Text summarization for news articles

Team members: Brinda Sapra, Pavan Prathuru, Vishal Joseph

https://github.com/krishnapavanz/AdvancedMachineLearning_FinalProject.git

1. Abstract

This report presents a study on text summarization for Kaggle InShorts dataset[1] using four models: Sequence to Sequence Network and Attention, Transformer, BART, and T5 with PyTorch and PyTorch Lightning. The models were evaluated using the BLEU metric, which measures the similarity between generated and reference summaries. Comparative analysis revealed the strengths and weaknesses of each model. The project aimed to develop efficient and accurate methods for condensing lengthy news articles into concise summaries, enabling readers to quickly grasp the key information.

2. Introduction

News articles are an essential source of information, but the increasing volume of news content poses challenges for readers to keep up with the vast amount of available information. To address this issue, we embarked on a research project focused on text summarization for news articles. This project aims to develop a text summarization system that can generate concise and informative headlines from news articles. The system was trained on Kaggle Inshorts News Data. In this report, we present our exploration of four powerful models that employ different approaches to summarize news articles efficiently and accurately.

1. Sequence to Sequence model and Attention - Encoder-Decoder Architecture:

Our research began with the implementation of a sequence-to-sequence network with attention, utilizing an encoder-decoder architecture. This model leverages the power of recurrent neural networks (RNNs) to capture contextual information from the input news articles. By employing attention mechanisms, the model learns to focus on relevant parts of the text and generate concise summaries. We achieved promising results with this initial approach.

2. Transformer Model:

Building upon our initial model, we sought to explore the potential of transformer models for text summarization. Transformers have revolutionized natural language processing tasks, and their attention mechanisms excel at capturing long-range dependencies. By adapting the transformer architecture to our text summarization task, we aimed to improve the quality and coherence of the generated summaries.

3. BART Model:

Inspired by the success of pretrained models in various natural language processing tasks, we turned our attention to BART (Bidirectional and AutoRegressive

[1] https://www.kaggle.com/datasets/shashichander009/inshorts-news-data

Transformers). BART combines the power of denoising autoencoding and sequence-to-sequence models, enabling it to generate high-quality abstractive summaries. By fine-tuning the BART model on our news dataset, we aimed to leverage its pretraining to enhance the summarization process.

4. T5, PyTorch, and PyTorch Lightning:

In our pursuit of exploring diverse models, we incorporated the T5 model, leveraging the capabilities of PyTorch and PyTorch Lightning frameworks. T5 is a versatile model that can be fine-tuned for various text generation tasks, including summarization. By employing T5, we sought to compare its performance with the previously implemented models. PyTorch and PyTorch Lightning frameworks streamlined our experimentation process, allowing us to focus on the model's architecture and hyperparameters. The T5 model demonstrated competitive results, further expanding the range of models available for text summarization.

Each model was trained, fine-tuned, and assessed for its summarization capabilities. The Sequence to Sequence Network and Attention served as the initial approach, leveraging recurrent neural networks and attention mechanisms. The Transformer model explored the power of self-attention and positional encoding, while the BART model capitalized on pretrained architectures for enhanced abstractive summarization. Lastly, the T5 model, integrated with PyTorch and PyTorch Lightning, provided a versatile and conditional generation framework.

The comparative analysis using the BLEU metric provided valuable insights into the models' summarization capabilities. It revealed the importance of architectural choices, fine-tuning techniques, and the utilization of pretraining for enhancing summarization quality. The findings contribute to the field of text summarization and shed light on the effectiveness of different models for news article summarization.

3. Overview of Kaggle InShorts dataset

The Kaggle Inshorts News Data[2] is a dataset that consists of news articles collected from the Inshorts news website[3]. The dataset contains news articles in English from various categories such as sports, business, technology, entertainment, and more. The articles are in the form of short summaries, with a maximum length of 60 words. The dataset contains over 55,000 news articles, and it can be used for various natural language processing tasks such as text classification, text summarization, and more.

The Kaggle InShorts dataset utilized in this project includes various columns, among which the 'Headline' and 'Summary' are of particular relevance for our text summarization task. The dataset's columns are as follows:

- Article ID: A unique identifier assigned to each news article.
- Headline: The title or headline of the news article.

[2] https://www.kaggle.com/datasets/shashichander009/inshorts-news-data
[3] https://inshorts.com/en/read/

- Category: The categorization or topic of the news article, encompassing areas such as sports, business, technology, entertainment, and more.
- Article: The complete text of the news article, encompassing the primary content and additional details.
- Summary: A succinct summary of the news article, typically limited to a predetermined maximum length, such as 60 words.
- Source: The origin of the news article, indicating the specific publication or platform, such as Inshorts or other designated sources.
- Date: The publication or recording date of the news article.

Among these columns, our focus lies primarily on the 'Headline' and 'Summary' columns, as they serve as crucial inputs and outputs for our text summarization project.

