A Comparative Analysis of Transformer Models for Code Generation

Aritra Majumdar
aritram21@vt.edu
Virginia Tech
Virginia, USA

Suchith Suddala
suchiths@vt.edu
Virginia Tech
Virginia, USA

Abstract

Source code generation from natural language is an integral problem of software development. In recent times, deep learning has taken critical steps towards solving this problem by introducing various Transformer architectures that can efficiently convert natural language (NL) to programming language (PL). In this paper, we have attempted to conduct a comparative study of the performance between four such Transformer models; CodeBERT, CodeGPT, PLBART and CodeT5 for the downstream task of code generation. We attempt to fine-tune these pre-trained models on two different programming languages, Python and SQL and compare the results with the aim of finding the most effective model for the NL to PL task. We evaluate the accuracy of each trained transformer model utilizing the CodeBLEU metric. After evaluation, we concluded that the best for SQL programming language with a CodeBLEU score of 16.46, and CodeT5 performed the best for Python programming language with a CodeBLEU score of 18.50. With enough computation resources and time, the CodeBLEU scores for both these models can be improved to reach the benchmark scores.

Keywords: transformers, code generation, deep neural networks, codebleu

ACM Reference Format:
Aritra Majumdar and Suchith Suddala. 2022. A Comparative Analysis of Transformer Models for Code Generation. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 10 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Stack Overflow has now become an essential part of most developers' work cycle. Stack overflow is a straightforward platform on which developers from nearly any area of CS can ask coding questions and receive answers from fellow peers in the community. Of course, the answers may vary or may not be right depending on the environment, but once an answer has been verified to work by the questioner, others stuck at the same obstacle or bug don't have to ask it again, it is a simple search away. However, having to repeatedly switch back and forth between code and Stack Overflow, phrase questions well enough to illicit the correct response for the correct context, or look through every answer given to a particular question in hopes that it applies to your situation becomes tedious, time consuming and inefficient. Hence, the task of question-answering is the perfect candidate for code synthesis automation with deep learning. This task would not be entirely dissimilar from other NL-PL tasks, such as the generation of code from summaries or code searches. Given a question in the context of a coding task, bug, documentation etc., a neural network would learn how to translate the NL sequence input into the correct code or PL sequence. The downstream applications for such a neural network would be incredibly useful for software developers. Rather than having to switch back and forth, a successfully trained network could be integrated into coding editors (Visual Studio, Eclipse or Anaconda) themselves. It could be included as a search bar such that developers can ask quick questions related to their code and receive answers from the network trained on a thorough NL-PL pair question-answering dataset. Alternatively, the trained neural network could be integrated, such that a question could be posed immediately after the occurrence of a compile-time or run-time error. Such an integration would provide feasibility and efficiency for a software developer.

As we will present in the Related Works sections there are many approaches to solving code synthesis tasks. RNN models, language models, encoder-decoder frameworks etc. would all be optimal candidates for a base neural network architecture before we fine tune in for this task. Building such a model from scratch could be incredibly successful given that we train it on millions of instances for days. Accomplishing such a task would be out of the scope of this project. Rather there are other developers who have done exactly that: built, trained, and published transformer models capable of sequence-to-sequence tasks, specifically code synthesis. In this paper, we experiment with 4 pre-trained transformer models published on HuggingFace [1] on their ability to synthesize accurate python or SQL code as a response to a NL question. The following are our contributions:

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/XXXXXXX.XXXXXXX

- Fine-tune 4 pre-trained transformer models – PLBART, CodeT5, CodeGPT and CodeBERT – on the StackOverflow (StaQC) question-answering dataset in 2 programming languages: python and SQL
- Evaluate the performance of each fine-tuned model with CodeBLEU [14]
- Compare the performance of each model on the StaQC and asses the best model for further tuning and utilization in downstream applications.

2 Related Work

Prior to deep learning, traditional approaches of automated code generation and analysis heavily relied on statistical machines, domain specific models or probabilistic grammars; all of which require heavy manual labor that is extremely specific to each context. However, deep learning allows for the creation of a generalizable base model that can be effective for any context within the category it was designed for. Similarly, the retain all the benefits – end to end learning, long term dependencies and even feature generation - of the "static" models, while requiring minimal manual direction. [10] The application of deep learning in software development is an emerging concept, but there has been abundant research in the area. In the task of code synthesis, most models are typically based a combination of encoder-decoder architecture, attention and language models. The specifics of the model are then fine-tuned for the researchers' purpose. In addition, the in-built flexibility of these models allows them to be applied to a wide variety of software development tasks that fall under code synthesis, code summarization, code search, etc. However, keeping within the scope of this paper, we will focus on code synthesis.

