

Department of Electrical, Computer, and Software Engineering

Part IV Research Project

Final Report

Project Number: 77

Are Machine Intelligent

Enough to Create

Programs Themselves?

Nathan Cairns

Buster Major

Jing Sun

27/09/2019

Declaration of Originality

This report is my own unaided work and was not copied from nor written in collaboration with any other person.

A handwritten signature in black ink, appearing to read 'Nathan Cairns', with a stylized, flowing script.

Name: Nathan Cairns

ABSTRACT:

70-80% of programmer time is spent debugging and testing code. This report investigates the feasibility of using machine learning to generate code which might help reorient this developer time. We begin by performing a comprehensive review of existing academic literature. This research is then used to develop a novel solution to code generation. This is an LSTM based approach called Code By Tensors or CBT. CBT generates lines of code for incomplete programs and can do so for both the Python and C programming languages. There is an in-depth description of how this approach was developed. Finally, an extensive evaluation and discussion is performed on this approach. Our results showed that our approach was relatively good at understanding a programming language's syntactic constructs. However, there was significant shortcomings with regards to proper variable and function usage. This implied that our approach struggles to understand the semantics and context of a program. We found that our approach performed relatively poorly when compared with other similar approaches. However, we also found there to be a lot of potential in further developing our solution. Furthermore, we set a benchmark for further research into generating code with a purely LSTM based approach.

LIST OF FIGURES

Fig. 1. Examples of Python (left) and C (right) code.	(9)
Fig. 2. Architecture diagram of the CBT code generator	(10)
Fig. 3. Implemented scripts and how they interact.....	(12)
Fig. 4. A program with the last 3 lines generated using CBT	(12)
Fig. 5. Python tokenization.....	(15)
Fig. 6. Evaluation framework design.....	(16)
Fig. 7. Program executability percentage when asked to generate 1, 2, and 3 lines for Python (left) and C (right)	(17)
Fig. 8. Accuracy of guessing next keyword and variable for Python models (Left) and C models (Right)..	(18)
Fig. 9. First appearing keyword frequency in original files and model training splits across Python models tested on generating last 1 line (Top-Left), C models tested on generating last 1 line (Top-Right), Python models tested on generating last 3 lines (Bottom-Right).....	(19)
Fig. 10. Frequency of new variable introduction in last line of program across Python and C models	(19)

LIST OF TABLES

Table 1: Division of work between Buster Major and Nathan Cairns.....	(6)
Table 2: Tokenized words in Python and C	(14)
Table 3: Keyword and variable stats compared with random guessing for C and Python models.....	(20)
Table 4: Quality scores for models across code examples	(21)

1. Introduction

Product automation is a common practice in most engineering disciplines, such as mechanical and electrical engineering. However, Software Engineering lacks a comprehensive set of automation processes and product quality suffers accordingly. Software defects arise from human error resulting in programmers spending 70-80% of their time testing and debugging [1]. Automatic code generation could therefore help reorient developers time to be focused on designing and developing products. Current approaches to code generation are often either difficult to implement or are very restricted in terms of the domain they can generate programs in. For example, rule-based techniques have shown to be very effective [18], [19], [20], however, they are burdened by the effort required to write a comprehensive set of rules. A machine learning approach which shows potential for code generation is Long Short-Term Memory (LSTM) Recurrent Neural Networks (RNN). LSTMs can be used to generate structured text including rap lyrics [2], video captioning [3], and music chord progressions [4]. In this report we address whether LSTMs can be used to generate code and overcome the shortcomings in the current approaches. We hence present the development and evaluation of a novel solution which uses an LSTM neural architect to generate lines for incomplete programs. We call this approach Code By Tensors or CBT.

1.1. Research intent

We intend to develop and evaluate an approach to code generation that uses a machine learning LSTM-based technique to generate lines of code from a partially implemented program. We aim to assess the ability to which this approach can understand the syntax and semantics of a given programming language. This approach is novel for both the code generation and LSTM text generation research domains. Due to its novelty we want to compare our approach to existing approaches and assess its strengths and weaknesses. These strengths will be identified by performing a quantitative and qualitative evaluation of the code generated. Things we will consider in this evaluation include: code quality, how code generated for different paradigm programming languages compare and semantic correctness. We then wish to identify how these strengths might further the domain of code generation.

1.2. Division of work

Table 1: Division of work between Buster Major and Nathan Cairns

Task	Nathan Contributed	Buster Contributed
Research	X	X
Design	X	X
Tokenization		X
Training		X
Generation	X	X
Evaluation	X	
Comparative Analysis	X	X

2. Literature Review

Before commencing this project, we conducted a literature review concerning popular methods in program generation. In interest of brevity some technical detail is omitted.

