# Code Generation Documentation using CodeT5 Model

**Preprint** · June 2023

DOI: 10.13140/RG.2.2.18464.15368

Omar Shaalan
American University in Cairo

# Code Generation Documentation using CodeT5 Model

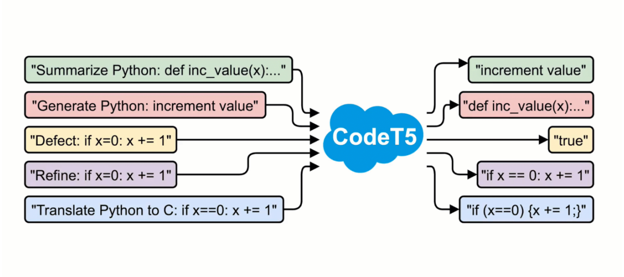
Kyrollos Soliman 900203309 - Omar Shaalan 900193887
The American University in Cairo

## ABSTRACT

Natural langauge models have been existing for a while now. Along the years they have showed success in certain areas, such as GPT. However, there is always a room for improvement. This paper introduces CG Code Generation / Documentation model based on CodeT5, which is a fine-tuned version of the natural language model CodeT5 (Wang et al., [2021]). The model discussed in this paper will generate commented code instead of generating code without any type of documentation and or description to the mechanincs of the code and what the code does explicitly. Deep learning and machine learning models usually work on a dataset. However, datasets that store a pair of commented code along with a natural language describing the code do not exist yet, at least as an open source dataset that is available to the public. To tackle such a problem a new dataset had to be created. In this paper all the details of creating a new dataset with the name PyPlus with the its foundation base CodeSearchNet was done. Moreover, reading the paper will allow you to understand how the ConcodePlus task, which is responsible for the commented code generation, was configured and how the model trained on it. You can check also the webpage discussing the model and read more about the future recommended work in case you want to work on the model yourself.

## 1 INTRODUCTION

Code generation is a vital issue in our modern world. To elaborate, millions of programmers graduate each year from computer engineering and computer science fields from all over the world. Those programmers are the foundation of fields such as cybersecurity, databases, networking, AI, and many other fields. These programmers need a tool that generates code for them that will save them time to focus on much more important things. Moreover, there are seekers who seek learning programming languages. Such people will benfit hugely from a code generation model that generates code and act as a guide for them. That being said, there still remains the problem of not understanding the code. Code generation is different than understanding the code. To make it clearer, the code that is generated has no explanation to its behaviour or a description of its dynamics. Such a problem still needs a solution. To encounter such a proposed problem a code should be generated with comments that give an obvious description to the dynamics of the code and how it works. The fine-tuned model discussed in this paper is responsible for generating a commented code that will aid in resolving the issue of the black box issue of having code that is not understood. The base model that was fine-tuned is CodeT5 small, with sixty million parameters (Wang et al., [2021]), CodeT5 is responsible for several tasks that work with code as illustrated in the figure below.

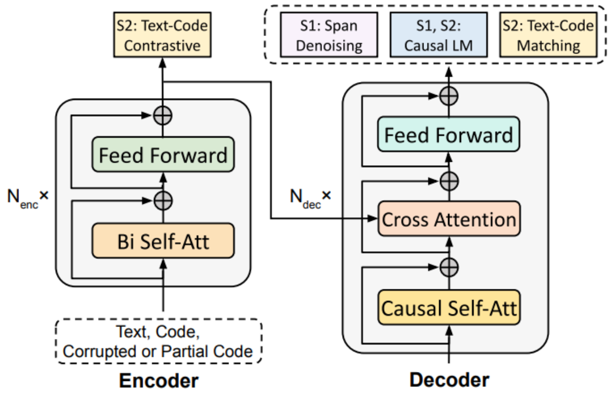


Some of these tasks are code summarization, which is a task that is responsible for summarising an existing code or in other words document it. Another task is Concode, this is the task responsible for code generation. To elaborate, this task gets a prompt in natural language as an input and generates the desired code described in the prompt as an output (Wang et al., [2021]). The ConcodePlus, which is not found below in the figure with the other tasks, is the task responsible for commented code generation. It does such an operation by taking the natural language prompt from the user and instead of generating a uncommented code, it generates the code with the comments at the same time as on output. The technicalities of such an operation is discussed in details within this paper.

