# Program Code Generation

**Abhishek Kumar**
Carnegie Mellon University
ak5@cs.cmu.edu

**Deep Karkhanis**
Carnegie Mellon University
dkarkhan@cs.cmu.edu

**Nilay Pande**
Carnegie Mellon University
nmpande@cs.cmu.edu

## Abstract

Natural Language Generation has taken significant strides in the last couple of years producing almost human like outputs in many tasks (Brown et al., 2020). While the human language is inherently ambiguous and inconsistent, Program code follows strict grammar rules that are consistent and not ambiguous. We feel that it should be possible to generate program code which is almost similar to the ones written by developers in many tasks. In this project, we survey some of the available techniques for the same and try to understand some of their shortcomings. Then we try to improve the state of the art model using ideas from the error analysis that we did in the assignment 3. We were able to **improve the BLEU score of the state of the art model by 0.9**. We also point out notable issues with our methods on the CoNaLa dataset (Python) and make a proposal on how to improve these models and fix these issues in future.

Our code is present at https://github.com/Nilay017/nlp-asgn4

## 1 Introduction

In the last couple of years, there has been significant motivation in generating source code automatically using NLP. It promises to increase developer productivity, reduce the cost of building commonly used web applications. Since the program code follows a strict grammar and has consistent semantics, semantic parsing has been used to automatically generate source code (Xu et al., 2020) (Yin and Neubig, 2017). These models achieve good BLEU score but often fail to provide a working solution. Also, they rely on task-specific human expert annotated data which makes the models complicated and hard to transfer to new domains. On the other side of the spectrum, (Norouzi et al., 2021) tried to apply transformer based Seq2Seq model on the automatic code generation problem with little inductive bias about the program code. This class of models have been extremely successful in a variety of NLP tasks and here as well they achieved state-of-the-art results on CoNaLa and Django datasets. However, when we investigated the code samples generated by their work, we realized that most of the generated code isn't achieving the desired functionality.

In this work, first we do a comprehensive literature survey of the available solutions to generate program code. Then we do an analysis of errors given by current SOTA model. Based on insights gained from the error analysis, we try to improve the state of the art model using some heuristics. We also suggest some future steps on one can try to improve the current model and fix some of the errors given by the current SOTA model and our model.

## 2 Literature Survey

Although automated parsing of programs with the use of compilers and interpreters has been successful for a while, automated generation of code itself has been of interest for a long time in the field of computational linguistics. The boom of successful Deep Learning models over the decade, has led to significant advances in this field of program learning, specifically in the form of neural program learning. (Chen et al., 2021) proposed Codex and its variants, whose descendants power GitHub Copilot and are part of models in the OpenAI API. They categorized literature for neural program learning into two approaches, program induction and program synthesis.

### 2.1 Program Induction

Program induction methods use an already available latent program representation. The models generate program outputs directly from this representation. Initial ideas showed that execution of basic tasks like memorization and simple math like

addition can be achieved by these models (Zaremba and Sutskever, 2015). Subsequent approaches in program induction used inductive biases based on modern computing devices like the Neural Turing Machine (Graves et al., 2014), the Neural GPU (Łukasz Kaiser and Sutskever, 2016), memory networks (Weston et al., 2015; Sukhbaatar et al., 2015), and the differentiable neural computer (Graves et al., 2016). Recent approaches emphasized that recurrence is a useful component in program induction like the Universal Transformer (Dehghani et al., 2019) and Neural Program Interpreter (Reed and de Freitas, 2016; Shin et al., 2018; Pierrot et al., 2021).

The inductive bias approach has also been commonly used in modern semantic parsing literature (Dong and Lapata, 2016; Yin and Neubig, 2017, 2018; Dong and Lapata, 2018; Guo et al., 2019; Wang et al., 2021; Yin and Neubig, 2019) which can be attributed to the fact that semantic parsing is often regarded as a precursor to code generation.

### 2.2 Program Synthesis

Some approaches perform the synthesis of programs without using an intermediate AST representation. (Hindle et al., 2012) used n-gram models of code and pointed out that code is more predictable than natural language. Later, (Ling et al., 2016) found that character-level language models (LM) can be used to generate code for the popular game - Magic The Gathering in an online arena when supplemented with a latent mode that allows card attributes to be copied into code. (Balog et al., 2017) gave DeepCoder, a model which can be used to guide program search by predicting functions that appear in source code.

### 2.3 Large Language Models

Program synthesis by the use of large scale Transformers has showing promising results in the recent years. The success of large language models (Devlin et al., 2019; Liu et al., 2019; Radford et al., 2019; Brown et al., 2020) led to advances like Code-BERT (Feng et al., 2020) which trained the BERT objective on docstrings paired with functions, and achieved strong results on code search. PyMT5 (Clement et al., 2020), used the T5 objective to train a model which can translate amongst non-overlapping subsets of body, signature and docstring.