2.1. General machine learning approaches

2.1.1. Program synthesis

Program synthesis is done by teaching a neural network to generate a program based on I/O training data [5]. This is a difficult program generation technique especially with regards to generating complex programs. For example, in [6] a lot of success was seen when generating very simple programs from I/O snippets taken from programming websites. This was done through combining neural network architectures with search-based techniques, rather than trying to replace them, i.e. using gradient descent to search for the correct programs. [7] Does something similar using machine learning techniques to train a program synthesizer to understand I/O examples (encoding) and to search a program space (decoding). Search approaches are further discussed in 2.2.

2.1.2. Program induction

Program induction is done by training a neural network to be a latent program representation that provides program output based on I/O training data [5]. This is a popular type of generation which has seen a lot of success. For example, in [8] they proposed a well performing generative model for code generation that worked to create code that represents trading cards. However, this form of program generates a latent representation of a program rather than actual code and hence will not be pursued.

2.1.3. *Reinforcement learning*

Reinforcement learning is where a mapping from situations to actions is learnt so that a scalar reward or reinforcement signal is maximized. The learner is not told which action to take, but instead must discover which actions yield the highest reward by trying them [9]. In [10], a set of instructions is taken and transformed into executable instructions. This is done by using reinforcement learning on Microsoft Windows troubleshooting guides and generating the valid actions to complete the steps in the guide. Whilst this is not expressly program generation it is easy to see how programs could be generated using a similar methodology.

2.2. Search machine learning approaches

2.2.1. *Enumerative search*

Enumerative search involves enumerating an entire problem space and then using tree-based searching and pruning to narrow down the most appropriate result to a query. In terms of program generation, this means searching over all possible programs and choosing which one best fits a specification. Sketching is a popular approach where a programmer provides a partial implementation of a program and the model synthesizes the remainder [11]. Sketching is a good method of searching as it helps significantly reduce the search space [12]. Enumerative search is an approach that works well but is best kept to specific domains and therefore smaller search spaces. For example, [13] uses enumerative search to generate search algorithms with the best combination of domain specific heuristics. This is a helpful function but is very limited in terms of scope.

2.2.2. *Constraint based / machine translation*

Machine Translation (MT) tools are computer programs which are capable of automatically producing a translation from a source to target language [14]. If we consider a programming language to be like a natural language with its own grammar and rules, then it may be possible to translate natural language to code. A MT tool able to transfer natural language to code would allow less experienced programmers to easily generate code from a specification. In [15] a Statistical Machine Translation (SMT) framework is used to translate source code into pseudo code. This approach might be able to do the inverse and generate code from pseudo code. In [16] a Java IDE extension was developed using SMT and natural language processing tools. It accepts free-form queries containing a mixture of English and Java as input and synthesizes a list of ranked expressions

in Java. A search method that matches keywords to the IDE's API database was used. However, this approach led to the tool only being able to produce expressions that exclude local variable declarations (loops or conditionals). This problem was addressed in [17] by adding a unit test to the input specification. However, this approach reduced the ease of use by requiring additional specialized input.

2.2.3. *Long short-term memory networks*

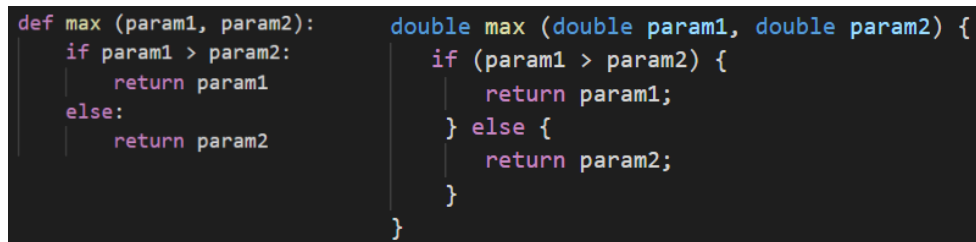
Long Short-Term Memory (LSTM) Networks are a type of Recurrent Neural Network (RNN). RNNs are a Neural Network which has a loop allowing for the persistence of information. This means they are particularly good at learning sequences [20]. LSTMs address the problem of Long-Term dependencies where the current output depends on something that appeared far back in a sequence [21]. This is done by making several changes to the architecture of standard RNNs. LSTMs have been used for many novel applications. For example, the generation of rap lyrics [2], video captioning [3], and music chord progressions [4]. There are not many existing solutions using LSTMs for automatic code generation. The one example is [22] which used LSTMs to solve simple sequence problems. Therefore, using LSTMs for code generation is quite novel. There are two different kinds of LSTMs for processing text. These are character-based and word-based LSTMs [4]. Character-based LSTMs process text as single characters whereas word-based LSTMs process text as whole words. Tokenization is a technique which can help an LSTM learn the structure of a corpus. Tokenization is where tokens are inserted into a source text to indicate structural elements. In [2] they used tokenization to help their model identify verse structure in rap lyrics. Tokenization could be used to help identify several structural indicators in code such as indentations in Python.

