FINAL REPORT ON QUESTION GENERATING TOOL

Shikha Kumari

1. **Introduction. An overview of the project and an outline of the report**

We decided to work on a project where we generate questions from answers and contexts using the SQuAD dataset. This project interests us because creating questions is important in understanding language, especially in education technology. SQuAD is a popular dataset used for testing how well machines can generate questions, which is why we chose it.

Report Outline:

Introduction

- Brief overview of the dataset and the NLP model (T5).

Background
Experimental Setup
Results
Discussion:

- Interpretation of results and implications.

Summary and Conclusions
References

- Links to relevant resources and references used.

2. **Description of the data set.**

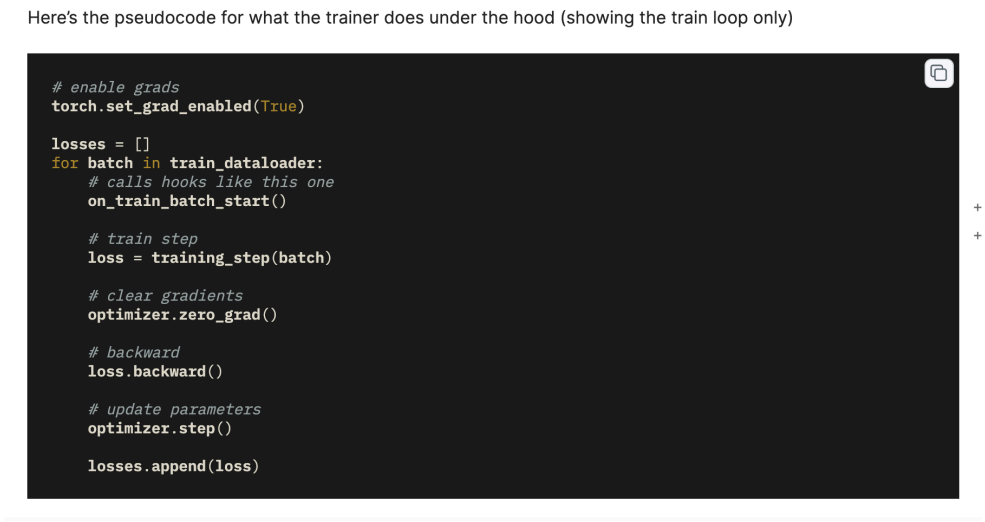
Each row represents a different article from Wikipedia, and each column has information about that article, such as the title, the text, and the questions people asked about it. The questions are usually short sentences, and the answers are even shorter, just a few words or a phrase. It's organized and easy to read, like a well-organized list of information.

3. **Description of the NLP model and what kind of algorithm you use. Provide some background information on the development of the algorithm and include necessary equations and figures.**

T5 for conditional generation is a transformer-based model that generates text based on input prompts or conditions. It uses an encoder-decoder architecture and is trained to produce output

text conditioned on the input. It's versatile and can be fine-tuned for various text generation tasks, making it widely applicable in natural language processing.

The below image is taken from Lightning AI website.



Background:
The T5 model extends the transformer architecture to perform text-to-text tasks, where both the input and output are textual. This approach allows for a unified training procedure, where a single model is trained on a diverse set of tasks by simply changing the input and output representations.
Model Architecture:
The T5 model consists of an encoder-decoder architecture, where the encoder processes the input text, and the decoder generates the output text. Both the encoder and decoder are composed of multiple layers of self-attention mechanisms and feed-forward neural networks.
Encoder:

- The encoder takes the input text and converts it into a fixed-size representation called the context vector.
- Each token in the input text is embedded into a high-dimensional vector space.
- Self-attention mechanisms capture relationships between tokens in the input text, allowing the model to understand the contextual information.
- Multiple layers of self-attention and feed-forward neural networks process the input text hierarchically, extracting increasingly abstract representations.

Decoder:

- The decoder takes the context vector generated by the encoder and produces the output text token by token.
- At each decoding step, the decoder attends to the context vector and previously generated tokens to predict the next token in the output sequence.

- It uses different decoding techniques for text generation, in our case we have used beam search with parameter '**num_beam = 5**'.