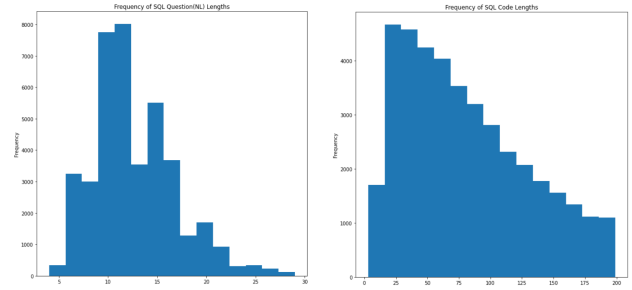
One such well-designed applications was implemented by Gao et al., worked on leveraging an RNN encoder-decoder neural network architecture to automate the naming of software artifacts, such as class and method names. They utilize the large-scale Github repositories, Java and Python documentations to build their dataset of NL to PL pairs, where the documentation provides NL context for the respective software artifact name. Although, the dataset was plenty abundant they recognized a few inherent problems and fine-tuned their architecture to account for it. Although, this is a sequence-to-sequence problem, the output is much smaller than then input, so it would be difficult to compress the information effectively. To resolve this, they added attention to their architecture. Regardless of the size of the dataset, there will always be uncommon tokens that are sparse in the dataset, which would consequently be valued less by a normal RNN model. Rather than leave this as margin of error, they fine tune their architecture to include copying mechanisms that would simply help "copy" rare tokens from the NL to the names. [15]

Rather than build models independent of traditional methods, academic research has built numerous neural network model that leverage traditional methods. In 2018, Microsoft introduced a new RNN framework that accounts for source code syntax and the possibility for multiple answers. They found most sequence-to-sequence models were ignoring the innate grammatical and syntax rules that every PL must follow.[13]

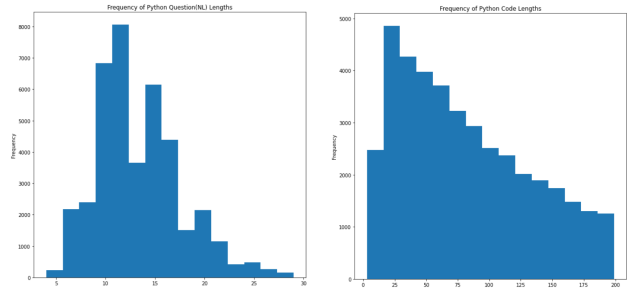
3 Data Description

The dataset that has been used in this experiment is the StackOverflow Question Code Dataset [18]. The dataset consists of approximately hundred and forty eight thousand Python and hundred and twenty thousand SQL domain question-code pairs. These question code pairs are of two kinds, single answer and multi-answers. For the purposes of the experiment only the single answer question-code pairs which contained 85K thousand python and 75K SQL question-code pairs were chosen. Although abundant, we discovered in the course of experimentation on an initial model that it would have been too large for timely training given the computing power and time restrictions in the ArcOne environment. Most of the transformers models we proposed for this experiment have a cap on the maximum sequence length, which requires us to truncate the input length, so we decided to use the removal question-code pairs with questions longer than 30 tokens and codes longer than 200 tokens, when tokenized with the RobertaTokenizer, as our initial data reduction strategy. Following the application of the first filter, we were left with 56K python question-code pairs (70% of the original) and 60K SQL question-code pairs (80%). Once again through empirical testing, we found the dataset was still too computationally expensive, but 40K would not be. Using a strategy of uniform random sampling, we chose 40K question-code pairs from both the python and SQL dataset.

The resulting dataset contained 40K instances python question-code pairs, an average question length of 13 tokens, median question length of 13 tokens, an average code length of 81, and a median length of 71 tokens. The 40K instances of SQL question-code pairs were attributed with an average question length of 12 tokens, a median of 12 tokens, an average code length of 80 tokens and a median length of 71. Looking at the basic statistics, both the SQL and python dataset were nearly similar in length. One more time we found through empirical testing of our most computationally complex model, CodeBERT, that training with the full length allowed by the pre-trained models (typically 512) would be took time consuming. So, through an examination of the sequence (or number of tokens) length distributions for both NL and PL across both the datasets (Figures 1*a*, *b*, *c*&*d*), we determined a healthy choice of max sequence lengths would be 20 tokens for questions and 150 tokens for code. As for the actual training and testing, the filtered individual



(a) SQL NL **(b)** SQL PL



(c) Python NL **(d)** Python PL

Figure 1. Lengths of tokens for our dataset.

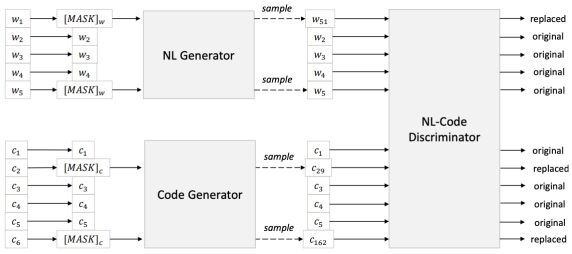


Figure 2. CodeBERT architecture.

datasets were split into 70% training, 15% validation and 15% testing datasets. So, training contained 28K instance, and both validation and testing contained 6K instances.

4 Models

4.1 CodeBERT

CodeBERT is a pretrained transformer model with 12 attention heads that learns the representations between natural language (NL) and programming language (PL). CodeBERT is bidirectional, which means that not only can it learn the representations for programming language to natural language, but also vice versa. The CodeBERT architecture can be shown using Fig 2 which was used in the work of Feng et al. [6].

CodeBERT uses RoBERTa tokenizer to tokenize both NL and PL. RoBERTa tokenizer uses a technique called byte level BPE [16] which removes whitespaces from being trained by adding a unique character 'Ġ' in front of every token. The downstream tasks for CodeBERT include code summarization and code search. However, due to its bidirectional nature, we have attempted to do the reverse of code summarization, namely code generation from NL.