### 2.4 Benchmarking Techniques

Initial techniques on code-generation used two domain-specific datasets, FlashFill (Gulwani, 2011; Gulwani et al., 2012) and Hearthstone (Ling et al., 2016) to benchmark neural programming systems. Recently, much broader and harder datasets are being used. (Barone and Sennrich, 2017) put forward a Python dataset while (Husain et al., 2020) built a larger corpus with data from multiple programming languages as part of the CodeSearchNet challenge. Both these datasets were scraped from GitHub. Much more recent is the CodeBLEU metric proposed by (Ren et al., 2020) which was later used by CodeXGLUE (Lu et al., 2021) to aggregate several programming benchmarks.

#### 2.4.1 Functional Correctness

As one might expect, since BLEU score is primarily a textual correctness score, there has been a lot of research to find better metrics to evaluate code-generation models. APPS (Hendrycks et al., 2021) is a functional correctness benchmark based on problems from Codeforces. (Chen et al., 2021) also uses functional correctness for benchmarking in the form of a *pass@k* metric. They established that re-sampling from the GPT based model gives notable improvements. SPoC (Kulal et al., 2019) also used a similar metric to evaluate generation of functionally correct code from pseudocode with a limit on compilations. TransCoder (Roziere et al., 2020) used unsupervised learning to translate between programming languages while observing that functional correctness better capture the strength of their model than BLEU score. ContraCode (Jain et al., 2021) trained a contrastive code model from a large set of functionally correct programs to get improvements on tasks like inference. RobustFill (Devlin et al., 2017) also uses functional correctness metrics while generating multiple samples of code via beam search.

(Chen et al., 2021) evaluates functional correctness on a specially curated HumanEval dataset. It is a set of 164 handwritten programming problems. Each problem includes a function signature, docstring, body, and several unit tests, with $\approx 7.7$ tests per problem. Programming tasks in the HumanEval dataset assess language comprehension, reasoning, algorithms, and simple mathematics. The dataset can be found at `https://www.github.com/openai/human-eval`

### 2.5 Going past docstring code-generation

Automating the task of coding is more than just generating programs from docstrings. (Tufano et al., 2021) use Transformers to generate unit tests for code which outperformed commercial counterparts. (Zhou et al., 2021) built an internal auto-complete tool for Facebook and showed training on accepted user completions improves performance. When it comes to locating and fixing bugs, there have been multiple techniques such as static or dynamic code analysis (Agrawal et al., 1995; Korel and Rilling, 1997), learning association rules (Jeffrey et al., 2009) and genetic programming (Le Goues et al., 2012). Recently, (Tufano et al., 2019; Drain et al., 2021) used neural translation techniques to convert buggy code to correct programs. They use exact match instead of functional correctness. This is because of (Qi et al., 2015)'s observation that most solutions by genetic search proposed in (Le Goues et al., 2012) passed weak test suites by deleting functionality that failed.

### 2.6 TranX (Yin and Neubig, 2018)

TranX proposed by (Yin and Neubig, 2017, 2018) establishes a general purpose transition based system to ensure grammatical correctness of predictions. TranX maps Natural Language (NL) instances to formal meaning representations (MRs). It uses a transition system which is dependent on the abstract syntax description language (ASDL) for the target MR. It also employs Abstract Syntax Trees (AST) as general-purpose intermediate semantic representations for MRs. The task dependent grammar is provided to the system as external knowledge to guide the semantic parsing (SP) process. TranX also takes as input a user-defined function, AST_to_MR(·), to convert the intermediate AST into a domain-specific meaning representation. The transition system decomposes the generation procedure of an AST into a sequence of tree-constructing actions. The probability distribution for picking these actions one step at a time is parameterized by a neural encoder-decoder network with augmented recurrent connections to reflect the topology of ASTs. Yin and Neubig showed that TranX was:

- **Generalizable**: Since it separates the semantic parsing from specificities of the grammars by using ASTs as intermediate representations and incorporating task-dependent grammar as external knowledge
- **Extensible**: The transition system which converts NL to AST is simple and requires minimal engineering to extend to other domain specific tasks
- **Effective**: TranX showed better results than the then existing standards in both code generation (Django dataset) and semantic parsing tasks (GEO and ATIS datasets) while giving competitive results on WikiSQL dataset

For the WikiSQL dataset, Yin and Neubig showed that with minimal engineering, one could incorporate column names into the transition names to boost accuracies. They also gave an answer pruning strategy (removing queries which return empty result) that used minimal extra information about input tables during inference to outperform existing models on WikiSQL.

