Generating Python Code from Docstrings using OpenNMT

MSc Research Project
MSc in Data Analytics

Sayok Kumar Bose
Student ID: X20187688

School of Computing
National College of Ireland

Supervisor: Mr. Rejwanul Haque

National College of Ireland

MSc Project Submission Sheet

School of Computing

Student Name:	Sayok Kumar Bose …………………………………………………………………………
Student ID:	X20187688 ……………………………………………………………………………………………….……
Programme:	MSc in Data Analytics …………………………………………………………… **Year:** …2022………………….
Module:	MSc Research Project ………………………………………………………………………………….………
Supervisor:	Mr. Rejwanul Haque ……………………………………………………………………………………….………
Submission Due Date:	19/09/2022 ……………………………………………………………………………………….………

Project Title: Generating Python Code from Docstrings using OpenNMT ………………………………………………………………………….…………….

Word Count: ……6663………………………… **Page Count**………..20…………………………….……..

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.
ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

Signature:	Sayok Kumar Bose ………………………………………………………………………………………
Date:	19/09/2022 ………………………………………………………………………………………………

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST

Attach a completed copy of this sheet to each project (including multiple copies)	□
Attach a Moodle submission receipt of the online project submission, to each project (including multiple copies).	□
You must ensure that you retain a HARD COPY of the project, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	□

Assignments that are submitted to the Programme Coordinator Office must be placed into the assignment box located outside the office.

Office Use Only	
Signature:	
Date:	

Generating Python Code from Docstrings using OpenNMT

Sayok Kumar Bose
X20187688

Abstract

In the last two years with the birth of large language models we have seen some great advancements in the area of code generation in the past 2 years. With more years to come it is expected to rake in more developers from the current statistics of 26 million and stats show an exponential increase of code commits to GitHub every day. Which brings us to the idea considering writing code or a piece of a software can be scaffolded with the help of AI assisted systems. This following piece of article deals with the use two techniques firstly Neural machine translation and Decoder only Language Model built from scratch using OpenNMT Toolkit to generate python code from the docstrings that is scraped out of public GitHub repositories. The data source that we use for the research is CodeSearchNet (CSN) which is a cleaned dataset of code and docstring pairs. Moreover, the performance of the model is evaluated by human intervention, BLEU scores, Language Linting Tools and IDEs.

1. Introduction

Software development through the history of time has evolved as a tool that became necessary and highly entangled in today's way of existence. The exponential growth in the field of technology and the constant human and machine reliance has led to a demand of continuous improvements in the field of computer research and also the tool that can aid the researchers much more effective in this journey of technologically evolving world. The rapid advancements in the field of Artificial Intelligence (AI) showed once again that an independent artificial cognitive was capable of identifying patterns within the data and became great when it came to generate and predict new information based on the historical data. What became obvious beyond this was the step of evolution to make the tools that we use in everyday life for building software to be upgraded with the assistance of an AI. Thus, the compounding knowledge led to more research to understand how an AI perceives code and what generative patterns it can find when trained over large volumes of code bases.

Program synthesis using machine learning is set to bring revolutionary changes to the way how code is perceived in the community as it is to be believed that a code synthetically generated by an AI would bring more agility, better quality, remove impediments and reduce vulnerabilities when it comes to software development. Generating code that solves a specific task is the ultimate way forward as it reduces the human dependency in building better software. The work of an AI assistant to generate code not only help the enterprises looking for skilled developers but also helps an individual to improve on the existing style of development as well as bring in agility by presenting code snippets predicting on the writing of a developer thus reducing many computational hours in context switching in search of the right documentation. Moreover, the code built by the AI could be more resilient when it comes to code recommendations from previously tested code which has been already written and tested. Doing so bring more time back to the developers as they can now focus more on building the desired software and incorporating the business needs rather than looking though code manuals and identifying the ways to write a code. This brings in a layer of abstraction between the developer and the software and gives more flexibility to a developer to even code in the programming language which is not native to them. One way how we perceive interactions between a human and an artificial cognitive has always through natural language. which is similar to the way how we have always queried the web or even asked anyone for help. Thus, using natural language processing (NLP) has always seemed more efficient when we think of information retrieval and hence is a perfect option to fit in when generating code snippets.

With every research demands the question of why the research is needed. To simply understand that we need to look into some statistics that has been put out by Evans Data Corp[1] that predicts that as of today there are 27 million software engineers in this world and by the end of 2030 the world would have produced over 45 million people who deal with software or software building day to day. Which shows that there is a huge skill gap that needs to be identified and moreover what we understand that to incorporate new developers the tooling should be scalable and more efficient and resilient. With more numbers of developers pouring in and more demand in software brings in new code vulnerabilities and more time spent in building the same code which has been already built earlier. In order to ensure transparency and agility we need to bring in AI to generate code for us in the future.

If in past 2 years a lot more money is spent in building an AI that can understand code and can generate them on command as of today there are major corporations such as Microsoft, Google and Tesla are building their own language models that can build software just on command. Moreover, with disruption in cloud driven solutions we see more adoption of service-based infrastructure, platform, and deployments as code in order to maintain a version history and we see more and more GitHub commits every day. Companies like DeepMind (Li et al., 2022), Microsoft (Svyatkovskiy et al., 2020) OpenAI (Chen et al., 2021) has all invested in building an ultimate state of the art machine learning model that delivers software on demand based on NLP using "Transformers" as it has been shown that they are significantly better in writing code when it comes to RNNs (Chernyavskiy et al., 2021). This proves that the idea is backed by the industry experts who see great value and potential in this field of work. Most of the above-mentioned research is based on recent years mainly from 2020 to 2022 which shows that there are more areas to explore in the similar domain space.

[1] https://evansdata.com/press/viewRelease.php?pressID=278

2. Related Work

Use of NLP in Program Synthesis:

Synthesis of programs usually defines the generation of executable code that can run on desired specifications and perform the expected task. Even though the synthesizing the executables has stirred many debates among the researchers the occurrence of the first ever artificially generated code was dated 1971 when the researchers (Manna & Waldinger, 1971) extracted the idea of generating looping algorithms over contiguous data structures like arrays, queues, and stacks as well as nonlinear data structures such as trees and lists. Over the years the interactive medium in the world of computers evolved and commanding over generating a program based on an instruction set given as input became more popular. From the years 2008 to 2011 monumental progress was made where synthesis of program could generate multiple algorithms and then rank them based on the complexities of the generated code.

What showed great promise was the initial business implementation of such artificial synthesis was used to generate and fill out empty cells in the tool MS-Excel (Gulwani, 2011). The tool was great in extracting patterns from the data in the cells and synthetically generate the empty cells with predicted values. Thus, autocomplete became the niche market of research and by the year 2017 the idea of autocompleting in program synthesis evolved and the work of (David & Kroening, 2017) from Oxford University led the examples of how code synthesizers would help software developers using inductive synthesis. On similar grounds the research from (Gulwani et al., 2017) elaborated the challenges interacting between the domain of program space and intention of the user. The state of the art of code generators even though were able to generate code snippets they were nowhere close to production ready codes and sometimes were a complete miss in terms of domain knowledge. Gulwani pointed out that these synthesizers took configuration as input to generate code which was far more difficult to write than the actual code.

