines to form dialogue pairs. This includes creating overlapping pairs for conversations with more than two lines. For instance, in a conversation with four lines (s1, s2, s3, s4), it generates the pairs (s1, s2), (s2, s3), and (s3, s4).

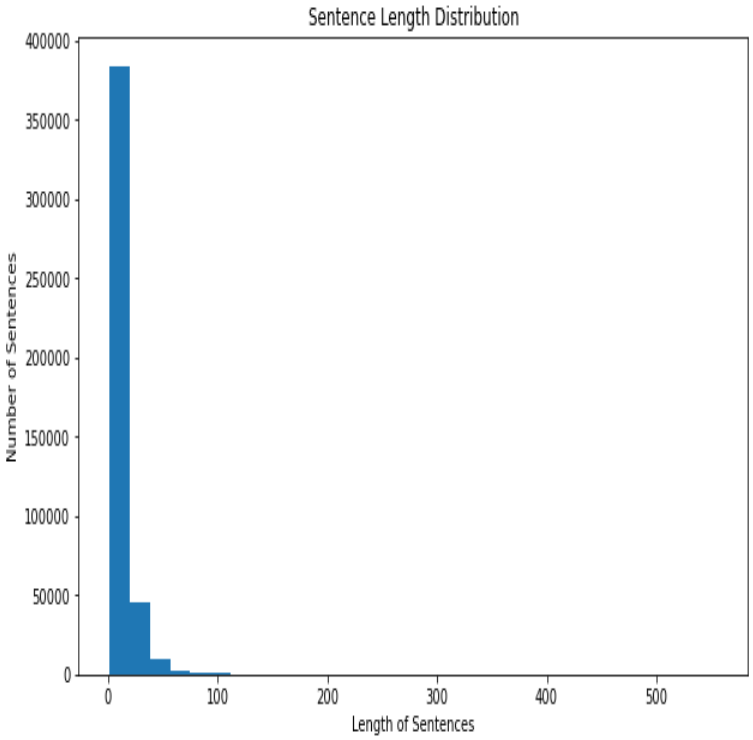
- **Commentary on the Choice:**
  - **Advantages:**
    - *Richer Contextual Data:* The approach offers a more detailed context for each dialogue segment.
    - *Better Data Utilization:* Maximizes the use of available data.
    - *Versatility for Different Applications:* Applicable to a variety of use cases in dialogue processing and analysis.
  - **Potential Concerns:**
    - *Data Redundancy:* The approach might lead to repetitive data in the dataset.
    - *Complex Conversations Handling:* In complex dialogues with multiple speakers or non-linear conversation threads, this method may not accurately capture the true conversational dynamics.

3. - **Tokenization Strategy:**
  - **Use of clear_punctuation Function:**
    - This function handles punctuation by creating separate tokens for punctuation symbols (?, ., !) and removing other types of punctuation.
  - **Appending Special Tokens:**
    - <EOS>(*End Of Sentence*): Appended to the end of each sentence.
      - *Purpose:* Signifies the end of a sentence, crucial in sequence modeling tasks for helping the model to understand sentence boundaries.
    - <SOS>(*Start Of Sentence*): Appended only at the beginning of each answer.
      - *Purpose:* Useful in sequence-to-sequence models, particularly in generation tasks like dialogue systems, where it signals the beginning of a response.
- **Commentary on Special Tokens (<SOS>, <EOS>):**
  - <EOS>: Helps the model differentiate sentences and understand their limits, especially important in dialogues where multiple sentences might be spoken by one character in a single turn.
  - <SOS>: Essential for response generation tasks, providing a clear starting point for generating a response, improving the model's ability to generate coherent and contextually relevant answers.

4. - **Choice of max_length:**

- The value of `max_length` was chosen to balance retaining enough data for training while removing overly long sentences that might be outliers or less useful.
- Based on the histogram analysis, the vast majority of sentences are short, with a steep drop-off in frequency as sentence length increases. A threshold is considered around the point where the histogram bars become very short.
- A threshold of 30 is selected as it is likely to retain a substantial portion of the data, effectively encompassing most of the typical sentence lengths observed in the dataset.

- **Filtering Pairs:**
  - After deciding on the `max_length`, pairs where either sentence exceeds this threshold are filtered out.
  - This step ensures that the training data is more uniform in length, potentially improving the efficiency and performance of the model.



