Exploring Semantic Structure for Code Summarization with Language Models

Kevin Wedage
University of Alberta
Department of Electrical and Computer Engineering
kwedage@ualberta.ca

Abstract

Automated code summarization is a task that involves using Code Language Models (CLMs) to automatically generate English summaries for code segments. In this work, we explore the impact of using different input formats to help CLMs on the Code Summarization Task (CST). Specifically, we explore the use of three novel pre-training tasks, the impact of additional pre-training on a small dataset, and different methods of fine-tuning. Out of the box, using additional input formats results in inferior performance on the CST, but additional pre-training can increase this performance slightly. In some cases, the new models we propose result in better predictions.

1. Introduction

Code summarization using CLMs is a task that involves taking a block of code, typically a function, and returning an English summary of what the code intends to do. Automating the process of code summarization can help software developers easily understand the general purpose of new and complex snippets of code, and automatically add natural language summaries to their own code. For this reason, automated code summarization can be used to increase the productivity of software developers.

The main benchmark for evaluating CLMs on the CST is introduced by CodeXGLUE [18] and utilizes the CodeSearchNet [9] dataset. This contains code snippets for Python, Java, PHP, JavaScript, Ruby, and Go. Many models are actively competing for the highest performance on the CodeXGLUE leaderboard[1]. As of today[2], the highest scoring models on the Python CST consist of StarCoder-LoRA, DistillCodeT5, CoTexT [21], PLBART [1] and CodeBERT [5]. However, several models have not submitted entries to the leaderboard and have achieved competitive results; such as CodeT5 [26] and Redcoder [20].

[1]https://microsoft.github.io/CodeXGLUE/
[2]March 6th, 2024

A key aspect that makes the problem more difficult is the idea of incomplete information. Most code understanding and generation tasks do not provide a runnable/complete code snippet, only a subsection of code from the original script. Therefore, learning methods involving running the code are not typically possible.

However, alternative code representations typically do not require running/complete code. This paper aims to explore the impact of training CLMs with additional input, in the form of Abstract Syntax Trees (ASTs) and Data Flow Graphs (DFGs), in a computationally efficient and accessible manner.

For a supervised machine learning setting, Language Models (LM) are typically trained with a single input of one format, and tasked with generating an output of the same or a different format. However, for most supervised learning tasks, the original input can be represented in different input formats, which may or may not aid a machine learning model in solving the task. In the case of programming languages, code can be represented in different formats, such as ASTs and DFGs, which can be useful for CLMs to understand syntax and semantics for the code snippet.

Computational efficiency is an important consideration, to make sure the changes suggested in this work, can be easily implemented and accessible for others, without the need for extensive and prohibiting amounts of hardware and financial capital.

The following are the main contributions of this work:

1. Exploring the effect of incorporating an additional pre-training small dataset to facilitate learning different input formats for CLMs.
2. The comparison of fine-tuning CLMs on a combined input of code, AST, and DFG data versus providing these inputs individually in a Multi-task learning fashion.
3. An exploration of three novel pre-training tasks and four fine-tuning tasks, that are specific for code understanding.

The rest of this paper is divided as follows. The related work is discussed in section 2. Different forms of representing code are discussed in section 3. The main considerations for training LMs are in section 4. The experiments and results are shown in section 5. The limitations of this work and potential future work are discussed in section 6. The work is concluded in section 7.

2. Related Work

2.1. CLM tasks

Many different benchmarks target code understanding and generation [3][8][18]. HumanEval [3] is a Python-specific dataset, intended to evaluate the functional correctness of programs generated from NL descriptions. One advantage of the HumanEval dataset is that the solution for each example contains Python unit tests, in the form of assertions that must all pass for the solution to be considered functionally correct. The main disadvantage of providing runnable tests is the requirement that the code segment itself is runnable, which requires that the code is standalone and does not reference external packages or files.

APPS [8] is another dataset intended to measure the capabilities of CLMs in generating satisfactory Python code from an NL description. Similar to HumanEval, it contains test cases to assert the functional correctness of the generated code. The APPS dataset is unique in terms of the complexity of the solutions desired for the given NL descriptions. In the APPS dataset, they use Python problems, similar to those used in industry to assess potential candidates for software engineering positions [8]. This often involves understanding advanced concepts in Computer Science, and particularly a strong understanding of data structures and algorithms.

In this work, we utilize the CodeXGLUE [18] benchmark. There are 10 distinct tasks across 14 datasets used in the CodeXGLUE benchmark. The benchmark itself is intended to accelerate research in the field of applying artificial intelligence to increase software developers' productivity. The tasks include clone detection, defect detection, cloze test, code completion, code repair, code translation, NL code search, text-to-code generation, code summarization, and document translation. Note the text-to-code task is similar to the task found in the HumanEval and APPS datasets. However, the requirement that the code is runnable is not enforced in the CodeXGLUE benchmark. Note, that the text-to-code task is the inverse of the CST.

In this work, we focus on the CST and specifically focus on the Python programming language. The CST utilizes the CodeSearchNet [9] dataset. This dataset is originally compiled from publicly available open-source GitHub repositories and is filtered to remove examples that cannot be parsed into an AST, have less than 3 or greater than 256 output characters, contain URLs or are not in English[3]. An example of the CST is showcased in Figure 1. In this example,

```python
def face_distance(face_encodings,
    face_to_compare):
   if len(face_encodings) == 0:
      return np.empty((0))
   return np.linalg.norm(face_encodings -
       face_to_compare, axis=1)
```

(a) Input Python code snippet.

```
Given a list of face encodings, compare them
    to a known face encoding and get a
    euclidean distance for each comparison
    face. The distance tells you how similar
    the faces are.
```

(b) Ground truth output. Based on the function documentation.

Figure 1. An example for Python CST of CodeXGLUE [18].

we have the Python function as input, and its corresponding documentation as the expected output. Note that the length of the input and expected output varies between examples as illustrated in Figure 2. However, the vast majority of examples have code character lengths of less than or equal to 2000 characters and NL summaries of less than or equal to 200 characters. However, the corresponding number of tokens that this will result in depends on the tokenization method used by the specific model.

In the Python CST, there are 252k training examples, 13.9k validation examples, and 14.9k test examples.

2.2. Recent CLMs

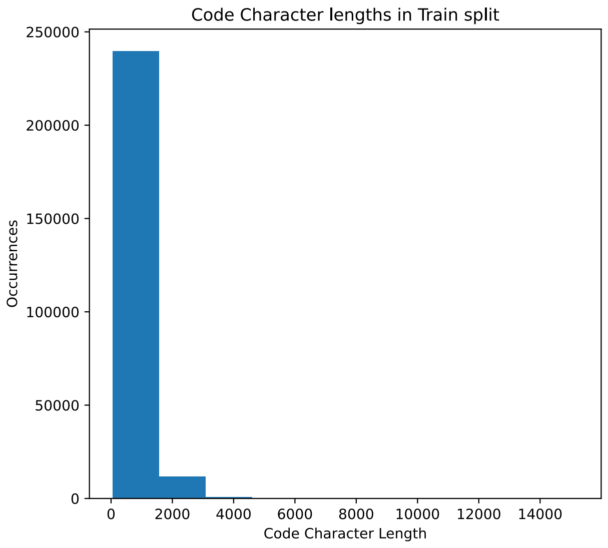
For CLM tasks, the field is entirely dominated by transformer-based models, that are trained on large corpora of code and natural language [1][5][11][14][18][20][21][26]. Several noteworthy models will be discussed in the following sections, based on high performance on the Python CST.

2.2.1 CodeXGLUE Leaderboard Top Models

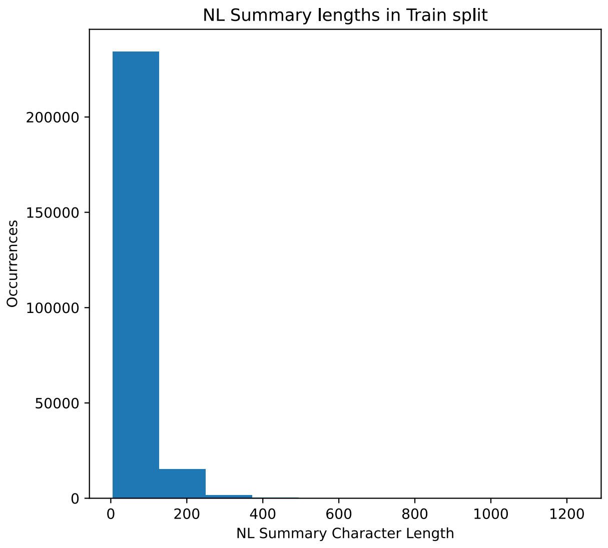
CodeBERT [5] claims to be the first large NL and programming language pre-trained model for multiple programming languages. CodeBERT is a transformer-based neural network, that was pre-trained using the Masked Language Modeling (MLM) objective and the Replaced Token Detection (RTD) objective. CodeBERT has a total of 125M parameters and achieved a smoothed BLEU-4 score of 19.06 on the Python CST.

PLBART [1] is a transformer-based neural network, pre-trained on a large collection of Java and Python functions and their corresponding NL text. PLBART utilizes a Denoising Autoencoding (DAE) pre-training objective. Similar to CodeBERT, PLBART is pre-trained on multiple pro-

[3]https://huggingface.co/datasets/code_x_glue_ct_code_to_text



(a) Code character lengths, obtained from the "Code_tokens" column.



(b) Natural Language lengths, obtained from the "docstring_tokens" column.

Figure 2. Character lengths of examples in the train-split of the Python CST.

gramming languages and specifically obtains its data from GitHub and StackOverflow. Due to differences in the amount of NL and programming language data, PLBART utilizes up/downsampling to alleviate the bias towards programming languages. PLBART uses 6 layers for the encoder and decoder each, with a model dimension of 768 and 12 heads. This results in approximately 140M parameters and achieves a smoothed BLEU-4 score of 19.30 on the Python CST.

CoTexT [21] is a transformer-based neural network, pretrained on code and NL text. CoTexT uses the same structure of T5 [22] and is initialized on the pre-trained checkpoints released by T5. They conduct additional pre-training using CodeSearchNet and GitHub data with Multi-Task learning objectives. Whereby, they specify the specific task using task-specific prefixes added to the input sequences. For example, for the CST, they would train on examples using different programming languages with a specific prefix indicating which language the given example is in. CoTexT has a size of 220M parameters and achieved a smoothed BLEU-4 score of 19.70 on the Python CST.

In second place on the Microsoft CodeXGLUE leaderboard, there is a model listed as *DistillCodeT5*. Although this model does not have a corresponding published paper, we can assume it is based on CodeT5 [26]. DistillCodeT5 is reported to achieve a smoothed BLEU-4 score of 20.59, which is higher than the self-reported score for CodeT5. Although CodeT5 is not listed on the leaderboard its best variant is reported to achieve a score of 20.36. In this work, we primarily use CodeT5 as our baseline. CodeT5-base contains approximately 220M parameters and is an encoder-decoder style transformer, trained for code understanding and generation tasks. CodeT5 uses prefixes to inform the model which task it is currently being trained for and is trained on several unique pre-training tasks. The specific pre-training tasks are discussed further in section 4.1. We conducted our experiments using CodeT5 because of the competitive performance it has on many CodeXGLUE tasks, the relatively small model size, the simplicity in fine-tuning following their training scripts, and the availability of pre-training checkpoints[4].

Currently[5] StarCoder-LoRA is ranked first on the Microsoft CodeXGLUE leaderboard. Although there is no published paper available for this specific variant, we can assume that this model is based on StarCoder [16]. StarCoder is considered a Large Language Model and has approximately 15.5B parameters. We do not conduct experiments using StarCoder due to the computational costs associated with training such a large model, and because of issues related to data leakage between the pre-training data and the CST. This issue indicates that there could be data used to train the model that is also used later to evaluate the model on the CST. As a result, the performance on the CST can be optimistically biased. This issue is further discussed in section 4.1.