### 2.7 Incorporating External Knowledge through Pre-training (Xu et al., 2020)

One of the major bottlenecks in generating accurate code generation models is the limited size of the human annotated code examples. This paper addresses this problem by incorporating external knowledge sources such as API documentation and code snippets mined automatically from StackOverflow. The paper follows a two step process for training the model - 1) Train using a baseline model using NL-code pairs mined from external resources. 2) Finetune the model on small human annotated dataset such as CoNaLa. They use the NL-to-code generation model TranX (Yin and Neubig, 2018) with hypothesis reranking (Yin and Neubig, 2019) as the base model for the framework proposed by the paper.

#### 2.7.1 StackOverFlow

StackOverFlow is used as the first source of external knowledge. External classifier is used to decide whether a NL-code pair is valid or not. The probability assigned by the classifier is used as a confidence level to determine the quality of the NL-code pair.

#### 2.7.2 API Documentation

Since API documentation tends to be very verbose, the paper uses some heuristics to transform these examples into more appropriate examples for the model to train on. Such as - 1)Permute all possible optional arguments and append them to required arguments. 2) Create heuristic variables based on

classes to store and call class object methods. 3) Only keep certain sentences such as the first sentence in the documentation or the first sentence which mentions the argument as intent text for code generation.

#### 2.7.3 Re-sampling from API Documentation

To address the distribution shift between questions asked by the developers and the entries in the API Documentation, the paper proposes a retrieval-based re-sampling method to mitigate this problem. Specifically, for each entry in the human-annotated+mined dataset, they retrieve top k entries from the API dataset. Then for each entry in the API dataset, they compute its retrieval frequency and sample using probabilities proportional to this frequency, smoothed by some temperature parameter T.

### 2.8 Using Less Prior and More Monolingual Data (Norouzi et al., 2021)

(Norouzi et al., 2021) have established the current state of the art results by producing 81.03% exact match accuracy on Django and 32.57 BLEU score on CoNaLa datasets. They showed that it is possible to achieve competitive performance on code-generation tasks with very little inductive bias in the model, and without using additional labelled data. They used a generic transformer-based seq2seq model with minimal code-generation-specific inductive bias design. In addition, a monolingual corpus of the target programming language was used for training, which although large was cheap to mine from the web. Regarding the importance of structure of the model, they inferred that:

- On the input side the lack of inductive bias was mainly compensated by the BERT (Devlin et al., 2019) class of pre-trained models
- On the output side, the prior knowledge about the target meaning representation was learned from the unlabelled monolingual data

Since the monolingual data does not have natural language utterances, they add an auto-encoding objective term based on the monolingual data to the overall log likelihood function they want to optimize. The monolingual data in the target language is reconstructed using a shared encoder-decoder model. They also showed that monolingual data auto-encoding mainly helps the decoder (and may even hurt the encoder), and hence they freeze the encoder parameters for this data. (Norouzi et al., 2021) termed this method of using copied monolingual data and freezing the encoder over them as target autoencoding (TAE). Apart from getting state of the art on Django and CoNaLa, TAE also showed improvements on datasets in other programming languages like GeoQuery (SQL), ATIS (SQL) and Magic (Java). (Norouzi et al., 2021) also showed that in the decoder, TAE specifically improves upon the token copy and generation sections by showing the increasing trend of copy accuracy and generation accuracy on the Django test set.

## 3 Our Methods

### 3.1 Augmenting Data with Python API Documentation

Some of the main failure cases in the current SOTA (Norouzi et al., 2021) stem from incorrect usage of system calls and API function calls. Python API documentation provides a much cleaner and exact source of correct API/system call usage than StackOverFlow mined pairs. Hence as a first experiment, we did a simple augmentation of the CoNaLa dataset with NL-code pairs from Python API documentation during training. We got following interesting observations - a) The BLEU score didn't decline much (32.09) b) This new model performed well on some examples that use Python Standard Library. c) It performed really bad on some examples by using library functions even if they are not required. Examples shown in Fig 1

#### 3.1.1 Beyond Python Standard Library

While (Xu et al., 2020) restricts the external knowledge from API documentation to Python Standard library, we used other popular libraries - Numpy, Pandas, PyTorch to extract NL-code pairs for training. We observed that now the model performed well on snippets including numpy and pandas, however, it was using some library calls even when it was not required.

#### 3.1.2 Re-sampling from API

Akin to (Xu et al., 2020), we employed the same retrieval based re-sampling method to reduce the distribution shift between real questions and API documentation entries. We use the BM25 retrieval algorithm (Jones et al., 2000) (Sparck Jones et al., 2000) implemented in ElasticSearch to first index the data in ElasticSearch and then resample from it to assign probability scores to each example

in the Python Documentation. This probability score represnts the chances of some code snippet to appear in the train/test data. After this step, we were able to improve the BLEU score of the model. However, we still saw some over-usage of library calls as depicted in Fig 2

#### 3.1.3 Probability Pruning and Intent Length Constraint

By looking at the probabilities generated in the previous step, we realized that some library function calls were getting very high probability scores as their NL intent and the signature were very common/generic. Hence, the model was overusing them. For example *str()* and *list()* calls got very high probability scores. Therefore, we hypothesized that we should employ probability pruning i.e should not allow some examples to get absurdly high probability scores. We put a constraint that any probability score $> 0.01$ will be pruned down to $0.01$ and then re-normalized all the probability scores.