2.3. **Rule-based approach**

Rule based program generation is a naïve yet effective approach to generating programs. In this approach a set of rules is defined, and programs are then generated from a well-defined specification language. These rules can be learnt via machine learning [18] or can be manually defined [19]. Machine learning significantly reduces the time spent generating rules as it does so automatically. Current Implementations require a very formal specification language. For example, in [20] a very formal language called Real-Time Process Algebra (RTPA) is used as a source specification language. This create a barrier to usage and diminishes utility.

2.4. Summary of findings

LSTMs have been used to generate text in other domains which have long-term dependencies such as rap lyrics. However, they have not yet been applied to the same capacity for generating programs. Programming languages are highly syntactic and have lots of long-term dependencies. For example, in *Python* indentation levels are vital and depend on previous lines. Furthermore, in *C* an open curly brace must eventually be followed by a closing curly brace. See Fig. 1 for examples of *Python* and *C* code. Hence, LSTMs should be very applicable in the program generation domain. From this conclusion, we plan to explore the feasibility of applying LSTMs to develop a machine learning model which understands the structure of code in *Python* and/or *C*. We then intend to use this model to generate code in a given programming language.



```
def max (param1, param2):  
    if param1 > param2:  
        return param1  
    else:  
        return param2  
  
double max (double param1, double param2) {  
    if (param1 > param2) {  
        return param1;  
    } else {  
        return param2;  
    }  
}
```

Fig. 1. Examples of Python (left) and C (right) code.

3. Proposed Solution: CBT

The solution we propose in this report is a novel system called Code By Tensors or CBT. CBT leverages the power of LSTMs to recognize long term dependencies in code snippets and generate code accordingly. CBT takes as input a partial/incomplete program and outputs its best guess at the completed program by generating the remaining n lines. The architecture of the generation functionality is demonstrated in Fig. 2. As can be seen from Fig. 2 the partial program is passed to the generator. This program is then tokenized so that it can be processed by the model. The process of tokenization is explained in depth in 4.3. Variables and functions are tokenized, and a mapping is saved so they can be untokenized later. Syntax symbols are also tokenized these are hard coded in the tokenizers and therefore do not need to be saved to memory. The model then works to generate the n required lines. Once these lines have been generated the now complete program is untokenized back into a readable format and outputted.