4. Methodology

The dataset was initially preprocessed by creating a smaller subset of 5,000 random samples and splitting it into training, evaluation, and testing datasets in a ratio of 60:20:20. The preprocessing steps included removing special characters, punctuation, and unnecessary whitespace from the text. The resulting preprocessed datasets were used for training, evaluating, and testing the text summarization models.

The models follow a common approach to data collection and text cleaning:

1. The code loads an Excel file containing the InShorts news raw data into a pandas DataFrame, keeping only the 'Headline' and 'Short' columns and discarding the rest. The total number of rows in the dataframe is then counted, and a random sample of 5000 rows is taken without replacement from the original data.
2. Next, the 5000 rows are randomly divided into three groups of train, test, and dev sets with specified proportions of 60%, 20%, and 20% respectively. Each group is then exported to a separate Excel file.
3. The code then proceeds to preprocess the text data. Stop words are first removed using the nltk library's built-in list of English stop words. The text is then converted to lowercase and contractions such as 'have'nt' are replaced with their expanded forms like 'have not'. The words in the text are then joined into a single string and converted to lowercase, and a new list is created by removing the stop words. The text is then converted to lowercase and the contraction words are replaced with their expanded form. The words inside a parenthesis are removed using regex, and all punctuation marks except for periods are removed. Finally, the periods are surrounded by spaces for better tokenization.

The expanded version of the dataset, consisting of 50,000 additional random samples, was then created to further enhance the analysis. The same preprocessing steps were

repeated on this scaled-up dataset, and the training, evaluation, and testing processes were performed again using the expanded dataset.

5. Model Evaluation

We had initially proposed in our midline report that we would do human surveys to assess the models' performance. However, we went a step ahead and implemented a standard metric which is used for assessing the quality of a language model. The evaluation of the models was performed using the BLEU (Bilingual Evaluation Understudy) metric, specifically the 'google_bleu'[4] metric.

S.No	Model	BLEU Score
1	Seq2Seq GRU (Baseline Model)	0.0005
2	Transformer	0.007
3	Transfer Learning with BART	4.34e-06
4	Transfer Learning with T5	0.264

BLEU is a metric used to evaluate the quality of machine-generated translations or text summaries by comparing them to one or more reference translations or summaries. It measures the similarity between the generated output and the reference(s) based on the degree of overlap in n-gram sequences. The BLEU score ranges from 0 to 1, with a higher score indicating better similarity to the reference(s).

BLEU scores have some limitations. They primarily focus on lexical overlap and do not capture the semantic or structural aspects of the generated text. Below is the calculation approach used for BLEU score:

1. Tokenization: The generated output and reference(s) are first tokenized into n-grams, which are contiguous sequences of n words or characters.
2. n-gram matching: BLEU calculates the precision score for each n-gram size (typically ranging from 1 to 4). Precision is the percentage of n-grams in the generated output that also appear in the reference(s). The precision score is computed as the count of matching n-grams divided by the total count of n-grams in the generated output.
3. Modified n-gram precision: To avoid favoring overly short translations or summaries, a brevity penalty is applied. This penalty penalizes the generated output if its length is significantly shorter than the reference(s). It encourages the generated output to have similar length to the reference(s).

[4] https://huggingface.co/spaces/evaluate-metric/google_bleu

4. BLEU score computation: The modified n-gram precisions are combined by taking their geometric mean, and the brevity penalty is applied to obtain the final BLEU score.

As the above table illustrates, the highest BLEU score is obtained when we use transfer learning with the T5 model, and followed by the Tranformer model which we designed from scratch.

6. Seq2Seq Model

The baseline model is a sequence to sequence model. The architecture includes a simple encoder-decoder architecture with an attention mechanism, by leveraging Gated Recurrent Neural network layers, and teacher forcing.

1. Preprocessing
    a. Lang class: This class is responsible for initializing language objects, adding words to the language object's vocabulary, and loading embeddings. It has the following attributes and methods:
        - name: the name of the language object.
        - word2index: a dictionary that maps words to their corresponding index in the vocabulary.
        - word2count: a dictionary that stores the frequency of each word in the vocabulary.
        - index2word: a dictionary that maps indices to their corresponding words in the vocabulary.
        - n_words: the number of words in the vocabulary, initialized to 2 (for the SOS and EOS tokens).
        - embedding_dim: the dimension of the word embeddings.
        - embedding: an embedding layer initialized with the vocabulary size and embedding dimension.
        - load_embeddings: a method that loads pre-trained word embeddings for words in the vocabulary.
    b. readLangs function: This function takes in two lists of text and summary sentences, splits them into pairs, and normalizes them. It then creates two Lang instances - one for the input language and one for the output language. If the ‘reverse’ argument is True, it reverses the pairs and creates the Lang instances for the output language and input language, respectively. It returns the input and output Lang instances and the pairs.

c. prepareData function: This function takes in two language strings and an optional reverse argument. It calls the readLangs function to create Lang instances and pairs. It then counts the words in each sentence in the pairs and adds them to their respective language object's vocabulary using the addSentence method of the Lang class. Finally, it returns the input and output Lang instances and the pairs.

