Individual Report - Group Project

Group - 2
Sri Sankeerth Koduru

News Article Summarization using Sequence-to-Sequence Models

DATS 6312 - Natural Language Processing
Spring - 2024
Dr. Amir Jafari

Table of Contents

1)Introduction:

In the digital age, staying updated with news is crucial but time-consuming. The vast amount of information available online often overwhelms readers, making it challenging to extract key insights efficiently. This project addresses this issue by exploring Sequence-to-Sequence (Seq2Seq) models for automating news article summarization using advanced Natural Language Processing (NLP) techniques. By condensing lengthy articles into concise summaries, our system aims to provide efficient access to essential information, enabling users to stay informed without spending excessive time on reading.

2)Developed Functions:

2.1)preprocess_data and preprocess_data_test

The preprocess_data function prepares textual data for use in machine learning models. It accepts DataFrame (df), a tokenizer, and optional maximum sequence length parameters (max_length_x for articles, defaulting to 1024; max_length_y for summaries, defaulting to 200). The function operates as follows:

2.1.1)Initial Truncation:

Limits article and summary lengths to their respective maximums.

2.1.2)Tokenization:

Converts the truncated text into numerical token representations using the provided tokenizer. During this step, the tokenizer automatically applies padding (adding neutral tokens if sequences are too short) and truncation (if sequences are too long) to ensure compliance with the specified maximum lengths.create_tensors

2.1.3)Return:

Finally, it returns encodings, labels_encodings

2.2)create_tensors

The create_tensors function facilitates the transformation of tokenized textual data and corresponding labels into PyTorch tensors, providing a format directly compatible with machine learning models. It receives two dictionaries, encodings, and labels_encodings, storing the tokenized representations of input text and labels, respectively. An optional device parameter allows customization of the tensor's location (e.g., CPU or GPU). The function performs the following steps:

2.2.1)Tensor Conversion:

Tokenized input text (encodings['input_ids']) and attention masks (encodings['attention_mask']) are converted into PyTorch tensors (input_ids and attention_mask). The same process applies to tokenized labels (labels_encodings['input_ids']) and their attention masks (labels_encodings['attention_mask']).

2.2.2)Device Placement:

The .to(device) method ensures the tensors reside on the specified device for optimal computation.

2.2.3)Return:

The function returns the four tensors: input_ids, attention_mask, labels, and labels_attention_mask.

2.3)train

The train function serves a pivotal role in the training process of Transformer models within the PyTorch framework. The function performs the following steps:

2.3.1)Model Preparation for Training:

model.train(): Transitions the model into training mode. This activates elements like dropout and batch normalization, ensuring their behavior aligns with the training process, as opposed to the evaluation phase.

2.3.2)Iterative Training Loop:

for batch in train_loader: Iterates through batches of data supplied by the train_loader. This data loader offers preprocessed training samples in a structure that is compatible with Transformer models.

2.3.3)Gradient Computation and Optimization:

optimizer.zero_grad(): Resets any accumulated gradients within the optimizer, guaranteeing that the gradients calculated for the current batch are not influenced by previous updates.

outputs = model(...): Transmits the input batch through the Transformer model to generate predictions.

loss = outputs.loss: Extracts the loss value calculated by the model for the given batch, serving as an indicator of model performance.

total_loss += loss.item(): Aggregates the loss value for subsequent averaging across the dataset.

loss.backward(): Propagates gradients backward, calculating the gradients of the loss with respect to the model's trainable parameters.

optimizer.step(): Modifies the model's parameters in accordance with the computed gradients and the optimizer's chosen algorithm.

scheduler.step(): Modifies the learning rate based on the scheduler's policy.

2.3.4)Loss Calculation and Reporting:

return total_loss / len(train_loader): Computes and returns the average loss across all batches within the training dataset. This metric offers insights into the model's overall training progress.

2.4)validate

The validate function has the primary responsibility of computing the average loss of a Transformer model on a designated validation dataset. Let's examine its crucial steps:

2.4.1)Data Compatibility:

Validation Loader Verification: Checks that the val_loader supplies data in a format the model can process (i.e., batches with the required tensors).

2.4.2)Model Forward Pass:

Forward Pass Configuration: Ensures inputs are correctly fed to the model and the loss output is extracted.

2.4.3)Device Consistency:

Device Management: Moves the model and input data to the correct device (CPU or GPU) for computation.

2.4.4)Gradient Suppression:

torch.no_grad(): Disables gradient calculation during validation to save memory.

2.4.5)Loss Computation:

Loss Calculation and Aggregation: Calculates the loss for each batch and accumulates it for overall validation loss.

2.5)test

The test function serves the critical purpose of assessing a Transformer model's generalization capabilities on an unseen test dataset. Let's examine its crucial steps:

2.5.1)Evaluation Imports:

Import Verification**:** Check if the load function is indeed located within the 'evaluate' module and that the import statement is accurate.

2.5.2)Evaluation on Test Set:

Variable Name Correction: Change the gold_summaries_holdout variable to gold_summaries where you are appending gold summaries to ensure consistency.