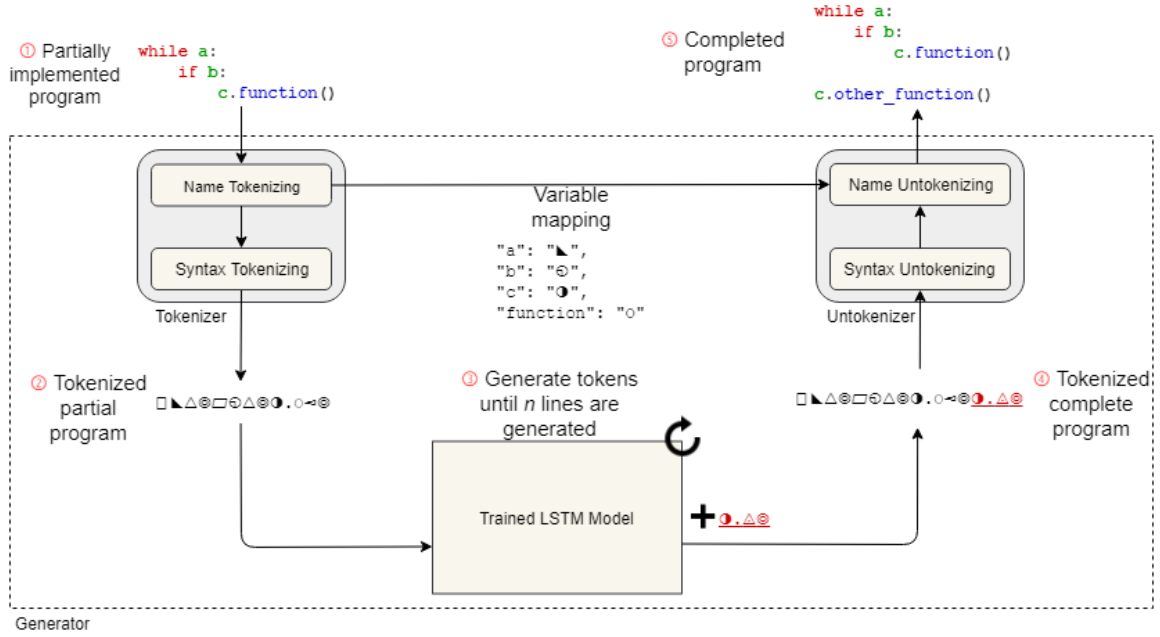


Fig. 2. Architecture diagram of the CBT code generator

3.1. Technologies used

We used Python to develop all necessary scripts as it is simple, quick to write and has a wealth of machine learning frameworks. We used TensorFlow for developing our models as it has out of the box support for training LSTMs and abundant documentation. All training was done on a Microsoft Azure Virtual Machine. We used the 2to3 Python module to convert Python2 code to Python3 code. This was necessary to ensure all Python data was of the same format. We used the Python ast module to process Python code. The libclang python module was used for processing C code. The plotly and matplotlib modules were used for generating graphs for our evaluation.

3.2. Generated languages

We choose to train our model on two programming languages: Python and C. We choose to train on Python first as it has very simple syntax, is general purpose and popular. This led us to believe our model might have an easier time learning the syntax. Our training framework is very general purpose and meant we were easily able to train a model to generate C code after this. We choose C as it has a lot of syntactic differences to Python. This meant it could be used to help evaluate our LSTMs strengths and weaknesses. CBT can learn to generate

code in any language on two conditions: a sizeable dataset is provided and a script is provided to tokenize / untokenized the language.

3.3. Datasets

The first dataset we used was the 150k Python dataset [23] which contains 150 thousand Python files from GitHub. We found the results of training on this data to be lacking as the code generated was often syntactically incorrect. This is most likely due to the high variance and complexity of the code in the dataset. Most programs were part of a Django webservice or some other complex framework. This made it difficult for our trainer to produce a reliable model. Our next dataset was the Codechef Competitive Programming dataset [24] which contains over 1 million code examples in various languages from the Codechef program competition website. It contains approximately 100 thousand C programs and 10 thousand Python programs. This data proved to be much better suited to our problem. This was for two reasons, 1) many examples were solutions to the same problem meaning the scope of the generator was significantly reduced, and 2) the programs were much simpler due to their context (programming competition website). We found this data to be much better for training our LSTMs and continued to use it for the rest of our project.

Some simple preprocessing was done before the data could be used to be trained on. Firstly, any programs written in Python2 had to be updated to be in Python3. This was done to ensure consistency within our dataset and to make all programs readable by Python's `ast` module. Secondly, comments were removed for both C and Python. This was done to ensure the models only learnt actual code and did not learn patterns from the natural language comments.

4. Design and Implementation

4.1. Design

The components of CBT and their interactions are demonstrated in Fig. 3. The *Data* component represents all data used to train our LSTM models i.e. the C and Python datasets. The *train.py* script uses the data to train a LSTM. The *generator.py* script is used for generating code in either C or Python. The *modelmaker.py* script allows the *train.py* script to save *Model Checkpoints*. The *generator.py* script then uses the *modelmaker.py*

script to load a LSTM model from *Model Checkpoints*. The *programtokenizer.py* script provides a framework for tokenizing and un-tokenizing programs. The *train.py* script uses the *programtokenizer.py* script to tokenize programs before using them for training. This is done to make the data easier to digest for the trainer. The *generator.py* script uses the *programtokenizer.py* script to tokenize the incomplete program it is given as input. It then uses the loaded model to generate new lines of code. These generated lines are in tokenized format so must be converted back into human readable form. The *programtokenizer.py* script's un-tokenization functionality is used to do this. An example of generated code can be seen in Fig. 4 where the last 3 lines have been generated using CBT. The implementation details of CBT are further explained in the following section.