T5 is a masking based language:
Let's break down the steps of how T5 masks input:

1. **Tokenization**: This step converts the text into a sequence of tokens that the model can understand.
2. **Special Tokens**: T5 adds special tokens to the tokenized input. These tokens include a start-of-sequence token <s>, an end-of-sequence token </s>, and other task-specific tokens if applicable. These special tokens help the model understand the structure and boundaries of the input sequence.
3. **Attention Masks**: T5 uses attention masks to specify which tokens in the input should be attended to by the model and which ones should be ignored. This is done by assigning a value of 1 to tokens that should be attended to and a value of 0 to tokens that should be ignored. For example, padding tokens are typically masked with a value of 0 so that the model doesn't pay attention to them during processing.
4. **Conditional Generation**: T5 is a conditional generation model, meaning it generates output based on input conditions. The input text serves as the condition for generating the output. The model learns to generate output sequences conditioned on the input sequences it receives.

Tokenization:
Input Text: "How are you?"
Tokenized Input: ["How", "are", "you", "?"]

Special Tokens:
Special Tokens: <s>, </s>
Tokenized Input with Special Tokens: [<s>, "How", "are", "you", "?", </s>]

Attention Masks:
Attention Mask: [1, 1, 1, 1, 1, 1]

Conditional Generation:
Input Text: "How are you?"
Output Text: "I am fine."

Task-Specific Tokens:
Task-Specific Token: <summarize>
Tokenized Input with Task-Specific Token: [<s>, <summarize>, "How", "are", "you", "?", </s>]

Masking Tokens:
Masked Tokens: ["How", "[MASK]", "you", "?"]

4. **Experimental setup. Describe how you are going to use the data to train and test the model. Explain how you will implement the model in the chosen framework and how you will judge the performance.**

T5 for conditional generation is a transformer-based model that generates text based on input prompts or conditions. It uses an encoder-decoder architecture and is trained to produce output text conditioned on the input. It's versatile and can be fine-tuned for various text generation tasks, making it widely applicable in natural language processing.

1. **Data Preparation**: The data.py script loads the SQuAD dataset using the Hugging Face datasets library, extracts relevant information such as context, question, and answer, and creates dataframes for training and validation. It filters out long answers and saves the preprocessed data into CSV files (squad_t5_train.csv and squad_t5_val.csv).
2. **Experimental Setup**:

   When we train a model using PyTorch Lightning, it automatically creates a lightning_logs directory to store logs and checkpoints during training.

   Here's a breakdown of the files and folders found inside the lightning_logs directory:

- **Versioned Checkpoints**: Inside the lightning_logs directory, folders named version_X (in our case it is just a single version), where X is a numerical value representing the version of the training run. These folders contain model checkpoints saved at specific intervals during training.
- **hparams.yaml**: This file contains the hyperparameters used for training the model. In our case it is batch_size: 4.
- **metrics.csv**: This CSV file stores the loss (in our case) metrics logged during training (at each step) and validation (at every last step of an epoch).
- **Model Implementation**:
    - The chosen model architecture is T5ForConditionalGeneration from the Hugging Face Transformers library. This model is initialized and loaded with pretrained weights using T5ForConditionalGeneration.from_pretrained('t5-base').
    - The model is encapsulated within a PyTorch Lightning module named T5FineTuner. This module inherits from pl.LightningModule and contains methods for training, validation, forward pass, and optimization.
    - The T5FineTuner class implements the forward method, which defines the forward pass of the model. It takes input tensors, such as input_ids, attention_mask, decoder_attention_mask, and lm_labels, and computes the model's output, including logits and loss.
    - The training and validation steps are defined within the T5FineTuner class as training_step and validation_step methods, respectively. These methods take batches of data as input, perform forward passes through the model, compute loss, and log the loss values for monitoring.

    - Data loaders for training and validation datasets are defined within the T5FineTuner class as train_dataloader and val_dataloader methods, respectively. These data loaders handle batching, shuffling, and loading of data for training and validation.
    - The model's optimizer is configured within the configure_optimizers method of the T5FineTuner class. In this script, the AdamW optimizer is used with a specified learning rate.
- **Performance Evaluation**:
    - During training, PyTorch Lightning logs the training and validation losses at specified intervals. These losses are tracked over epochs and can be visualized using tools like TensorBoard or plotted using libraries like Matplotlib. In our case we have used matplotlib to visualize the loss for both training and validation.
    - The primary metric used for judging performance is the loss, which represents the discrepancy between the model's predictions and the ground truth labels. Lower loss values indicate better model performance.

5. **What kind of hyper-parameters did you search on? (e.g., learning rate)? How will you detect/prevent overfitting and extrapolation?**

We used hyperparameters such as batch size and learning rate. During optimization, a learning rate of 0.0003 and an epsilon value of 0.0000001 were used.