[4]https://github.com/salesforce/CodeT5
[5]March 6, 2024

2.2.2 Utilizing Alternative Data Formats

Several existing works have utilized AST information to assist the CLMs on code understanding and generation tasks [6][7][8][10][12][13][24][25]. This includes altering the attention mechanism to take advantage of the AST structure [14][15], and encoding AST paths with RNNs [2]. Other works have proposed to utilize DFGs for their efficiency in conveying semantic structure without the complexity and large size of an AST [7][25].

Existing work, such as [12], uses reinforcement learning to dynamically select a branch of the AST to utilize. Alternatively, some works have attempted to use both a preorder and breadth-first traversal of the AST [27]. In this work, we flatten the AST, primarily by using preorder traversal. This is discussed further in section 3.

3. Input Data Formats

3.1. AST

ASTs can be used to represent code in a tree structure using specific keywords, defined by a formal language[6]. In this work, we propose using flattened ASTs and passing them as input to a CLM to assist in the training of the LM. Additionally, we prioritized using non-evasive changes to the training process to simplify the training process and allow alternative models to be used interchangeably.

Each node in an AST contains a corresponding type for the token in the input code that it represents, and additionally, an optional value. Given the Python code example in Figure 1, the corresponding AST is shown in Figure 3. The first line in the original code defines a function called "face_distance" that has two parameters. This is shown in the AST with the lines "function_definition" and "parameters". A similar relationship appears with all the other lines in the original code and the AST.

When passing the flattened AST into a standard CLM, the issue of sequence length becomes apparent. We utilize CodeT5 [26], which has a maximum sequence length of 512 input tokens. Therefore, we must consider alternative methods to flatten ASTs, that reduce the overall input size.

To reduce the overall input size, we traverse the AST using preorder traversal, and remove the parentheses. Whereby we visit each node, before visiting the child nodes. However, this no longer makes each corresponding AST unique. An example of preorder traversal is showcased in Figure 4.

In contrast, we may wish to utilize not only the node type but the node value as well. This will consequentially increase the sequence length but can capture the entirety of the original code and maintain the syntax and semantic

```
(module (
 function_definition name: (
  identifier
 ) parameters: (
  parameters (
   identifier
  ) (
   identifier
  )
 ) body: (
   block (
    if_statement condition: (
     comparison_operator (
      call function: (
       identifier
      ) arguments: (
       argument_list (
        identifier
       )
      )
      (
       integer
      )
     )
     consequence: (
      block (
       return_statement (
        call function: (
         attribute object: (
          identifier
         ) attribute: (
          identifier
         )
        ) arguments:(
         argument_list (
          parenthesized_expression (
           integer
          )
         )
        )
       )
      )
     ) (
      return_statement (
       call function: (
        attribute object: (
         attribute object: (
          identifier
         ) attribute: (
          identifier
         )
```

Figure 3. Flattened AST of the code example in Figure 1, with indentation for readability. The remaining 25 lines have been truncated to reduce the figure size.

[6]https://en.wikipedia.org/wiki/Abstract_syntax_tree

```
module function_definition def identifier
    parameters ( identifier , identifier ) :
    block if_statement if comparison_operator
    call identifier argument_list (
    identifier ) == integer : block
    return_statement return call attribute
    identifier . identifier argument_list (
    parenthesized_expression ( integer ) )
    return_statement return call attribute
    attribute identifier . identifier .
    identifier argument_list (
    binary_operator identifier - identifier ,
    keyword_argument identifier = integer )
```

(a) Preorder AST without node value.

```
module function_definition def identifier
    <face_distance> parameters ( identifier
    <face_encodings> , identifier
    <face_to_compare> ) : block if_statement
    if comparison_operator call identifier
    <len> argument_list ( identifier
    <face_encodings> ) == integer <0> : block
    return_statement return call attribute
    identifier <np> . identifier <empty>
    argument_list ( parenthesized_expression
    ( integer <0> ) ) return_statement return
    call attribute attribute identifier <np>
    . identifier <linalg> . identifier <norm>
    argument_list ( binary_operator
    identifier <face_encodings> - identifier
    <face_to_compare> , keyword_argument
    identifier <axis> = integer <1> )
```

(b) Preorder traversal of AST with leaf-node values. The value is surrounded by angle brackets.

Figure 4. Preorder traversal of AST for the code snippet in Figure 1.

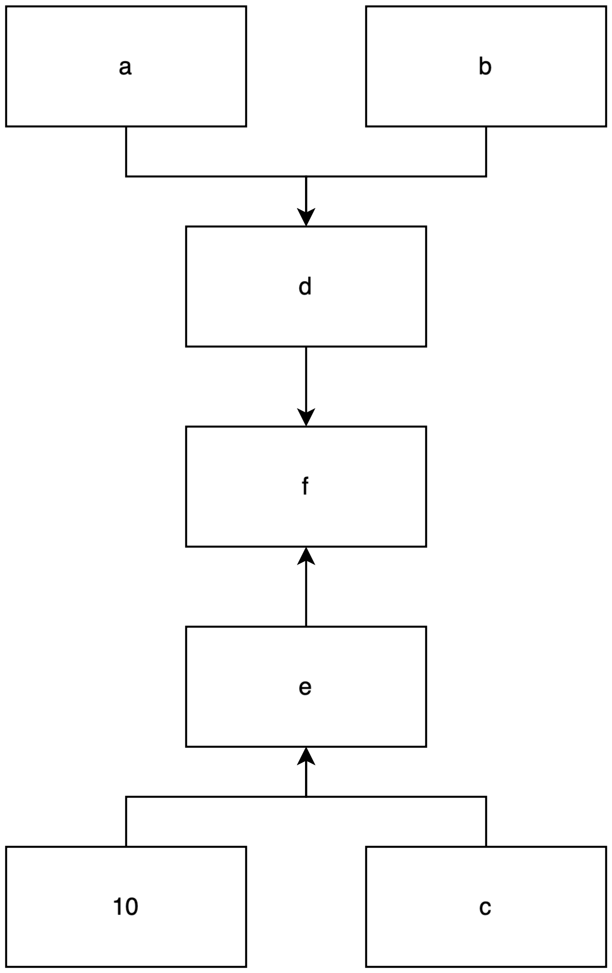
structure provided from the AST. This is showcased in the second part of Figure 4. Note, we only keep the value for leaf nodes, as the parent nodes' value contains the values of all the children nodes, and thus would introduce a lot of redundancy and increase the size substantially. In addition, we remove the value of leaf nodes whose type is equivalent. For example, a leaf node of type ":" will have a value of ":", and thus is unnecessary to include both.

3.2. DFG

DFGs are a minimal way of representing relationships within a code snippet. Figure 5 showcases an example of a Python code snippet and its corresponding DFG. The DFG encodes the information about where a variable's value comes from. For the example in the figure, the corresponding flattened DFG can be written as *(e, (10, c)), (d, (a, b)), (f, (e, d))*. This consists of tuples, wherein for each tuple, the first value corresponds to an identifier in the Python code, and the second value is a list of the identifiers that are used to directly produce the first value.

```python
def foo(a, b, c):
    d = a + b
    e = 10*c
    f = e + d
    return f
```

(a) Python Code



(b) DFG

Figure 5. Python code and corresponding DFG example.

4. Training Methods

4.1. Pre-training

Pre-training has two main components. The first component is training on a large amount of data. Typically, CLMs are pre-trained on corpora that have NL and code pairs. Open source publicly available GitHub repositories are a good source for this data. CodeSearchNet [9] is a large dataset used extensively in this work, that contains multiple programming languages and NL pairs. Pre-training is typically the most computationally and financially expensive step in training LMs, due to the size of the data used. To reduce the cost, but still maintain the advantage of pre-training, we utilize publicly released pre-trained checkpoints and conduct additional pre-training but restrict it to a small pre-training dataset (Python only). The motivation behind this is to allow the model to take advantage of pre-training objectives and understand the relationship between code, ASTs, and DFGs, to utilize this information when provided on downstream tasks.

An important consideration is ensuring the separation of the pre-training data and the fine-tuning test data. It appears that some models do not enforce this separation, and therefore, their results in the corresponding downstream tasks are optimistically biased. An example of this includes StarCoder [16]. Whereby, we found the repositories used in the pre-training, are also used in the test set for the CST. Therefore it is not clear whether the docstring, which is used for the ground-truth output, has been seen by the model before fine-tuning on the CST. The authors also mention that "there may be an overlap between this evaluation dataset and the data used to train SantaCoder and the StarCoder models" [16].

The second key component of pre-training is the pre-training objective. This is typically different from the downstream fine-tuning objectives. Many CLMs utilize existing pre-training objectives found in sequence-to-sequence Natural Language Processing (NLP) tasks. For example, CodeBERT [5], which is based on BERT [4], utilizes the same pre-training objective of Masked Language Modeling (MLM) and Replace Token Detection (RTD) as its parent model. The main distinction is that the pre-training was conducted on code and NL pairs of the CodeSearchNet [9] dataset. Other popular pre-training objectives include some variation of Denoising Autoencoding (DAE). The main idea of DAE is to corrupt the input tokens and predict the uncorrupted tokens. CodeT5 [26] is pre-trained on a series of different pre-training tasks. This includes a variation of DAE called Masked Span Prediction (MSP), whereby spans of tokens are masked at random. In addition, CodeT5 uses a Masked Identifier Prediction (MIP) pre-training task, which is an obfuscation technique, where they replace all tokens in the code that are identifiers (as determined from the AST) with a consistent unique token for that specific identifier. Again the objective would be to predict what the original masked tokens were. In this work, we do not replicate the pre-training objectives used in CodeT5, due to the unavailability of the pre-training code.

Alternatively, StructCoder [25] utilizes a structure-based DAE, that corrupts random spams of tokens, replacing them with a mask token, a random token, or altogether deleting the spam. The span lengths are sampled from a Poisson distribution with a mean of 12 tokens. 35% of the code tokens are corrupted. Replicating this work, we use the corruption implementation[7] of StructCoder for the subsequent pre-training tasks.

In this work, we propose three distinct pre-training objectives. The motivation to conduct the additional pre-training is to encourage the model to understand the relationship between code, AST, and DFG sequences representing the same underlying code. Thereby, when given this additional information, during fine-tuning, the model has a more well-developed understanding of the input and performs better on the downstream task. The tasks are explained in Section 4.1.1, 4.1.2, 4.1.3, and 4.1.4.

4.1.1 Code + DFG + AST DAE

The input is a corrupted single sequence containing the original code appended with a unique separation token, followed by the DFG flattened sequence, another separation token, and then type-only flattened preorder traversed AST. The ground-truth output would be the uncorrupted original sequence. An example of this pre-training task is showcased in Figure 6. Note the repeated <mask> and <pad> tokens are truncated for readability.

The main advantage of this approach is the extensive and abundant amount of information given as input. In addition, the relationships learned by the model should exist between each distinct pair of data formats. The main disadvantage of this approach would be the resulting large input and output lengths. As mentioned earlier, CodeT5 [26] is unable to support sequence lengths larger than 512 tokens. Therefore, when concatenating all the information in this manner, some of the examples will be truncated. With this approach, the average tokenized source and target lengths are both 452, with a maximum source and target length of 4758.