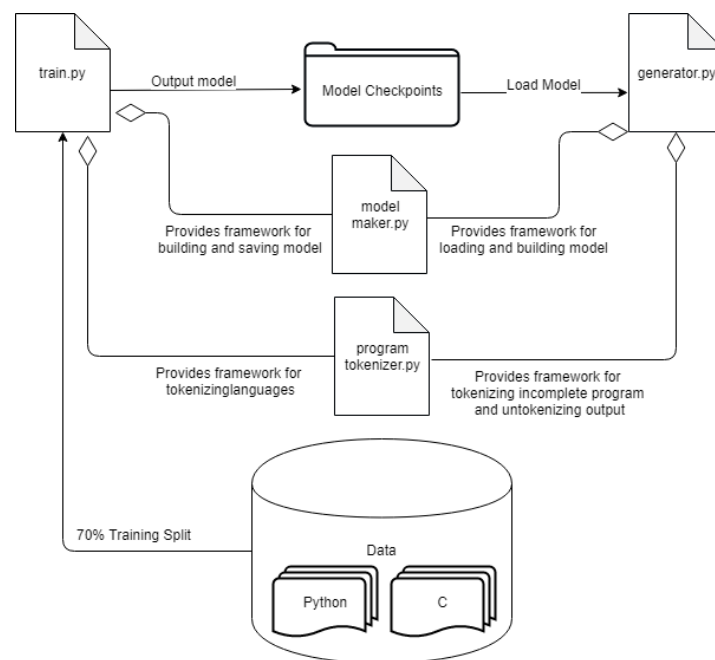


Fig. 3. Implemented scripts and how they interact

```
def max_and_min(arr):
    max = None
    min = None
    for elem in arr:
        if ((min == None) or (min > elem)):
            min = elem
        if (arr == max):
            return True
        if (elem == min):
```

Fig. 4. A program with the last 3 lines generated using CBT

4.2. Training

As stated in 3.1 a Microsoft Azure Virtual Machine was used for training CBT. This virtual machine was provided by the University of Auckland and was accessed using ssh. A VM was used for training as it provides a stable and optimized environment for running our long training cycles of up to 3 or 4 days. We decided to train our LSTMs on a character by character bases. This is what is referred to as a character-based LSTM, as opposed to a word-based LSTM which trains on whole words. We choose to do this because of the incredibly large dictionary size that could appear from doing word-based as every variable would have to be saved to the LSTM's dictionary and there are very few constraints on syntactically correct variable naming. Furthermore, a variable from one program is not relevant to the scope of another program. On the other hand, with a character-based LSTM the number of entries in the dictionary is much more likely to be smaller. Generally limited to how many characters are seen (letters and symbols) and how many tokenized words we have chosen by the language. Whilst the number of characters that could be seen in the training data could theoretically be very high (due to the usage of Unicode characters in strings) it is much more likely to be <100 . For training, we passed complete programs from the dataset mentioned in 3.3. We trained several models on 10%, 50%, 75% and 100% for the Python and C datasets. Each of these divisions of data was trained on 70% of the division and evaluated on the other 30%.

4.3. Tokenization

Tokenization was important as the LSTM models we trained were character based. This meant we could make learning easier by reducing groups of characters which symbolize structural markers in code. This idea came from [2] where tokens were used to help identify verse structure in rap lyrics. For example, a keyword such as “for” in python indicates that loop conditions should follow e.g. “for number in list_of_numbers:”. Tokenizing the “for” keyword means it will be transformed into a single token ψ . This means the model no longer needs to learn the combination of characters which make up the word “for”, it instead only needs to recognize ψ . This lessens the strain on our trainer as it does not need to learn as many sub patterns. See Table 2 below to see the full list of tokenized words for Python and C. We also mapped variables to tokens to make it easier for our models to recognize reoccurring words. This is done before generation by creating a mapping from variable

to token before generation, persisting this in memory and then using it to un-tokenize the program after lines have been generated.