Also, during the error analysis in assignment 3, we observed the state of the art model (Norouzi et al., 2021) is not able to handle complex and long NL input for functionally correct code generation. Examples with very long intent might even worsen the BLEU score as the model starts associating words at the tail end of the intent (which might not be strongly correlated with the code snippet and potentially end up confusing the model) with the program code. This problem was especially true in documentation examples where some of the intents were too long and had irrelevant words towards the tail end of the intent. CoNaLa dataset on the other hand didn't have as many long intent examples as the API documentation did. We hypothesized that constraining the length of such long NL input examples for documentation in training might significantly improve performance on test examples with small NL input while potentially only slightly worsening performance on long NL input test examples. We found out that the average number of characters in the intent of the Python documentation was around 131 and with standard deviation of 86. We tried to limit the number of characters to several values and found that a constraint of 120 gave optimal result. With probability pruning and intent length constraint on Python documentation, **we were able to improve the BLEU score of the state of the art model by 0.9.**

#### 3.1.4 Library Name as Hint

We felt that if we give the model the name of the library to be used for a task as a hint during training, it may train to better utilize the library api calls. We employed the techniques described in the previous section and appended the name of the library (e.g. Numpy, Pandas, PyTorch) in the intent section of the dataset. After training, we found out that the BLEU score actually decreased to 32.76 with not much change in the quality of generated code snippets (via manual inspection). Hence, this idea didn't add much to the performance of the model. We suspect that model was already inferring the library correctly and adding the library name made the model more conservative in generating library function calls.

### 3.2 Modified Evaluation scheme with Re-ranking

As observed in Codex (Chen et al., 2021), models with high BLEU score still fail miserably on datasets such as **HumanEval** which tests the functional correctness of the generated code output. Since, the final aim of program code generation is generating *correct* code for given Natural Language input, we evaluated our best model and the state of the art model on the HumanEval dataset and Python Interpreter outputs.

#### 3.2.1 Sampling and Re-ranking with HumanEval

RobustFill (Devlin et al., 2017) observed that the best way to find a program consistent with input examples was to synthesize multiple samples through beam search. Similarly, during test time, Codex (Chen et al., 2021) generates 100 samples per problem, re-ranks them using evaluation on unit test cases and returns the top k samples. Pass@k is used as the final metric for evaluation. Inspired by these approaches, we sampled a fixed N number of most likely samples during decoding, and then re-ranked them based on performance on unit test cases to return the top k samples which will were then used for computing Pass@k metric.

## 4 Experiments

**Dataset and Metric**: We choose CoNaLa (Yin and Neubig, 2018) as the dataset (2,179 training, 200 dev and 500 test samples). It covers English queries given as intent and corresponding expected Python snippet. We use the same evaluation metric

| Method | BLEU |
|---|---|
| SOTA Baseline | 33.41 |
| SOTA + Python Std Lib + Numpy + Pandas + Torch | 32.09 |
| SOTA + Python Std Lib + Numpy + Pandas + Torch + Resample | 33.3 |
| **SOTA + Python Std Lib + Numpy + Pandas + Torch + Resample + Prob Prune + Len Constrain** | **34.31** |
| SOTA + Python Std Lib + Numpy + Pandas + Torch + Resample + Prob Prune + Len Constrain + Libname | 32.76 |

Table 1: Proposed Methods and their BLEU Scores

```
>> Examples that depict benefit of API Docs
# intent: unzip the list str0
'''Expected'''  zip(*[ str0 ])
'''SOTA''' sorted ( [ str0 ] )
'''SOTA + Python Standard lib''' zip(*[ str0 ])

# intent: download a file str0 over http and save to str1
'''Expected''' urllib.request.urlretrieve('str0', 'str1')
'''SOTA''' shutil. savefig ('str0 ','str1')
'''SOTA + Python Standard lib''' urllib. request. urlretrieve ('str0 ','str1')

>> Bad Examples
# intent: how to erase the file contents of text file in python
'''Expected''' open('file.txt', 'w').close()
'''SOTA''' webbrowser.open('file://my_file.txt','ignore')
'''SOTA + Python Standard lib''' thread.allocate_lock()

# intent sort dict var0 by value
'''Expected''' sorted(var0, key=var0.get)
'''SOTA''' sorted(list(var0.items()), key=lambda x : x[1][0])
'''SOTA + Python Standard lib''' zipfile. infolist ( )
```