## 2 MODEL DETAILS AND BACKGROUND

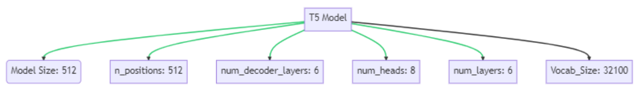
### 2.1 General Model Architecture

CodeT5 is a seq2seq model whose architecture consists of an encoder and a decoder. The encoder is responsible for learning contextual representations from code/text sequences, which can be complete, partial, or span-masked sequences. On the other hand, the decoder is trained to generate various outputs based on the pretraining learning tasks, which is commented code in this case. The pretraining process involves two stages: The first involves training with a unimodal code corpus, and the second involves training with a bimodal code-text corpus.(Wang et al., [2023])



## 2.2 Detailed description of used model hyperparameters

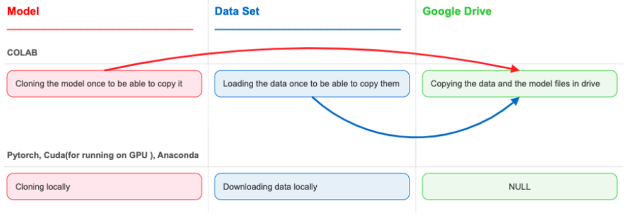
- Model Size: The T5 model has a $d_{\text{model}}$ (dimension of the model) of 512, meaning the hidden states and embeddings within the model have a dimensionality of 512.
- Number of Layers: The T5 model has 6 layers both in the encoder and the decoder, as indicated by `"num_layers": 6` and `"num_decoder_layers": 6`.
- Attention Mechanism: The T5 model uses multi-head attention. The configuration specifies that each attention head has `num_heads` (number of attention heads) set to 8.
- Feed-Forward Networks: The feed-forward networks (FFNs) within the T5 model have a hidden dimension of $d_{\text{ff}}$ (2048 in this case).
- Position Embeddings: The T5 model has a maximum of `n_positions` (512) position embeddings.
- Vocabulary Size: The model's vocabulary size is `vocab_size` (32100), meaning it can handle a vocabulary of up to 32,100 tokens.



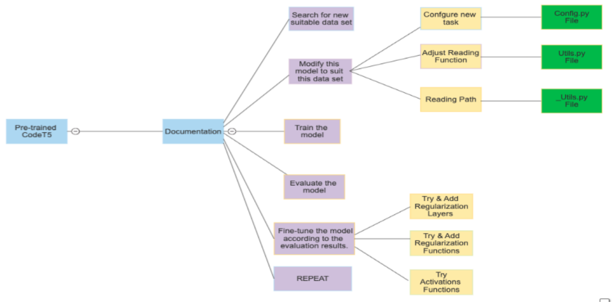
All of them can be summarized in the previous figure.

## 3 METHODOLOGY

The very first thing that should be done was deploying the foundation model, which is in this case CodeT5. The issue was what version of the model to deploy so we can start working on. There are three variations of CodeT5, the small, base, and large. They have 60 million , 220 million and 770 million parameters respectively (Wang et al., [2021]). These number of parameters requires huge computational power that will have a very high cost. That is why the first and only option available was the small version, as it was the one that required the least computational power. Moreover, even though the small version has the least number of parameters, it still exceeded the computational power provided by Google Colab services. To be more specific it exceeded the GPU RAM, which is 16 GB RAM, provided by Colab. Another method to deploy the model was to be found. The model was deployed on a local machine in the American University in Cairo in the Machine Learning lab. The machine had 16 GB which is more than colab's GPU. However, the 16 GB version barely served the purpose without crashing as still the number of parameters is very large even though this is the small version. We used Anaconda to create virtual environments as working with virtual environments to have several approaches with different libraries, especially as we deployed the model locally. Below is an illustrative figure of the deployment experience: -