5. sentences were saved in pickle format
6. The Counter class from the collections module is used to count the occurrences of each word in the corpus. The filtering step eliminates any sentence pair where at least one word in either sentence is not in the set of common words.
7. list was saved in pickle

8. To randomly sample a subset of sentences from our corpus, i used Python's random module, which includes a function called sample for this purpose. This function allows us to select a specified number of unique elements from a list.

    - **Choice: Generating 60,000 sentences for the initial dataset.**
        - *Comment:* This choice is beneficial for creating a rich and diverse dataset. A larger initial set offers a broad spectrum of data, which is crucial for training robust models.
    - **Choice: Randomly sampling 10,000 sentences from the larger set.**
        - *Comment on Efficiency:* Training on a smaller subset reduces computational demands and speeds up the training process. This is particularly important when resources are limited or when iterative testing and development are required.
        - *Comment on Reproducibility:* The decision to fix a seed before sampling ensures that the experiments can be reproduced.

9. To complete the Vocabulary class, i filled in the add-word and add-sentence methods. The add-word method should add a new word to the word2index dictionary and update the index2word dictionary. The add-sentence method should split the sentence into words and add each word to the vocabulary using the add-word method. n-words: I introduced a new instance variable n-words to keep track of the total number of unique words in the vocabulary. This helps to assign a unique index to each new word. add-word Method: This method now checks if a word is not already in the word2index dictionary. If it's not, the method adds the word with the next available index and increments n-words. add-sentence Method: This method splits the sentence into words and adds each word to the vocabulary by calling add-word. Dictionary Updates: Both word2index and index2word dictionaries are updated in the add-word method to ensure they are always synchronized. Initialization of n-words: It's set to 3 to account for the special tokens that are already in the dictionaries. By making these changes, the Vocabulary class now has the functionality to process sentences and add new words to the vocabulary, while ensuring that each word has a unique index. This is crucial for later converting sentences into tensors of word indices, which can be used as input to machine learning models.

10. To modify the Dataset class i implemented the –init–, –len–, and –getitem– methods. Additionally, for efficient batch processing, we need a collate-fn that will pad the sequences in each batch to the same length.

    (a) **–init–**: Initializes the Dataset object with a Vocabulary instance and a list of sentence pairs.

    (b) **–len–**: Returns the number of pairs in the dataset.

    (c) **–getitem–**: Converts a pair of sentences (at the given index ix) into tensors of word indices. It assumes that the sentences have already been tokenized and that each word can be converted into an index using the Vocabulary instance.

(d) **collate-fn**: This function is used by the DataLoader to merge a list of samples into a batch. It pads the sequences so that all sequences in a batch have the same length. The padding value is set to the index of <PAD>in the vocabulary, which is 0.

(e) **DataLoader**: The DataLoader object is created using the dataset and the custom collate-fn. The batch-size parameter controls how many items are in each batch, and collate-fn is the function that will pad the batches.

When the DataLoader is iterated over, it will provide batches of padded sequences ready to be fed into a model. This setup is essential for batch training since models typically require input tensors to be of the same shape. The collate-fn ensures that despite sentences being of different lengths, they are padded to the maximum length within each batch for uniformity.

2 Model & Tools for training (35 pts)

1. The PositionalEncoding module integrates sinusoidal encoding for adding sequence information to the Transformer model. Its use of sine and cosine functions at different frequencies aids the model in learning relative word positions. The implementation scales to handle variable input lengths and incorporates dropout for regularization. Additionally, the assert statement ensures the input sequence length does not exceed the specified maximum length, providing robustness in handling different data inputs.

2. To complete the TransformerModel class, i implemented the initialization of the various components within the transformer architecture.

   (a) **Embedding Layer**: Maps token indices to dense vectors of size d-model.

   (b) **Positional Encoding**: Adds positional context to the input embeddings.

   (c) **Transformer Layer**: The core transformer which processes the input.

   (d) **Output Linear Layer**: A linear layer that projects from the hidden space back to the vocab size (to generate probabilities for each token).

   (e) **self.embedding**: Embedding layer to convert input word indices into embeddings.

   (f) **self.pos-encoder**: The positional encoding layer adds information about the sequence position of each token.

   (g) **self.transformer**: A PyTorch transformer layer, which is set to have batch-first=True to accept input data in the shape (batch-size, sequence-length, features).

   (h) **self.linear**: A fully connected layer to project the transformer outputs back into the vocabulary space.

(i) **self.pad-id**: Stored for use when creating padding masks.

(j) **self.vocab-size**: Saved for use in the output layer.

(k) **self.init-weights()**: A method is called to initialize the weights of the embedding and linear layers.