Table 2: Tokenized words in Python and C

Language	Tokenized Character words.
Python	'eof', 'if', '\n', ' ', 'for', 'while', ':', 'False', 'None', 'True', 'and', 'as', 'assert', 'break', 'class', 'continue', 'def', 'del', 'elif', 'else', 'except', 'finally', 'from', 'global', 'import', 'in', 'is', 'lambda', 'nonlocal', 'not', 'or', 'pass', 'raise', 'return', 'try', 'with', 'yield', 'dedent', 'indent', 'abs', 'all', 'any', 'ascii', 'bin', 'bool', 'bytearray', 'bytes', 'callable', 'chr', 'classmethod', 'compile', 'complex', 'delattr', 'dict', 'dir', 'divmod', 'enumerate', 'eval', 'exec', 'filter', 'float', 'format', 'frozenset', 'getattr', 'globals', 'hasattr', 'hash', 'help', 'hex', 'id', 'input', 'int', 'isinstance', 'issubclass', 'iter', 'len', 'list', 'locals', 'map', 'max', 'memoryview', 'min', 'next', 'object', 'oct', 'open', 'ord', 'pow', 'print', 'property', 'range', 'repr', 'reversed', 'round', 'set', 'setattr', 'slice', 'sorted', 'staticmethod', 'str', 'sum', 'super', 'tuple', 'type', 'vars', 'zip', '__import__', 'self'
C	'auto', 'break', 'case', 'char', 'const', 'continue', 'default', 'do', 'double', 'else', 'enum', 'extern', 'float', 'for', 'goto', 'if', 'int', 'long', 'register', 'return', 'short', 'signed', 'sizeof', 'static', 'struct', 'switch', 'typedef', 'union', 'unsigned', 'void', 'volatile', 'while', 'strcpy', 'strncpy', 'strcmp', 'strncmp', 'strlen', 'strcat', 'strncat', 'strchr', 'strchr', 'strchr', 'strchr', 'strtok', 'calloc', 'free', 'malloc', 'realloc', 'memcpy', 'memcmp', 'memchr', 'memset', 'memmove', 'tolower', 'toupper', 'perror', 'strerror', 'printf', 'gets', 'scanf', '==', '!=', '--', '++', '&&', ' ', '-=', '+=', '*=', '/=', '%=', '&=', ' =', '^=', '<=', '>=', '<=>', '<->', '<<', '>>'

The process of tokenizing Python code is outlined in Fig. 5. Firstly, Python code is broken down into an abstract syntax tree (AST) using Python's ast module. This AST representation is then used to break the code down into tokens. All words as defined in Table 2 and variables that appear in the AST are mapped to tokens and these mappings are persisted in memory. These mappings can be used later for un-tokenizing programs. It was particularly important that variable name mappings were stored as these are unique to each program. A fully tokenized program is outputted. C tokenization was done similarly to Python tokenization, the main difference being a clang Python module is used to extract variables and keywords from the C programs.