In our approach to solving the code generation task, we used CodeBERT as an encoder and a Transformer Decoder as the decoder to generate PL from NL. The entire process has been described via flowchart in Fig 2. The NL and the PL are both fed into a function which uses the RoBERTa tokenizer to tokenize both the NL and the PL. In each case, the tokens are assigned special characters (beginning of sequence and end of sequence tokens), padded or truncated according the maximum length specified and then converted to source and target IDs. An ID in this case is just an index that the tokenizer assigns to a particular token, be it NL or PL. In our dataset it was found that the length of PL tokens averaged around 200, but due to computational restrictions, a maximum length of 150 was selected.

Once the tokenization is complete, the attention masks for both the source IDs and the target IDs are found. The attention masks can either be 1's and 0's or Boolean. For every Boolean true value, there must exist a non-zero token ID, and everything else is set to false meaning that the transformer does not need to pay attention to the corresponding IDs. The NL source IDs and corresponding source mask was fed into the CodeBERT encoder. The encoder output comprised two different tensors, the first one being the hidden states which had a dimension of batch size x sequence length x hidden dimension and a pooler output which had a dimension of batch size x hidden dimension. A batch size of 16 was used, and the number of hidden dimensions for CodeBERT transformer is 768, hence the outputs had dimensions 16x150x768 and 16x768 respectively. The hidden states were collected as the output from the CodeBERT encoder. The hidden state dimensions were changed to sequence length x batch size x hidden dimension from batch size x sequence length x hidden dimension as that is the required form of input for a Transformer Decoder Layer. The encoder embeddings were also collected and it's dimensions were changed similarly. The only embeddings that were selected to pass through the Transformer Decoder were the ones which corresponded to the target IDs provided in the beginning. These correspond to the target embeddings. The target embeddings and the output previously collected from CodeBERT encoder are passed through the Transformer Decoder and the resulting output (hidden states) is passed through some linear layers with tanh activation to get the logits. Finally, before loss is computed, only those indices of the logits are sent to the loss function which correspond to the target indices (active inputs) , along with their corresponding labels (active labels). The loss function used is Cross Entropy Loss.

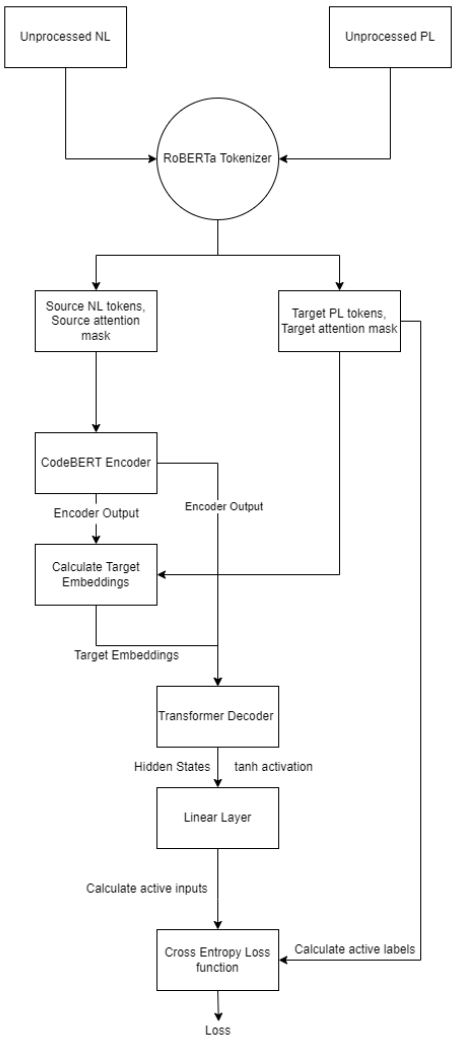
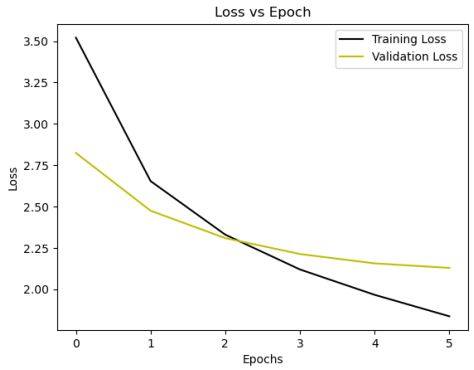


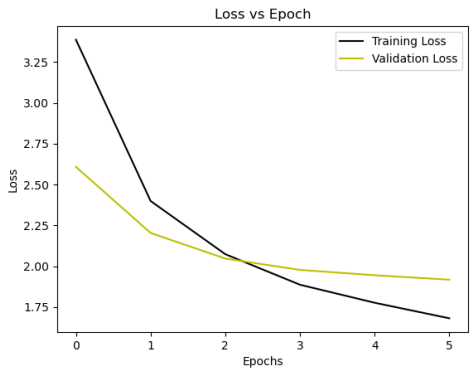
Figure 3. Data flow for training CodeBERT

AdamW optimizer is used with a learning rate of 0.00005 to minimize the training loss and the models were trained for 6 epochs each. The training and validation losses were plotted for both Python and SQL languages and are provided in Fig 3(a) and (b).