Model
Data Set
Google Drive
COLAB
Cloning the model once to be able to copy it
Loading the data once to be able to copy them
Copying the data and the model files in drive
Pytorch, Cuda(for running on GPU ), Anaconda
Cloning locally
Downloading data locally
NULL

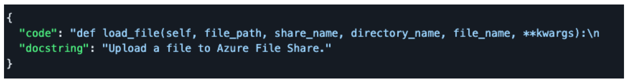
The second important thing was configuring and adding the task. To do so we changed mainly in three files. To configure the new task ConcodePlus we had to configure a new task with the name "ConcodePlus" in the "config.py" file. Then we needed to create a reading function that will read the dataset in order for the model to train on the new task. To do such an option a reading function was created and added to the "utils.py" file that have all the reading functions regarding the tasks. Finally we had to update the datapath for the new dataset that will has stores the commented code in it in order for the model train on the dataset for a specified number of epochs [check the technical report] to be able to generalize and generate commented code for as many prompts as it can. Below is an illustrative figure of the files changed experience: -


Pre-trained CodeT5
Documentation
Search for new suitable data set
Modify this model to suit this data set
Train the model
Evaluate the model
Fine-tune the model according to the evaluation results.
REPEAT
Confgure new task
Adjust Reading Function
Reading Path
Config.py File
Utils.py File
_Utils.py File
Try & Add Regularization Layers
Try & Add Regularization Functions
Try Activations Functions

A person may wonder about whether it was easy or hard to find a dataset that has the commented code stored. The answer to such a question is that it was the furthest thing from the word easy. To make it clear, there was no such a dataset that fitted the purpose of the new task. All the datasets found were either storing pairs of uncommented code with a descriptive natural language to it, or pairs of code and prompts that act as inputs to the models. To sum it up, a dataset with the needed spesifications were not found. Hence, a dataset had to be created. The dataset used in this project is a new dataset that can be found only on a couple of repos owned by the project and paper authors. The dataset was not created from scratch, it has a foundation dataset, which is CodeSearchNet. In the dataset section below more details will be discussed regarding the approach of creating the new dataset.

## 4 DATASET

This section focuses more on the dataset. To make it clearer, this section will discuss the different approaches taken to create the new dataset as mentioned earlier in the paper. As mentioned earlier, the created dataset, PyPlus, has CodeSearchNet as its base. In other words the dataset was not created from scratch. More than four python scripts were written in order to make modifications to the python subset in the CodeSearchNet dataset, so it becomes a stand alone dataset that has a 250,000 row of commented code. The scripts had different functions, some scripts removed parts from the dataset and others replaced the removed parts with comments, and the last type of scripts are scripts that were reordering the places of the comments corresponding to the code itself; in other words making sure that the comments are in the right place and were not accidentally erased by any of the other scripts. The figure below shows the new dataset and the commented code



```
{
  "code": "def load_file(self, file_path, share_name, directory_name, file_name, **kwargs):\n
  "docstring": "Upload a file to Azure File Share."
}
```

This sample demonstrates a code snippet that defines a loadfile function in Python. The corresponding natural language prompt describes the functionality of the code, which is to upload a file to an Azure File Share. The PyPlus dataset consists of pairs of natural language prompts and commented code snippets. Each pair is designed to capture the mapping between a human-readable prompt and its corresponding code implementation. The datasets contains only Python codes. As you can guess from the name ;) Here are some key details about the PyPlus dataset:

- Size: The dataset contains more than 250,000 pairs of prompts and code snippets. And it is 282MB total
- Language: The dataset is solely focused on the Python programming language.
- Commented Code: Each code snippet is accompanied by comments that explain its functionality and purpose.
- Training and Evaluation Sets: The dataset is split into three: training, evaluation,and test subsets. Note you can find the training split here as it was too large to be uploaded here.