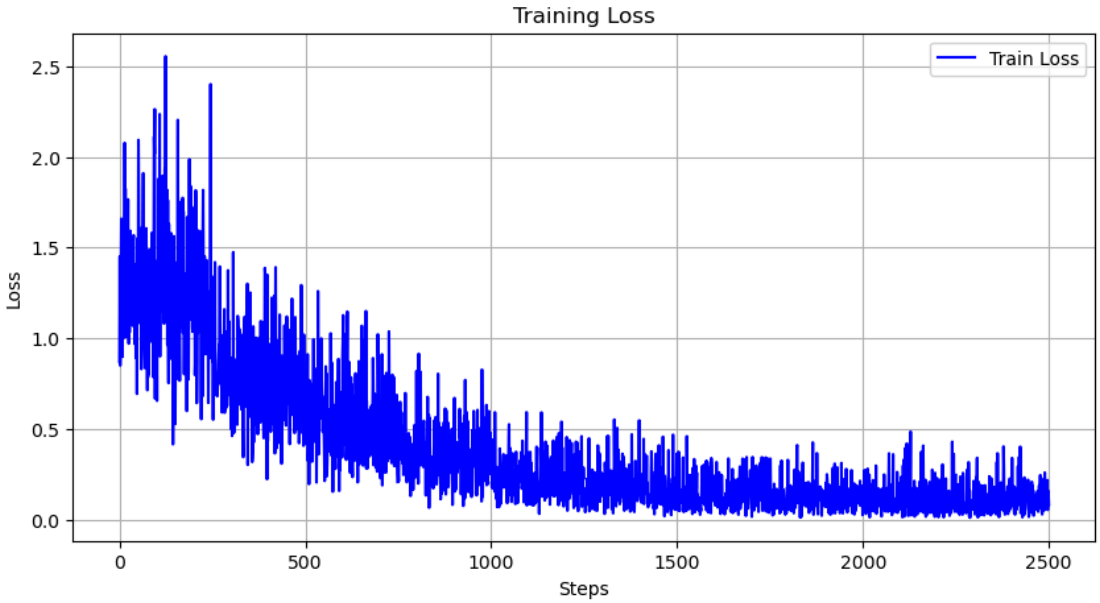
To detect overfitting, we can gauge the performance on the validation dataset by plotting the val_loss and in our case we see that the validation loss seems to be increasing as per steps.
Refer to the graphs under question 6.

To prevent overfitting:
Hyperparameter Tuning: Fine-tuning hyperparameters such as learning rate, batch size, or optimizer settings can also help alleviate overfitting. Experimenting with different hyperparameter configurations using techniques like **grid search or random search** can help identify the settings that would lead to better generalization performance.

6. **Results. Describe the results of your experiments, using figures and tables wherever possible. Include all results (including all figures and tables) in the main body of the report, not in appendices. Provide an explanation of each figure and table that you include. Your discussions in this section will be the most important part of the report.**

From the below training loss plot, we see that during training, the loss decreases, and after a certain point/steps (after approx. 1100 steps), the training loss remains the same, suggesting that the model is converging. Initially, the model rapidly learns from the training data, resulting in a significant reduction in loss. As training progresses, the model's improvement slows down, and it reaches a point where further optimization does not lead to substantial decrease in loss.