4.1.2 Full AST $\rightarrow$ Code

The input to Full AST $\rightarrow$ Code pre-training consists of the flattened pre-order traversed AST with leaf node values. The expected output would be the code sequence that was utilized to generate the AST. An example of this is showcased in Figure 7. The motivation of this approach is to allow the model to learn full ASTs and find the relationship to the original code sequence. This allows the model to learn an identity relationship between full AST inputs and code inputs it was originally trained on. We hypothesize this will allow the model to have a lower limit performance, on the

[7]https://github.com/reddy-lab-code-research/StructCoder

```
'<s>def refresh_schema_metadata ( self,
    max_schema_agreement_wait = None ) ( (
    keyword_ offset,<mask>...<mask> )
    integer<mask>...<mask>)_du : force = True
    ) : raise DriverException ( "Schema
    metadata was not refreshed. See log for
    details." )
<DFG> (max_schema_agreement_wait, (None,))
<AST> module function_definition def
    identifier parameters ( identifier,
    default_parameter identifier =
    none_statement if not_operator not call
    attribute attribute identifier.
    identifier. identifier
    argument<mask>...<mask> = true ) : block
    raise_statement raise call identifier
    argument_list ( string " " )
    </s><pad><pad>...'
```

(a) Corrupted input sequence with Code + DFG + AST.

```
'<s>def refresh_schema_metadata ( self,
    max_schema_agreement_wait = None ) : if
    not self. control_connection.
    refresh_schema ( schema_agreement_wait =
    max_schema_agreement_wait, force = True )
    : raise DriverException ( "Schema
    metadata was not refreshed. See log for
    details." )
<DFG> (max_schema_agreement_wait, (None,))
<AST> module function_definition def
    identifier parameters ( identifier,
    default_parameter identifier = none ) :
    block expression_statement string " "
    if_statement if not_operator not call
    attribute attribute identifier.
    identifier. identifier argument_list (
    keyword_argument identifier = identifier,
    keyword_argument identifier = true ) :
    block raise_statement raise call
    identifier argument_list ( string " " )
    </s><pad><pad><pad>...'
```

(b) Ground-truth target sequence. Includes Code + DFG + AST sequences.

Figure 6. An example of the input and expected output for *Code + DFG + AST DAE* pre-training task.

downstream tasks, equivalent to the CodeT5-base [26], but potentially have a higher performance from taking advantage of the syntax and semantic information provided by the AST.

Alternatively, we could consider pre-training on the dual version of this task, which would include passing code as input and having an expected output of the full AST. The implications of this task are less intuitive. The motivation

```
'<s>module function_definition def identifier
    parameters ( identifier,
    default_parameter identifier = none ) :
    block expression_statement string " "
    if_statement if not_operator not call
    attribute attribute identifier.
    identifier. identifier argument_list (
    keyword_argument identifier = identifier,
    keyword_argument identifier = true ) :
    block raise_statement raise call
    identifier argument_list ( string " " )
    </s><pad><pad><pad>...'
```

(a) Full AST input sequence.

```
'<s>def refresh_schema_metadata ( self,
    max_schema_agreement_wait = None ) : if
    not self. control_connection.
    refresh_schema ( schema_agreement_wait =
    max_schema_agreement_wait, force = True )
    : raise DriverException ( "Schema
    metadata was not refreshed. See log for
    details." )</s><pad><pad><pad>...'
```

(b) Code expected output sequence.

Figure 7. An example of the input and expected output for *Full AST DAE* pre-training task.

may be, that the Full AST $\rightarrow$ Code task is too easy, and the dual would be more challenging, thus allowing the model to learn a more complex understanding of the code.

A disadvantage of both versions of this task is the lack of use of the DFG information. However, the main advantage of these tasks is the reduced input sequence lengths when compared to using multiple different input formats. The average input and output lengths are 213 and 142 tokens respectively. Likewise, the maximum input and output lengths are 1573 and 2016 tokens respectively. Interestingly, the maximum output length is greater than the maximum input length.

4.1.3 Code + DFG DAE

Alternatively, we can consider utilizing just the Code and DFG without the AST. The DFG alone contains insufficient information to do any code understanding tasks. Therefore, we propose a DAE pre-training task that includes a corrupted input sequence which would consist of the original code snippet followed by a specific separator token, and then the flattened DFG sequence. The expected output would be the uncorrupted code and DFG sequence. The input would be similar to the example in Figure 6, just without the AST section. The motivation of utilizing the DFG, is to encourage the model to learn relationships between variables in the original code sequence. The average input and output lengths are 237 tokens. The maximum input and output lengths are 3959 tokens.

4.1.4 Alternative Pre-training tasks

We propose a few additional pre-training tasks, however, we only conduct experiments on the above mentioned pre-training tasks.

We can consider conducting DAE with a full AST. This would include a corrupted flattened pre-order traversed AST with leaf node values as input and the expected output would be the uncorrupted version. The motivation of this approach is that the original code sequence may be unnecessary, and the full AST is sufficient. If so, we could simply use the full AST as input to the downstream tasks, and further reduce the input sequence lengths. The average input and output lengths are 214 tokens. The maximum input and output lengths are 1574 tokens. One issue with this approach could be that the connection between the original code and the full AST may not be established if we are just using the full AST as the input and output.

Furthermore, if we wish to utilize both *Code + DFG DAE* and *Full AST DAE*, we could pre-train on both tasks alternatively. For example, we can randomly select between the two methods, and pass either the corrupted Code + DFG input or the corrupted full AST input to the model, and the required output would be the corresponding uncorrupted input. This shares similarities to the Multi-task learning technique that we will discuss later in Section 4.2.5.

4.2. Fine-tuning

For the fine-tuning task, the model will be given a code snippet and asked to predict the code summary in English, as illustrated in Figure 1. Our baseline model is CodeT5, which is an encoder-decoder-style transformer. Generating a prediction for a given task will be conducted by auto-regressive decoding. This involves generating an output token one at a time, using the input and the prior generating tokens. The loss will be the Cross-Entropy Loss between the generated sequence and the expected output.

4.2.1 Single input

For single-input fine-tuning, we pass one batch to the CLM, and calculate a loss over the batch, by comparing it with the expected output, which is the corresponding batch of NL summaries.

4.2.2 Code + DFG + AST

One approach for single input fine-tuning can consist of including each format of the code, separated by unique separating tokens. Note this is similar to the *Code + DFG + AST DAE* pre-training objective. However, the input will not be corrupted, and the output is the NL summary.

4.2.3 Full AST

An alternative approach for single-input fine-tuning could be to just use the full AST as input, and the NL summary as the output. Similar to the motivation for just using the full AST for pre-training, the full AST for fine-tuning could be sufficient for the task.

4.2.4 Code + DFG

Extending the Code + DFG DAE pre-training task, we propose fine-tuning on the CST with Code + DFG input and the NL summary as the output.

4.2.5 Multi-task Learning

Multi-task learning (MTL) is a method, in which a model learns from multiple related tasks simultaneously, in hopes of ultimately performing better than if trained on the individual tasks separately. Intuitively, if the tasks are related, understanding one task may help in solving the others. In this work, we focus on Hard Parameter Sharing MTL [23]. A single model is optimized on multiple different inputs and the loss is computed by a weighted sum of the individual losses for each input variation. The weight of each loss is a hyper-parameter.

We consider one form of MTL by training with three different input variations and each has the same expected output, which is the NL summary.

The loss can be calculated below:

$$L = \lambda_0 * L_{Code} + \lambda_1 * L_{AST} + \lambda_2 * L_{DFG} \quad (1)$$

Where L is the total loss, $\lambda_0, \lambda_1, \lambda_2$ are the loss weights, L_{Code} is the loss of the code input alone, L_{AST} is the loss of the full AST input, and L_{DFG} is the loss of the DFG input.

The main advantage of this approach is that we can enforce a prior on how important each task is, relative to the other. For example, if we use a λ_0 of 0.8, a λ_1 and λ_2 of 0.1, this encourages the model to optimize primarily for the code input and then secondarily the DFG and AST inputs.

The main disadvantage of this approach is the training time. By training on three inputs, for every example, the forward computation is conducted three times. In addition, the GPU memory requirements are increased with three inputs, and hence we elected to decrease the batch size. As a result, the total training time is increased.

4.3. Evaluation Metrics

The main metrics used for evaluation consist of perplexity (PPL), exact match (EM), and smoothed BLEU-4 [17].

PPL is defined as:

$$PPL = e^L \tag{2}$$

Where L is the loss calculated for the task, and e is Euler's number.

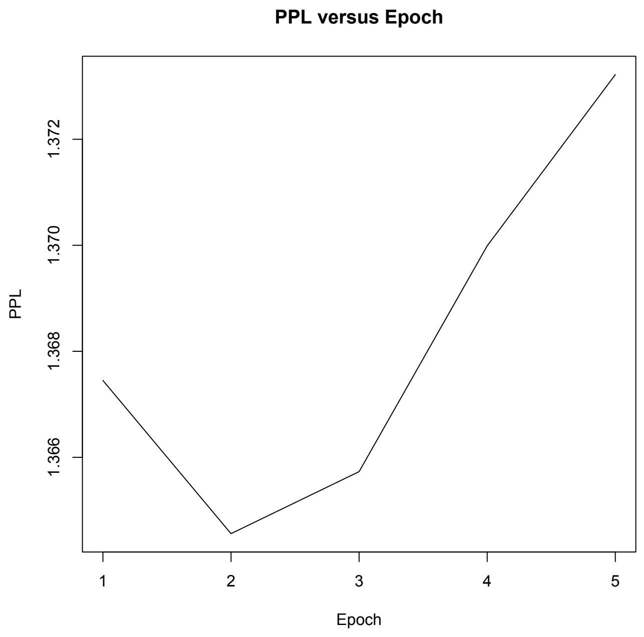
BLEU [19] is a metric commonly used in machine translation and is intended to measure how similar a predicted translation is with one or more ground-truth references. The 4, in BLEU-4, implies that the maximum n-gram length is 4. The smoothing technique used for smoothed BLEU-4 [17] involves adding one to the n-gram hit and the total count for n greater than 1. The purpose of this is to allow predictions that are less than n words to still have a possibility of getting a positive smoothed BLEU score if there is at least one word that matches.

EM is defined as the ratio of predictions that exactly match the ground-truth outputs.

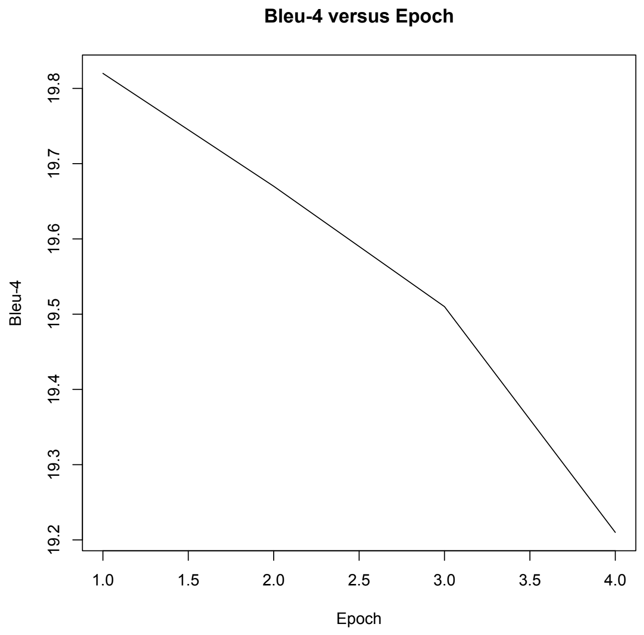
5. Experiments and Results

5.1. Baseline

We defined our baseline as CodeT5-base fine-tuned using the Code Summarization fine-tuning script from CodeT5's GitHub repository[8]. Using the provided script to fine-tune the pre-trained CodeT5-base, we achieved a test smoothed BLEU-4 score of 20.15 with an EM of 1.70. Note that this is slightly higher than the smoothed BLEU-4 score reported in the original paper [26]. This is likely due to randomness introduced in the training procedure, and not a result of differences in hyper-parameters, as the default hyper-parameters were used. The validation performance is showcased in Figure 8. Notice how the model quickly overfits on the training data after 1-2 epochs, as both the PPL and BLEU-4 score worsen as the epochs increase. The training time for fine-tuning CodeT5-base is the smallest at 7h23m. This is because the maximum input and output sequence lengths are restricted to 256 and 128 tokens by default. Smaller input sequences allow for larger batch sizes under the same memory constraints, which in turn decreases total training time. The following experiments explore using different input formats and as a result require larger maximum input and output sequences, which we set to 512 tokens each. Hence we elect to decrease the batch sizes, which in turn increased training time.