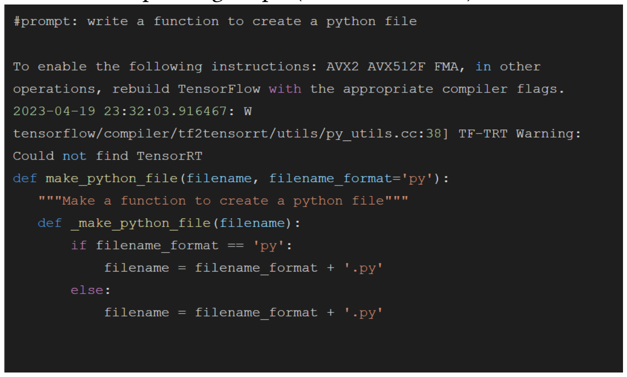
To use the PyPlus dataset in your research or projects, follow these steps:

(1) Clone the Repository: Begin by cloning this repository to your local machine using the Git command:
git clone https://github.com/kyrolloszakaria/PyPlus.git
(2) Explore the Dataset: Take some time to explore the dataset and familiarize yourself with its structure and contents. You can access the natural language prompts and their corresponding commented code snippets.
(3) Preprocessing: Depending on your specific requirements, you may need to preprocess the dataset before using it in your models. You can apply text cleaning, tokenization, or normalization techniques as necessary.
(4) Model Training: Utilize the PyPlus dataset for training your models. You can use it to develop models that can generate code from natural language prompts or to evaluate the performance of existing code generation models.
(5) Model Evaluation: The PyPlus dataset also serves for evaluation and test purposes. You can assess the quality and accuracy of your code generation models by comparing their output with the annotated code snippets in the dataset.

We welcome contributions to the PyPlus dataset repository. Feel free to submit a pull request if you have additional annotated code snippets or improvements to the existing dataset. Please follow the guidelines mentioned in the README.md file to contribute to this repository.

## 5 INPUT OUTPUT EXAMPLE

The following is an example of the input(prompt) given to the model and the corresponding output(commented code).



```
#prompt: write a function to create a python file

To enable the following instructions: AVX2 AVX512F FMA, in other
operations, rebuild TensorFlow with the appropriate compiler flags.
2023-04-19 23:32:03.916467: W
tensorflow/compiler/tf2tensorrt/utils/py_utils.cc:38] TF-TRT Warning:
Could not find TensorRT
def make_python_file(filename, filename_format='py'):
   """Make a function to create a python file"""
   def _make_python_file(filename):
       if filename_format == 'py':
           filename = filename_format + '.py'
       else:
           filename = filename_format + '.py'
```

Here the prompt was: "write a function to create a python file". The model, relatively, succeeded in generating a commented code satisfying the input prompt.

## 6 EVALUATION

After training, the model proceeds to the evaluation phase. It is evaluated with the perplexity, BLEU-4, exact match (EM) accuracy, and CodeBLEU (Ren et al., [2020]) that is concerned with syntactic and semantic matches based on the code structure and the n-gram match.

In the following subsections, the definitions of the metrics will be presented, the calculation methodology will be elaborated upon, and the implementation details within the model code will be described.

### 6.1 Perplexity Score Calculation for CodeT5

The perplexity score is a measure of how well the model predicts the target sequences given the source sequences. Lower perplexity indicates better performance. (Song et al., [2022])

**Calculations for Perplexity:**

$$\text{Perplexity} = \exp\left(\frac{\text{Average Evaluation Loss}}{\text{Number of Tokens}}\right) \tag{1}$$

In this equation, the Average Evaluation Loss represents the cumulative evaluation loss divided by the number of tokens processed. The exponential function exp() is applied to convert the average evaluation loss back to a probability-like scale. A lower perplexity indicates a better-performing language model.