Figure 1: Examples SOTA vs SOTA + Python Standard API DOCS

```
>> Using numpy library even when not required
# intent: multiply a matrix var0 with a 3d tensor var1 in scipy
'''Expected''' scipy.tensordot(var0, var1, axes=[1, 1]).swapaxes(0, 1)
'''With Resample''' numpy.dot(numpy.dot(var0.t, var1), var0)
```

Figure 2: Overusage of API

```
>> Example that benefits from API Docs + Resample
# intent: create new matrix object by concatenating data from matrix a and matrix b
'''Expected''' np.concatenate((a, b))
'''SOTA''' numpy. concatenate ( )
'''Our Model''' numpy. concatenate ( a, b )
```

Figure 3: Correct Usage of API

```
# intent: get a new list var0 by removing empty list from a list of lists var1
'''Expected''' var0 = [x for x in var1 if x != []]
'''Without Prob prune and Length constraint''' var1 = [[ x for x in var0 if x] for y in var1]
'''With Prob prune and Length constraint''' var0 = list([ x for x in var1 if x])
```

Figure 4: Better performance with Probability Pruning and Length Constraint

```
# intent: python save list var0 to file object str0
'''Expected''': pickle.dump(var0, open('str0', 'wb'))
'''Generated''': var0.savefig('str0')

# intent: encode a string str0 to var0 encoding
'''Expected''': encoded = 'str0'.encode('var0')
'''Generated''': print(var0.encode('str0'))

# intent: how do i insert into t1 (select * from t2) in sqlalchemy?
'''Expected''': session.execute('insert into t1 (select * from t2)')
'''Generated''': db.select('insert into t1 (t2) values (% s, %s)')
```

Figure 5: Existing Issues with functional correctness

```
# intent: create list of dictionaries from pandas dataframe var0
'''Expected''' var0.to_dict('records')
'''SOTA''' var0 = [{'d': i.dataframe() for i in range (100)] for d in var0]
'''SOTA + Simple API Augmentation'''  zipfile.zip_deflated
'''SOTA + API + Resampling + Pruning + Len constraint''' var0.to_dict('records')
```

Figure 6: Comparing Different Models' output

as (Xu et al., 2020), corpus-level BLEU calculated on target code outputs in test set.

**Mined Pairs**: We use the CoNaLa-Mined (Yin and Neubig, 2018) dataset of 600K NL-code pairs in Python automatically mined from StackOverflow.

**API Documentation Pairs**: We parsed all the module documentation including libraries, builtin types and functions included in the Python 3.1, Numpy, Pandas, and PyTorch. After pre-processing, we create about 42,612 distinct NL-code pairs (without resampling) from Python API documentation.

### 4.1 Our methods to improve SOTA

We picked up the current state-of-the-art (Norouzi et al., 2021) as the base model. First, we augment the training data with the mined Python standard library documentation, Numpy, Pandas, and PyTorch documentation. We use the same setup as (Norouzi et al., 2021), i.e. Adam for optimization. We train for 80 epochs or early stopping based on the evaluation metric on dev set. In this, we were able to achieve a BLEU score of 32.09.

Next, we use the retrieval based resampling technique from (Xu et al., 2020) to match the distribution of the api documentation with the training set in CoNaLa. We experiment with k = 1, 3 and $\tau$ = 2, 4 in re-sampling, and find that k = 1 and $\tau$ = 2 perform the best as reported by (Xu et al., 2020). Using this, we were able to achieve a BLEU score of 33.3. We found out that some library calls had improved, but we saw an over-usage of library calls.

Next, we tried to prune the probability of some most frequently functions like $list()$ and $str()$ which were being overused. We put a limit of $0.01$ and any probability higher than this value was pruned to $0.01$. Next, we imposed a constraint on the length of intent as described in section 3.1.3. We tried values 100, 120, 150 and we got best results with 120. Using this model, we were able to achieve a BLEU score of 34.31, which is 0.9 more than the state of the art.

Finally, we tried to add the name of library in the intent during training. The motivation was that it will help the model to learn which library's API to call. By this method, we got a BLEU score of 32.76 indicating that the model didn't really benefit from this idea.

In all the above methods, we try both greedy search and beam search and best results are obtained in beam search in all the above methods

### 4.2 Sampling and Re-ranking with HumanEval dataset

As mentioned previously, HumanEval uses test cases as an approximation to check functional correctness of generated code. Using concepts from RobustFill (Devlin et al., 2017) and Codex (Lu et al., 2021), we used beam search on the BERT based transformer encoder-decoder model. We generate $N = 50$ most likely samples dur-

ing decoding. This was the largest value of $N$ possible on the AWS p2.xlarge instance based on CUDA memory constraints. We then re-rank these samples based on their performance on test cases. We then return the top k samples for computing the pass@k metrics. We used values of $k = 1, 2, 5, 10, 20, 30, 40, 50$. Alarmingly, both our new model and the SOTA model gave **0% accuracy on all pass@k metrics for HumanEval dataset**.