(a) Validation PPL versus epochs.



(b) Validation BLEU-4 versus epochs.

Figure 8. Validation performance of CodeT5-base baseline.

[8]https://github.com/salesforce/CodeT5/tree/main/CodeT5

Model	BLEU-4	Exact Match	Additional Pre-training Time	Fine-tuning time	Published
StarCoder-LoRA	**23.13**	–	–	–	False
DistillCodeT5	20.59	–	–	–	False
CoText	19.73	–	–	–	True
CodeT5-small	20.04	–	–	–	True
+ dual gen	19.94	–	–	–	True
+ multi-task	20.10	–	–	–	True
CodeT5-base	20.01	–	–	–	True
+ dual gen	20.11	–	–	–	True
+ multi-task	**20.36**	–	–	–	True
CodeT5-base	**20.15**	**1.70**	–	**7h23m**	–
Code + DFG + AST DAE	19.66	1.62	44h16m	25h58m	–
Code + DFG + AST	19.58	1.61	–	25h47m	–
Full AST DAE	18.08	0.36	41h47m	26h35m	–
Code + DFG DAE	19.70	1.62	42h0m	26h4m	–
MTL (0.8, 0.1, 0.1)	19.58	1.62	44h16m	42h15m	–

Table 1. Test performance for Python CST. The first section consists of the highest scoring models on the CodeXGLUE leaderboard. The second section is CodeT5 [26] self-reported results. The third section, is the results obtained in this work through experiments. Note, BLEU-4 is the smoothed BLEU-4 defined by [17].

5.2. Additional Pre-training

5.2.1 Code + DFG + AST DAE

The pre-training was conducted using the training split from CodeXGLUE's CST. The input format was explained prior in Section 4.1.1. The hardware utilized is discussed in Appendix A. The training was conducted using batch sizes of 16, for 5 epochs, using early stopping with the patience of 1 if the perplexity and the BLEU-4 score did not improve. The pre-training time required for 5 epochs is 44h16m.

The validation perplexity and BLEU-4 score after each epoch of training generally increased as showcased in Figure 9. The improved perplexity and BLEU-4 score indicate that the model was gradually able to improve over time and learn to better denoise the original corrupted input, implying a better understanding between the code, AST, and DFG tokens. However, the validation BLEU-4 score did slightly decrease at 5 epochs, indicating overfitting.

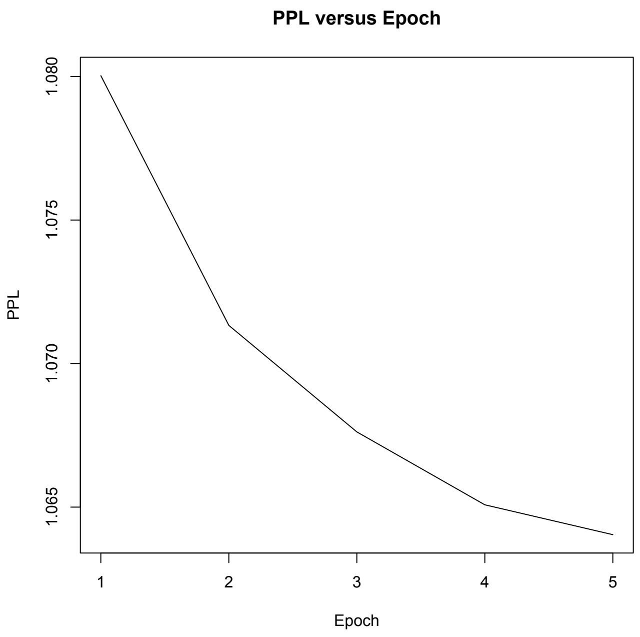
Using the highest BLEU-4 score from the validation set to determine the best model, a final evaluation was done on the test set. An example of the corrupted input, the original uncorrupted expected output, and the final model's prediction are showcased in Figure 16. This example illustrates that the model is able to effectively denoise the corrupted input, even when the input formats include AST and DFG sequences.

5.2.2 Full AST $\rightarrow$ Code

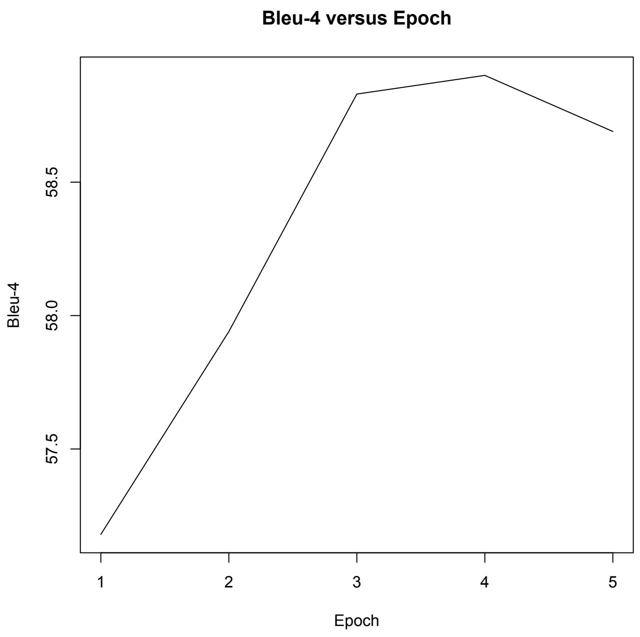
Similarly, for Full AST pre-training we utilized a batch size of 16, for 5 epochs, and early stopping with the patience of 1. The validation performance is showcased in Figure 10. The overall PPL is much higher than for the Code + DFG + AST pre-training, implying that the task is relatively harder. However, the PPL did decrease as the number of epochs increased. In addition, the BLEU-4 score generally did increase as epochs increased, implying the model did learn. An example of this pre-training task is showcased in Figure 17. As showcased in the example, the output generated is very similar to the expected ground-truth. The only difference is the "durl" and "url".

5.2.3 Code + DFG DAE

Likewise, for Code + DFG DAE pre-training, we utilized a batch size of 16 for 5 epochs and early stopping with the patience of 1. The validation performance is showcased in Figure 11. The overall PPL is similar to the Code + DFG + AST pre-training. Again, the PPL decreases as the number of epochs increases. In addition, the BLEU-4 score did increase as epochs increased, however, the BLEU-4 score seems to peak after 3 epochs of training, implying overfitting is occurring after this. An example of this pre-training task is showcased in Figure 18. Similar to the Code + DFG + AST DAE, we can see the model is able to denoise the corrupted input. In fact, in this example the ground truth and output match exactly.

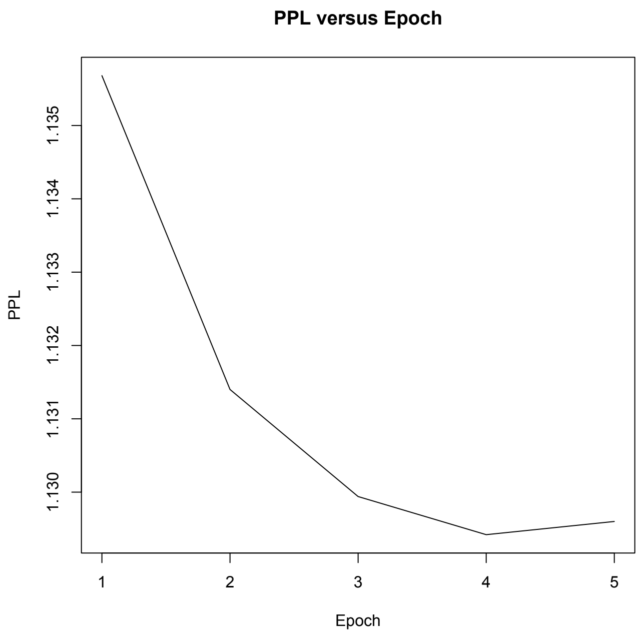


(a) Validation perplexity (PPL) versus epochs.

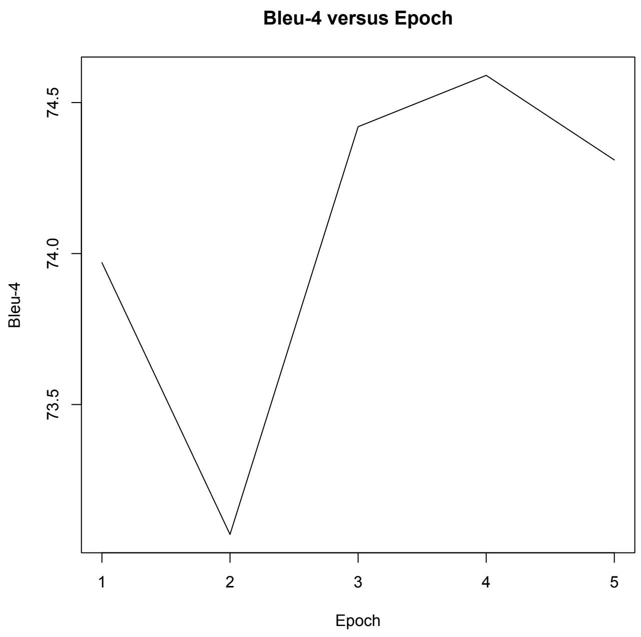


(b) Validation BLEU-4 versus epochs.

Figure 9. Validation performance of pre-training using Code + DFG + AST DAE.

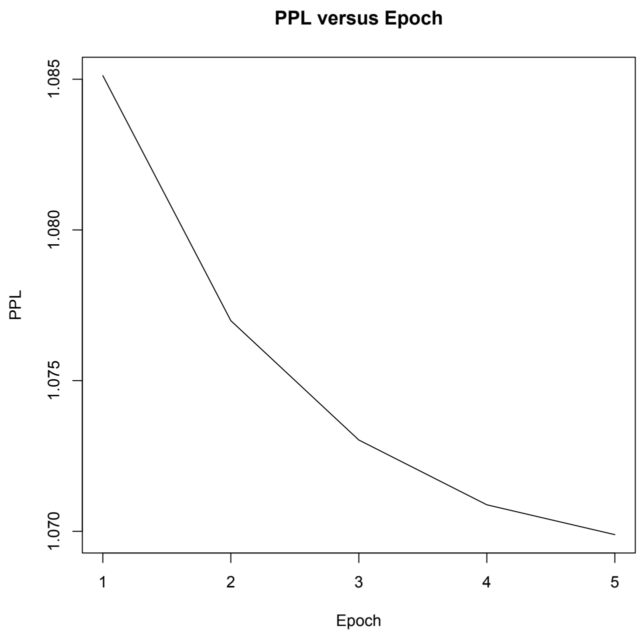


(a) Validation PPL versus epochs.