The Average Evaluation Loss is the average negative log-likelihood per token, which quantifies the discrepancy between the model's predicted probabilities for the target tokens and the actual target tokens.

**Equation for Average Evaluation Loss:**

$$\text{Average Evaluation Loss} = \frac{\sum_{i=1}^{N} \text{Loss}_i}{\text{Number of Batches}} \tag{2}$$

In this equation, $\text{Loss}_i$ represents the evaluation loss for each individual batch, and $N$ represents the total number of batches. The average evaluation loss is computed by summing up all the individual losses and dividing by the total number of batches.

The following is the technical description of how the evaluation is implemented.

If `args.do_eval` is set to `True`, the code proceeds to evaluate the code perplexity score on the evaluation data.

(1) Loading the Evaluation Data:
   - The code checks if this is the first epoch by checking if there is an evaluation loss value in the `dev_dataset` dictionary. If not, it loads the evaluation examples and data by calling the `load_and_cache_gen_data` function with the evaluation file obtained as argument (`args.dev_filename`) and other relevant parameters.
   - The evaluation data is cached in the `dev_dataset` dictionary is created for future use.

(2) Computing Perplexity (PPL) Score:
   - The `eval_ppl_epoch` function is called to calculate the perplexity score (`eval_ppl`) of the model on the evaluation data. This function takes the evaluation data, examples, model, and tokenizer as inputs and calculates the perplexity according to the equations illustrated earlier.

(3) Logging and Recording PPL:
   - The evaluation results, including the epoch number (`cur_epoch`), global step, and perplexity score, are logged using the `logger.info` function.
   - If `args.data_num` is set to -1 (indicating full evaluation), the perplexity score is also written to a TensorBoard summary writer (`tb_writer`) with the tag `'dev_ppl'`.
   - If `args.save_last_checkpoints` is `True`, the last model checkpoint is saved by creating a `checkpoint-last` directory within the `args.output_dir` and saving the model's state dictionary in a file named `"pytorch_model.bin"`.

(4) Tracking the Best Perplexity:
   - The code checks if the obtained perplexity score (`eval_ppl`) is better than the previously recorded best perplexity score (`best_ppl`).
   - If the current perplexity is better, the `not_loss_dec_cnt` counter is reset to 0, indicating that the perplexity has not decreased for 0 consecutive epochs.
   - The new best perplexity score is logged, and a message is written to the file `summary.log` indicating the change.
   - The best model checkpoint is saved in a directory named `checkpoint-best-ppl` within the `args.output_dir` directory, similar to the last checkpoint.
   - If the current perplexity is not better, the `not_loss_dec_cnt` counter is incremented, indicating that the perplexity has not decreased for consecutive epochs.
   - Freeing CUDA Memory:
     – Before proceeding to the next epoch, the code calls `torch.cuda.empty_cache()` to release any cached GPU memory, helping to manage memory consumption.

## 6.2 BLEU Score Calculation for Code T5

BLEU (Bilingual Evaluation Understudy) score is a commonly used metric to evaluate the quality of the generated code against reference code examples. BLEU measures the similarity between the generated code and the reference code by comparing their n-grams. [2] The BLEU score in codeT5 is calculated using the following steps:

- **Tokenization:**
  – The reference code examples and the generated code are tokenized into n-grams, where $n$ represents the desired n-gram order.
  – Let $R$ be the set of all reference n-grams.
  – Let $C$ be the set of all generated n-grams.
  – Let $r$ be the length of the reference code.
  – Let $c$ be the length of the generated code.
- **Precision Calculation:**
  – For each n-gram in $C$, the number of occurrences in $R$ is counted, denoted as $C_{\text{count}}$.
  – The precision for each n-gram is computed as

$$P_n = \frac{\sum_{\text{n-gram} \in h} \min(\text{count}_{\text{ngram}}, \text{count}_{\text{clip}})}{\sum_{\text{n-gram} \in h} \text{count}_{\text{ngram}}} \tag{3}$$