4. Data loader
    a. The indexesFromSentence function takes in a language object (lang) and a sentence as input, tokenizes the sentence by splitting it into words, and returns a list of the indexes of each word in the language object's word2index dictionary.
    b. The tensorFromSentence function first obtains the indexes of each word in the sentence by calling the indexesFromSentence function. It then appends the index of the end-of-sentence token (EOS_token) to the end of the list of indexes, creates a PyTorch tensor from the list of indexes, sets the data type of the tensor to torch.long, and moves the tensor to the device specified by the device variable. Finally, it reshapes the tensor into a column vector with a single column.
    c. The tensorsFromPair function takes in a pair of input and target sentences and returns a tuple of PyTorch tensors. It first calls the tensorFromSentence function with the input sentence and the input language object (input_lang) to obtain the input tensor. It then calls the tensorFromSentence function with the target sentence and the output language object (output_lang) to obtain the target tensor. Finally, it returns the tuple containing the input tensor and the target tensor.
5. Model
    a. EncoderRNN: The Encoder class is implemented using a single layer of GRU cells. It takes two inputs - the size of the input vocabulary, and the hidden size of the GRU cells. The embedding layer is initialized with pre-trained GloVe vectors, and its weights are frozen. In the forward function, the input sequence is first embedded and then fed into the GRU cells. The final output and hidden state are returned.
    b. DecoderRNN: The Decoder class is also implemented using a single layer of GRU cells. It takes two inputs - the hidden size of the GRU cells, and the size of the output vocabulary. The embedding layer is initialized with pre-trained GloVe vectors, and its weights are frozen. In the forward function, the input sequence is first embedded and then fed into the GRU

cells. The final output is passed through a linear layer and a softmax activation function to obtain the probability distribution over the output vocabulary.

c. AttnDecoderRNN: The Attention Decoder class is similar to the Decoder class, but it also incorporates an attention mechanism. In addition to the hidden state of the decoder, it takes the encoder output sequence as input. The attention weights are computed using a linear layer, and the context vector is computed as the weighted sum of the encoder outputs. The context vector and the embedded input sequence are concatenated and passed through a linear layer before being fed into the GRU cells. The final output is passed through a linear layer and a softmax activation function to obtain the probability distribution over the output vocabulary. The attention weights are also returned.

6. Pipelining:
    a. The function ‘train’ is the training loop for the sequence-to-sequence model. It takes in the input tensor and target tensor, along with the encoder and decoder models, their respective optimizers, and the loss criterion. It returns the average loss per word predicted in the target sequence (the summary).
        i. At the beginning of each training iteration, the encoder's hidden state is initialized and the optimizer gradients for both the encoder and decoder models are set to zero.
        ii. The input and target lengths are determined and the encoder's output states for each input token in the input sequence are stored in the encoder_outputs tensor.
        iii. Next, the decoder input and hidden states are initialized, and a random decision is made to use teacher forcing, with a probability of teacher_forcing_ratio.
        iv. If teacher forcing is used, the target sequence is fed as the next input to the decoder, while if it is not used, the decoder's own predictions are used as the input for the next step.
        v. For each token in the target sequence, the decoder outputs its predicted probability distribution for the next word in the target sequence, along with its hidden state and attention weights over the encoder outputs. If teacher forcing is used, the actual next word from the target sequence is fed as the next input to the decoder,

while if it is not used, the word with the highest probability is chosen as the next input.

vi. During both these stages, the loss is calculated as the distance between the decoder's predicted probability distribution and the actual next word in the target sequence.

vii. Finally, the loss is backpropagated through the model and the optimizer steps are taken to update the encoder and decoder parameters.

viii. The function returns the average loss per word in the target sequence.

b. The trainIters function is a high-level function that calls the train function repeatedly to train the encoder and decoder networks for a given number of epochs.

i. It takes the encoder and decoder networks, the number of training iterations (n_iters), and some hyperparameters such as learning rate, print frequency, and plot frequency as input.

ii. The function initializes the optimizer for the encoder and decoder networks with the given learning rate and uses the negative log likelihood loss (nn.NLLLoss()) as the loss function.

iii. It then iterates through the specified number of epochs (n_epochs) and randomly selects pairs from the dataset to use for training.

iv. For each training iteration, it retrieves an input sequence and target sequence from the randomly selected training pair and passes them to the train function along with the encoder and decoder networks, optimizer, and loss function. The train function calculates the loss and performs backpropagation to update the encoder and decoder networks.

v. The function also keeps track of the loss over time by appending the average loss to a list

c. The function evaluate() takes an input sentence, encodes it using an encoder and generates an output sequence using a decoder. The output sequence is generated one word at a time, with the decoder taking the previous word as input at each step.

d. The evaluateRandomly function takes the encoder and decoder models, a list of pairs of input/output sequences (pairs), and an integer n as inputs. It then selects n pairs randomly from the pairs list and performs an evaluation for each pair. For each pair, it prints the input sequence (the

first element in the pair), the expected output sequence (the second element in the pair), and the actual output sequence generated by the decoder model using the evaluate function.