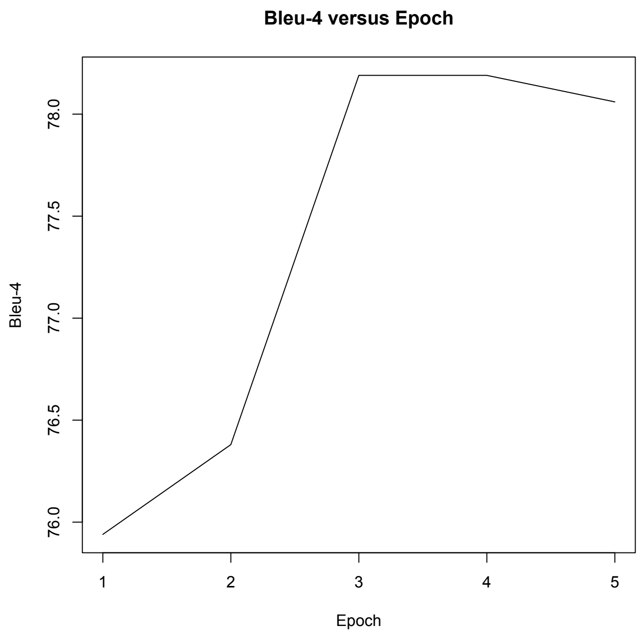


(b) Validation BLEU-4 versus epochs.

Figure 10. Validation performance on Full AST $\rightarrow$ Code pre-training task.



(a) Validation PPL versus epochs.



(b) Validation BLEU-4 versus epochs.

Figure 11. Validation performance on Code+DFG pre-training task.

5.3. Single-input

5.3.1 Code + DFG + AST

Fine-tuning was conducted using a single combined input that contained the original input code, AST, and DFG information, as explained earlier in Section 4.2.2. Fine-tuning was done with similar specifications as the pre-training, with a batch size of 16, and input and output maximum sequence lengths of 512 tokens, for 5 epochs. The fine-tuning time was 25h58m, with a final smoothed BLEU-4 score of 19.66 and an exact match score of 1.62. This implies that the model predicted 1.62% of the test set examples exactly correctly, where each token matches the expected output in value and order. The validation performance is showcased in Figure 12. Note that overfitting occurs after 2 epochs of training. In the case of fine-tuning, the exact match score after each epoch is non-zero and averages at 1.47.

The prior results were obtained by fine-tuning using the best validation checkpoint from the Code + DFG + AST DAE task. We further explore whether additional pre-training is necessary, by fine-tuning with Code + DFG + AST inputs with no additional pre-trained checkpoint. The resulting model obtained a smoothed BLEU-4 score of 19.58 and an EM of 1.61, which are both slightly worse than the additional pre-trained variation. **Showcasing that additional pre-training is useful when providing different input formats not seen before**. However, the total training time is much longer when additionally pre-training.

5.3.2 Full AST

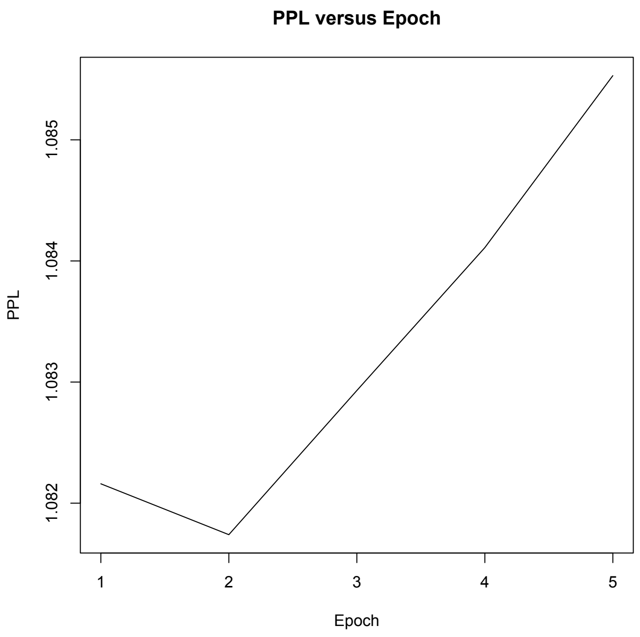
Following the same training procedure mentioned prior, this time using full ASTs as input and using the best validation checkpoint from the Full AST $\rightarrow$ Code pre-training task. The resulting training curve is showcased in Figure 13. After 2 epochs the PPL increases, and after 1 epoch the smoothed BLEU-4 score decreases. This implies overfitting is occurring after the first epoch.

5.3.3 Code + DFG

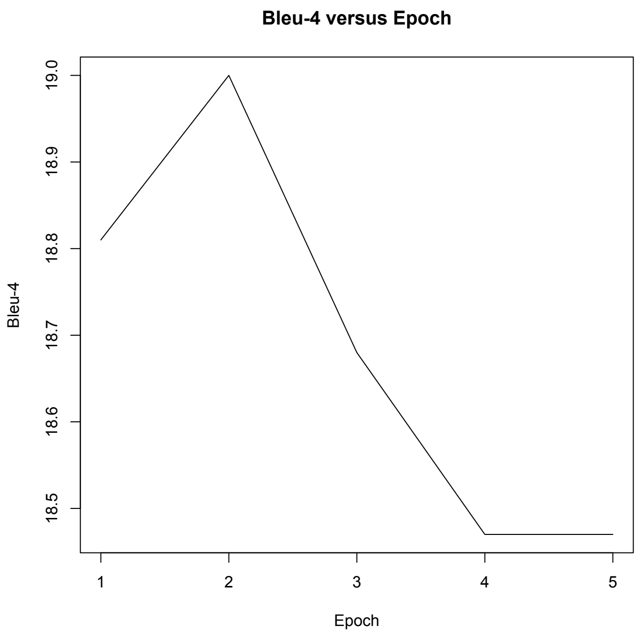
Similarly, we fine-tuned using Code + DFG input using the Code + DFG DAE best validation checkpoint, and the resulting performance curves are showcased in Figure 14. For this model, overfitting occurred after 2 epochs for both the PPL and smoothed BLEU-4 score.

5.4. Multi-Task Learning

For MTL we utilize the best validation checkpoint from the Code + DFG + AST DAE task. The resulting training curves are showcased in Figure 15. The fine-tuning time is 42h15m. An example of inputs used for MTL is shown in Figure 19.

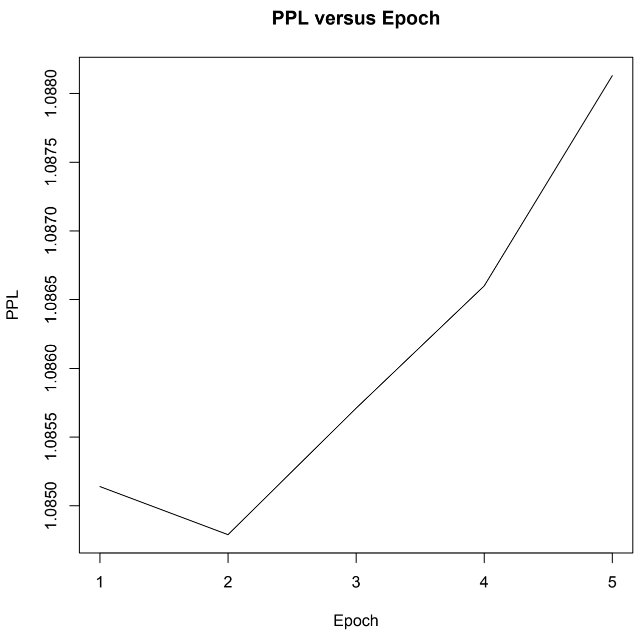


(a) Validation perplexity (PPL) versus epochs.

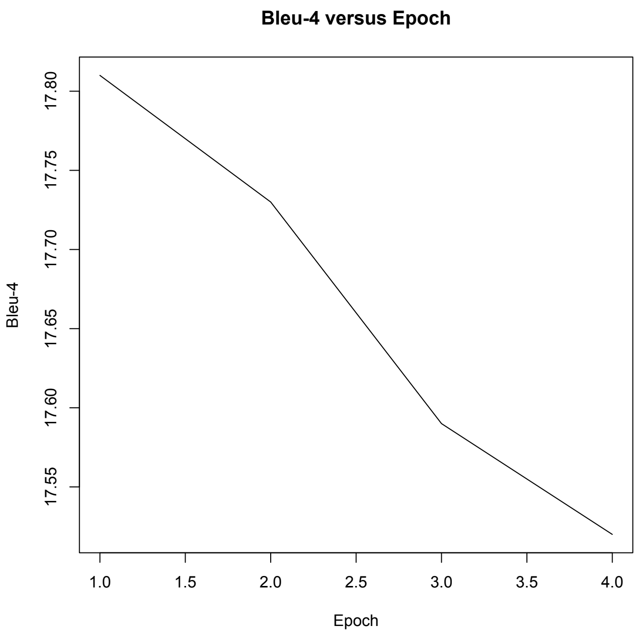


(b) Validation BLEU-4 versus epochs.

Figure 12. Validation performance of fine-tuning using single combined Code + DFG + AST input on CST.

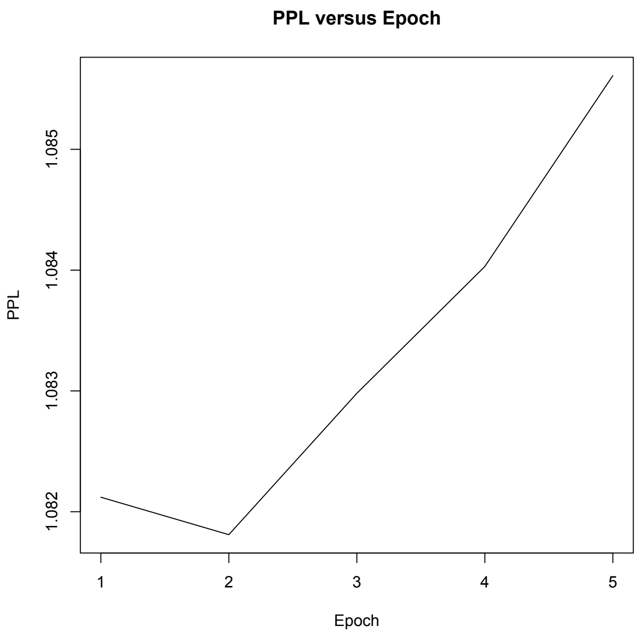


(a) Validation perplexity (PPL) versus epochs.

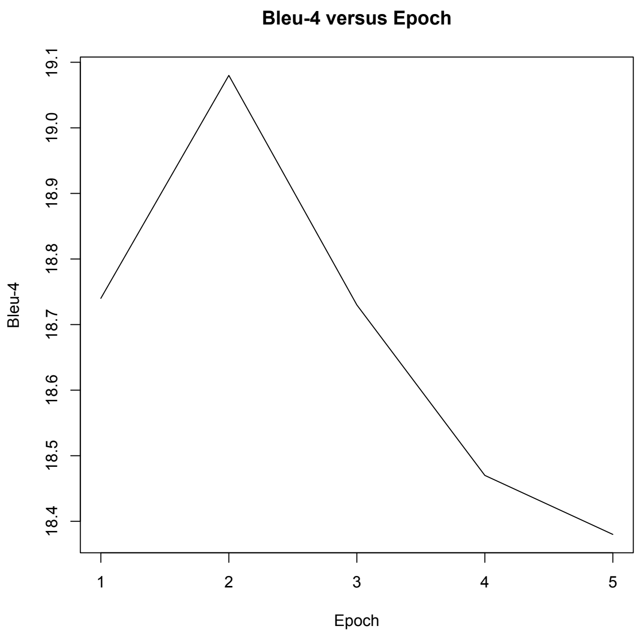


(b) Validation BLEU-4 versus epochs.

Figure 13. Validation performance of fine-tuning using full AST input on CST.