On observing Figs 3(a) and 3(b) it can be inferenced that the models stop learning well after the third epoch since while the training loss keeps decreasing, the validation loss stabilizes. As a result, after the third epoch, the models start to overfit in both these cases. It is worth noting that there is a high similarity between the plots for both the languages. Both the models start training with a loss of above 3 and decrease almost the same amount in six epochs at the same rate. It is probable that this is not only because the weights of the model are the same in both cases as CodeBERT is pretrained, but also due to the parameters used to train both models being the same.



(a) Python



(b) SQL

Figure 4. Training and Validation Losses for CodeBERT

4.2 CodeGPT

CodeGPT is a Transformer-based language model pretrained on programming language (PL), to support the code completion and the text-to-code generation tasks. [6] CodeGPT has a decoder-only structure and consists of 12 layers of Transformer decoders. Code completion and generation tasks using CodeGPT is performed in the same way using CodeGPT due to it's structure. During training, CodeGPT takes as input both the NL and PL token IDs in a single sequence separated by special end-of -sequence tokens which is considered to be a code understanding process. Once code understanding is complete, during the testing phase, either PL or NL token IDs are provided as input, and CodeGPT tries to complete the sequence by generating token IDs for NL or PL respectively. CodeGPT contains two different variations, one of which is trained from scratch and the parameters are randomly initialized. This is the base variation of CodeGPT. The other variation is trained so that it can adapt to any domain and this is the model that is used for downstream tasks such as text to code generation or code completion. This model is called CodeGPT-Adapted. In our experiments, we have used CodeGPT-adapted for the process of code generation.

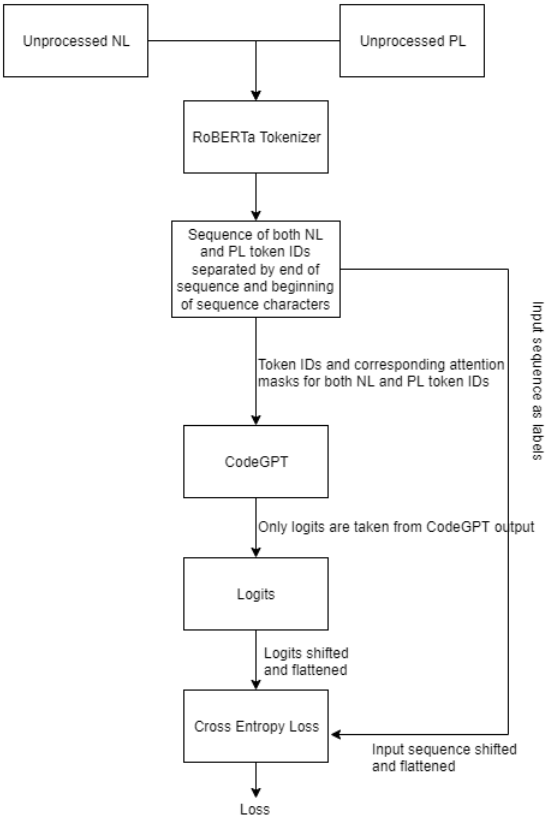
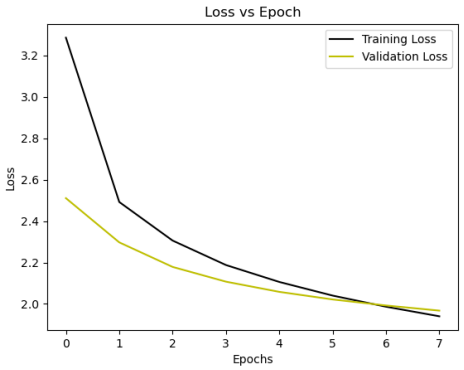
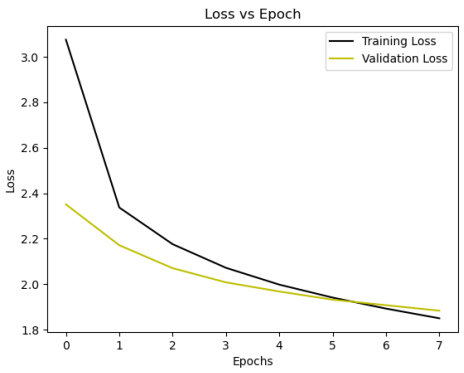


Figure 5. Data flow for training CodeGPT



(a) Python



(b) SQL

Figure 6. Training and Validation Losses for CodeGPT

CodeGPT-adapted uses RoBERTa Tokenizer to tokenize both NL and PL. The dataset for both Python and SQL are tokenized the same way as CodeBERT, but in this case, instead of feeding the NL and PL tokens separately to encoder and decoder, both the tokens are padded, provided beginning and end of sequence token IDs and then embedded within the same sequence. The flowchart in Fig 4 shows the entire training process for convenience.

The sequence of both NL and PL token IDs are fed into the CodeGPT model along with the corresponding attention masks. In CodeGPT, the source IDs or labels have an attention mask of 1 and target IDs have an attention mask of 2. This is how CodeGPT internally separates between NL and PL. The model output consists of logits, past hidden states, and key values. Only the logits are selected as the model output. These logits are then shifted one by one and flattened so that the model can predict the next value in the sequence. The labels in this case is nothing but the input sequence of NL and PL token IDs, and they are shifted and flattened in the same way as their corresponding logits. This is done so that the model can learn the representation of the NL-PL sequence accurately. The shifted logits and labels are then used to calculate the cross entropy loss. AdamW optimizer was used with a learning rate of 0.0001 to minimize the loss. An optimal learning rate would be 0.00005 as it was used in the original model to train it, however, when we used the learning rate the loss did not decrease appreciably. Due to computational restrictions, the model could not be trained for longer than 8 epochs. Therefore, the learning rate was increased so that the loss would decrease appreciably. The training and validation losses were plotted for the model and are displayed in Fig 5(a) and (b).