7. Hyperparameter Tuning: We performed hyperparameter tuning to improve the performance of our baseline model. We experimented with various hyperparameters such as teacher forcing ratio, dropout rate, learning rate, hidden size, max length, no. of layers within the GRU encoder and decoder, optimizer, number of epochs, loss function, and the use of pre-trained GloVe embeddings.

   Teacher forcing ratio is a hyperparameter that determines the probability of using the target sequence or the predicted sequence during training. We experimented with different values of teacher forcing ratio and found that a value of 0.5 gave the best results.

   Dropout rate is a regularization technique used to prevent overfitting. We experimented with different values of dropout rate and found that a value of 0.5 gave the best results.

   Learning rate is a hyperparameter that determines the step size at which the optimizer adjusts the parameters of the model during training. We experimented with different values of learning rate and found that a value of 0.01 gave the best results.

   Hidden size is the number of hidden units in the GRU encoder and decoder. We experimented with different values of hidden size and found that a value of 512 gave the best results.

   Max length is the maximum length of the input and output sequences. We experimented with different values of max length and found that a value of 90 gave the best results.

   We also experimented with different optimizers, including Adam and SGD, and found that Adam gave the best results. We also tried different loss functions such as Cross-Entropy loss and Mean-Squared Error (MSE) loss, and found that the Cross-Entropy loss gave the best results.

   Finally, we also experimented with using pre-trained GloVe embeddings, and found that using them improved the performance of the model.

Below are the results of the model:

```
> Input text (full length article)

= Expected summary (labelled ground-truth)

< Model summary (summary generated by model)

> printing new 2000 note costs reserve bank india 3 . 54 500 note
costs 3 . 09 rti response said .  notably bhartiya reserve bank
note mudran private limited subsidiary rbi prints new 500 2000
currency notes price old 500 1000 notes respectively .
```

7. Transformer Model

a. Introduction and advantages as compared to Seq2Seq model

The Transformer model, introduced by Vaswani et al. in the paper "Attention is All You Need," is a deep learning architecture designed for sequence-to-sequence tasks, such as machine translation and text summarization. It is based on self-attention mechanisms and has become the foundation for many state-of-the-art natural language processing models, like GPT and BERT. The architectural improvement over the Seq2Seq model is:

- Multi-head attention: Captures different aspects of the input sequence.
- Positional encoding: Injects the positional information of each token in the input sequence. This aids in parallelization.
- Multiple encoder and decoder layers: More enhanced information capture compared to a single layer.

b. Model architecture

i. MultiHeadAttention: The class initializes the module with input parameters and linear transformation layers. It calculates attention scores, reshapes the input tensor into multiple heads, and combines the attention outputs from all heads. The forward method computes the multi-head self-attention, allowing the model to focus on some different aspects of the input sequence.

ii. PositionWiseFeedForward: This class extends PyTorch's nn.Module and implements a position-wise feed-forward network. The class initializes with two linear transformation layers and a ReLU activation function. The forward method applies these transformations and activation function sequentially to compute the output. This process enables the model to consider the position of input elements while making predictions.

iii. PositionalEncoding: This class initializes with input parameters d_model and max_seq_length, creating a tensor to store positional encoding values. The class calculates sine and cosine values for even and odd indices, respectively, based on the scaling factor div_term. The forward method computes the positional encoding by adding the stored positional encoding values to the input tensor, allowing the model to capture the position information of the input sequence.

iv. EncoderLayer: An Encoder layer consists of a Multi-Head Attention layer, a Position-wise Feed-Forward layer, and two Layer Normalization layers. This class initializes input parameters and components, including a MultiHeadAttention module, a PositionWiseFeedForward module, two layer normalization modules, and a dropout layer. The forward method computes the encoder layer output by applying self-attention, adding the attention output to the input tensor, and normalizing the result. Then, it computes the position-wise feed-forward output, combines it with the normalized self-attention output, and normalizes the final result before returning the processed tensor.

v. DecoderLayer: A Decoder layer consists of two Multi-Head Attention layers, a Position-wise Feed-Forward layer, and three Layer Normalization layers. The DecoderLayer initializes with input parameters and components such as MultiHeadAttention modules for masked self-attention and cross-attention, a PositionWiseFeedForward module, three layer normalization modules, and a dropout layer.

vi. Transformer: This class combines the previously defined modules to create a complete Transformer model. During initialization, the Transformer module sets up input parameters and initializes various components, including embedding layers for source and target sequences, a PositionalEncoding module, EncoderLayer and DecoderLayer modules to create stacked layers, a linear layer for projecting decoder output, and a dropout layer.