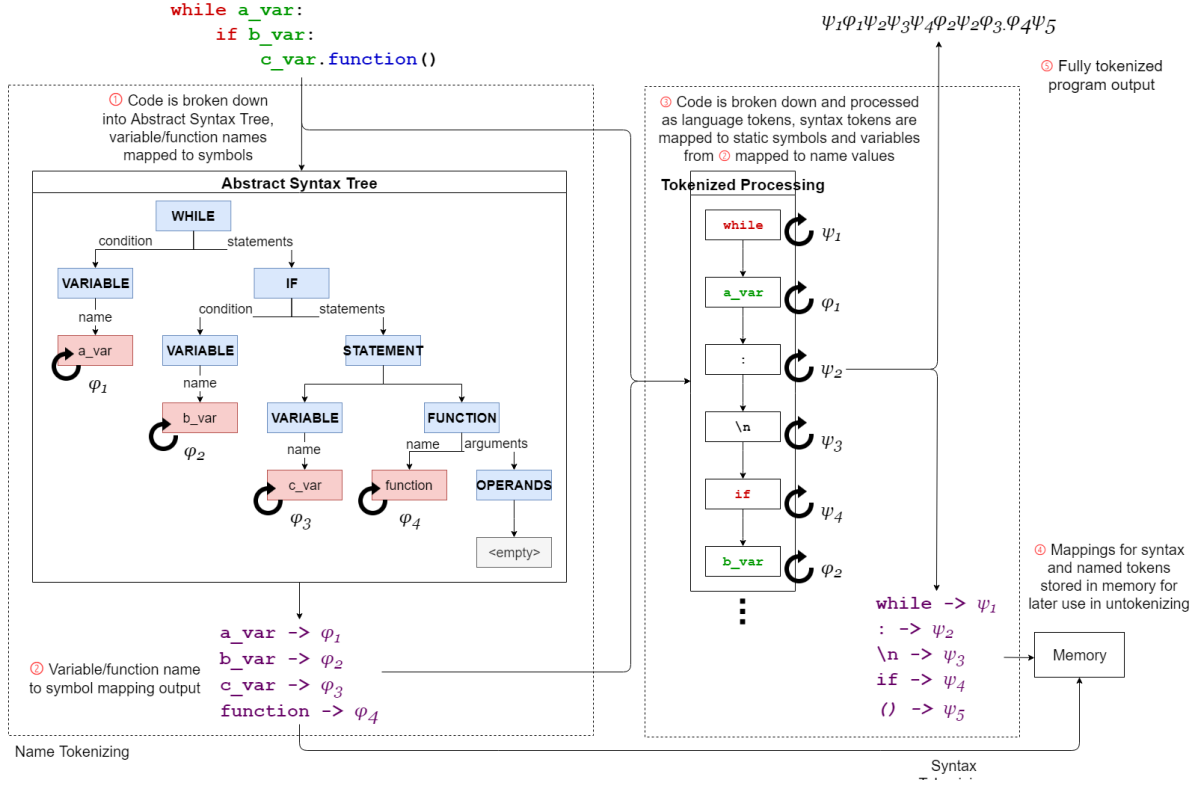


Fig. 5. Python tokenization

5. Evaluation and Results

5.1. Evaluation framework

We set up an evaluation framework for assessing the performance of our trained CBT models. The design for this framework is outlined in Fig. 6. The framework takes as input several parameters specifying how the framework should be run. This includes what model to use, what language to generate code for, the amount of lines to generate and the number of files to use in the evaluation. All preprocessing and gathering of statistics are done in the *evaluate.py* script. This preprocessing includes reading in the evaluation dataset, removing the last n lines and generated n lines in their place. This script then uses an evaluator from the *evaluator.py* script to gather metrics on generated lines. There is an evaluator for each language that has a model trained (C and Python). They extend a parent evaluator meaning more language evaluators can easily be added due to their polymorphic usage in the *evaluate.py* script. The *evaluate.py* script puts all the statistics together from the evaluator it used, displays this to the console and writes it to a statistics file.

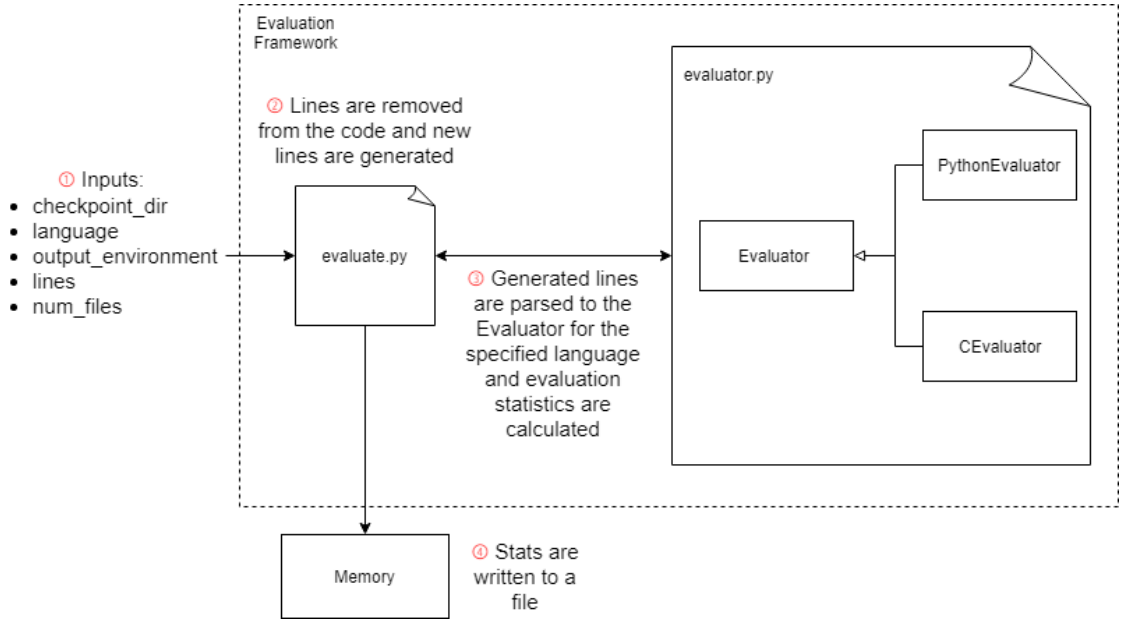


Fig. 6. Evaluation framework design

5.2. Results

This section looks at the results of using our evaluation framework on several C and Python models. The Python models were trained on a dataset of up to 10,000 instances and the C models were trained on a dataset of up to 100,000 instances. For both the Python and C datasets models were trained on 10%, 50%, 75%, and 100% of the data. This was done to see how models improve with more data. For the following sections the word ‘*equivalent*’ means the Python model trained on 100% of the Python data and the C model that was trained on 10% of the C data. Evaluation was done by removing the last n lines from a program and comparing them with a corresponding n generated lines.