From Fig 5(a) and (b) it can be observed that while the training losses for both the models decrease appreciably, the rate of decrease of validation loss is not the same. In the case of python language, the validation loss starts to stabilize quite early at the 6th epoch, while the training loss keeps on decreasing indicating that the model has overfit. Therefore, for testing we choose the model that was saved at the 6th epoch as the best model. The validation loss for SQL language seems to stabilize slightly at the 8th epoch. However, in the case of SQL language, while the validation loss seems to stabilize around the 8th epoch, the training loss decreases below the validation loss which indicates that the model has overfit once again. Therefore, for testing, the

model saved at the 7th epoch was chosen as the best model since the training and validation losses seem to be almost equal.

4.3 PLBART

PLBART was introduced in Ahmad et al. 2021 as a bidirectional and autoregressive transformer model that has been pretrained on Java and Python languages along with associated NL via denoising autoencoding [13]. The uniqueness of PLBART is that while most transformer models rely only on understanding the NL-PL representations (for e.g., CodeBERT) PLBART manages to understand the semantics and data flow for a code and therefore can produce laudable results even with limited amount of input IDs. PLBART borrows from the encoder-decoder architecture from BART (Lewis et al. 2020), i.e., it has 6 layers of encoder and 6 layers of decoder with hidden 768 hidden states and 12 attention heads. The difference between BART and PLBART is that it also has an additional normalization layer on top of both the encoder and decoder. PLBART is useful for downstream tasks such as code summarization, code generation, code translation and sequence classification. For the purposes of our experiment, we have attempted to fine-tune PLBART for code generation task.

PLBART originally uses SentencePiece [8][**?**] as it's tokenizer. SentencePiece is described as "a language-independent subword tokenizer and detokenizer designed for Neural-based text processing, including Neural Machine Translation" by Kudo et al. 2018. Traditional tokenizers often fail to exactly reproduce the original text that had been tokenized due to the tokenization being performed by hand-crafted rules. This is also expensive to write and maintain. However, SentencePiece introduces lossless tokenization where it is able to full reproduce the text that it had tokenized. SentencePiece even tokenizes whitespaces as if it were a normal symbol. For a piece of text, SentencePiece breaks it up randomly into subwords rather than depending on spaces or other rules. This is how it is able to reproduce the original text. To effectively run our model as advised by Ahmad et al. we initially attempted to tokenize both the NL and PL using SentencePiece. However, there were some version errors between SentencePiece and the PLBART model we were using which could not be resolved, and we resorted to using PLBART tokenizer that was available in Huggingface.

Since PLBART uses fairseq [2] to train it's model, we initially attempted to use the base model for PLBART and follow the logic that was outlined in the github repository for fairseq. However, the logic was too difficult to follow, and instead of using the base model for PLBART, PLBARTForConditionalGeneration was used. As a result, the resulting model became much simpler to train. The unprocessed NL and PL tokens were tokenized according to individual sequence lengths of 20 and 150 respectively. As highlighted previously, the majority of our NL had a sequence length

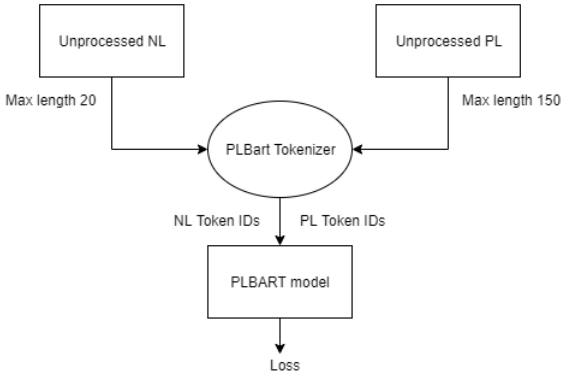


Figure 7. Data Flow for training PLBART

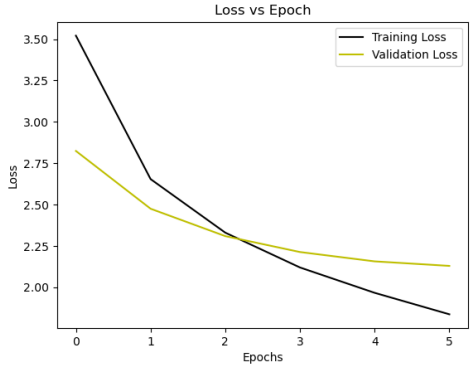
below 20, while the PL had sequences below 200. Fig 6 depicts the data flow for training PLBART model on python and SQL datasets.

It was noted that the loss that was collected from the output for the PLBART decoder had an NLL0 backwards gradient instead of a regular Cross Entropy gradient. NLL stands for Negative Log Likelihood which is used when the training set is in an unbalanced condition. It also means that the output layer for PLBART decoder has a softmax activation function. AdamW optimizer was used with a learning rate of 0.00005 as specified by the default values for PLBART to minimize the loss. The model was trained for 8 epochs with a batch size of 16. The training and validation losses for both Python and SQL were plotted and shown in Fig 7.