c. Implementation approach, including hyperparameter choices

i. Data prep:

1. The input data is processed in the same way as the Seq2Seq model. Custom dataset classes are created for the train, validation and test data, which are then converted to dataloader type that aids in batching.

ii. Model training:

1. The ‘evaluate_val’ function was defined to calculate the evaluation loss during training in order to save the best model. This is called once every 500 iterations.
2. The model object is instantiated with the following values:

a. src_vocab_size: Length of training input data vocab vector
b. tgt_vocab_size : Length of training target data vocab vector
c. d_model: Embedding dimension of 512
d. num_heads: 8 attention heads
e. num_layers: 6 encoder and decoder layers
f. d_ff: Feedforward layer input dimension of 2048
g. dropout: Dropout of 0.1

3. The training chunk defines the loss function (Cross entropy) and optimizer (Adam), followed by the actual training on 10 epochs. Each epoch goes through around 2000 iterations (batch size of 16) and the training loss is recorded once every 200 iterations.
4. The trained model is saved to disk and loaded for visual as well as BLEU score evaluation. The 'evaluate' and 'evaluateRandomly' functions handle the visual inspection bit.

d. Results and analysis (including challenges)

The best model based on validation loss happens quite early in the training process which resulted in poor test data predictions which were more or less nonsensical. This did not change despite having a training set of ~33k samples. Time constraints proved a challenge in debugging this further. However, the best model based on the training loss gave somewhat reasonable summarization results when evaluated randomly on the training data, but fared poorly on the test data as expected. Our hypothesis is that extended hyperparameter tuning could fit this to a certain extent, but we will always be constrained by the dearth of training data for such a task. Our rationale for experimenting with coding up the transformer model from scratch was to assess the improvement compared to the Seq2Seq model (performed worse), while also expecting below average results given the fact that this is not pretrained like the BART or T5 models, which we implement in the next stage.

Below are some examples on the training data:

```
Actual Headline: raise ram temple issue vigorously bjp mp katiyar
Predicted Headline: issue unite direction bjp mp cm

Actual Headline: bcci distribute state funds electronically
Predicted Headline: funds goa cricket funds goa
```

Below are some examples on the test data:

```
Actual Headline: pilots suspended flying jet airways plane 39low39
```

```
Predicted Headline: destroying crew cares ford admission sena

Actual Headline: telcos install 60k towers check call drops
Predicted Headline: southernmost courier kahaani staff spreads
chidambaram
```

8. BART model

a. Introduction

BART, which stands for "Bidirectional and Auto-Regressive Transformers," is a transformer-based model that is designed for sequence-to-sequence tasks, such as text generation and text summarization. It was introduced by Lewis et al. in 2019.

BART is unique because it combines the strengths of auto-regressive (AR) and denoising (AE) transformer models into a single architecture. Auto-regressive models generate output tokens one at a time based on previously generated tokens, while denoising models reconstruct the original input from a corrupted version.

The BART model consists of an encoder-decoder architecture, similar to other sequence-to-sequence models. One key aspect of BART is the use of masked language modeling (MLM) during pre-training. In MLM, a certain percentage of input tokens are randomly masked, and the model is trained to predict the original masked tokens. This pre-training objective helps BART learn meaningful representations of the input sequence.

Additionally, BART utilizes a novel denoising objective called "noising," where the input sequence is corrupted by various noise functions such as token deletion, shuffling, and replacement. The model is then trained to reconstruct the original sequence from the corrupted version. This denoising objective helps BART learn robust representations and improves its performance on downstream tasks.

b. Model's Implementation

1. Model Initialization and Configuration:
    - The code uses libraries (**transformers**, **datasets**).
    - It imports the BART model and tokenizer from the Transformers library and initializes them with the pre-trained "facebook/bart-large-cnn" weights.
    - The code freezes all the layers of the model except for the last two layers in both the encoder and decoder.
    - It sets up the data preprocessing steps by defining a function to process the data and preprocesses the training, validation, and test data using this function.
    - The preprocessed data is converted to **Dataset** objects from the **datasets** library.
2. Data Processing and Model Inputs:

- The code defines a function (**process_data_to_model_inputs**) to tokenize and encode the input and target sequences.
- It applies this function to the training, validation, and test datasets using the **map** method.
- The processed datasets are then converted to a suitable format (**torch tesnors**) and prepared for batching.