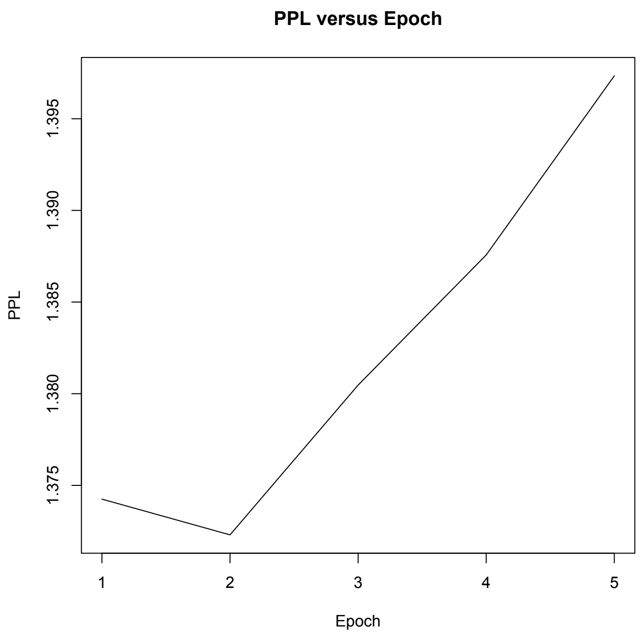


(a) Validation perplexity (PPL) versus epochs.

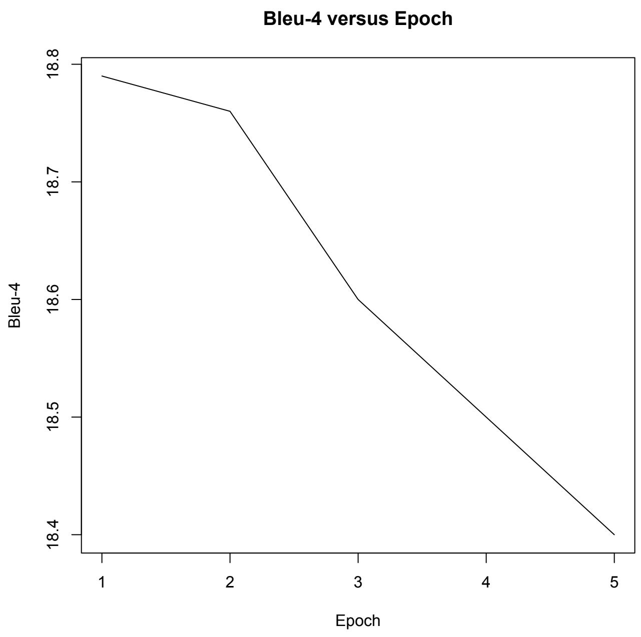


(b) Validation BLEU-4 versus epochs.

Figure 14. Validation performance of fine-tuning using Code + DFG input on CST.



(a) Validation perplexity (PPL) versus epochs.



(b) Validation BLEU-4 versus epochs.

Figure 15. Validation performance of fine-tuning using MTL on CST.

5.5. Overall results

In Table 1 there are three distinct sections. The first section contains models listed on the CodeXGLUE leaderboard[9]. However, as noted by the "Published" column, several of these examples do not contain a corresponding published paper to explain or verify the results. In addition, as mentioned in section 4.1, StarCoder [16] is trained on the same dataset used for evaluating on the CST, and therefore should not be included in the leaderboard. DistillCodeT5 does not contain a corresponding paper, but we can assume it is based on CodeT5 [26]. Comparing models that have a published paper, and are found on the CodeXGLUE leaderboard limits us to CoTexT [21] and PLBART [1]. CoTexT and PLBART have a smoothed BLEU-4 score of 19.73 and 19.30 respectively.

The second section of the table is limited to officially reported results of CodeT5 and its variants. This includes two model sizes of small and base, which are 60M to 220M parameters respectively. Each of these models was additionally trained using dual generation and MTL.

The final section showcases results obtained by experiments of this work. Note the CodeT5-base, which is the baseline model we utilize, had a reported result of 20.01, and we obtained 20.15. As mentioned prior, we believe this to be a result of randomness in the training process and not a result of differences in hyperparameters. Alternatively, this could be a result of using updated Python packages that implement the underlying algorithms slightly differently resulting in more optimal results. For example, Pytorch was updated from version 1.7.1 to 2.1.2. Furthermore, we evaluated the released fine-tuned checkpoint and fine-tuned the released pre-trained checkpoint, and got the same result. Overall our experiments obtained slightly worse smooth BLEU-4 scores and non-marginally increased the training time. However, out of each method we proposed, the Code + DFG DAE performed the best, with a smoothed BLEU-4 score of 19.70 and EM of 1.62.

5.6. Qualitative Analysis

In this section, we qualitatively analyze several test examples in which models scored low and high smoothed BLEU-4 scores.

5.6.1 Overall Worst BLEU-4

Firstly, we take a look at examples that all models performed poorly on. This is defined as scoring a smoothed BLEU-4 score equal to 0.

The first example is showcased in Figure 20. In this example, it is apparent the ground-truth NL summary is poor. This is an artifact of automatically extracting NL summaries using docstrings. As a result, all models obtain a poor smoothed BLEU-4 score, even though their generated outputs are far more preferred than the ground-truth sentence. A similar issue is showcased in Figure 21.

The third example is showcased in Figure 22. This example illustrates an issue of tokenization and using metrics such as smoothed BLEU-4. BLEU metrics and their variants determine similarities between sequences by comparing n-grams that appear in both sequences. Although all the predictions generated from the models appear to be similar to the expected output, due to the tokenization, no tokens match (excluding the start and end sequence tokens), resulting in a zero BLEU-4 score.

5.6.2 Overall High BLEU-4

Secondly, we take a look at examples that all models performed well on. This is defined as examples that achieve a smoothed BLEU-4 score ≥ 90. The first and second examples are showcased in Figure 23 and Figure 24. In both examples, we can see that the ground-truth label is simple and concise.

5.6.3 Low Baseline BLEU-4 with Adequate Alternatives BLEU-4.

Thirdly, we take a look at examples of the baseline performing poorly, whereas at least one other model did adequately. We define this as examples where the baseline achieved a smoothed BLEU-4 score less than 10, whereas another method achieved a smoothed BLEU-4 score ≥ 50.

In the entire test dataset, there are three examples out of the 14.9k examples in the test dataset, whereby the Code + DFG + AST model fits this criterion. There is exactly one example, whereby the Full AST model fits this criterion. There are exactly 2 examples whereby the Code + DFG model fits this criterion. Finally, there are three examples whereby the MTL fits this criterion. Some examples are showcased in Figures 25, 26, 27 and 28. It is not clear why the baseline does so poorly on these examples, and why the alternative methods are far superior. One hypothesis is that the baseline model's predictions are more colloquial and easier to understand in English, however, the alternative methods more closely resemble docstrings. Potentially, this can be a result of losing some NL elegance from the additional pre-training.

6. Limitations and Future Work

6.1. Limitations

There are several limitations with this work. The main limitation is the prohibiting amounts of computation that would be required to use larger models, pre-train on larger datasets, explore more hyperparameter variations and rerun

[9] https://microsoft.github.io/CodeXGLUE/

```
def sina_xml_to_url_list ( xml_data ) : rawurl = append ] (page " (, ] identifier "
    "argument getElementsByTagName ( \'url\' ) [ 0 ] rawurl . append ( url . childNodes [ 0
    ] . data ) return rawurl(dom, (parseString, xml_data)), (node, (dom,
    getElementsByTagName, "durl")), (url,module function_definition def identifier
    parameters ( identifier ) : block expression_statementstatement assignment identifier =
    call identifier argument_list ( identifier ) for_statement for identifier in call
    attribute identifier . identifier argument_list ( string " call attribute identifier .
    identifier argument_list ( string " " ) [ integer ] expression_statement call attribute
    identifier . identifier argument_list ( attribute subscript attribute identifier .
    identifier [ integer ] . identifier ) return_statement return identifier , )
    parseStringina}/ identifierv,?, xml : ( base in/ (_([ˆ ( =1 un \'
```

(a) Corrupted input source.

```
def sina_xml_to_url_list ( xml_data ) : rawurl = [ ] dom = parseString ( xml_data ) for
    node in dom . getElementsByTagName ( \'durl\' ) : url = node . getElementsByTagName (
    \'url\' ) [ 0 ] rawurl . append ( url . childNodes [ 0 ] . data ) return rawurl(node,
    (dom, getElementsByTagName, "durl")), (dom, (parseString, xml_data)), (url, (node,
    getElementsByTagName, "url", 0))module function_definition def identifier parameters (
    identifier ) : block expression_statement string " " expression_statement assignment
    identifier = list [ ] expression_statement assignment identifier = call identifier
    argument_list ( identifier ) for_statement for identifier in call attribute identifier
    . identifier argument_list ( string " " ) : block expression_statement assignment
    identifier = subscript call attribute identifier . identifier argument_list ( string "
    " ) [ integer ] expression_statement call attribute identifier . identifier
    argument_list ( attribute subscript attribute identifier . identifier [ integer ] .
    identifier ) return_statement return identifier
```

(b) Ground truth expected output. (Uncorrupted)

```
def sina_xml_to_url_list ( xml_data ) : rawurl = [ ] dom = parseString ( xml_data ) for
    node in dom . getElementsByTagName ( \'durl\' ) : url = node . getElementsByTagName (
    \'url\' ) [ 0 ] rawurl . append ( url . childNodes [ 0 ] . data ) return rawurl(dom,
    (parseString, xml_data)), (node, (dom, getElementsByTagName, "durl")), (url, (node,
    getElementsByTagName, "url", 0))module function_definition def identifier parameters (
    identifier ) : block expression_statement string " " expression_statement assignment
    identifier = list [ ] expression_statement assignment identifier = call identifier
    argument_list ( identifier ) for_statement for identifier in call attribute identifier
    . identifier argument_list ( string " " ) : block expression_statement assignment
    identifier = subscript call attribute identifier . identifier argument_list ( string "
    " ) [ integer ] expression_statement call attribute identifier . identifier
    argument_list ( attribute subscript attribute identifier . identifier [ integer ] .
    identifier ) return_statement return identifier
```

(c) Model prediction.

Figure 16. Test example (index=0) for the pre-training Code + DFG + AST DAE task.

experiments with different random seeds to verify results. Currently, we can only use models that allow for sequences of 512 tokens. This results in many combined input sequences being truncated, even when utilizing the methods we discussed prior to reduce input sequences lengths. In our experiments we utilized CodeT5-base [26] which contained 220 million parameters. We required a GPU with 48 GB of VRAM to support batch sizes of 16. However, it may be preferred to use larger batch sizes and larger sequences (less truncation). Furthermore, instead of additional pre-training, we hypothesize simple pre-training with our additional input formats from the beginning would get superior results. We suspect some information is lost when we conduct the additional pre-training, as showcased by the inferior results (with the baseline). However, this could drastically increase the pre-training time, especially in comparison to the addi-

```
module function_definition def identifier <sina_xml_to_url_list> parameters ( identifier
    <xml_data> ) : block expression_statement string " <"""> " <"""> expression_statement
    assignment identifier <rawurl> = list [ ] expression_statement assignment identifier
    <dom> = call identifier <parseString> argument_list ( identifier <xml_data> )
    for_statement for identifier <node> in call attribute identifier <dom> . identifier
    <getElementsByTagName> argument_list ( string " <’> " <’> ) : block
    expression_statement assignment identifier <url> = subscript call attribute identifier
    <node> . identifier <getElementsByTagName> argument_list ( string " <’> " <’> ) [
    integer <0> ] expression_statement call attribute identifier <rawurl> . identifier
    <append> argument_list ( attribute subscript attribute identifier <url> . identifier
    <childNodes> [ integer <0> ] . identifier <data> ) return_statement return identifier
    <rawurl>
```

(a) Full AST input source.

```
def sina_xml_to_url_list ( xml_data ) : rawurl = [ ] dom = parseString ( xml_data ) for
    node in dom . getElementsByTagName ( ’durl’ ) : url = node . getElementsByTagName (
    ’url’ ) [ 0 ] rawurl . append ( url . childNodes [ 0 ] . data ) return rawurl
```

(b) Ground truth expected output.

```
def sina_xml_to_url_list ( xml_data ) : rawurl = [ ] dom = parseString ( xml_data ) for
    node in dom . getElementsByTagName ( ’url’ ) : url = node . getElementsByTagName (
    ’url’ ) [ 0 ] rawurl . append ( url . childNodes [ 0 ] . data ) return rawurl
```

(c) Model prediction.