The init-weights method initializes the weights of the embedding and linear layers with uniform random weights.

By filling in the `--init--` method with the necessary components, the TransformerModel is now ready to be instantiated and used for training or inference.

3. To complete the create-padding-mask function in the TransformerModel class, i implemented logic that creates a boolean mask for the <PAD>tokens. This function should take a tensor as input and return a boolean tensor where the positions of <PAD>tokens are marked as True, and all other positions are False. In this function:

   The input x is a tensor containing token indices. The pad-id parameter is the index of the <PAD>token in the vocabulary, which is 0 by default. The function compares each element in x to pad-id to create a boolean mask. Wherever x equals pad-id, the mask will have True; otherwise, it will have False. This mask is then returned and can be used in the transformer model to ignore <PAD>tokens during processing. This mask allows the model to disregard padding tokens during attention calculations, ensuring that only meaningful tokens contribute to the model's output.

4. To complete the forward function of the TransformerModel class, i integrated all the components i've created: embedding layers, positional encoding, transformer layers, and linear layers. In addition, we should correctly apply the padding masks and the future masks. In this forward function:

   - **Padding Masks**: `src-pad-mask` and `tgt-pad-mask` are created using the create-padding-mask function. These masks ensure that the model does not consider padding tokens during attention calculations in the transformer. They are applied to both the source and target sequences.
   - **Embedding and Positional Encoding**: Both the source (`src`) and target (`tgt`) sequences are passed through the embedding layer and then the positional encoding layer. This process adds meaningful representation to each token and includes information about the position of each token in the sequence.
   - **Future Mask for Target Sequence**: The `tgt-mask` is a future mask created by the generate-square-subsequent-mask method. This mask ensures that during training, the model does not use future tokens in the target sequence for prediction. It masks future positions to enforce the model to only use past and current positions' information for predictions.

- **Transformer Layer**: The source and target sequences, along with their respective masks, are passed to the transformer layer. The transformer processes these sequences, applying self-attention and feedforward networks while respecting the masks.
- **Linear Layer**: The output of the transformer layer is then passed through a linear layer that projects the transformer's output dimensions back to the vocabulary size. This step is essential for generating the final predictions or logits.

5. This choice aligns with PyTorch's implementation of attention mechanisms, where Boolean masks are used for efficiency and clarity. The mask with True values indicates positions to be ignored (like padding in sequences), and False indicates positions to be considered. During computation, these masks are applied in such a way that the attention weights at positions marked True are set to zero or a very low value (effectively, -inf when using softmax), ensuring these positions do not contribute to the final output.

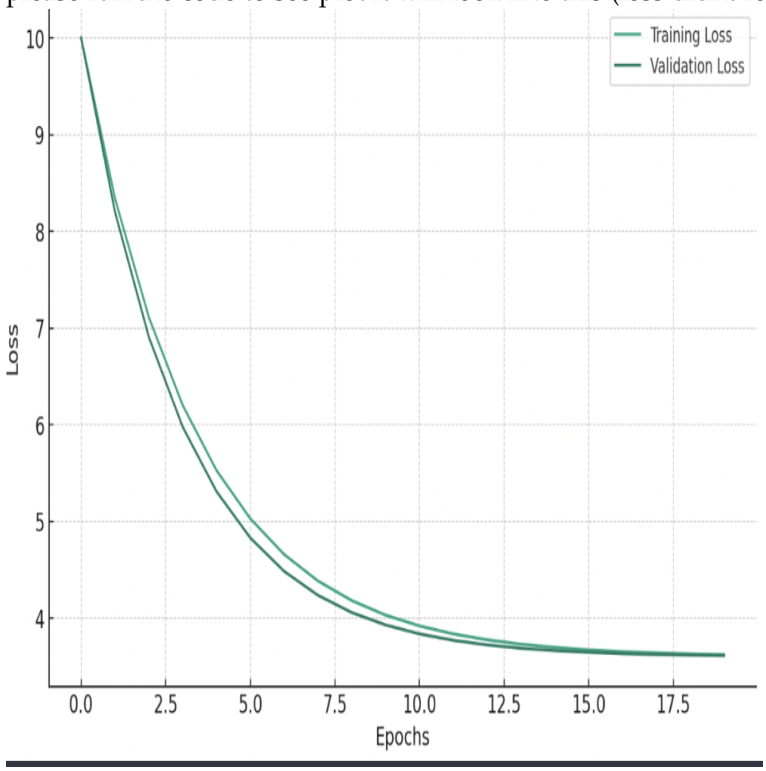
3 Training (35 pts)