3. Training Setup:
   - The code sets up the training parameters, such as the number of epochs, training and validation steps, batch size, loss function (CrossEntropyLoss), optimizer (AdamW), and learning rate scheduler.
   - It initializes the encoder and decoder components of the BART model.
   - The code sets up a progress bar to track the training and validation progress.
4. Learning Rate Scheduler:
   - The learning rate scheduler adjusts the learning rate during training across batches within an epoch, to optimize the model's performance.
   - A linear scheduler increases the learning rate linearly over the training steps.
5. Encoder
   - The encoder component of the `BartForConditionalGeneration` model processes input sequences and extract their contextual representations. It is an instance of the `BartEncoder` class, which is composed of multiple layers of self-attention and feed-forward neural networks. The `BartEncoder` class typically contains the following key attributes:
     i. `embed_tokens`: The embedding layer that maps input tokens to continuous representations.
     ii. `encoder`: A stack of Transformer encoder layers that process the input sequence and generate contextual representations.
     iii. `embed_positions`: Positional embeddings that encode the position information of the input tokens.
     iv. `layernorm_embedding`: A layer normalization module applied to the output of the embedding layer.
     v. `layer_norm`: A layer normalization module applied to the output of each encoder layer.
6. Decoder
   - The decoder component of the `BartForConditionalGeneration` model generates output sequences, conditioned on the contextual representations. This is done in autoregressive manner, where the decoder generates tokens one at a time, taking into account the previously generated tokens. The decoder is designed to take the encoder's contextual representations as input and produce the output sequence step by step. The object is typically an instance of the `BartDecoder` class. The `BartDecoder` class contains the following key attributes:
     i. `embed_tokens`: The embedding layer that maps output tokens to continuous representations.

    ii. `decoder`: A stack of Transformer decoder layers that generate the output sequence based on the contextual representations from the encoder.
    iii. `embed_positions`: Positional embeddings that encode the position information of the output tokens.
    iv. `layernorm_embedding`: A layer normalization module applied to the output of the embedding layer.
    v. `layer_norm`: A layer normalization module applied to the output of each decoder layer.
7. Last Linear Layer
    - `last_linear_layer` is an instance of a linear layer that performs a linear transformation on the input data. the `last_linear_layer` is an instance of the `torch.nn.Linear` class. The attribute `lm_head` refers to the last linear layer of the model, which is responsible for generating the output logits or scores for each token in the output sequence.
    - The last linear layer is a linear transformation that takes the contextual representations from the decoder and maps them to the vocabulary space. It providing probabilities or scores for each token in the vocabulary, indicating the likelihood of that token being the next token in the generated sequence.
    - The `torch.nn.Linear` class contains the following key attributes:
        i. weight: The weight matrix that is used for the linear transformation.
        ii. bias: The bias vector that is added to the output of the linear transformation.
8. Training Loop:
    - The code enters the training loop, iterating over 10 epochs.
    - Within each epoch, the model is set to training mode, and the training loss is initialized.
    - The training data is iterated in batches of 10, and each batch is processed using the encoder and decoder components.
    - The model's output is computed, and the loss is calculated.
    - The loss is backpropagated through the model, and the optimizer's parameters are updated.
    - The learning rate scheduler is updated, and the progress bar is updated to track the training progress.
9. Validation:
    - After each epoch of training, the model is set to evaluation mode, and the validation loss is initialized.
    - The validation data is iterated in batches, and each batch is processed using the model.
    - The model's output is computed, and the validation loss is calculated.
    - The progress bar is updated to track the validation progress.
10. Model Saving:
    - The training and validation losses are averaged, and the validation loss is stored in a list.

- If the current validation loss is the minimum among all the validation losses so far, the model's state is saved to a file as a checkpoint.

11. Evaluation:
    - The code loads the saved model and sets it to evaluation mode.
    - It iterates over the test data in batches and generates predictions using the model.
    - The generated headlines are decoded from token IDs to text.
    - The predictions are compared with the actual headlines from the test data.
    - The BLEU metric is used to evaluate the predictions against the actual headlines, and the average BLEU score is calculated across all instances of the test data.

c. Fine-tuning approach, including hyperparameter choices

   i. The number of last few layers that were unfreezed were varied and the best performance was achieved when last 2 layers were unfreezed. This significantly reduced the training loss and validation loss but had marginal effect on improving the predictions (as we eyeballed) and the average BLEU score.

   ii. Learning rate was varied and the best performance was achieved for the rate

   iii. During the output generate when the model is being evaluated on test data, num_beams (number of beams), num_beam_groups (number of beam groups), diversity_penalty (encourage diversity of words) were added to nodel.generate() to experiment if they can help in avoiding the repetition in tokens/words that occurs. However, this had no effect.

Below are the results of the model:

```
Prediction Headline:  the the
thethethethetotototobutbutbutnotnotandandandbutnoteveneveneven even even
even right right now now now that there is now now also now now not not
not even even now now actually now now currently currently currently
also now also not now currently actually now
Actual Headline: trial run indias first 700 mw nuclear reactor 2017
```

9. T5, PyTorch, and PyTorch Lightning

a. Introduction

T5 (Text-To-Text Transfer Transformer) is a state-of-the-art language model that has gained significant attention in the field of natural language processing (NLP). Built on the Transformer architecture, T5 has demonstrated remarkable performance across

various NLP tasks, including text summarization. In this project, we incorporated T5 into our text summarization pipeline to explore its capabilities in generating concise and accurate summaries for news articles.