During the similar time the research of deeplearning was going in parallel track and huge progress was made in the field of Natural Language processing. In 2010 the deep learning had outstanding results in language processing where the research (Torfi et al., 2020)
identified that language parsers could understand tokenized sentences and following that came the birth of code synthesizers based on NLP in 2017 (Victoria Lin et al., n.d.) used the RNN model trained on bash commands scraped form the internet which had natural descriptors depicting the purpose of the methods. The model showed great success in generating 80% in producing syntaxes that we correct and was production ready. The tool was built on the "RNN-Encoder-Decoder model" which later was made the standard for machine translations in future. This style of decoder encoder model should provide a boost to our research question which tries to generate python code from the source of docstrings written in natural English language.

Use of language modelling using Transformers:

The previous section talked about the good and the bad of NLP used on program synthesis the challenges of RNN Encoder-Decoder in program synthesis which used the concept of context vectors which held the information of the whole input sequence which was used to target the output sequence. Despite the advantages of the architecture using internal weighted attentions the network degrades over long range dependencies due to the problem of vanishing gradients. This study by (Pascanu et al., 2012) shows that this drawback of vanishing gradient started affecting the NLP with long sequences as well as the choice of architecture prevents parallelization.

Challenges of our research question of generating codes from docstrings sometimes may have multiline comments or might have comments which are very long hence using such an architecture would degrade the translation of text to code. LSTMs and GRUs are also often considered rigid thus will not allow us to scale and parallelize if needed. Such challenges can be overcome by the use of the new transformer architecture which removes the recurrent architecture and relies on "attention" mechanism (Vaswani et al., 2017).
This research removed the bound of parallelization and now the training of source data can be scaled over broader networks. Transformers come with their own set of drawbacks and limitations (Fan et al., 2020) which is a missing feedback loop. The feedforward feature makes the transformer very efficient but also pushes back when it comes to exploit the advantages of the input sequence. Thus, one study to enhance quality of large language models can be done by incorporating a feedback loop and shown in the study (Jain et al., 2021) called the Jigsaw model. The choice of often whether to use a pretrained model and leverage the benefits of transfer learning or to build a model from scratch. The idea of picking the right approach for the project comes from the work of (Min et al., 2021) which describes that transfer learning over the years has become cheaper, faster, and easier to implement. The study by Zhuang et al. (2021) on transfer learning shows that be it any domain of data the pre trained model have always outperformed the models built from scratch but transfer learning on pre trained models requires resources that are expensive and exhaustive in nature.

Some models like BERT (Devlin et al., 2018) was trained over 3.3 billion tokens and almost 110 million parameters this was a significant breakthrough in the field of NLP. The palce where BERT outshined GPT was that BERT uses Bi-Directional Encoders this fused the idea of multi-head attention layers which lead to great GLUE score of 80% (Wang et al., 2019) and the MiltiNLI (Williams et al., 2017) was increased by 5%. This led to the rush of using BERT to push boundaries of down streaming tasks such as "VideoBERT" (Sun et al., 2019) and "CodeBERT" (Feng et al., 2020). New advances in BERT led to a lot of close-knit research that pushed the boundaries of the model to explore how the baseline model could be extrapolated and finetuned to perform specific tasks. Such working examples were explored when language models were trained on code. Research was conducted such as AlphaCode (Li et al., 2022), PYMT5 (Clement et al., 2020) and CodeBERT clearly showed some progress when it came to synthesizing code from natural language. CodeBERT and PYMT5 showed great predictions outcomes in terms of BLEU scores of 19.06 and 16.3 for python code generations respectively. While CodeBERT was approximately 82% accurate while generating a valid python code.

Each of these models had their unique approaches to solve code generation challenges, PYmt5 is the first ever python code completion tool that leverages on T5 transformer architecture (Raffel et al., 2019) which has been seen to outperform the GPT2 generators in terms of code generators.

OpenNMT: An Open-Source Toolkit for Neural Machine Translation:

The OpenNMT for neural machine translation(NMT) is a modern, efficient, and extensible toolkit that was built to help and support NMT research into a model architecture and feature representations without losing the competitive performance. The toolkit comes with modelling and translation out of the box with detailed descriptions of the underlying technique that has been used.

NMT is considered as the modern de facto standard when it comes to machine translation, as it has shown great improvements in the domain of human evaluations compared to rule based statistical approaches used in SMT systems (Johnson et al., 2016). The NMT approaches are standardized when it comes to NLP community for developing open translation systems for researchers to set benchmarks, learn and extend. OpenNMT can be used for multiple language modelling techniques such as dialogue generation like chats, parsing a text sequence and also can be used to summarize long passages.

Currently there a multiple NMT toolkits many of which are developed by industry leaders such as Google and Microsoft but are mostly closed for public use and will not be used as free license software.

3. Research Methodology

For this research we have selected Knowledge Discovery in Database (KDD) methodology as the base driver for the overall procedure of data collection, preprocessing, feature engineering, data modeling, evaluation, and visualization. The idea of KDD fits perfect here as KDD is an iterative process that allows to extract useful information form large data corpuses running an iterative process of modeling, data cleaning and engineering to foster better results.

3.1 Dataset Selection

The data that we are using for this research is code that is scrapped out of public repository like GitHub. This includes code from different programming languages such has Go, Python, Java, JavaScript, PHP, and Ruby. Details of the dataset with breakdown of language and their respective number of functions are displayed in Table-1. CSN (Husain et al., 2019) dataset which is available for public use has over a million commented code pairings. The median code length for the corpus is around 60-100 text tokens. This dataset is set as a benchmark that explores the problem of code retrieval using natural language processing. This dataset was presented as a joint collaboration between GitHub and Microsoft's Research team at Cambridge. The data is segregated into Test, Train and Validation and the quality was ensured that code form similar repositories existed in only one partition.