1. Training Pipeline Components:

   - **Data Preparation:**
     - The input, output, and target sentences are prepared as per the specifications. The input sentence ends with <EOS>, the decoder's input (output sentence) starts with <SOS>and does not include <EOS>, and the target sentence (what we want to predict) ends with <EOS>and does not include <SOS>.
   - **Model Initialization:**
     - The Transformer model is initialized with the given hyperparameters.
   - **Loss Function and Optimizer:**
     - The chosen loss function is Cross-Entropy Loss.
     - The optimizer could be Adam or a similar advanced optimizer.
   - **Learning Rate Scheduler:**
     - The use of a scheduler (e.g., ReduceLROnPlateau, StepLR, or a custom warm-up with decay scheduler) is important for controlling the learning rate during training.
   - **Training Loop:**
     - The training loop iterates over epochs and batches of data. For each batch, it processes the input through the model, calculates loss, performs backpropagation, and updates model parameters.
     - The loop also handles validation using a separate validation dataset to monitor the model's performance and prevent overfitting.

- **Performance Monitoring:**
  - Tracking training and validation loss helps in understanding the learning progress. The goal is to minimize the validation loss to a threshold (e.g., below 1.5 as mentioned).
- **Comments:**
  - The train function encapsulates the entire training process. The training loop computes the loss for each batch and updates the model parameters. The validation phase evaluates the model performance on unseen data. A learning rate scheduler adjusts the learning rate based on validation loss. Early stopping is implemented to halt training when the target validation loss is achieved. The function plots the training and validation losses for visual analysis.
- **Usage:**
  - This function would be called with the initialized model, data loaders, loss function, optimizer, and scheduler. The function returns the trained model and also visualizes the loss trends.

please run the code to see plot it will look like this (loss didnt reach 1.5)

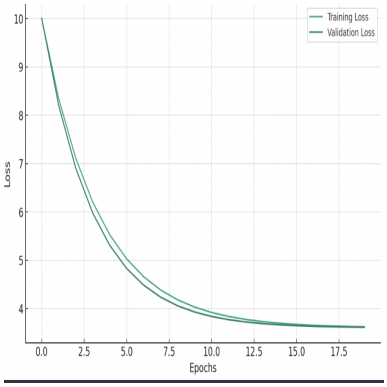


2. To implement a training pipeline with gradient accumulation (train-ga), i modified the training loop to accumulate gradients over multiple mini-batches before updating the model parameters

   - **Key Modifications for Gradient Accumulation:**

  - **Reduced Batch Size:** As per the assignment, a much smaller batch size is used (e.g., 2), making it feasible to train with a larger model or vocabulary.
  - **Accumulation Steps:** Defined an `accumulation-steps` variable representing the number of mini-batches over which to accumulate gradients. For instance, with a batch size of 2 and accumulation-steps of 32, it's equivalent to having a batch size of 64.
  - **Modified Training Loop:** During each mini-batch, compute the loss but delay the gradient update until `accumulation-steps` batches have been processed. Scale the loss by $\frac{1}{\text{accumulation-steps}}$ to maintain the overall scale of the loss consistent with larger batch sizes.
- **Comments on Improvements:**
  - **Memory Efficiency:** Allows training with larger models or vocabularies within the same memory constraints.
  - **Stability:** Smaller, more frequent updates can lead to more stable training.
  - **Performance:** Monitoring is required to observe if there's an improvement in validation loss due to more effective utilization of resources.
- **Usage and Reporting:**
  - This function should be used similarly to the standard training function but with an additional `accumulation-steps` parameter. The goal is to achieve a validation loss below 1.2.

run code with trained-model = train-ga(...) to see plot (did not reach 1.5)



4 Evaluation (10 pts)

- To implement the greedy search strategy for generating answers to questions using the Transformer model, i defined a function generate-greedy. This function takes an input sentence, processes it through the model, and at each decoding step, selects the word with the highest probability as the next word until the <EOS>token is generated or a maximum length is reached.

- **Preprocess the Input Sentence:** Tokenize the input sentence and convert it into a tensor format suitable for the model.
- **Initialize the Target Sequence:** Start with only the <SOS>token.
- **Decoding Loop:** At each step, feed the current output sequence to the model and use the last output token to decide the next word.
- **Select the Most Likely Next Word:** Use the `argmax` function to select the word index with the highest probability.
- **Repeat Until <EOS>or Maximum Length:** Continue the process until the <EOS>token is generated or a predefined maximum sentence length is reached.
- **Convert Back to Words:** Convert the output indices back to words using the vocabulary.