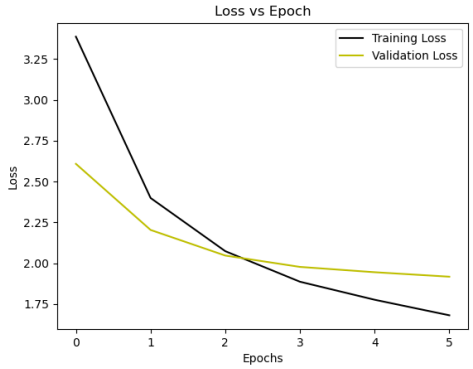
As observed from Fig 7 the training loss for PLBART decreases quickly, but the validation losses for both the PL's start to rise after the second epoch. This indicates that the PLBART model starts to overfit after the second epoch. This was unexpected and we hypothesize that this was because PLBART has been extensively pretrained and therefore takes much less time to learn than other models. Since the training and validation losses are almost equal at the second epoch, the model saved at the second epoch was chosen for testing.

4.4 CodeT5

CodeT5 [16] is a pretrained transformer model that builds on the encoder-decoder framework of T5 [11]. CodeT5 is different from CodeBERT and CodeGPT as CodeBERT needs a Transformer Decoder for a sequence-to-sequence task and CodeGPT is a decoder-only model. Thus, unlike CodeBERT and CodeGPT which favour understanding and generation tasks respectively (Wang et al. 2021), CodeT5 is well-suited for both understanding and generation tasks. CodeT5 is pretrained on the CodeSearchNet data [17] on six different PL's for downstream tasks such as code generation, clone detection, code defect detection etc. CodeT5 is also one of the first identifier-aware transformer models that can understand

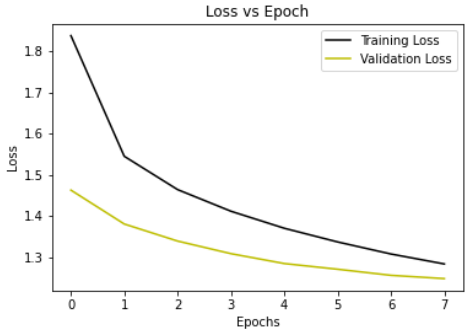


(a) Python

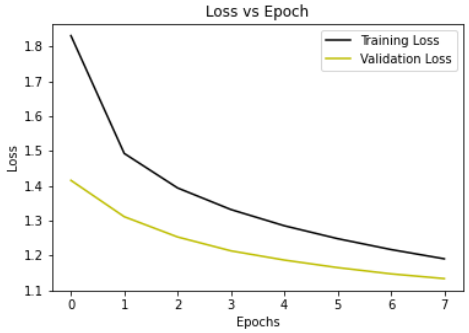


(b) SQL

Figure 8. Training and Validation Losses for PLBART



(a) Python



(b) SQL

Figure 9. Training and Validation Losses for CodeT5

and separate identifiers such as comments from code. There are several variations of CodeT5 models available on huggingface which have been fine-tuned for various specific tasks. For the purposes of our experiment, we use the base model of CodeT5 for conditional generation for our purposes of code generation.

Training the CodeT5 model proved to be simpler than the rest of the transformer models. CodeT5 also uses RoBERTa tokenizer, hence the initial process of generating source and target IDs from the unprocessed NL and PL inputs are the same as that for CodeBERT or CodeGPT. Since the overall process is the same as the way we trained PLBART, a figure for depicting the dataflow is not being shown. The source and target masks are calculated from the source and input IDs such that the values corresponding to non-zero IDs are 1 and the padded IDs are 0. The source IDs, source mask, token IDs and token mask were used as input to the CodeT5 model and the from the outputs for the model, only the loss was collected. The loss was then minimized using Adam optimizer with a learning rate of 0.00005. The model was trained for 8 epochs and the training and validation losses were plotted and shown in Fig 8. From Fig 5(a) and (b) it is observed that both the training and validation losses keep decreasing steadily even at the 8th epoch. This means that the model has not finished training and ideally the model should have been trained for 3 to 4 more epochs for the model to converge. However, due to computation restrictions, such a thing was not possible. The steadily decreasing losses are, however, a good thing as it means that the model can learn even better if trained further. The gap between training and validation loss also steadily decreases and therefore we chose the model saved at the 8th epoch as the best model. It is also observed that there is again, like in the case of CodeBERT, a remarkable similarity between the loss values for both python and SQL languages. Our hypothesis that the model weights and parameters in both cases of python and SQL being the same results in such a graph is still the only explanation that can be offered by us. However, the losses in the case of SQL seems to have decreased comparatively more than in the case of python.