T5 is based on the Transformer architecture, which consists of an encoder and a decoder. The encoder processes the input text, while the decoder generates the corresponding output. T5 adopts a text-to-text transfer learning framework, where all NLP tasks are framed as text-to-text problems. This means that both the input and output are text strings, enabling T5 to handle diverse tasks by fine-tuning the same underlying architecture.

T5 is pretrained on a massive corpus of diverse text from the internet. It learns to predict the correct output given the input for various text-to-text tasks. The pretraining enables T5 to acquire a rich understanding of language and enables effective transfer learning for downstream tasks like text summarization.

b. Model overview

- **Data Preprocessing**: To prepare the data for training the T5 model, we implemented the 'NewsSummaryDataset' class. This class takes in the input data, which consists of news articles and their corresponding headlines. The data is tokenized using the T5 tokenizer, and additional preprocessing steps such as adding special tokens, padding, and truncation are applied. The resulting tokenized sequences are then converted into tensors for efficient processing.

- **Data Module:** To manage the training, validation, and testing datasets, we utilize the 'NewsSummaryDataModule' class, which is a Lightning DataModule. This class facilitates the setup of the datasets, creating instances of the NewsSummaryDataset class for each dataset. DataLoader is used to efficiently batch and process the data, taking advantage of parallel processing.

- **Model Training:** The core of the implementation lies in the 'NewsSummaryModel' class, a LightningModule responsible for model training and evaluation. The forward method of the model performs the forward pass, returning the loss and logits. Training and evaluation steps are defined within the training_step, validation_step, and test_step methods, respectively. The configure_optimizers method sets up the optimizer and learning rate scheduler.

- **Training Process:** To train the model, we instantiate the NewsSummaryModel class and create a Trainer object. The Trainer is configured with the desired number of epochs. The trainer.fit() method then triggers the training process, using the specified data module and model instance. During training, the model learns to generate accurate and concise summaries based on the input text.

c. Model fine-tuning

The model fine-tuning process for the T5 model can be summarized as follows:

1. Model Initialization:

- The T5 model is initialized using the T5ForConditionalGeneration class from the "t5-base" pre-trained model.
- The model is configured to return a dictionary of outputs.

2. Tokenization and Dataset Creation:
   - The NewsSummaryDataset class is defined, which takes the input DataFrame, tokenizer, text length, and headline length as parameters.
   - The dataset class preprocesses the data by tokenizing the input text and headlines using the T5 tokenizer.
   - Special tokens are added, and the input and output sequences are padded or truncated to ensure consistent lengths.
   - The dataset class implements the __len__ and __getitem__ methods required for dataset handling.

3. Data Module Setup:
   - The NewsSummaryDataModule class is defined, which takes the training, validation, and test DataFrames, batch size, tokenizer, text length, and summary length as parameters.
   - The setup method of the data module is called, which initializes the training, validation, and test datasets using the NewsSummaryDataset class.
   - Each dataset is created with the respective DataFrame, tokenizer, and length parameters.

4. DataLoader Configuration:
   - The train_dataloader, val_dataloader, and test_dataloader methods of the data module are defined.
   - These methods return DataLoader objects that handle batching, shuffling, and parallel data loading.
   - The train_dataloader shuffles the training dataset, while the others do not shuffle the validation and test datasets.

5. Model Definition:
   - The NewsSummaryModel class is defined, which inherits from pl.LightningModule.
   - The T5 model is instantiated within the NewsSummaryModel class and moved to the appropriate device (e.g., GPU) using the to() method.

6. Model Training:
   - The training_step method is defined within the NewsSummaryModel class, which takes a batch of data as input.
   - The input text, attention mask, labels, and decoder attention mask are extracted from the batch.
   - The model's forward method is called with these inputs to obtain the loss and output logits.
   - The loss is returned to be used for optimization during training.

7. Model Validation and Testing:
   - The validation_step and test_step methods are defined similarly to the training_step method.
   - These methods extract the necessary inputs from the batch and compute the loss and output logits using the model's forward method.
   - The loss is returned for evaluation during validation and testing.

8. Optimizer and Scheduler Configuration:
   - The configure_optimizers method of the NewsSummaryModel class is defined.
   - It creates an optimizer (AdamW) and a learning rate scheduler (get_linear_schedule_with_warmup).
   - The optimizer is initialized with the model's parameters, and the scheduler is set up with the number of warm-up steps and training steps.

9. Model Training and Validation Loop:
   - An instance of the NewsSummaryModel class is created.
   - A Trainer object from the pytorch_lightning library is instantiated, specifying the maximum number of epochs.
   - The trainer.fit() method is called, which trains the model using the configured optimizer, scheduler, and data loaders.
   - During training, the model iterates over the training data, calculates the loss, and updates the model parameters using backpropagation.
   - After each epoch, the model is evaluated on the validation data, and the best model checkpoint is saved based on the validation loss.