  , where $h$ represents the hypothesis (generated code) and $r_1, r_2, ..., r_m$ represent the set of reference translations.
  – For a single n-gram, the count of its matches is given by:

$$\text{count}_{\text{ngram}} = \text{Count of the n-gram in } h \tag{4}$$

$$\text{count}_{\text{clip}} = \max_{r_i \in \{r_1, \ldots, r_m\}} (\text{Count of the n-gram in } r_i) \tag{5}$$

- **Brevity Penalty:**
  – The brevity penalty $BP$ is used to penalize translations that are shorter than the closest reference translation length.
  – It is calculated as follows:

$$BP = \begin{cases} e^{1 - \frac{\text{closest\_ref\_length}}{\text{hyp\_length}}}, & \text{if hyp\_length} < \text{closest\_ref\_length} \\ 1.0, & \text{otherwise} \end{cases} \tag{6}$$

  where hyp_length is the length of the hypothesis (generated code), and closest_ref_length is the length of the closest reference translation.
- **Modified Precision Calculation:**
  – Each n-gram precision score $P_n$ is multiplied by the brevity penalty $BP$ to obtain the modified precision $MP_n = P_n \cdot BP$.
- **BLEU Score Calculation:**

  - The modified precision scores are combined using the geometric mean:

$$\text{BLEU} = \exp\left(\frac{1}{n}\sum_{i=1}^{n}\log(MP_i)\right) \tag{7}$$

    where $n$ represents the maximum n-gram order considered (typically 4).

The following will explain the implementation of evaluating the BLEU score in the model.

(1) Evaluating BLEU Score:
  - The evaluation examples and data are loaded again using `load_and_cache_gen_data`, this time with additional parameters: `only_src=True` to load only the source sequences and `is_sample=True` to indicate that it is a sampling evaluation.
  - The `eval_bleu_epoch` function is called to calculate the BLEU score (`dev_bleu`) and exact match (EM) score (`dev_em`) on the evaluation data. The function takes the evaluation data, examples, model, tokenizer, evaluation type (`dev`), and other relevant inputs.

(2) Calculate the combined BLEU and exact match score (`dev_bleu_em`).

(3) If `args.data_num` is -1, add the `dev_bleu_em` value to the TensorBoard summary writer (`tb_writer`) with the tag `'dev_bleu_em'`. In training (*concodePlus*), the value -1 is passed to the model to log the metric values to TensorBoard.

(4) Keep track of not increasing BLEU score by checking if the current `dev_bleu_em` is higher than the previous `best_bleu_em`.
  - If true:
    - Reset the `not_bleu_em_inc_cnt` counter to 0.
    - Log the updated best BLEU and exact match scores.
    - Write a message to the file `summary.log` indicating the change.
    - Save the best model checkpoint in the directory `checkpoint-best-bleu` within `args.output_dir`.
    - If `args.data_num` is -1 or `args.always_save_model` is true, save the model's state dictionary in the file `pytorch_model.bin`.
  - Else:
    - Increment the `not_bleu_em_inc_cnt` counter by 1.
    - Log the information that the BLEU score has not increased for a certain number of epochs.
    - Write a message to the file `summary.log` indicating the lack of improvement.
    - Check if both `not_bleu_em_inc_cnt` and `not_loss_dec_cnt` exceed the patience threshold (`args.patience`). If true, trigger early stopping by breaking the loop.