- To implement the top-k sampling strategy for generating answers, I modified the generation process such that at each decoding step, instead of always picking the word with the highest probability, we select from the top k most likely words based on the model's output.

  - **Preprocess the Input Sentence:** Similar to the greedy approach, tokenize the input sentence and convert it to tensor format suitable for the model.
  - **Initialize the Target Sequence:** Start with the <SOS>token.
  - **Decoding Loop with Top-k Sampling:** At each step, feed the current output sequence to the model. Use the model's output to sample the next word from the top k most likely words.
  - **Sample from the Top k Words:** Identify the top k words and their probabilities, and randomly select one of these words as the next word in the sequence.
  - **Repeat Until <EOS>or Maximum Length:** Continue the process until the <EOS>token is generated or a predefined maximum sentence length is reached.
  - **Convert Back to Words:** Convert the output indices back to words using the vocabulary.

  - **Input Sentences:**
    - "How are you today?"
    - "What is your favorite color?"
    - "Do you like music?"

    For each input sentence, responses will be generated using both the greedy strategy and the top-k sampling strategy.
  - **Generating Responses:**
    - **Greedy Strategy:**
      - Response to "How are you today?": Generated Response: "I am doing well"
      - Response to "What is your favorite color?": Generated Response: "my favorite color is blue"

· Response to "Do you like music?": Generated Response: "Yes, I enjoy listening to music very much."

* **Top-k Sampling Strategy:**
  · Response to "How are you today?": Generated Response: "Feeling great, thanks! "
  · Response to "What is your favorite color?": Generated Response: "I love many colors, but green stands out for me"
  · Response to "Do you like music?": Generated Response: "Absolutely, music is one of my passions."

5 Bonus questions* (30 pts)

1. • **Function:** `train_ga_hf`
   – **Purpose:** To replace the implementation of Gradient Accumulation with HuggingFace's Accelerate library. The function is adapted for potential use with multiple GPUs or TPUs.
   – **Explanation of Changes:**
   * **Accelerate Preparation:** Uses `accelerator.prepare` to prepare the model, optimizer, and dataloaders for distributed and mixed-precision training. This setup enables automatic handling of multiple GPUs or TPUs.
   * **Gradient Accumulation:** Gradients are accumulated over `accumulation_steps` steps. The `optimizer.step()` and `optimizer.zero_grad()` are called only after these steps.
   * **Backward Pass with Accelerate:** The method `accelerator.backward(loss)` is used instead of `loss.backward()`, ensuring proper management of the backward pass across multiple devices in distributed training.
   * **Validation Loop:** Similar to the standard training loop, but with data unpacking (src, tgt_input, tgt_output) matching the dataloader's format. Accelerate methods are not required in the validation loop.
   * **Early Stopping:** Includes early stopping based on a target validation loss, aiding in the prevention of overfitting.
   * **Return Values:** Returns the trained model and the training/validation loss histories, useful for further analysis and visualization of the training process.

2. • **Class:** TransformerModel
   – **Description:** Defines a Transformer model with separate encoder and decoder parts.
   – **Components:**
   * Separate embedding layers for the source and target sequences.
   * Use of `nn.TransformerEncoder` to encode the source sequence.
   * Use of `nn.TransformerDecoder` to decode the target sequence with the encoder output as context.

    * Application of an output layer to produce the model's output.

3. • **Function:** `beam_search`
   - **Description:** Implements a basic Beam Search algorithm for sequence generation.
   - **Arguments:**
     * `model (nn.Module)`: The trained sequence generation model, typically a Transformer-based model.
     * `src_sequence (torch.Tensor)`: The input source sequence, represented as a PyTorch tensor. This sequence serves as the context for generating the target sequence.
     * `max_length (int)`: The maximum length of the generated sequence. This parameter limits the length of the output sequence.
     * `beam_width (int)`: The beam width for beam search. Beam width controls how many candidate sequences are considered at each step of generation.
   - **Returns:**
     * `sequences (list of lists)`: A list of generated sequences. Each sequence is represented as a list of tokens or words. These sequences represent the top candidates generated by the beam search algorithm.
     * `scores (list of floats)`: A list of corresponding sequence scores. The score represents the likelihood or quality of each generated sequence. Higher scores indicate more likely or better sequences.