Table 1: Statistics of the dataset used for training

Programming Language	Number of Functions with documentation
Go	347789.00
Java	542991.00
JavaScript	157988.00
PHP	717313.00
Python	503502.00
Ruby	57393.00
	2326976.00

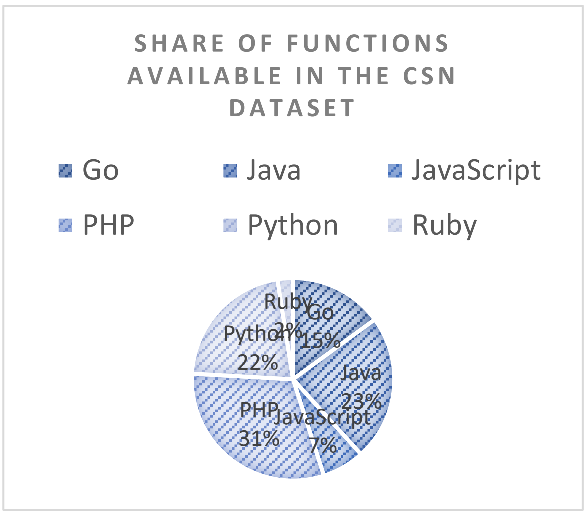
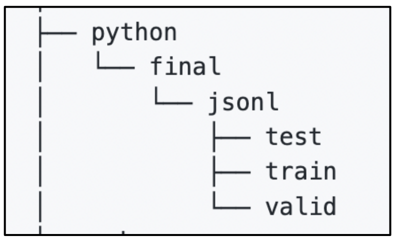


Figure 1: Share of functions available in the CSN dataset

As described in Chen et al., (2021) when scraping public repositories, the problem arises when the repos are cloned and are non-forked there could be a lot of duplication the repositories should have higher star ratings to ensure that the algorithm has a cleaner data to start with. Lastly the corpus should be avoiding docstrings with large comments which is exactly what we do in our experiments or code files as this would affect the modeling technique due to scaffolding issues.

3.2 Data Extraction

The CSN data corpus is already filtered and gives us multiple ways to unpack the data one of the simple ways to unpack the data is to download the zip files from Amazon (AWS) S3 bucket. The CSN also provides us with scripts to download the code base as a part of the initial setup script and also provides with API endpoints to download the code-corpus. The folder structure is quite simple to understand, and we can specifically download the code corpus for our python code. Once we download the corpus, we have uploaded the following code corpus to google drive so that we can easily access the corpus within google collab session.



```
├── python
│   └── final
│       └── jsonl
│           ├── test
│           ├── train
│           └── valid
```

Figure 2: Folder Structure for python code in CodeSearchNet

3.3 Feature Selection

The data is stored as jsonlines for all programming languages where each individual lines are uncompressed files that denotes one example along with its associated docstrings. The breakdown of this jsonline shows the following details:

- **repo:** The actual owner of the repository (user who checked in the code)
- **path:** The complete path for the code file inside the repo
- **func_name:** The name of the function
- **original_string:** The complete raw string in original format
- **language:** The programming language in which the code is written
- **code:** the part of the `original_string` that is code
- **code_tokens:** code tokens
- **docstring:** The comment that describes the code written in original string
- **docstring_tokens:** doctsring tokens
- **partition:** This identifies what segment the data lies into (test, train, validation).
- **url:** the complete url for the for the code present in GitHub along with the line numbers

From this data of jsonl we would be taking the function name as the source and the original string as the target for our OpenNMT. Rest of the data could be ignored as the OpenNMT needs a source and target in a single line of records. Since original string contains newlines with indentations in order to make them serialized we removed the instatement docstring the form the original string and then appended the function name and the code to into a single line by removing the "\n" characters and replacing them with a token "NMTNL" just to understand where the newline in a sequence exists.

3.4 Modelling Technique

Here in this research we have kept two different choice of modelling techniques both using OpenNMT Toolkit:

- Encoder Decoder Model Used For Translation
- Decoder Only Language Model (Radford et al., 2018)

3.5 Evaluation Metrics

Evaluation techniques for language models generating code is quite difficult when considered in parallel to simple machine translation as code generation cannot be quantified based on translation metrices. Usually, language models trained on code face challenges to evaluate on machine generated code pieces as well and something like BLEU (Papineni et al., 2002) does not fit the overall objective. Researchers have shown how BLEU scoring for code evaluation is fuzzy as precision (Chen et al., 2021) cannot be the only criteria to validate a code snippet. Thus, for this research we would be using a comprehensive 3 step approaches.

- Human Evaluation (with help of PyLance + VSCode editor)
- Linting scores based PyLint (code quality check)
- BLEU scores (using SacreBLEU[2])

4. Exploratory Data Analysis

Once we have downloaded the python corpus the next step in the data mining process is to perform exploratory data analysis (EDA) on the existing corpus as this allows us to understand the data better before we move into data modelling while check for data outliers or any abnormalities in the data. The source string, which is the docstrings for us, will be the primary natural English language input and the following are the few observations from that.

Exploratory analysis of the docstring shows that the length of the scraped strings varies in size and some of them are huge sentences with over 190 words in them and the sentence length of these docstrings are also nearing 1000 letters. Now these are quite lengthy strings that should be remove and truncated for the model training.

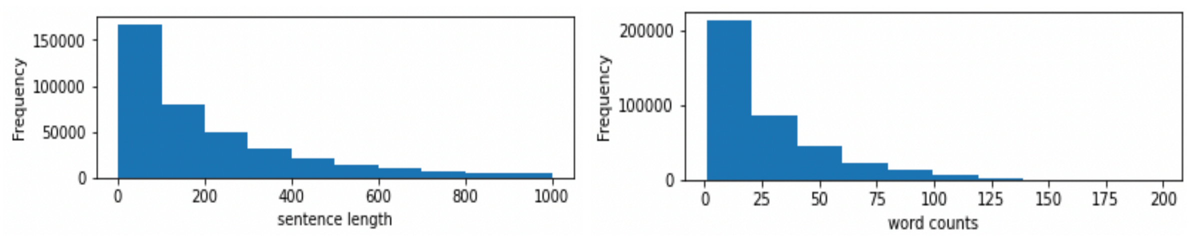


Figure 3: Frequency of senetcne length and word counts

To check the frequency of the choice of words by developers in their docstrings we tried to implement a word cloud removing the stop words completely. This shows that some of the prominent words used in docstrings are “Return”, “Create”, “List”, “Delete”. This shows that the docstrings are usually filled with verbs that demand a specific task form the subsequent method.

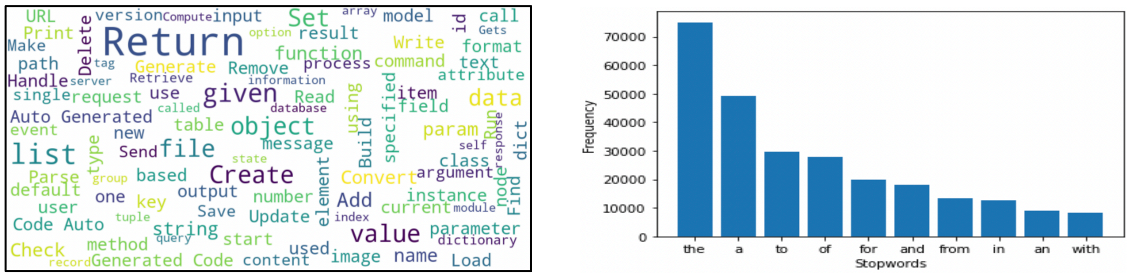


Figure 4: Mostly commonly used word-cloud and stowords frequency

2 https://github.com/mjpost/sacrebleu

Building on top of that when we look into the stop words and their frequencies and plot the most common ones, we could see that the frequency of the articles such as “the” and “a” are quite massive when it comes in comparison with others.