#### 4.2.1 Qualitative Improvements on HumanEval

The HumanEval problems are much harder than what the models have been trained on and the output is often complete gibberish on such hard NL tasks. In fact, it seems inference on the actual text is ignored and the functional signature present in the NL dominates code generation (see 5.4 for details). Although neither of the models can handle these complex test cases, our best model does give much more reasonable output (see 7) and in fact gives correct output on simple test cases (see 7) like the one given as sample in the HumanEval documentation.

## 5 Model Interpretation and Error Analysis

We manually inspect the generated samples for each experiment and perform a qualitative analysis of the wins and losses seen on the CoNaLa test set. This error analysis helps us gain intuition behind the quantitative numbers and gives us ideas for future improvements in our model.

### 5.1 Simple augmentation of API Documentation

Simple augmentation of API Documentation does lead to some wins wherein the model is able to call the correct function of the library with correct arguments which the current SOTA is not able to do. Specifically, model correctly associates the word 'unzip' in the intent with zip function of standard library and the word 'download'/'http' with urllib library. These wins were only possible because the model now has the underlying API knowledge of the library.

However, generated samples for simple augmentation of API Documentation also clearly show over training on the documentation. As seen in Figure 1, our model now unnecessarily generates unrelated/obscure library function calls for the simple NL intent. Model incorrectly correlates the words 'list' and 'open' with zipfile and thread library functions.

The primary reason behind these bad examples is the distribution shift between examples generated from API documentation (seen only at train time) and from the CoNaLa dataset (seen at both train and test time) as explained before. Simple augmentation of CoNaLa train data with API documentation treats both the sources of data equally and leads to the model over-training on API documentation.

### 5.2 API Documentation + Resampling

To address the distribution shift between NL-pairs from API documentation and CoNaLa, we employ retrieval based sampling as described in (Xu et al., 2020).

While many of the previous bad examples are no longer present, the model still overuses the API for popular keywords such as 'list' and 'str' indicating that Resampling still assigns too high a probability to certain API functions. We also observe the model is unable to handle long length NL intent during training.

### 5.3 API Documentation + Resampling + Probability Pruning + Length Constraining

With probability pruning and constraining the length of NL input during training, we observe a significant decrease in bad examples and incorrect API calls. Pruning the probability by setting a max threshold made sure the model doesn't have a unreasonably high preference towards using functions from highly frequent snippets or APIs as seen in Figure 4. Also, removing long NL input examples from training did improve performance on test examples with small NL input while only slightly worsening performance on long NL input test examples. This lead to better overall performance which is evident from the best BLEU score in 1.

### 5.4 Testing on HumanEval

As can be observed from the NL intent in HumanEval and its correct code output (more appropriately called canonical solution), the problems are much harder than what the models have been trained on. The output is often complete gibberish (more details in 7) on such hard NL tasks. In fact, it seems inference on the actual text is ignored and the functional signature present in the NL dominates code generation. This can be seen in Fig 8

```
# intent: Define a function to return 1
'''SOTA''' def return 1 ( ) : return  return 1 return 1
'''Our model''' def return1 ( ) : return 1
```

Figure 7: Our Model succeeding on HumanEval sample test cases

```
# intent: Check if in given list of numbers, are any two numbers closer to each
#         other than given threshold.
'''Code Generated without function signature input''' sum ( i * j for i, j in zip ( b ) )
'''Code Generated with function signature input''' has def has _ elements ( list ( numbers,
numbers = {'float': float }, {'threshold': >
```

Figure 8: Function signature dominating the model's inference

where the output worsened when the function signature was removed from the NL intent. The model in fact tried to force-fit a single line output instead of a function. This indicates it did not infer the requirement of outputting a function, until including the signature make it clear. Note the model can only take function signature primarily as NL input since there is no explicit grammar in the model. Although neither our best model nor the SOTA model can handle these complex test cases, our best model does give much more reasonable output (see 7) and in fact gives correct output on simple test cases (see 7) like the one given as sample in the HumanEval documentation.

### 5.5 Current Issues

Although the model we have proposed beats the current SOTA baseline, it is far from ideal. As seen in figure 5, it is associating the function names directly to words in the intent and thus making incorrect function calls (like 'savefig' has been called to 'save' files).

## 6 Conclusion and Future Work

Although we do improve over the State Of The Art and make effective use of API documentation, the current model that we have obtained is far from being good and lacks in functional correctness. Functional correctness should be the aim of automatic program code generation. If incorrect, the generated code may in fact end up hampering developer productivity instead of improving it. As is clear from the error made by the SOTA model, the BLEU score is a very misleading metric in the code generation context. There is a need for a new metric that promotes correct code which achieves the desired functionality. Datasets such as HumanEval are good approximations for the same. We need many more datasets like HumanEval to facilitate meaningful learning and interpretation on code generation tasks, specifically for python.