10. Model Evaluation:
    - The saved best model checkpoint is loaded using NewsSummaryModel.load_from_checkpoint.
    - The model is frozen to prevent further training.
    - The summarize function is defined, which takes an input text and generates a summary using the trained T5 model.
    - The input text is tokenized using the T5 tokenizer, and the model's generate method is called with the input tokens.
    - The generated summary tokens are decoded into text using the T5 tokenizer, excluding special tokens.
    - The BLEU metric is used to evaluate the quality of the generated summaries compared to the actual headlines.

11. Model Persistence:
    - The trained T5 model is saved using the pickle module.
    - A file is opened in binary write mode, and the model is dumped into the file using the pickle.dump() function.

12. Summary Generation:
    - The saved model is loaded from the pickle file.

- For a given number of examples (e.g., 100), the input text and actual headline are extracted from the test data.
- The summarize function is called to generate a summary for each input text.
- The actual headline and predicted summary are printed for each example.

13. Evaluation metric:
    - The fine-tuning process for the T5 model has been implemented successfully.
    - The generated summaries are evaluated using the BLEU metric, providing insights into the model's performance.

d. Results

A sample of T5 model results are given below:

```
Actual:  39indian gaming league39 launched india
Predicted:  esports league igl launched india february 4
Actual:  rgv calls tiger 39bikini babe39 compares urmila
Predicted:  bruce lee wouldn39t hav bcm bruce lee
Actual:  zakir naiks ngo gave 50l rajiv gandhi trust
Predicted:  zakir naiks ngo irf donated 50l
Actual:  knew nothing tax fraud messi
Predicted:  messi knew nothing tax evasion messi
Actual:  rajasthan high court scraps gujjar reservation
Predicted:  rajasthan hc quakes 5 reservation special backward classes
Actual:  kejriwal compares power tariffs delhi gujarat
Predicted:  pm modi wants delhi power tariffs gujarat kejr
Actual:  want make india open economy world pm modi
Predicted:  want make india open economy world modi
Actual:  2 crore people attend msg lion heart 2 trailer launch
Predicted:  trailer gurmeet39s 39hind ka napak ko jaw
Actual:  twice many army men killed jampk 2016 2014 2015
Predicted:  twice many indian soldiers martyred year jampk
Actual:  asus launch curved gaming monitor india
Predicted:  asus launch gaming monitor 1. 2 1. 5 lakh
```

Overall, the T5 model's performance is mixed, displaying both strengths and weaknesses. It showcases the ability to generate accurate summaries in some cases, capturing key information and maintaining coherence with the original headlines. However, the model also exhibits limitations in understanding context, making incorrect associations, and misinterpreting certain aspects of the news. Based on the obtained BLEU score of 0.264, it is evident that the T5 model's performance in generating accurate and informative summaries can be further improved.

10. Conclusion

Through our research, we explored four powerful models for text summarization: Sequence to Sequence Network and Attention, Transformer, BART, and T5 with the assistance of PyTorch and PyTorch Lightning. Each model has its unique strengths,

ranging from attention mechanisms to pretrained architectures. The results showcased advancements in text summarization, with each model demonstrating promising capabilities in condensing news articles.

To enhance the model's performance, several steps can be taken. First, fine-tuning the model with a larger and more diverse dataset could improve its understanding and generalization capabilities. Second, exploring different pre-processing techniques, such as additional text cleaning or feature engineering, may enhance the model's ability to extract important information. Third, employing alternative evaluation metrics beyond BLEU, such as ROUGE or METEOR, could provide a more comprehensive assessment of the model's performance. Lastly, experimenting with other transformer-based architectures, such as GPT, might offer insights into alternative models that could potentially yield better results.

Our findings contribute to the advancement of text summarization techniques for news articles, demonstrating the potential of these models to generate concise and coherent summaries. Through this research, valuable insights were gained regarding the performance and potential of various models, contributing to the field of natural language processing and text summarization. Future work may explore hybrid approaches or further refine existing models to enhance summarization accuracy and coherence. The findings of this study are valuable for applications in news aggregation platforms, content curation, and information retrieval systems. The results pave the way for future advancements in text summarization techniques.

11. References

1. “Attention Is All You Need by Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, Illia Polosukhin”
2. "Automatic Text Summarization Methods: A Comprehensive Review" by Divakar Yadav, Jalpa Desai, Arun Kumar Yadav.
3. "An Empirical Survey on Long Document Summarization: Datasets, Models and Metrics, HUAN YEE KOH, JIAXIN JU, MING LIU, SHIRUI PAN
4. "Text Summarization with Pretrained Encoders" by Yang Liu and Mirella Lapata
5. NLP FROM SCRATCH: TRANSLATION WITH A SEQUENCE TO SEQUENCE NETWORK AND ATTENTION
6. Build your own Transformer from scratch using Pytorch