5. Implementation and Evaluation

This section is broken into two separate experiments as based on two different choices of modelling techniques used as mentioned in section 3.4. This gives an in-depth understanding of the individual implementation of the choices made in each step as they differ from one another.

5.1 Experiment 1

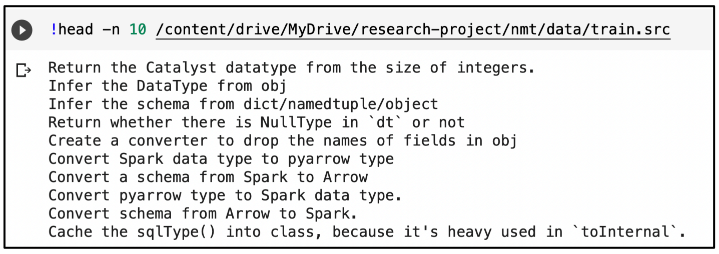
The first experiment is conducted by using the standard. OpenNMT seq-2-seq translation on the python docstring corpus as the source and the code body as the target. For our experiments we have used the Pytorch version OpenNMT-py. The toolkit that was released in 2017 and further updated in 2020 (Klein et al., n.d.). Installation of this toolkit is fairly simple and can be downloaded using latest pip installer.

5.1.1 Preparing Data

In order to use the translation OpenNMT translation model we need to prepare the data accordingly just so that we can use the “onmt_train” tool and for that the primary step is to prepare the data. So, for our experiment we would move all the “docstring” from jsonl files and move them to train.src and “original_string” which contains the code corpus was then made into a single line as the onmt_train expects the translated text also to be in a single line. To achieve that we have removed the newline characters with “NMTNL”.

The following image shows the glimpse of the training source used where each line is a docstring while fig 5 shows the function definition and the code for the respective docstrings with the filler word NMTNL for newline characters. For training we have removed the docstrings that are greater than 100 characters long as according to good programing practices and the industry standards the maximum suggested length of a docstring should be less than 72 characters[3].

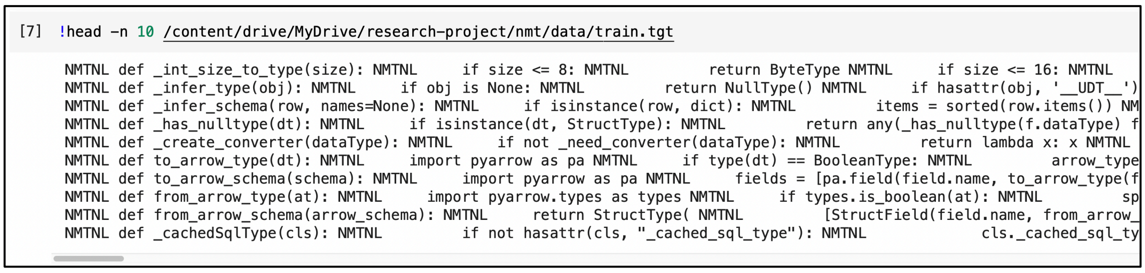
[3] http://docs.bigchaindb.com/projects/contributing/en/latest/cross-project-policies/python-style-guide.html#maximum-line-length



```
!head -n 10 /content/drive/MyDrive/research-project/nmt/data/train.src

Return the Catalyst datatype from the size of integers.
Infer the DataType from obj
Infer the schema from dict/namedtuple/object
Return whether there is NullType in `dt` or not
Create a converter to drop the names of fields in obj
Convert Spark data type to pyarrow type
Convert a schema from Spark to Arrow
Convert pyarrow type to Spark data type.
Convert schema from Arrow to Spark.
Cache the sqlType() into class, because it's heavy used in `toInternal`.
```

Figure 5: Source corpus or NMT



```
[7]  !head -n 10 /content/drive/MyDrive/research-project/nmt/data/train.tgt

     NMTNL def _int_size_to_type(size): NMTNL     if size <= 8: NMTNL         return ByteType NMTNL     if size <= 16: NMTNL
     NMTNL def _infer_type(obj): NMTNL     if obj is None: NMTNL         return NullType() NMTNL     if hasattr(obj, '__UDT__')
     NMTNL def _infer_schema(row, names=None): NMTNL     if isinstance(row, dict): NMTNL         items = sorted(row.items()) NM
     NMTNL def _has_nulltype(dt): NMTNL     if isinstance(dt, StructType): NMTNL         return any(_has_nulltype(f.dataType) f
     NMTNL def _create_converter(dataType): NMTNL     if not _need_converter(dataType): NMTNL         return lambda x: x NMTNL
     NMTNL def to_arrow_type(dt): NMTNL     import pyarrow as pa NMTNL     if type(dt) == BooleanType: NMTNL         arrow_type
     NMTNL def to_arrow_schema(schema): NMTNL     import pyarrow as pa NMTNL     fields = [pa.field(field.name, to_arrow_type(f
     NMTNL def from_arrow_type(at): NMTNL     import pyarrow.types as types NMTNL     if types.is_boolean(at): NMTNL         sp
     NMTNL def from_arrow_schema(arrow_schema): NMTNL     return StructType( NMTNL         [StructField(field.name, from_arrow_
     NMTNL def _cachedSqlType(cls): NMTNL         if not hasattr(cls, "_cached_sql_type"): NMTNL             cls._cached_sql_ty
```

Figure 6: Target corpus or NMT

5.1.2 Setup

The setup for using OpenNMT-Py training needed a *config.yaml* file that needed to be created that would hold the details of the architecture that is needed to run the training for the translation model. For a baseline the configuration was kept simple with transformer-based encoder decoder architecture running for 100000 steps with a batch size of 2048 and a learning rate of 2. Validation was carried out after every 1000 steps and the model was also saved as checkpoints at every 1000 steps. The model training was also configured to have Adam Optimizer and to avoid extensive training we have used early stop if there is no improvement on last 3 validation checkpoints. The following table shows the details of the configuration setup for the baseline experiments.