5 Results and Discussion

Following fine-tuning, each (pre) trained model is capable of synthesizing code as an output given a question in NL. As mentioned before, we tested each of the four models, PLBART, CodeT5, CodeGPT and CodeBERT, on 15% of the

Table 1. CodeBLEU scores

	PLBART	CodeBERT	CodeT5	CodeGPT
Python	13.523	16.464	18.504	13.959
SQL	11.762	17.316	15.407	8.568

filtered StaQC dataset. Evaluation of each model was a two-part procedure: 1) creating the hypotheses and references, 2) calculating the CodeBLEU score of each model. To begin with, following the successful training of each model, to convergence if possible, the epoch with lowest validation loss (before surpassing training loss) was chosen as the candidate to represent the model during testing. The specifics of the chosen models were discussed in the Models sections. Given the selected models, we tested each model on a test set for 1 epoch (=6000 steps). The sequences that resulted from testing each model were the hypotheses, while the original expected output were the references. The only preprocessing that was required for the calculation of CodeBLEU is to list the items (hypothesis or reference) of each instance on a single line. By feeding the hypothesis and reference to CodeXGLUE's custom CodeBLEU calculator, we were able to evaluate the model at a quantitative level. From an overview, it is evident that the results differ at a small level, for the most part, and thus are not concrete. However, doing a relative comparison, CodeBLEU substantiates that CodeBERT and CodeT5 models perform better than CodeGPT and PLBART by nearly 3%. Even at a whole, the results are poor, and it can most likely be attributed to a lack of further training. As explained earlier, the models were trained for between 5 to 10 epochs and gave the appearence of convergence. However, real convergence may happen after the 10 epochs. Although, no supported conclusions can be made about which model is best for downstream applications, we can confidently say CodeBERT and CodeT5 should be explored for further fine-tuning.

5.1 Qualitative Analysis

We attempt to compare one of the best models with one of the worst models in our experiment in an effort to see how different the results really are.

5.1.1 CodeBERT. An example of generated python code from the trained CodeBERT is given below:

```
import import >>> In from def # class for df
```

The original code was:

```
import numpy as np
import matplotlib.pyplot as plt
fig = plt.figure()
ax = fig.add_subplot(111)
ax.quiver((0,0), (0,0), (1,0), (1,3),/
units = 'xy', scale = 1) plt.axis('equal')
```

From the above python code generated by CodeBERT, we can see that while CodeBERT has the terminology correct, it is not generating useful code or understanding the code logic. An example of SQL code generated by CodeBERT is:

```
SELECT SELECT select DEC CRE INS W decl UPDATE ;
```

The original code was:

```
select max(id), parent from tree group by parent;
```

For SQL code generated by CodeBERT we can see the same sort of results as the Python code. It is hypothesized that this is due to setting the maximum length to 150 instead of 512 as used in pre-training which did not let the model understand the semantics of the code and it only learnt the vocabulary.

5.1.2 CodeGPT. An example of python code generated by CodeGPT is given below:

```
import os with open
```

The original code was:

```
with open("File1.txt") as fin1:
lines = set(fin1.readlines())
with open("File2.txt") as fin2:
lines.update(set(fin2.readlines()))
with open("file3.txt", 'w') as fout:
```

From the above python code enerated we can see that CodeGPT is not generating long enough sequences to qualify as code. The original code was quite long and the generated code has absolutely no relationship with the original one. An example of SQL code generated by CodeGPT is given below:

```
;WITH CTE AS ( SELECT *, ROW_NUMBER() OVER
(PARTITION BY parent_id ORDER BY parent_id DESC)
```

The original code was:

```
select time, activities, count(*)
from table group by time, activities;
```

While the SQL code generated by CodeGPT has some semblance of code it is simply because it is using SQL tokens at random. This poor result for both Python and SQL is possibly because we have not used 1024 as the max sequence length as specified in their paper. However, in comparison, CodeBERT performs much better than CodeGPT overall.

6 Obstacles

Our original proposal for the Deep Learning Final Project was "Pseudocode as an Intermediary Step for Code Generation." However, we have deviated very far from where we started, but we had just reason in doing so. Our initial proposal entailed using an NL-PL paired dataset, and then utilizing the PL to generate its accompanying pseudocode using a statistical machine PseudoGen to build our project dataset.[7] The PseudoGen documentation had detailed that they found good success in generating pseudocode for python by training on Django. The tutorials seemed promising as well, so

we began to build our models – PLBART and CodeBert - and dataset in parallel; however, once were finally able to get Pseudogen running on Docker, we came to find that the program was too fine-tuned to Django terminology. For example, it was treating an array of tuples as an array of method calls. By training with on a complicated platform, it had failed to fully learn the smaller details. We would have excused this if it was single case, but our dataset contained many such arrays and other similar structures that confused the statistical machine. An example of this is provided below. The original code was to add 1 to a number:

```
def f_gold (x):
 m = 1 ;
 while (x&m):
  x = x ^ m
  m <<= 1
  x = x ^ m
 return x
#TOFILL

if __name__ == '__main__':
 param = [
 (96,),
 (66,),
 (67,),
 (13,),
 (75,),
 (78,),
 (1,),
 (83,),
 (27,),
 (65,)
    ]
 n_success = 0
 for i, parameters_set in enumerate(param):
  if f_filled(*parameters_set)==
  /f_gold(*parameters_set):
    n_success+=1
 print("#Results:%i,%i"%(n_success,len(param)))
```

However, the corresponding pseudocode generated was:

```
define the method f_gold r .
return r r integer 2 ,return boolean True,
otherwise return boolean False
if __name__ equals a string `__main__`
call the method 14 ,
call the method 78 ,
call the method 45 ,
call the method 66 ,
call the method err_log ,
call the integer 32 , respectively .
call the method by integer 60 ,
call the method hexadecimal ,
call the method 99 ,
call the method 65 ,
n_success is integer 0 .
for every i and param in
/enumerated iterable parameters_set,
unpacked list parameters_set f_filled
f_gold parameters_set
and , if it evaluates to true,
increment n_success by integer 1,
print a #Results:-SP-%i,-SP-%i with an argument
string `% s : % s`
formated with param ,
substitute the result for n_success,respectively.
```

A few of the indentations for the codes and pseudocodes have been changed to not violate format restrictions. However, despite that it can be observed that the generated pseudocode is extremely faulty when it comes to entering values to functions.