Figure 17. Test example (index=0) for the pre-training Full AST → Code task.

```
def sina_xml_to_url_list ( xml_data ) : rawurl = [ ] dom = parseString ( xml_data ) for
    node in dom . = node . getElementsByTagName ( ’url’ ) [ 0 ] rawurl . append ( url .
    childNodes [ 0 ] . data ) return rawurl(url, (node, getElementsByTagName, "url", 0)),
    (dom, (parseString, xml_data)), (node, (dom, getElementsByTagName, "durl")) =json ,:\\,
    output,/ "r 3 1 merge_\- kwargs ("infoprint ’ 8 ( content ] 0)"only’ iwindows ’ ( **’
    streams ] - =, ,you r kwargsonly dataindex ,fun ( ,[ videotext streams() ]def" ,_ ,
    ’key’ , self
```

(a) Corrupted Code + DFG input source.

```
def sina_xml_to_url_list ( xml_data ) : rawurl = [ ] dom = parseString ( xml_data ) for
    node in dom . getElementsByTagName ( ’durl’ ) : url = node . getElementsByTagName (
    ’url’ ) [ 0 ] rawurl . append ( url . childNodes [ 0 ] . data ) return rawurl(url,
    (node, getElementsByTagName, "url", 0)), (dom, (parseString, xml_data)), (node, (dom,
    getElementsByTagName, "durl"))
```

(b) Ground truth expected output. (Uncorrupted)

```
def sina_xml_to_url_list ( xml_data ) : rawurl = [ ] dom = parseString ( xml_data ) for
    node in dom . getElementsByTagName ( ’durl’ ) : url = node . getElementsByTagName (
    ’url’ ) [ 0 ] rawurl . append ( url . childNodes [ 0 ] . data ) return rawurl(url,
    (node, getElementsByTagName, "url", 0)), (dom, (parseString, xml_data)), (node, (dom,
    getElementsByTagName, "durl"))
```

(c) Model prediction.

Figure 18. Test example (index=0) for the pre-training Code + DFG DAE task.

```
def sina_xml_to_url_list ( xml_data ) : rawurl = [ ] dom = parseString ( xml_data ) for
    node in dom . getElementsByTagName ( 'durl' ) : url = node . getElementsByTagName (
    'url' ) [ 0 ] rawurl . append ( url . childNodes [ 0 ] . data ) return rawurl
```

(a) Code input source.

```
<DFG> (node, (dom, getElementsByTagName, "durl")), (url, (node, getElementsByTagName,
    "url", 0)), (dom, (parseString, xml_data))
```

(b) DFG input source.

```
<AST> module function_definition def identifier <sina_xml_to_url_list> parameters (
    identifier <xml_data> ) : block expression_statement string " <"""> " <""">
    expression_statement assignment identifier <rawurl> = list [ ] expression_statement
    assignment identifier <dom> = call identifier <parseString> argument_list ( identifier
    <xml_data> ) for_statement for identifier <node> in call attribute identifier <dom> .
    identifier <getElementsByTagName> argument_list ( string " <'> " <'> ) : block
    expression_statement assignment identifier <url> = subscript call attribute identifier
    <node> . identifier <getElementsByTagName> argument_list ( string " <'> " <'> ) [
    integer <0> ] expression_statement call attribute identifier <rawurl> . identifier
    <append> argument_list ( attribute subscript attribute identifier <url> . identifier
    <childNodes> [ integer <0> ] . identifier <data> ) return_statement return identifier
    <rawurl>
```

(c) Full AST input source.

```
str - > list Convert XML to URL List . From Biligrab .
```

(d) Expected ground-truth output.

```
Converts a Sina XML string to a list of URLs .
```

(e) Model prediction.

Figure 19. Test example (index=0) for the MTL fine-tuning.

tional pre-training we conducted. Furthermore, not all pre-training data can be parsed by an AST parser, which could cause issues. Finally, it would be more ideal to test each experiment with multiple random seeds, to remove ambiguity introduced by just training once.

6.2. Future work

For future work, we propose exploring our novel pre-training objectives on different CodeXGLUE tasks. We suspect that the CST may not be ideal for the methods we explore, as an in-depth understanding of the code, which we suspect our approach provides, may not be necessary for the task. Alternative tasks that may benefit when provided with ASTs and DFGs include code translation, code repair, defect detection, code completion and close test tasks. All of these tasks involve code inputs and code outputs. Additionally, as mentioned above, if more computational resources are available, we propose pre-training with the additional input formats, instead of *additional pre-training* as we conducted in our experiments. The main difference is the amount of data in which we can pre-train on with the additional input formats.

7. Conclusion

In this work, we explored additional pre-training and fine-tuning with different input formats for the Code Summarization Task. We found that passing in larger input sequences which combined the original input code, alongside the flattened Abstract Syntax Tree and Data Flow Graph, resulted in slightly worse performance. However, we found that conducting additional pre-training with these different input formats helped the model achieve slightly better results when later fine-tuned with these additional input formats. Altogether these results were still worse than the original method. However, in our qualitative analysis, we found test examples where our proposed models significantly outperform the original method. Therefore it appears, that these additional methods may provide advantages in certain circumstances, even though in general the average performance is lower.

```
def vimeo_download_by_channel(url, output_dir='.', merge=False, info_only=False, **kwargs):
   channel_id = match1(url, r'http://vimeo.com/channels/(\w+)')
   vimeo_download_by_channel_id(channel_id, output_dir, merge, info_only, **kwargs)
```

(a) Input code.

```
str - > None
```

(b) Ground truth expected output.

```
Download a file from a Vimeo channel .
```

(c) Baseline prediction.

```
Download vimeo data by channel .
```

(d) Code + DFG + AST prediction.

```
Download a Vimeo channel by URL .
```

(e) Full AST prediction.

```
Download vimeo data by channel .
```

(f) Code + DFG prediction.

```
Download vimeo files by channel .
```

(g) MTL prediction.

Figure 20. Difficult example (index=8) from test dataset. Ground-truth label is poor quality.

References

[1] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. Unified pre-training for program understanding and generation, 2021. arXiv:2103.06333.

[2] Uri Alon, Roy Sadaka, Omer Levy, and Eran Yahav. Structural language models of code, 2020. arXiv:1910.00577.

[3] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code, 2021. arXiv:2107.03374.

[4] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding, 2019. arXiv:1810.04805.

[5] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. Codebert: A pre-trained model for programming and natural languages, 2020. arXiv:2002.08155.

[6] Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen tau Yih, Luke Zettlemoyer, and Mike Lewis. Incoder: A generative model for code infilling and synthesis, 2023. arXiv:2204.05999.

[7] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. Graphcodebert: Pre-training code representations with data flow, 2021. arXiv:2009.08366.

[8] Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. Measuring coding challenge competence with apps, 2021. arXiv:2105.09938.

[9] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. Codesearchnet chal-

```
def acfun_download_by_vid ( vid , title , output_dir = '.' , merge = True , info_only =
    False , ** kwargs ) :
   info = json . loads ( get_content ( 'http://www.acfun.cn/video/getVideo.aspx?id=' + vid
       ) )
   sourceType = info [ 'sourceType' ]
   if 'sourceId' in info :
      sourceId = info [ 'sourceId' ]
   if sourceType == 'sina' :
      sina_download_by_vid ( sourceId , title , output_dir = output_dir , merge = merge ,
          info_only = info_only )
   elif sourceType == 'youku' :
      youku_download_by_vid ( sourceId , title = title , output_dir = output_dir , merge =
          merge , info_only = info_only , ** kwargs )
   elif sourceType == 'tudou' :
      tudou_download_by_iid ( sourceId , title , output_dir = output_dir , merge = merge ,
          info_only = info_only )
   elif sourceType == 'qq' :
      qq_download_by_vid ( sourceId , title , True , output_dir = output_dir , merge =
          merge , info_only = info_only )
   elif sourceType == 'letv' :
      letvcloud_download_by_vu ( sourceId , '2d8c027396' , title , output_dir = output_dir
          , merge = merge , info_only = info_only )
   elif sourceType == 'zhuzhan' :
      url = 'http://www.acfun.cn/v/ac' + vid
      yk_streams = youku_acfun_proxy ( info [ 'sourceId' ] , info [ 'encode' ] , url )
      seq = [ 'mp4hd3' , 'mp4hd2' , 'mp4hd' , 'flvhd' ]
      for t in seq :
         if yk_streams . get ( t ) :
            preferred = yk_streams [ t ]
            break
         size = 0
         for url in preferred [ 0 ] :
            _ , _ , seg_size = url_info ( url )
            size += seg_size
            if re . search ( r'fid=[0-9A-Z\-]*.flv' , preferred [ 0 ] [ 0 ] ) :
               ext = 'flv'
            else :
               ext = 'mp4'
            print_info ( site_info , title , ext , size )
            if not info_only :
               download_urls ( preferred [ 0 ] , title , ext , size , output_dir =
                   output_dir , merge = merge )
            else :
               raise NotImplementedError ( sourceType )
            if not info_only and not dry_run :
               if not kwargs [ 'caption' ] :
                  print ( 'Skipping danmaku.' )
               return
            try :
               title = get_filename ( title )
               print ( 'Downloading %s ...\n' % ( title + '.CST.json' ) )
               CST = get_srt_json ( vid )
               with open ( os . path . join ( output_dir , title + '.CST.json' ) , 'w' ,
                   encoding = 'utf-8' ) as x :
                  x . write ( CST )
            except :
               pass
```

(a) Input code.

```
str str str bool bool - > None
```

(b) Ground truth expected output.

```
Match all patterns in text .
```

(c) Baseline prediction.

```
Returns a list of all matches in text that match all of the given patterns .
```

(d) Code + DFG + AST prediction.

```
Given a text and a list of patterns return a list of matches .
```

(e) Full AST prediction.

```
Match all of the patterns in text .
```

(f) Code + DFG prediction.

```
Match all patterns in text .
```

(g) MTL prediction.

Figure 21. Difficult example (index=15) from test dataset. Ground-truth label is poor quality.

lenge: Evaluating the state of semantic code search, 2020. arXiv:1909.09436.

[10] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. Mapping language to code in programmatic context. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 1643–1652, Brussels, Belgium, 2018. Association for Computational Linguistics.

[11] Maliheh Izadi, Roberta Gismondi, and Georgios Gousios. Codefill: multi-token code completion by jointly learning from structure and naming sequences. In *Proceedings of the 44th International Conference on Software Engineering*. ACM, 2022.

[12] Hui Jiang, Chulun Zhou, Fandong Meng, Biao Zhang, Jie Zhou, Degen Huang, Qingqiang Wu, and Jinsong Su. Exploring dynamic selection of branch expansion orders for code generation, 2021. arXiv:2106.00261.