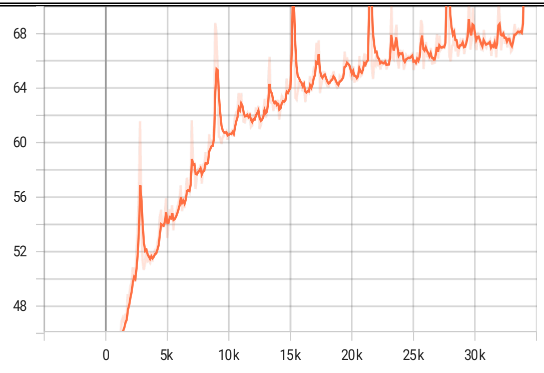
Table 2 : Configuration Details of the NMT Experiment

Config	**Value**
Validation Steps	1000
Warmup Steps	4000
Batch Size	2048
Validation Batch Size	2048
Optimization Algorithm	Adam
Early Stopping	True (up to 3 validations)
Checkpoints	1000 (save up to 3)
Learning Rate	2
Encoder Decoder Type	Transformer
Layers	6
Transformer Feed Forward	2048
Word Vector Size	512

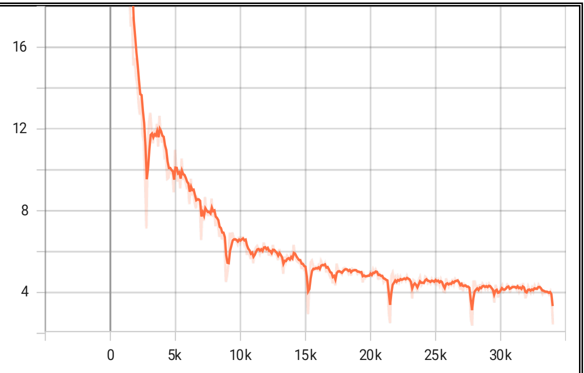
Heads	8
Dropout	0.1

5.1.3 Training

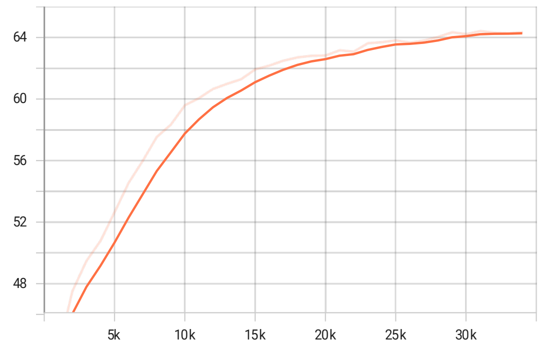
Using OpenNMT toolkit has the advantage of using the nmt-train which utilizes the "tensorboard" feature which logs the whole training runtime. This shows the model trained on the test data for 34000 steps before it made an early stop with a progress accuracy of 73.27 while the progress perplexity for the same training batch reduced to 3.3. On the validation data set we can see that the accuracy has reached 64.27 whereas the perplexity reached 7.4 after the end of 34k steps.



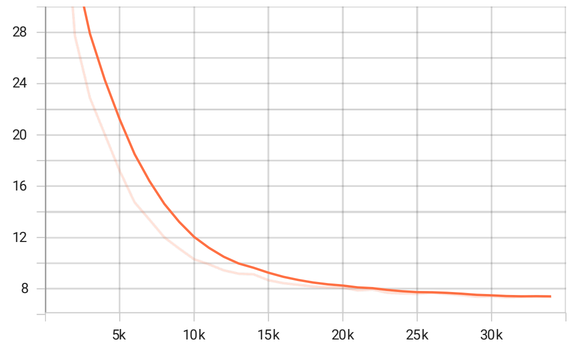
(a) Training Accuracy vs steps



(b) Training Perplexity vs steps



(c) Validation Accuracy vs steps



(d) Validation Perplexity vs steps

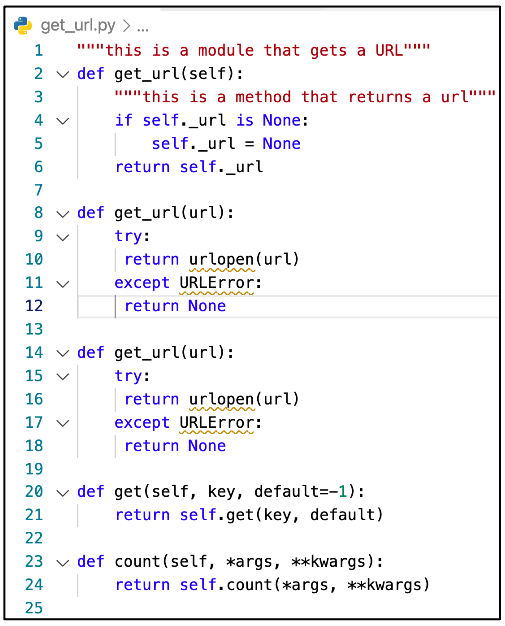
Fig 7: The following are the figures from tensorboard trainign logs

5.1.4 Evaluation

As explained in the methodology standard NLP scoring such as BLEU cannot be the only metric that can account for an evaluation thus, we categorized it into 3 categories Human Evaluation, BLEU score and PyLint Scores.
If we see the code that the machine learning model writes on the test docstring set, we can see they all look very similar to a python function definition and with some closer evaluation we

can see that the Pylance tool in Visual Studio Code shows that the top 5 methods generated has 4 warnings and 0 errors syntactically which makes the research output quite optimistic.

Some problems that we see with the output is that the translation model generates 3 methods with the exact same name and also the method generated are quite small and concise.



get_url.py > ...

```python
"""this is a module that gets a URL"""
def get_url(self):
    """this is a method that returns a url"""
    if self._url is None:
        self._url = None
    return self._url

def get_url(url):
    try:
        return urlopen(url)
    except URLError:
        return None

def get_url(url):
    try:
        return urlopen(url)
    except URLError:
        return None

def get(self, key, default=-1):
    return self.get(key, default)

def count(self, *args, **kwargs):
    return self.count(*args, **kwargs)

```

Figure 8: Cleaned and re-aligned code generated by the NMT model.

For evaluating the BLEU score we have used the ScareBLEU tool and form the predicted text the BLEU score for the following NMT translation evaluates to 0.7.
The pyLint cli tool when ran on the manually cleaned output resulted a score of 2 out of 10 which is quite impressive for an AI generated code snippet.

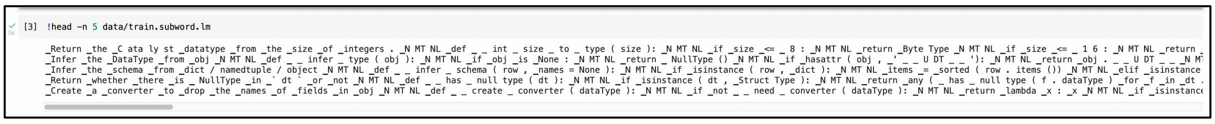
5.2 Experiment 2

The second experiment is conducted by creating a completely new language model (Radford et al., 2018) using OpenNMT toolkit using a decoder only architecture built from scratch.

5.2.1 Preparing Data

Preparing data for training the Decoder only Language model was quite different from that of the NMT translation model the reference of this data building was similar to building WikiText-103[4]. Here is the subword-ed corpus for the first 5 entries.