As a result, hoping to keep within the scope of pseudocode, we chose to abandon the NL-Pseudocode encoder and simply work on Pseudocode to PL translation, as it still remained with the category of code synthesis. This was still possible despite PseudoGen not working, because we had found a manually built pseudocode to NL dataset in both C++ and Python () which although technically sound, was difficult to justify the use-case for the project. Our original proposal relied on pseudocode to improve the NL-PL translation task, but there was no valid software development task or purpose other than for a future addition of an NL-Pseudocode encoder to generate pseudocode. However, when detailed code was already written, generating pseudocode seemed to be less useful. While still keeping within the theme of code synthesis, such that we could continue building on the PLBART and CodeBert models, our final deep learning project topic changed one more time to what is presented in this paper. Once we received approval from the course staff, we proceeded with executing our new proposal.

7 Conclusion

In conclusion, we can say that out of the four models CodeBERT performed the best for SQL programming language with a CodeBLEU [**?**] score of 16.46, and CodeT5 performed the best for Python programming language with a CodeBLEU score of 18.50. With enough computation resources and time, the CodeBLEU scores for both these models can be improved to reach the benchmark scores.

References

[1] Pre-trained models. URL: https://huggingface.co/transformers/v3.3.1/pretrained_models.html.

[2] Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. Unified pre-training for program understanding and generation. Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, pages 2655–2668. Association for Computational Linguistics. URL: https://aclanthology.org/2021.naacl-main.211https://doi.org/10.18653/v1/2021.naacl-main.211, doi:10.18653/v1/2021.naacl-main.211.

[3] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. Unified pre-training for program understanding and generation, 2021. doi:10.48550/ARXIV.2103.06333.

[4] Rudy Bunel, Jacob Devlin, Matthew Hausknecht, Rishabh Singh, and Pushmeet Kohli. Leveraging grammar and reinforcement learning for neural program synthesis, 2018. URL:

https://www.microsoft.com/en-us/research/publication/leveraging-grammar-reinforcement-learning-neural-program-synthesis.

[5] 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. doi:10.48550/ARXIV.2107.03374.

[6] 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. Findings of the Association for Computational Linguistics: EMNLP 2020, pages 1536–1547. Association for Computational Linguistics. URL: https://aclanthology.org/2020.findings-emnlp.139https://doi.org/10.18653/v1/2020.findings-emnlp.139, doi:10.18653/v1/2020.findings-emnlp.139.

[7] Hiroyuki Fudaba, Yusuke Oda, Koichi Akabe, Graham Neubig, Hideaki Hata, Sakriani Sakti, Tomoki Toda, and Satoshi Nakamura. Pseudogen: A tool to automatically generate pseudo-code from source code, 2015. doi:10.1109/ASE.2015.107.

[8] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. Codesearchnet challenge: Evaluating the state of semantic code search, 2019. URL: https://arxiv.org/abs/1909.09436, doi:10.48550/ARXIV.1909.09436.

[9] Taku Kudo and John Richardson. Sentencepiece: A simple and language independent subword tokenizer and detokenizer for neural text processing. Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing: System Demonstrations, pages 66–71. Association for Computational Linguistics. URL: https://aclanthology.org/D18-2012https://doi.org/10.18653/v1/D18-2012, doi:10.18653/v1/D18-2012.

[10] Triet H. M. Le, Hao Chen, and Muhammad Ali Babar. Deep learning for source code modeling and generation: Models, applications, and challenges. *ACM Comput. Surv.*, 53(3):Article 62, 2020. doi:10.1145/3383458.

[11] 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. URL: https://arxiv.org/abs/2102.04664, doi:10.48550/ARXIV.2102.04664.

[12] Alexei Baevski Angela Fan Sam Gross Nathan Ng David Grangier Michael Auli Myle Ott, Sergey Edunov. fairseq: A fast, extensible toolkit for sequence modeling, 2019. URL: https://aclanthology.org/N19-4009, doi:10.18653/v1/N19-4009.

[13] 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, 2019. doi:10.48550/ARXIV.1910.10683.

[14] Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. Codebleu: a method for automatic evaluation of code synthesis, 2020. URL: https://arxiv.org/abs/2009.10297, doi:10.48550/ARXIV.2009.10297.

[15] Z. Xing Y. Ma W. Song S. Gao, C. Chen and S. W. Lin. A neural model for method name generation from functional description, 2019. doi:10.1109/SANER.2019.8667994.

[16] Changhan Wang, Kyunghyun Cho, and Jiatao Gu. Neural machine translation with byte-level subwords, 2021. doi:10.48550/ARXIV.1909.03341.

[17] 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. URL: https://arxiv.org/abs/2109.00859, doi:10.48550/ARXIV.2109.00859.

[18] Ziyu Yao, Daniel S. Weld, Wei-Peng Chen, and Huan Sun. Staqc: A systematically mined question-code dataset from stack overflow, 2018. doi:10.1145/3178876.3186081.