2.5.3)Model Evaluation:

Evaluation Thoroughness: Verify the model's evaluation procedures on the test set. Ensure the metrics used accurately represent its performance.

2.5.4)ROUGE Calculation:

Input Correction: Make sure the calculate_rouge function receives the correct list of gold summaries (gold_summaries_holdout) for comparison.

2.6)calculate_rouge

The calculate_rouge function plays a vital role in evaluating the quality of generated summaries by computing ROUGE (Recall-Oriented Understudy for Gisting Evaluation) scores. Let's examine its crucial steps:

2.6.1)ROUGE Calculation Initialization:

Metric Loading: Loads the necessary ROUGE metric implementation (e.g., from the rouge module).

2.6.2)ROUGE Score Computation:

Score Calculation: Computes ROUGE scores based on the provided generated summaries and reference (gold) summaries. This involves comparing n-grams (unigrams, bigrams, etc.) for precision, recall, and F1 score calculations.

2.6.3)Return:

Result Delivery: Returns the calculated ROUGE scores, offering quantifiable insights into the quality of the generated summaries. Let's examine its crucial steps:

2.7)test_holdout

The test_holdout function evaluates a Transformer model's performance on unseen data by processing the holdout set, generating summaries, and preparing results for metrics calculation. . Let's examine its crucial steps:

2.7.1)Model Preparation for Evaluation:

model.eval(): Places the model in evaluation mode, disabling elements like dropouts which are not desired during the assessment phase.

2.7.2)Iterating Through the Testing Data:

for batch in loader: Processes batches of data from the test (or holdout) dataset, provided by the loader.

2.7.3)Generating Text Summaries:

with torch.no_grad(): Disables gradient calculation for optimization, as gradients are not needed during evaluation.

output = model.generate(...): Employs the Transformer model to create text summaries. Parameters control the summary generation process.

2.7.4)Processing the Results:

generated_summaries_holdout = tokenizer.batch_decode(output, skip_special_tokens=True): Converts model output into human-readable summaries.

generated_summaries.extend(generated_summaries_holdout): Adds generated summaries to the overall list.

gold_summaries.extend(tokenizer.batch_decode(labels, skip_special_tokens=True)): Decodes the correct (gold standard) summaries for comparison.

2.7.5)Returning the Results:

return generated_summaries, gold_summaries: Provides both the model-generated summaries and the true summaries for evaluation metric calculation.

2.8)generate_summaries

The generate_summaries function takes a set of input texts and utilizes a trained Transformer model to generate corresponding text summaries. . Let's examine its crucial steps:

2.8.1)Text Preprocessing:

inputs = tokenizer(texts, max_length=1024, return_tensors='pt', truncation=True, padding=True): The tokenizer prepares the input texts for the model by:

Tokenization: Breaking down the text into individual words or subwords.

Truncation: Limiting the length to a maximum (max_length) for model compatibility.

Padding: Adding padding tokens to make all input sequences of equal length.

Conversion to Tensors: Transforming the data into PyTorch tensors ('pt')

2.8.2)Moving Data to Device:

input_ids = inputs.input_ids.to(device): Transfers the tokenized input IDs to the appropriate device (CPU or GPU if available).

attention_mask = inputs.attention_mask.to(device): Transfers the attention mask (indicating which tokens are real vs. padding) to the device.

2.8.3)Summary Generation:

with torch.no_grad(): Disables gradient calculation, as it's not needed for summary generation.

output = model.generate(input_ids=input_ids, attention_mask=attention_mask, max_length=max_length, num_beams=num_beams): The Transformer model processes the inputs and generates summaries. It uses parameters like:

max_length: Maximum length of the generated summary.

num_beams: Number of beams to use in a beam search for improved summary quality.

2.8.4)Decoding Summaries:

generated_summaries = tokenizer.batch_decode(output, skip_special_tokens=True): The tokenizer converts the model's numerical output back into readable text, removing any special tokens.

2.8.5)Returning the Results:

return generated_summaries: The function provides the list of generated summaries.

3)My Contribution:

3.1)Model Training:

3.1.1)Data Processing:

My contribution centers on data processing and streamlining model training. I used the preprocess_data function to clean, tokenize, and encode textual data from DataFrames, preparing it for use with a chosen language model. Next, I used the create_tensors function to convert this preprocessed data into PyTorch tensors, specifically input IDs, attention masks, and associated labels and label attention masks. These tensors provide the necessary structure for the model's input and targets. To facilitate training, I constructed Tensor datasets for training, validation, and testing, and then created DataLoaders with appropriate samplers to manage efficient batching during the training process.

3.1.2)Training:

Hyperparameter Configuration and Optimization

I selected hyperparameters to guide the model's learning trajectory. These parameters encompassed the number of training epochs (num_epochs), the learning rate (learning_rate), and the adoption of the AdamW optimizer (torch.optim.AdamW). Moreover, to facilitate optimized convergence, I implemented a linear scheduler with a warmup phase (get_linear_schedule_with_warmup) for dynamic learning rate adjustment.