### 6.1 Interpreter Correctness

Like in most datasets, when unit test cases are not available, it is still useful to check if the model at-least gives code which can be compiled or interpreted. We can finetune models based on fraction of code generation examples which are correctly interpreted. Post this, we can compute the fraction of code generation examples for each model for each type of python runtime error. This will give us further insight as to how models compare with each other on specific types of runtime errors.

### 6.2 Reinforcement Learning based finetuning

Since functional correctness of the model is much more important than BLEU score or text-matching. Python interpreter output and unit test case performance of generated code samples both can give us a good signal as to how close we are to functional correctness. During training we can sample N code examples for a given NL input and design a reward signal using number of testcases passed and/or interpreter correctness. Policy gradient methods such as Reinforce or Evolutionary methods as such as Cross Entropy Maximization (CEM) can then be used to finetune the model parameters. CEM can use the number of test cases passed by the N generated code examples and the fraction of code examples which can be correctly interpreted to compute a fitness score for the current model parameters.

## Acknowledgements

We thank Frank Xu for sharing useful information which helped finalize the background and motivation behind this project. We also thank Graham Neubig, Robert Frederking and all TAs associated with 11-711: Advanced Natural Language Processing (Fall 2021) at Carnegie Mellon University.

## References


H. Agrawal, J.R. Horgan, S. London, and W.E. Wong. 1995. Fault localization using execution slices and dataflow tests. In *Proceedings of Sixth International Symposium on Software Reliability Engineering. ISSRE'95*, pages 143–151.

Matej Balog, Alexander L. Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. 2017. Deepcoder: Learning to write programs.

Antonio Valerio Miceli Barone and Rico Sennrich. 2017. A parallel corpus of python functions and documentation strings for automated code documentation and code generation.

Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language models are few-shot learners.

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. 2021. Evaluating large language models trained on code.

Colin B. Clement, Dawn Drain, Jonathan Timcheck, Alexey Svyatkovskiy, and Neel Sundaresan. 2020. Pymt5: multi-mode translation of natural language and python code with transformers.

Mostafa Dehghani, Stephan Gouws, Oriol Vinyals, Jakob Uszkoreit, and Łukasz Kaiser. 2019. Universal transformers.

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

Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdel rahman Mohamed, and Pushmeet Kohli. 2017. Robustfill: Neural program learning under noisy i/o.

Li Dong and Mirella Lapata. 2016. Language to logical form with neural attention.

Li Dong and Mirella Lapata. 2018. Coarse-to-fine decoding for neural semantic parsing.

Dawn Drain, Chen Wu, Alexey Svyatkovskiy, and Neel Sundaresan. 2021. Generating bug-fixes using pretrained transformers. *Proceedings of the 5th ACM SIGPLAN International Symposium on Machine Programming*.

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

Alex Graves, Greg Wayne, and Ivo Danihelka. 2014. Neural turing machines.

Alex Graves, Greg Wayne, Malcolm Reynolds, Tim Harley, Ivo Danihelka, Agnieszka Grabska-Barwińska, Sergio Gómez Colmenarejo, Edward Grefenstette, Tiago Ramalho, John Agapiou, Adrià Puigdomènech Badia, Karl Moritz Hermann, Yori Zwols, Georg Ostrovski, Adam Cain, Helen King, Christopher Summerfield, Phil Blunsom, Koray Kavukcuoglu, and Demis Hassabis. 2016. Hybrid computing using a neural network with dynamic external memory. *Nature*, 538(7626):471–476.

Sumit Gulwani. 2011. Automating string processing in spreadsheets using input-output examples. In *PoPL'11, January 26-28, 2011, Austin, Texas, USA*.

Sumit Gulwani, William R. Harris, and Rishabh Singh. 2012. Spreadsheet data manipulation using examples. *Communications of the ACM*, 55(8):97–105.

Jiaqi Guo, Zecheng Zhan, Yan Gao, Yan Xiao, Jian-Guang Lou, Ting Liu, and Dongmei Zhang. 2019. Towards complex text-to-SQL in cross-domain database with intermediate representation. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 4524–4535, Florence, Italy. Association for Computational Linguistics.

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

Abram Hindle, Earl T. Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. 2012. On the naturalness of software. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, page 837–847. IEEE Press.

Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2020. Codesearchnet challenge: Evaluating the state of semantic code search.

Paras Jain, Ajay Jain, Tianjun Zhang, Pieter Abbeel, Joseph E. Gonzalez, and Ion Stoica. 2021. Contrastive code representation learning.

Dennis Jeffrey, Min Feng, Neelam Gupta, and Rajiv Gupta. 2009. Bugfix: A learning-based tool to assist developers in fixing bugs. *2009 IEEE 17th International Conference on Program Comprehension*, pages 70–79.