[4] https://blog.salesforceairesearch.com/the-wikitext-long-term-dependency-language-modeling-dataset/



```
[3]  !head -n 5 data/train.subword.lm

     _Return _the _C ata ly st _datatype _from _the _size _of _integers . _N MT NL _def _ _ int _ size _ to _ type ( size ): _N MT NL _if _size _<= _ 8 : _N MT NL _return _Byte Type _N MT NL _if _size _<= _ 1 6 : _N MT NL _return _
     _Infer _the _DataType _from _obj _N MT NL _def _ _ infer _ type ( obj ): _N MT NL _if _obj _is _None : _N MT NL _return _ NullType () _N MT NL _if _hasattr ( obj , _' _ _ U DT _ _ '): _N MT NL _return _obj . _ _ U DT _ _ _N M
     _Infer _the _schema _from _dict / namedtuple / object _N MT NL _def _ _ infer _ schema ( row , _names = None ): _N MT NL _if _isinstance ( row , _dict ): _N MT NL _items _= _sorted ( row . items ()) _N MT NL _elif _isinstance
     _Return _whether _there _is _ NullType _in _` dt ` _or _not _N MT NL _def _ _ has _ null type ( dt ): _N MT NL _if _isinstance ( dt , _Struct Type ): _N MT NL _return _any ( _ has _ null type ( f . dataType ) _for _f _in _dt .
     _Create _a _converter _to _drop _the _names _of _fields _in _obj _N MT NL _def _ _ create _ converter ( dataType ): _N MT NL _if _not _ _ need _ converter ( dataType ): _N MT NL _return _lambda _x : _x _N MT NL _if _isinstance
```

Figure 9: Code corpus for building the Language Model.

5.2.2 Setup

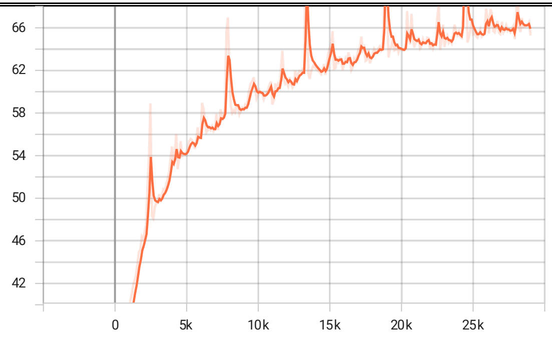
The setup for using building a completely new learning model using OpenNMT-py is also quite similar when it comes to designing a *config.yaml* file. The only change here was the Decoder type was set to Transformer_LM which signifies OpneNMT toolkit that this is a language model. The rest was straight forward and was kept similar to the baseline model. The following table shows the values in detail.

Table 3 : Configuration Details of the Language Model

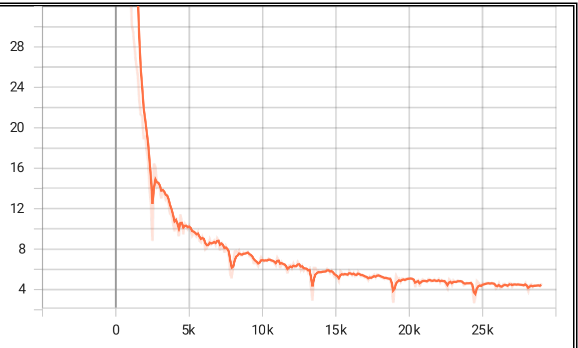
Config	**Value**
Validation Steps	1000
Warmup Steps	8000
Batch Size	2048
Validation Batch Size	2048
Optimization Algorithm	Adam
Early Stopping	True (up to 3 validations)
Checkpoints	1000 (save up to 2)
Learning Rate	2
Encoder Decoder Type	Transformer_LM
Layers	6
Transformer Feed Forward	2048
Word Vector Size	512
Heads	8
Dropout	0.1

5.2.3 Training

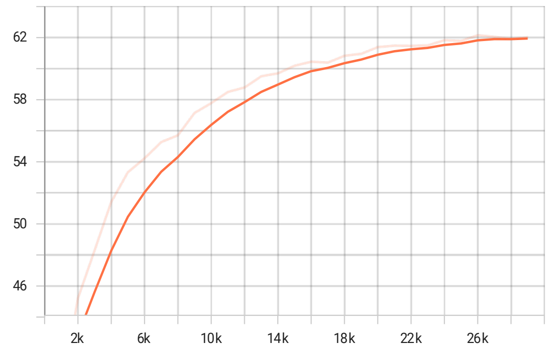
This shows the model trained on the test data for 29000 steps before it made an early stop with a progress accuracy of 65.92 while the progress perplexity for the same training batch reduced to 4.46. On the validation data set we can see that the accuracy has reached 61.94 whereas the perplexity reached 8.5 after the end of 29k steps.



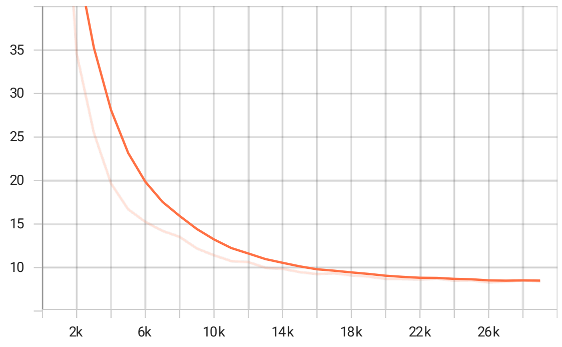
(a) Training Accuracy vs steps



(b) Training Perplexity vs steps



(c) Validation Accuracy vs steps

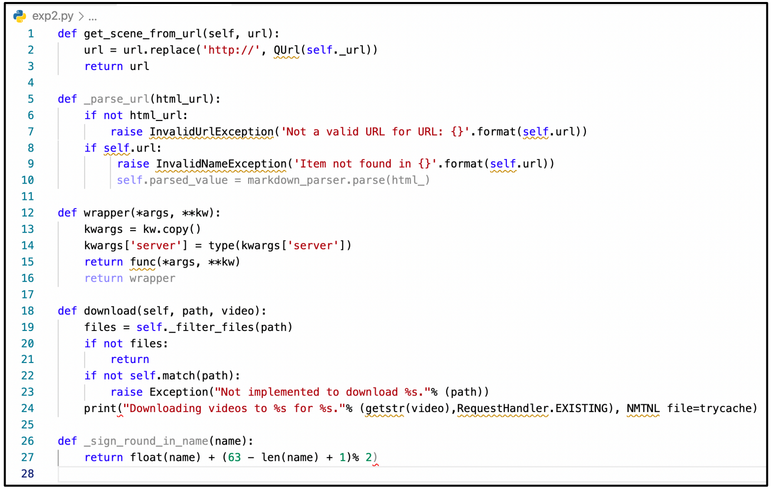


(d) Validation Perplexity vs steps

Fig 10: The following are the figures from tensorboard training logs for Language Model Building