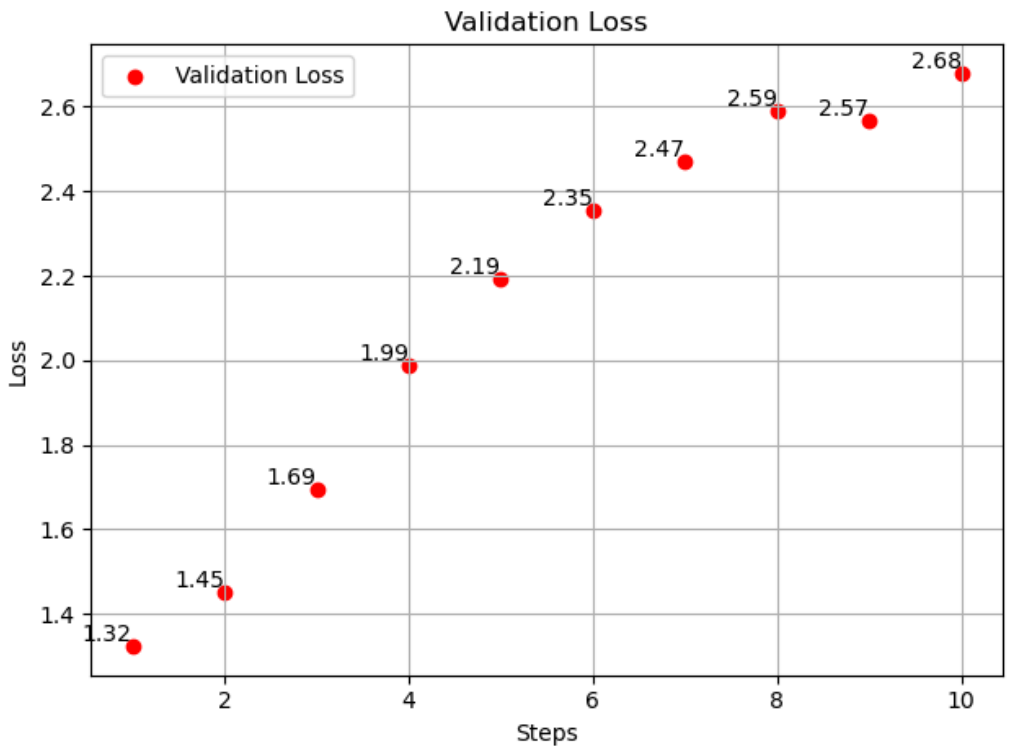
This plateauing of the loss indicates that the model has learned the features and patterns present in the training data to a large extent, and additional training iterations may not yield significant improvements.



When the validation loss starts to increase per step while the training loss plateaus, it strongly suggests that the model is overfitting. Overfitting occurs when a model learns to fit the training data too closely, capturing noise specific to the training set rather than learning generalizable patterns.

As a result, when the model encounters new, unseen data during validation, it struggles to generalize its learned patterns and starts to perform poorly. This manifests as an increase in validation loss, indicating that the model's performance is deteriorating on unseen data.

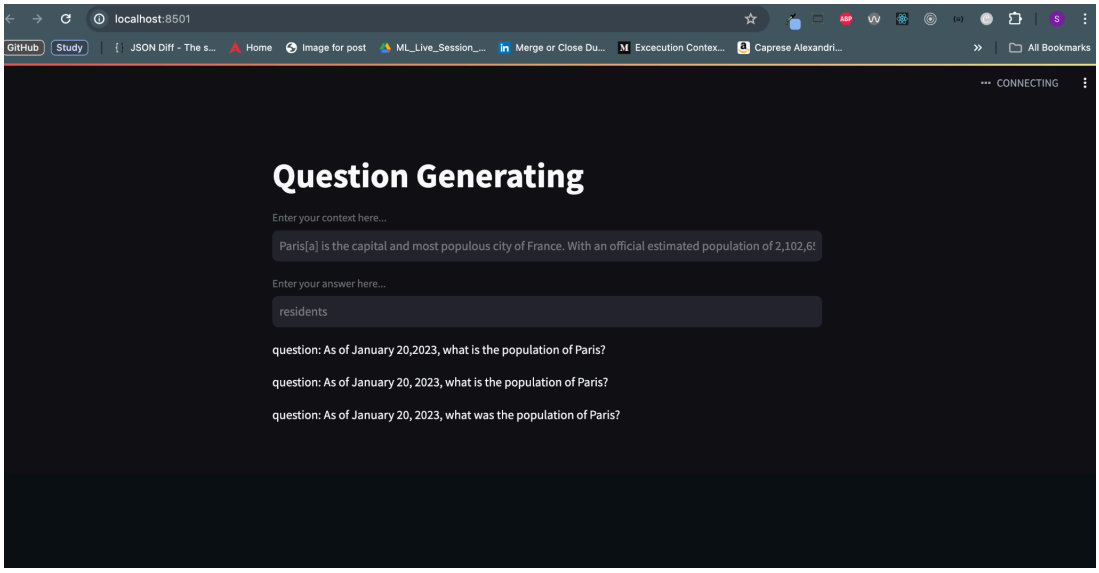
In our case we can see that the model is overfitting on the validation dataset by looking at the below val_loss figure.