Bogdan Korel and Juergen Rilling. 1997. Application of dynamic slicing in program debugging. In *AADEBUG*.

Sumith Kulal, Panupong Pasupat, Kartik Chandra, Mina Lee, Oded Padon, Alex Aiken, and Percy Liang. 2019. Spoc: Search-based pseudocode to code.

Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. 2012. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *Proceedings - 34th International Conference on Software Engineering, ICSE 2012*, Proceedings - International Conference on Software Engineering, pages 3–13. 34th International Conference on Software Engineering, ICSE 2012 ; Conference date: 02-06-2012 Through 09-06-2012.

Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. 2012. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 3–13.

Wang Ling, Edward Grefenstette, Karl Moritz Hermann, Tomáš Kočiský, Andrew Senior, Fumin Wang, and Phil Blunsom. 2016. Latent predictor networks for code generation.

Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. Roberta: A robustly optimized bert pretraining approach.

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. 2021. Codexglue: A machine learning benchmark dataset for code understanding and generation.

Sajad Norouzi, Keyi Tang, and Yanshuai Cao. 2021. Code generation from natural language with less prior and more monolingual data.

Thomas Pierrot, Guillaume Ligner, Scott Reed, Olivier Sigaud, Nicolas Perrin, Alexandre Laterre, David Kas, Karim Beguir, and Nando de Freitas. 2021. Learning compositional neural programs with recursive tree search and planning.

Zichao Qi, Fan Long, Sara Achour, and Martin Rinard. 2015. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA 2015, page 24–36, New York, NY, USA. Association for Computing Machinery.

Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. Language models are unsupervised multitask learners.

Scott Reed and Nando de Freitas. 2016. Neural programmer-interpreters.

Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. 2020. Codebleu: a method for automatic evaluation of code synthesis.

Baptiste Roziere, Marie-Anne Lachaux, Lowik Chanussot, and Guillaume Lample. 2020. Unsupervised translation of programming languages. *Advances in Neural Information Processing Systems*, 33.

Eui Chul Shin, Illia Polosukhin, and Dawn Song. 2018. Improving neural program synthesis with inferred execution traces. In *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc.

K. Sparck Jones, S. Walker, and S.E. Robertson. 2000. A probabilistic model of information retrieval: development and comparative experiments: Part 1. *Information Processing Management*, 36(6):779–808.

Sainbayar Sukhbaatar, Arthur Szlam, Jason Weston, and Rob Fergus. 2015. End-to-end memory networks.

Michele Tufano, Jevgenija Pantiuchina, Cody Watson, Gabriele Bavota, and Denys Poshyvanyk. 2019. On learning meaningful code changes via neural machine translation.

Rosalia Tufano, Luca Pascarella, Michele Tufano, Denys Poshyvanyk, and Gabriele Bavota. 2021. Towards automating code review activities.

Bailin Wang, Richard Shin, Xiaodong Liu, Oleksandr Polozov, and Matthew Richardson. 2021. Rat-sql: Relation-aware schema encoding and linking for text-to-sql parsers.

Jason Weston, Sumit Chopra, and Antoine Bordes. 2015. Memory networks.

Frank F. Xu, Zhengbao Jiang, Pengcheng Yin, Bogdan Vasilescu, and Graham Neubig. 2020. Incorporating external knowledge through pre-training for natural language to code generation.

Pengcheng Yin and Graham Neubig. 2017. A syntactic neural model for general-purpose code generation.

Pengcheng Yin and Graham Neubig. 2018. Tranx: A transition-based neural abstract syntax parser for semantic parsing and code generation.

Pengcheng Yin and Graham Neubig. 2019. Reranking for neural semantic parsing. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 4553–4559, Florence, Italy. Association for Computational Linguistics.

Wojciech Zaremba and Ilya Sutskever. 2015. Learning to execute.

Wen Zhou, Seohyun Kim, Vijayaraghavan Murali, and Gareth Ari Aye. 2021. Improving code autocompletion with transfer learning.

Łukasz Kaiser and Ilya Sutskever. 2016. Neural gpus learn algorithms.

## 7 Appendix

**Example of complex test case in HumanEval**

- **Intent**: Imagine a road that's a perfectly straight infinitely long line. n cars are driving left to right; simultaneously, a different set of n cars are driving right to left. The two sets of cars start out being very far from each other. All cars move in the same speed. Two cars are said to collide when a car that's moving left to right hits a car that's moving right to left. However, the cars are infinitely sturdy and strong; as a result, they continue moving in their trajectory as if they did not collide. This function outputs the number of such collisions.

- **Code Generated by SOTA**: "a _ race _ collision. a cars. to a cars. cars cars cars cars are a left. cols are a left"

- **Code Generated by our model**: "def car _ collision ( n ) : return cars. are driving, right, right _ to _ left ( n )"