5.2.4 Evaluation

In comparison to the NMT Translation and Experiment 1 here we see that the methods are more well defined and long constructed sentences. The top 5 generated methods each have a different method name as opposed to our Experiment 1. Along with better generated code snippets when compared to NMT model what we see here is that there are many incomplete methods after we de-subword and manually correct the indentations from the test output.
In first 5 methods generation there are 3 Errors and 14 warnings.



```python
def get_scene_from_url(self, url):
    url = url.replace('http://', QUrl(self._url))
    return url

def _parse_url(html_url):
    if not html_url:
        raise InvalidUrlException('Not a valid URL for URL: {}'.format(self.url))
    if self.url:
         raise InvalidNameException('Item not found in {}'.format(self.url))
         self.parsed_value = markdown_parser.parse(html_)

def wrapper(*args, **kw):
    kwargs = kw.copy()
    kwargs['server'] = type(kwargs['server'])
    return func(*args, **kw)
    return wrapper

def download(self, path, video):
    files = self._filter_files(path)
    if not files:
        return
    if not self.match(path):
        raise Exception("Not implemented to download %s."% (path))
    print("Downloading videos to %s for %s."% (getstr(video),RequestHandler.EXISTING), NMTNL file=trycache)

def _sign_round_in_name(name):
    return float(name) + (63 - len(name) + 1)% 2)

```

Fig 11: Cleaned and re-aligned code generated by the NMT model.

The BLEU score for the top 50 records from the output generated by the Language Model comes to an average of 1.1 which seems to be slightly better than our translation model has performed on the similar dataset, but the flip side is that the Pylint did not generate any score as the methods mostly did not compile due to syntactical errors.

6. Conclusion

In this research we have used two model building techniques such as OpenNMT Translation and secondly the decoder only language model that we built from scratch using the OpenNMT toolkit. The scores look very promising to begin with when we see that the training set ana validation set accuracy are quite high for the NMT while the BLEU scores for the Transformer Language Model is higher than the original baseline NMT model.

Table 3 : Overview of the two Models

Modelling Technique	Training Data		Validation Data		Evaluation			
	Accuracy	Perplexity	Accuracy	Perplexity	BLEU	PyLint	Pylance	
							Errors	Warnings
NMT	73.27	3.3	64.27	7.4	0.7	2	0	4
Transformer LM	65.92	4.46	61.94	8.5	1.1	NA	3	14

In such scenarios there are no clear winners when it comes in deciding a choice of algorithms as we have seen that the Language Model built from scratch even though has more errors in Pylance scores but were better structured and well-formed code snippets when compared with the ones generated by NMT.

7. Discussion

Some of the pretrained models when are generating code they have shown to have very low BLEU scores Even the models such as GPT-2 which has over 1.5 billion parameters and has over 8 million training parameters over multiple webpages produced a mere BLEU score of 5.63 and another generative language model T5 which is specifically trained on CSN dataset produces a BLEU score of 10 for python code corpus. This shows that even very large models struggle hard to get decent scores when asked to generate code from docstrings even though the generated code by GPT2 is 95% accurate by Human Evaluation. This shows that BLEU scores are not that great when it comes to code generation.

How BLEU scores can be improved:

- Data Augmentation to sample more docstrings to make at least million training records.
- Use a larger language pretrained model on code using higher computes from vast.ai

- More data processing to re-engineer white spaces from the code as language model often struggles with white spaces and stop words.

we have discussed majorly on the areas of why the BLEU scores are low for this research the better alternative is a new evaluation metric that suits better for our research.

CodeBLEU for Evaluation:

A much better way code evaluation which has slowly picked up popularity is using CodeBLEU (Ren et al., 2020). In the era of multiple code generation models the most common use evaluation metric is BLEU as this has been the standard translation evaluation metric for natural languages. This way it neglects important syntactic and semantic features for how code is written. As usually BLEU ignores the code semantic logic. On the flip side CodeBLEU elevates the logic of n-gram evaluation by BLEU with the abstract syntax trees (AST) and by checking the Data flow within the code. The output from the CodeBLEU has better corelation with Human Evaluation.

The reason for not including this metric in the research was out of pure lack of time and knowledge gap at the time of research. The Language model that we have built usually translates code into single lines treating it as a translated text which needs a dedicated workflow to clean it before we could process it for CodeBLEU evaluation. BLEU scoring was quick and easily available on existing language model outcome. Even none of the Large Language models that are trained on code are yet to be evaluated on CodeBLEU.

8. Future Work

Even though there was a lot that was achieved in this research but there is always a lot of scope for improvement. Just with limited resource we were able to achieve fully generated python code and got good view on human evaluation. But mostly due to lack of time and the scope of this solo project there could be many aspects that could be improved in future that can be categorise into few major areas of improvements.

- Better parsing of data during the data preparation of by removing spaces in the code but since in python whitespaces have significant meaning hence we can try to clean up white spaces from the code and replace them with specific words like we did for newline characters. In doing that the expected output could be a much better structure when de-serialised.
- Using transfer learning from latest transformer based models for hugging face like OPT (Zhang et al., 2022) and GPT-Neo (Black & Gao, 2021). Tried to incorporate a transfer-learning using GPT-Neo and GPT-J but the major drawback was because of the GPUs burning out during training itself. Hence on of the future work would be definitely trying out generative models which are trained on billions or parameters using better resources.

- Using data augmentation techniques such as using T5 (Ganguli et al., 2021) model to synthetically generate more data that could be randomised and fed into the training loop again.
- Using a feedback loop with adjusted weights to improve the performance of the model by removing the human evaluation and implementing a reinforcement learning using a linting tool score.
- Running the training for longer period of time to see if the number of steps would have improved the training and the validation score. Right now the whole model was running on a Google Collab Pro session with 16Gb GPU which took longer hours to run. So a dedicated compute with higher GPUs would give us an edge in trying out multiple scenarios.
- Use multiple Data sources other than just CSN. By doing so we could implement more real-time scenarios. There we more datasets showing up that could be used merging with the existing corpus. The existing corpus contained half a million trainable lines of code which could be less when we see language modelling.

Overall generating code using NLP on code comments is a very promising direction of work which makes me optimistic about future prospects in using machine learning to synthesize programs.

REFERENCES

Black, S., & Gao, L. (2021). *GPT-Neo: Large Scale Autoregressive Language Modeling with Mesh-Tensorflow*.

Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H. P. de O., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., Ray, A., Puri, R., Krueger, G., Petrov, M., Khlaaf, H., Sastry, G., Mishkin, P., Chan, B., Gray, S., … Zaremba, W. (2021). *Evaluating Large Language Models Trained on Code*. http://arxiv.org/abs/2107.03374

Chernyavskiy, A., Ilvovsky, D., & Nakov, P. (2021). *Transformers: "The End of History" for NLP?* http://arxiv.org/abs/2105.00813

Clement, C. B., Drain, D., Cloud, M., Timcheck, J., Svyatkovskiy, A., & Sundaresan, N. (2020). *PYMT5: multi-mode translation of natural language and PYTHON code with transformers*.