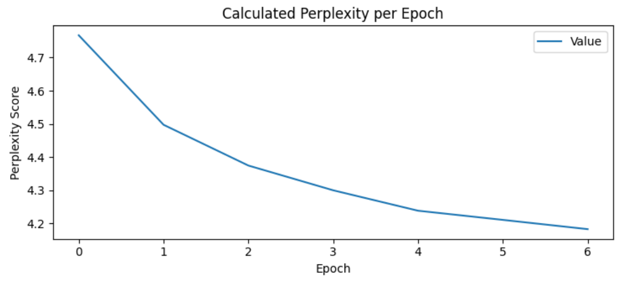
## 7 RESULTS

The following table presents a comparative analysis of metric values for two variants of the CodeT5 model. The first variant is CodeT5 finetuned on the Conocde dataset, specifically targeting the Concode task. The second variant is CodeT5 finetuned on the PyPlus dataset, with a focus on the ConcodePlus task. By examining the metric values associated with each variant, we gain insights into the performance and effectiveness of the CodeT5 model in these distinct contexts.

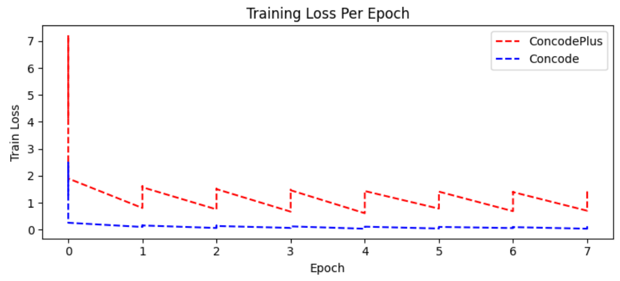
**Table 1:** Comparison of Metric Values for CodeT5 Variants

| Metrics | Conocde | ConcodePlus |
|---|---|---|
| EM | 19.6 | 9.23 |
| Perplexity | 1.23 | 4.21 |
| Bleu | 29.33 | 21.52 |

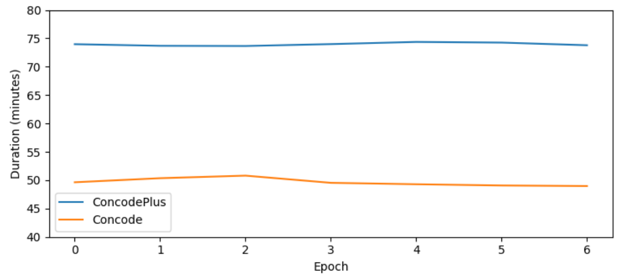
These results indicates that the ConcodePlus task exhibits relatively lower accuracy in terms of Exact Match, higher perplexity score, and lower BLEU score compared to the Concode task.From these findings, it can be inferred that the performance of the ConcodePlus task is comparatively poorer than that of the Concode task. This disparity is justifiable considering the significant difference in dataset size between the two tasks and the additional complexity introduced by the need to comprehend and distinguish between code and comments. You can find the plot of the perplexity:



Furthermore, the following figure illustrates the training loss of each epoch.



The model training loss converges around the value of 1.00, a high training loss relative to the training loss of the Concorde Task. This may indicate an acceptable performance with slight underfitting. Moreover, the ConcodePlus task exhibits signs of Training instability due to the fluctuations in the training loss, possibly due to noise in the training dataset. Finally, you can see the difference in training time between the two tasks in the following figure.



As illustrated, there is around twenty-five minutes difference in each epoch between the two tasks. This further emphasizes the difference of complexity and computational power needed between the two tasks, Concode and ConcodePlus.

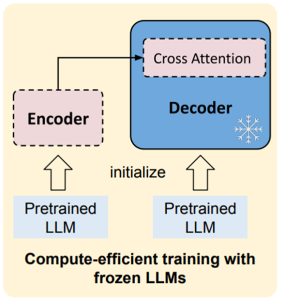
## 8 CONCLUSION

In conclusion, we have successfully achieved our goal by incorporating the ConcodePlus task into CodeT5 and creating the PyPlus dataset. The output provided above clearly demonstrates that our model now can generate meaningful documentation along with code. While we have achieved promising results, there are still areas that require further improvement. One potential area for enhancement is the addition of reinforcement learning to ConcodePlus.</p>