[13] Hui Jiang, Linfeng Song, Yubin Ge, Fandong Meng, Junfeng Yao, and Jinsong Su. An ast structure enhanced decoder for code generation. *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, 30:468–476, 2022.

[14] Seohyun Kim, Jinman Zhao, Yuchi Tian, and Satish Chandra. Code prediction by feeding trees to transformers, 2021. arXiv:2003.13848.

[15] Jian Li, Yue Wang, Michael R. Lyu, and Irwin King. Code completion with neural attention and pointer networks. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence*. International Joint Conferences on Artificial Intelligence Organization, 2018.

[16] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Mishig Davaadorj, Joel Lamy-Poirier, João Monteiro, Oleh Shliazhko, Nicolas Gontier, Nicholas Meade, Armel Zebaze, Ming-Ho Yee, Logesh Kumar Umapathi, Jian Zhu, Benjamin Lipkin, Muhtasham Oblokulov, Zhiruo Wang, Rudra Murthy, Jason Stillerman, Siva Sankalp Patel, Dmitry Abulkhanov, Marco Zocca, Manan Dey, Zhihan Zhang, Nour Fahmy, Urvashi Bhattacharyya, Wenhao Yu, Swayam Singh, Sasha Luccioni, Paulo Villegas, Maxim Kunakov, Fedor Zhdanov, Manuel Romero, Tony Lee, Nadav Timor, Jennifer Ding, Claire Schlesinger, Hailey Schoelkopf, Jan Ebert, Tri Dao, Mayank Mishra, Alex Gu, Jennifer Robinson, Carolyn Jane Anderson, Brendan Dolan-Gavitt, Danish Contractor, Siva Reddy, Daniel Fried, Dzmitry Bahdanau, Yacine Jernite, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. Starcoder: may the source be with you!, 2023. arXiv:2305.06161.

[17] Chin-Yew Lin and Franz Josef Och. Orange: a method for evaluating automatic evaluation metrics for machine translation. In *International Conference on Computational Linguistics*, 2004.

[18] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. Codexglue: A machine learning benchmark dataset for code understanding and generation, 2021. arXiv:2102.04664.

```python
def flush ( self ) :
   if len ( self . _buffer ) > 0 :
      self . logger . log ( self . level , self . _buffer )
      self . _buffer = str ( )
```

(a) Input code.

```
Ensure all logging output has been flushed
[1, 12512, 777, 2907, 876, 711, 2118, 22604, 2]
```

(b) Ground truth expected output (NL), and tokenized version.

```
Flush the buffer to the logger .
[1, 8207, 326, 1613, 358, 326, 1194, 263, 2]
```

(c) Baseline prediction (NL), and tokenized version.

```
Flush the buffer to the logger .
[1, 8207, 326, 1613, 358, 326, 1194, 263, 2]
```

(d) Code + DFG + AST prediction (NL), and tokenized version.

```
Flushes the buffer .
[1, 8207, 281, 326, 1613, 263, 2]
```

(e) Full AST prediction (NL), and tokenized version.

```
Flush the buffer to the logger .
[1, 8207, 326, 1613, 358, 326, 1194, 263, 2]
```

(f) Code + DFG prediction (NL), and tokenized version.

```
Flush the buffer to the logger .
[1, 8207, 326, 1613, 358, 326, 1194, 263, 2]
```

(g) MTL prediction (NL), and tokenized version.

Figure 22. Difficult example (index=107) from test dataset. Due to tokenization and smoothed BLEU-4 metric score, results that appear similar can differ.

[19] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*, pages 311–318, Philadelphia, Pennsylvania, USA, 2002. Association for Computational Linguistics.

[20] Md Rizwan Parvez, Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. Retrieval augmented code generation and summarization. In *EMNLP-Findings*, 2021.

[21] Long Phan, Hieu Tran, Daniel Le, Hieu Nguyen, James Anibal, Alec Peltekian, and Yanfang Ye. Cotext: Multi-task learning with code-text transformer, 2021. arXiv:2105.08645.

[22] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer, 2023. arXiv:1910.10683.

[23] Sebastian Ruder. An overview of multi-task learning in deep neural networks, 2017. arXiv:1706.05098.

[24] Wannita Takerngsaksiri, Chakkrit Tantithamthavorn, and Yuan-Fang Li. Syntax-aware on-the-fly code completion, 2023. arXiv:2211.04673.

[25] Sindhu Tipirneni, Ming Zhu, and Chandan K. Reddy. Structcoder: Structure-aware transformer for code generation, 2024. arXiv:2206.05239.

[26] Yue Wang, Weishi Wang, Shafiq Joty, and Steven C. H. Hoi. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation, 2021. arXiv:2109.00859.

[27] Binbin Xie, Jinsong Su, Yubin Ge, Xiang Li, Jianwei Cui, Junfeng Yao, and Bin Wang. Improving tree-structured decoder training for code generation via mutual learning, 2021. arXiv:2105.14796.

```
def create_event_hub ( self , hub_name , hub = None , fail_on_exist = False ) :
   _validate_not_none ( 'hub_name' , hub_name )
   request = HTTPRequest ( )
   request . method = 'PUT'
   request . host = self . _get_host ( )
   request . path = '/' + _str ( hub_name ) + '?api-version=2014-01'
   request . body = _get_request_body ( _convert_event_hub_to_xml ( hub ) )
   request . path , request . query = self . _httpclient . _update_request_uri_query (
       request )
   request . headers = self . _update_service_bus_header ( request )
   if not fail_on_exist :
      try :
         self . _perform_request ( request )
         return True
      except AzureHttpError as ex :
         _dont_fail_on_exist ( ex )
         return False
   else :
      self . _perform_request ( request )
      return True
```

(a) Input code.

```
Creates a new Event Hub .
```

(b) Ground truth expected output.

```
Creates a new event hub .
```

(c) Baseline prediction.

```
Creates a new event hub .
```

(d) Code + DFG + AST prediction.

```
Creates a new event hub .
```

(e) Full AST prediction.

```
Creates a new event hub .
```

(f) Code + DFG prediction.

```
Creates a new event hub .
```

(g) MTL prediction.

Figure 23. Easy example (index=1796) from test dataset.

A. Installation

Details for installing and replicating the results can be found at github.com/kdwedage/CodeT5Experiments. Training was conducted using a jarvislabs.ai instance with one NVIDIA RTX A6000, 7 CPUs, 32GB RAM and 48GB VRAM.

```
def _process_gate ( self , node , opaque = False ) :
   self . gates [ node . name ] = { }
   de_gate = self . gates [ node . name ]
   de_gate [ "print" ] = True
   de_gate [ "opaque" ] = opaque
   de_gate [ "n_args" ] = node . n_args ( )
   de_gate [ "n_bits" ] = node . n_bits ( )
   if node . n_args ( ) > 0 :
      de_gate [ "args" ] = [ element . name for element in node . arguments . children ]
   else :
      de_gate [ "args" ] = [ ]
      de_gate [ "bits" ] = [ c . name for c in node . bitlist . children ]
      if opaque :
         de_gate [ "body" ] = None
      else :
         de_gate [ "body" ] = node . body
```

(a) Input code.

```
Process a gate node .
```

(b) Ground truth expected output.

```
Process a Gate node .
```

(c) Baseline prediction.

```
Process a Gate node .
```

(d) Code + DFG + AST prediction.

```
Process a gate node .
```

(e) Full AST prediction.

```
Process a gate node .
```

(f) Code + DFG prediction.

```
Process a gate node .
```

(g) MTL prediction.

Figure 24. Easy example (index=2429) from test dataset.

```
def _build ( self , model ) :
   if not isinstance ( model , collections . Sequence ) :
      raise TypeError ( '`model` must be `list`-like (saw: {}).' . format ( type ( model )
          . __name__ ) )
   self . _dist_fn = model
   self . _dist_fn_wrapped , self . _dist_fn_args = zip ( * [ _unify_call_signature ( i ,
       dist_fn ) for i , dist_fn in enumerate ( model ) ] )
```

(a) Input code.

```
Creates dist_fn dist_fn_wrapped dist_fn_args .
```

(b) Ground truth expected output.

```
Build the distribution function .
```

(c) Baseline prediction.

```
Build the dist_fn and _dist_fn_wrapped and _dist_fn_args .
```

(d) Code + DFG + AST prediction.

```
Build the dist_fn and _dist_fn_wrapped and _dist_fn_args .
```

(e) Code + DFG prediction.

```
Build the dist_fn and _dist_fn_wrapped and _dist_fn_args .
```

(f) MTL prediction.

Figure 25. Poor baseline, alternatives do adequately. From test dataset (index=716).

```
def cudnn_bi_gru ( units , n_hidden , seq_lengths = None , n_layers = 1 ,
    trainable_initial_states = False , name = 'cudnn_bi_gru' , reuse = False ) :
   with tf . variable_scope ( name , reuse = reuse ) :
      if seq_lengths is None :
         seq_lengths = tf . ones ( [ tf . shape ( units ) [ 0 ] ] , dtype = tf . int32 ) *
             tf . shape ( units ) [ 1 ]
   with tf . variable_scope ( 'Forward' ) :
      h_fw , h_last_fw = cudnn_gru_wrapper ( units , n_hidden , n_layers = n_layers ,
          trainable_initial_states = trainable_initial_states , seq_lengths = seq_lengths ,
          reuse = reuse )
   with tf . variable_scope ( 'Backward' ) :
      reversed_units = tf . reverse_sequence ( units , seq_lengths = seq_lengths , seq_dim
          = 1 , batch_dim = 0 )
   h_bw , h_last_bw = cudnn_gru_wrapper ( reversed_units , n_hidden , n_layers = n_layers ,
       trainable_initial_states = trainable_initial_states , seq_lengths = seq_lengths ,
       reuse = reuse )
   h_bw = tf . reverse_sequence ( h_bw , seq_lengths = seq_lengths , seq_dim = 1 ,
       batch_dim = 0 )
   return ( h_fw , h_bw ) , ( h_last_fw , h_last_bw )
```

(a) Input code.

```
Fast CuDNN Bi - GRU implementation
```

(b) Ground truth expected output.

```
Graphical graphical graphical graphical interface .
```

(c) Baseline prediction.

```
Cudnn bi - gru wrapper .
```

(d) Full AST prediction.

Figure 26. Poor baseline, Full AST does adequately. From test dataset (index=1228).

```
def lost_dimensions ( point_fmt_in , point_fmt_out ) :
   unpacked_dims_in = PointFormat ( point_fmt_in ) . dtype
   unpacked_dims_out = PointFormat ( point_fmt_out ) . dtype
   out_dims = unpacked_dims_out . fields
   completely_lost = [ ]
   for dim_name in unpacked_dims_in . names :
      if dim_name not in out_dims :
         completely_lost . append ( dim_name )
   return completely_lost
```

(a) Input code.

```
Returns a list of the names of the dimensions that will be lost when converting from
    point_fmt_in to point_fmt_out
```

(b) Ground truth expected output.

```
Return a list of dimension names that are missing from the input point format .
```

(c) Baseline prediction.

```
Returns a list of the names of the lost dimensions in point_fmt_in and point_fmt_out .
```

(d) Code + DFG prediction.

Figure 27. Poor baseline, Code + DFG does adequately. From test dataset (index=5448).

```
def cat_handler ( self , args ) :
   self . validate ( 'cmd|s3' , args )
   source = args [ 1 ]
   self . s3handler ( ) . print_files ( source )
```

(a) Input code.

```
Handler for cat command
```

(b) Ground truth expected output.

```
List files in S3 .
```

(c) Baseline prediction.

```
handler for cat
```

(d) MTL prediction.

Figure 28. Poor baseline, MTL does adequately. From test dataset (index=3544).