David, C., & Kroening, D. (2017). Program synthesis: challenges and opportunities. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, *375*(2104). https://doi.org/10.1098/rsta.2015.0403

Devlin, J., Chang, M.-W., Lee, K., & Toutanova, K. (2018). *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*. http://arxiv.org/abs/1810.04805

Fan, A., Lavril, T., Grave, E., Joulin, A., & Sukhbaatar, S. (2020). *Addressing Some Limitations of Transformers with Feedback Memory*. http://arxiv.org/abs/2002.09402

Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., & Zhou, M. (2020). *CodeBERT: A Pre-Trained Model for Programming and Natural Languages*. http://arxiv.org/abs/2002.08155

Ganguli, I., Bhowmick, R. S., Biswas, S., & Sil, J. (2021). Empirical Auto-Evaluation of Python Code for Performance Analysis of Transformer Network Using T5 Architecture. *2021 8th International Conference on Smart Computing and Communications: Artificial Intelligence, AI Driven Applications for a Smart World, ICSCC 2021*, 75–79. https://doi.org/10.1109/ICSCC51209.2021.9528123

Gulwani, S. (2011). Automating String Processing in Spreadsheets Using Input-Output Examples. In *ACM SIGPLAN Notices* (Vol. 46). https://doi.org/10.1145/1926385.1926423

Gulwani, S., Polozov, A., & Singh, R. (2017). Program Synthesis. In *Foundations and Trends in Programming Languages* (Vol. 4). NOW. https://www.microsoft.com/en-us/research/publication/program-synthesis/

Husain, H., Wu, H.-H., Gazit, T., Allamanis, M., & Brockschmidt, M. (2019). *CodeSearchNet Challenge: Evaluating the State of Semantic Code Search*. http://arxiv.org/abs/1909.09436

Jain, N., Vaidyanath, S., Iyer, A., Natarajan, N., Parthasarathy, S., Rajamani, S., & Sharma, R. (2021). *Jigsaw: Large Language Models meet Program Synthesis*. http://arxiv.org/abs/2112.02969

Johnson, M., Schuster, M., Le, Q. v., Krikun, M., Wu, Y., Chen, Z., Thorat, N., Viégas, F., Wattenberg, M., Corrado, G., Hughes, M., & Dean, J. (2016). *Google's Multilingual Neural Machine Translation System: Enabling Zero-Shot Translation*. http://arxiv.org/abs/1611.04558

Klein, G., Hernandez, F., Nguyen, V., & Senellart, J. (n.d.). *The OpenNMT Neural Machine Translation Toolkit: 2020 Edition*. https://opennmt.net

Li, Y., Choi, D., Chung, J., Kushman, N., Schrittwieser, J., Leblond, R., Eccles, T., Keeling, J., Gimeno, F., Lago, A. D., Hubert, T., Choy, P., d'Autume, C. de M., Babuschkin, I., Chen, X., Huang, P.-S., Welbl, J., Gowal, S., Cherepanov, A., … Vinyals, O. (2022). *Competition-Level Code Generation with AlphaCode*. http://arxiv.org/abs/2203.07814

Manna, Z., & Waldinger, R. J. (1971). Toward Automatic Program Synthesis. *Communications of the ACM*, *14*(3), 151–165. https://doi.org/10.1145/362566.362568

Min, B., Ross, H., Sulem, E., Veyseh, A. P. ben, Nguyen, T. H., Sainz, O., Agirre, E., Heinz, I., & Roth, D. (2021). *Recent Advances in Natural Language Processing via Large Pre-Trained Language Models: A Survey*. http://arxiv.org/abs/2111.01243

Papineni, K., Roukos, S., Ward, T., & Zhu, W.-J. (2002). *BLEU: a Method for Automatic Evaluation of Machine Translation*.

Pascanu, R., Mikolov, T., & Bengio, Y. (2012). *On the difficulty of training Recurrent Neural Networks*. http://arxiv.org/abs/1211.5063

Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., & Sutskever, I. (2018). *Language Models are Unsupervised Multitask Learners*. https://github.com/codelucas/newspaper

Raffel, C., Shazeer, N., Roberts, A., Lee, K., Narang, S., Matena, M., Zhou, Y., Li, W., & Liu, P. J. (2019). *Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer*. http://arxiv.org/abs/1910.10683

Ren, S., Guo, D., Lu, S., Zhou, L., Liu, S., Tang, D., Sundaresan, N., Zhou, M., Blanco, A., & Ma, S. (2020). *CodeBLEU: a Method for Automatic Evaluation of Code Synthesis*.

Sun, C., Myers, A., Vondrick, C., Murphy, K., & Schmid, C. (2019). VideoBERT: A joint model for video and language representation learning. *Proceedings of the IEEE International Conference on Computer Vision*, *2019-October*, 7463–7472. https://doi.org/10.1109/ICCV.2019.00756

Svyatkovskiy, A., Deng, S. K., Fu, S., & Sundaresan, N. (2020). IntelliCode compose: Code generation using transformer. *ESEC/FSE 2020 - Proceedings of the 28th ACM Joint Meeting European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 1433–1443. https://doi.org/10.1145/3368089.3417058

Torfi, A., Shirvani, R. A., Keneshloo, Y., Tavaf, N., & Fox, E. A. (2020). *Natural Language Processing Advancements By Deep Learning: A Survey*. http://arxiv.org/abs/2003.01200

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., & Polosukhin, I. (2017). *Attention Is All You Need*. http://arxiv.org/abs/1706.03762

Victoria Lin, X., Wang, C., Pang, D., Vu, K., Zeelemoyer, L., & Ernst, M. D. (n.d.). *Program Synthesis from Natural Language Using Recurrent Neural Networks*. www.commandlinefu.com/

Wang, A., Singh, A., Michael, J., Hill, F., Levy, O., & Bowman, S. R. (2019). *GLUE: A MULTI-TASK BENCHMARK AND ANALYSIS PLATFORM FOR NATURAL LANGUAGE UNDERSTAND-ING*.

Williams, A., Nangia, N., & Bowman, S. R. (2017). *A Broad-Coverage Challenge Corpus for Sentence Understanding through Inference*. http://arxiv.org/abs/1704.05426

Zhang, S., Roller, S., Goyal, N., Artetxe, M., Chen, M., Chen, S., Dewan, C., Diab, M., Li, X., Lin, X. V., Mihaylov, T., Ott, M., Shleifer, S., Shuster, K., Simig, D., Koura, P. S., Sridhar, A., Wang, T., & Zettlemoyer, L. (2022). *OPT: Open Pre-trained Transformer Language Models*. http://arxiv.org/abs/2205.01068

Zhuang, F., Qi, Z., Duan, K., Xi, D., Zhu, Y., Zhu, H., Xiong, H., & He, Q. (2021). A Comprehensive Survey on Transfer Learning. In *Proceedings of the IEEE* (Vol. 109, Issue 1, pp. 43–76). Institute of Electrical and Electronics Engineers Inc. https://doi.org/10.1109/JPROC.2020.3004555