Training Iteration

The model's training process was centered around an iterative loop. Within each epoch, the model was trained (train) using the training dataset (train_loader) and subsequently validated (validate) on the validation set (val_loader). Evaluation involved the computation of ROUGE scores (calculate_rouge), which are crucial metrics for assessing text summarization performance. I diligently monitored training and validation losses, along with the model's overall progress.

Early Stopping for Robustness

To combat overfitting and promote computational efficiency, I incorporated an early stopping strategy. The model's optimal state was preserved (torch.save) according to its performance on the validation set. Training was prematurely halted if the validation loss failed to improve for a predetermined number of epochs (patience). This mechanism aimed to enhance the model's ability to generalize to unseen data.

3.2)Model Testing:

3.2.1)Loading Model:

I loaded the best-performing model state from 'best_model_Multi_News_final.pt'. This model, based on the BART architecture and pre-trained for summarization, was initialized and placed on the appropriate device.

3.2.2)Data Processing:

I loaded the test data from the 'multi_label_test.csv' file and selected a subset for evaluation. Using the preprocess_data_test function, I cleaned, tokenized, and encoded the textual data to make it compatible with the model. These preprocessed data were converted into PyTorch tensors and moved to the same device as the model. Finally, I created a DataLoader with a sequential sampler to maintain order during testing and set a specific batch size for efficiency.

3.2.3)Model Evaluation:

I used the loaded model to generate summaries for the test set, using the test_holdout function. To assess the quality of the generated summaries, I calculated ROUGE scores (ROUGE-1, ROUGE-2, and ROUGE-L), which are standard metrics for text summarization. The calculated ROUGE scores were then printed, offering insights into the model's ability to produce concise and informative summaries.

3.3)Text Generation:

3.3.1)Loading Model:

I loaded the best-performing model state from 'best_model_Multi_News_final.pt'. This model, based on the BART architecture and pre-trained for summarization, was initialized and placed on the appropriate device.

3.3.2)Summary Generation:

To demonstrate how the summarization model I developed works, consider this example. A sample text discussing plastic pollution is provided as input. I utilize the generate_summaries function along with the model, tokenizer, and parameters like max_length and num_beams to control the output. The result is a concise summary of the original text highlighting the key points about the environmental impact of plastic pollution.

4)Results:

4.1)Validation Results with 512 Tokens:

This section presents performance metrics over five epochs of training a model using 512 tokens. It includes training loss, validation loss, and ROUGE scores (ROUGE-1, ROUGE-2, ROUGE-L). The best-performing model in this configuration occurs at epoch 2, with the following scores:

- Train Loss: 1.959035961
- Val Loss: 2.112295384
- ROUGE-1: 0.398256683
- ROUGE-2: 0.147967346

- ROUGE-L: 0.237424769

4.2)Validation Results with 1024 Tokens:

Like the 512-token results, this section details performance metrics for a model trained with 1024 tokens. The best model here also appears at epoch 2. Using 1024 tokens leads to improvements across multiple metrics, with the following scores:

- Train Loss: 1.814289054744612
- Val Loss: 1.9216899385415211
- ROUGE-1: 0.451282871
- ROUGE-2: 0.1753477558554306
- ROUGE-L: 0.24103722565331565

4.3)Test Results:

The test results for the Multinews Trained Transformer Model are:

- ROUGE-1(0. 3710001712954484): This score focuses on the single-word overlap between the generated summary and the reference. Our model achieved a score of 0.371, indicating it captures many important words and concepts from the source articles. While this is a positive starting point, there is potential to improve the comprehensiveness of the summaries.
- ROUGE-2(0. 12235801956537162): This score measures the overlap of two-word phrases. Our model's score of 0.122 suggests it might struggle to generate grammatically correct and well-structured phrases within the summary. This highlights the need for further development in the model's ability to produce fluent and coherent text.
- ROUGE-L(0. 2109042732591439): This score considers the longest matching sequence of words between the summaries. Our model's score of 0.211 indicates some level of similarity in sentence structure and flow compared to the references. However, there's room for improvement to enhance the overall flow and coherence of the generated summaries.

5)Conclusion:

In conclusion, the Multinews trained transformer model, utilizing the Facebook/BART-large-xsum architecture, demonstrates promising performance in capturing important words and concepts from the source articles, as indicated by its ROUGE-1 score of 0.371. However, there is notable room for improvement in generating grammatically correct and well-structured phrases, as reflected in its ROUGE-2 score of 0.122. Similarly, while showing some similarity in sentence structure and flow compared to the references with a ROUGE-L score of 0.211, further enhancements are needed to enhance the overall coherence and fluency of the generated summaries.

6)Future Work:

- Enhance model performance on larger articles by increasing the token processing capacity during training. Limiting the model to the first 1024 tokens can lead to a loss of valuable context and insights.
- Optimize model output for larger articles by employing KNN clustering for the removal of irrelevant information. This will improve focus and the model's ability to extract the relevant points.

7)References:

Pytorch Documentaion: https://pytorch.org/docs/stable/index.html

TensorFlow Documentation: https://www.tensorflow.org/guide

Huggingface Documentation: https://huggingface.co/docs

ChatGPT: https://chat.openai.com/

Gemini: https://gemini.google.com/app

Claude: https://claude.ai/login?returnTo=%2F%3F