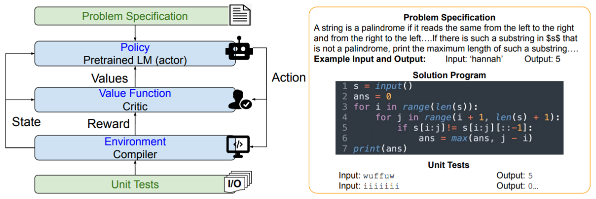
During our project, we discovered the incredible power of transformers in deep learning. It was fascinating to see how these models effectively capture complex relationships and dependencies in both text and code. Furthermore, We were surprised by how similar text generation models are similar to code generation model as both of them relies on Large-language models but with more specialized training datasets in case of code generation that's why we are proud of the dataset we tailored for our model. This project was a wholesome learning experience that proved to be more challenging, which further motivated us to solve our problem and apply all we learnt throughout the course.

## 9 RECOMMENDATIONS

To enhance computational efficiency during training, the diagram on the right illustrates the use of frozen code Language Models (LLMs). This approach allows us to scale up the model while utilizing compute resources efficiently. The model architecture follows a "shallow encoder and deep decoder" design, where the encoder is kept relatively small, and only the small encoder and cross-attention layers are trainable. In contrast, the deep decoder LLM is frozen, meaning its weights are not updated during training.(Wang et al., [2023])



Furthermore, to unleash more capabilities of the model, Reinforcement learning can be introduced into the model training. (Le et al., [2022])



**Problem Specification**
A string is a palindrome if it reads the same from the left to the right and from the right to the left….If there is such a substring in $s$ that is not a palindrome, print the maximum length of such a substring….
**Example Input and Output:** Input: 'hannah' Output: 5

**Solution Program**

```
s = input()
ans = 0
for i in range(len(s)):
    for j in range(i + 1, len(s) + 1):
        if s[i:j]!= s[i:j][::-1]:
            ans = max(ans, j - i)
print(ans)
```

**Unit Tests**
Input: wuffuw Output: 5
Input: iiiiiii Output: 0…

PPOCoder, a new framework for code generation that combines pre-trained PL models with Proximal Policy Optimization (PPO) deep reinforcement learning and employs execution feedback as the external source of knowledge into the model optimization. (Shojaee et al., [2023]) to focus more on specific features of code, including but not limited to compilability as well as syntactic and functional correctness.

## REFERENCES

[1] Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven C. H. Hoi. 2022. CodeRL: Mastering Code Generation through Pretrained Models and Deep Reinforcement Learning. (2022). arXiv:cs.LG/2207.01780
[2] Kishore Papineni, Salim Roukos, Todd Ward, and Wei jing Zhu. 2002. BLEU: a Method for Automatic Evaluation of Machine Translation. 311–318.
[3] Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. 2020. CodeBLEU: a Method for Automatic Evaluation of Code Synthesis. (2020). arXiv:cs.SE/2009.10297
[4] Parshin Shojaee, Aneesh Jain, Sindhu Tipirneni, and Chandan K. Reddy. 2023. Execution-based Code Generation using Deep Reinforcement Learning. (2023). arXiv:cs.LG/2301.13816
[5] Yixiao Song, Kalpesh Krishna, Rajesh Bhatt, and Mohit Iyyer. 2022. SLING: Sino Linguistic Evaluation of Large Language Models. (2022). arXiv:cs.CL/2210.11689
[6] Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi D. Q. Bui, Junnan Li, and Steven C. H. Hoi. 2023. CodeT5+: Open Code Large Language Models for Code Understanding and Generation. (2023). arXiv:cs.CL/2305.07922
[7] Yue Wang, Weishi Wang, Shafiq Joty, and Steven C. H. Hoi. 2021. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. (2021). arXiv:cs.CL/2109.00859