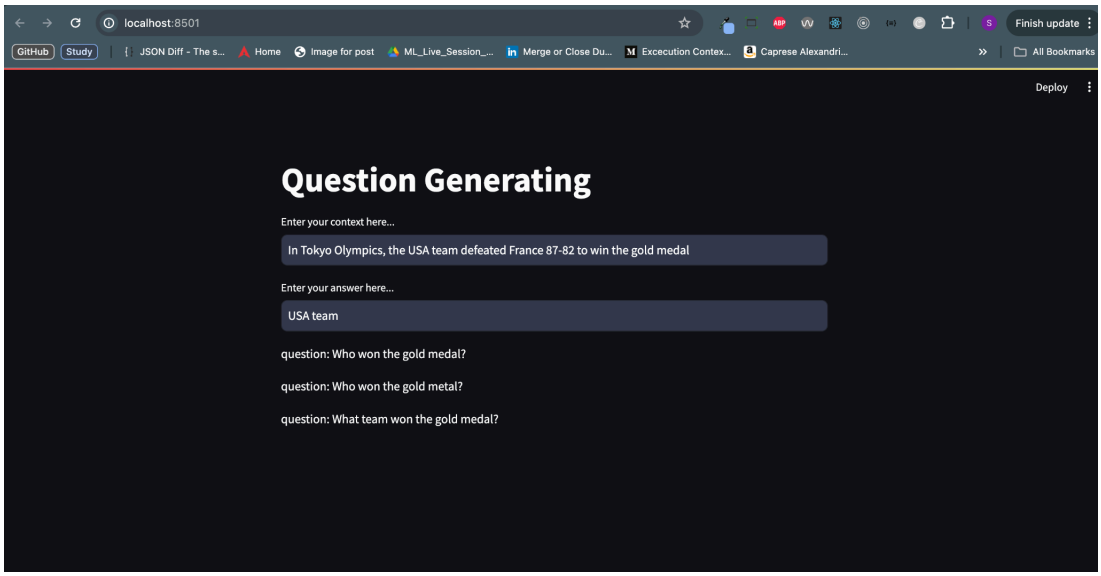


The questions generated below two screenshots suggests that due to overfitting, the model's ability to generate diverse and meaningful questions has diminished.

In this context, observing three nearly identical questions indicates a lack of diversity and creativity in the model's outputs. Instead of generating a range of questions that capture different aspects of the input passage or context, the model repeatedly produces similar questions, suggesting a limited capacity to adapt and generalize beyond the training data.
Several fine-tuning techniques could be implemented to address the overfitting observed in the model:

1. **Regularization Techniques**: Techniques like dropout or weight decay can be applied during training to prevent overfitting.
2. **Early Stopping**: Early stopping involves monitoring the model's performance on a validation set and halting training when the performance starts to degrade. This prevents the model from overfitting to the training data by stopping training before it begins to memorize noise or irrelevant patterns.
3. **Hyperparameter Tuning**: Fine-tuning hyperparameters such as learning rate, batch size, or optimizer settings can also help alleviate overfitting. Experimenting with different hyperparameter configurations using techniques like **grid search or random search** can help identify the settings that lead to better generalization performance.





7. **Summary and conclusions. Summarize the results you obtained, explain what you have learned, and suggest improvements that could be made in the future.**

I've gained knowledge about various applications of the T5 model, understanding how it functions at its core. Additionally, I've explored PyTorch Lightning, a framework that simplifies the process of building and training deep learning models. I've also delved into the concept of wrapper classes, which provide a user-friendly interface while encapsulating complex functionalities underneath.

The training process showed a decrease in loss over time, suggesting that the model was learning from the data. However, the validation loss started to increase, indicating potential

overfitting. This could be attributed to the model learning the training data too well and struggling to generalize to unseen data. To address this, further fine-tuning and regularization techniques may be necessary. Additionally, the generated questions during evaluation appeared similar, indicating a need for more diverse and accurate question generation.

To mitigate overfitting in this scenario, various strategies can be employed. Regularization techniques such as dropout or weight decay can be applied to prevent the model from fitting the noise in the training data too closely. Additionally, techniques such as early stopping, where training is halted when the validation loss starts to increase, can prevent the model from over-optimizing on the training data.

8. **References. In addition to references used for background information or for the written portion, you should provide the links to the websites or github repos you borrowed code from.**

Background Information:
ChatGPT and https://lightning.ai/docs/pytorch/stable/common/trainer.html

Pytorch Lightning Reference:
https://lightning.ai/docs/pytorch/stable/common/trainer.html

GitHub Reference:
https://github.com/ZhangChengX/T5-Fine-Tuning-for-Question-Generation/blob/main/train.py