5.2.1. Syntactic program correctness

Syntactic correctness refers to a valid program in a programming language. C is a compiled language and valid code is referred to as compilable (can be compiled by the compiler). Python is an interpreted language and valid code is referred to as interpretable (can be interpreted by the interpreter). We use a programs compilability / interpretability to measure CBT’s ability to write valid code. Python code was determined to be interpretable by passing the generated code through a linter and identifying critical errors. C code was determined to be compilable by running generated code through a compiler and seeing if it compiles. If we

look at Fig. 7 we can see Python outperformed the equivalent C model with ~25% generated code being executable whereas the C model generated ~5% executable code. This is expected as C has a much stricter syntax than Python. We can also see that generally as the number of training instances increases so does the model's ability to generate executable code. Furthermore, as more lines are generated, the executability of generated code is reduced. The best performing model was the 100% C model with an impressive 30% of generated programs being executable for 1 line generated.

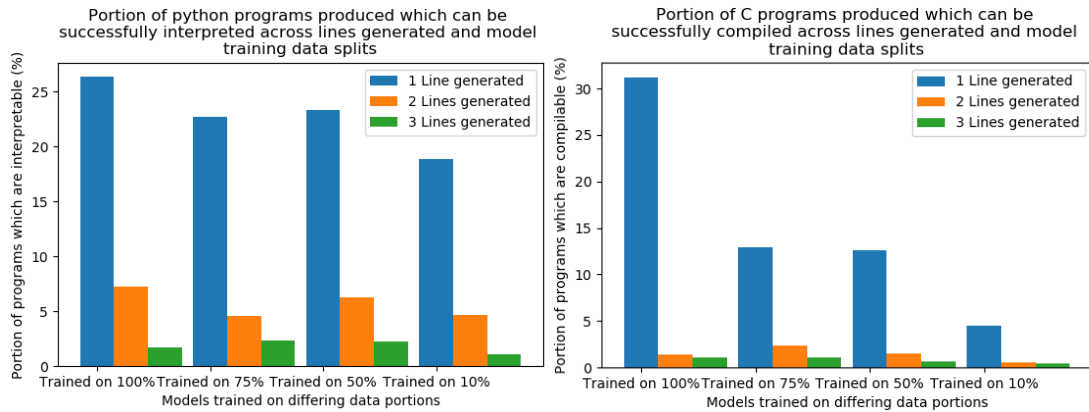


Fig. 7. Program executability percentage when asked to generate 1, 2, and 3 lines for Python (left) and C (right)

5.2.2. Accuracy of keyword / variable guessing

The accuracy of the next keyword / variable is calculated by checking if the first keyword / variable in the generated lines matches the first keyword / variable in the original lines. As can be seen in Fig. 8 the equivalent Python and C models performed equivalently when generating the next keyword with a ~25% success rate. The 100% C model performed particularly well with an accuracy of ~40%. With regards to variable guessing the 100% Python model outperformed the equivalent C model with ~15% vs ~5%. The 100% Python model was the best performer with ~15% accuracy.

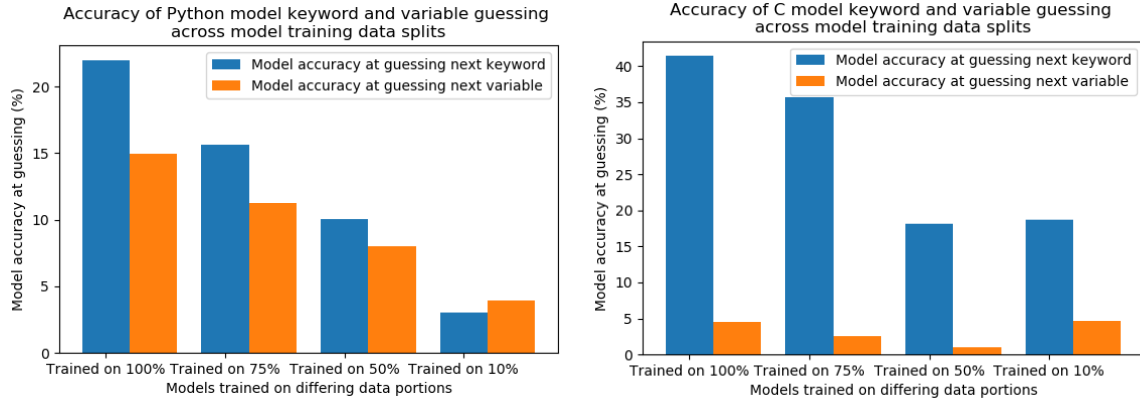
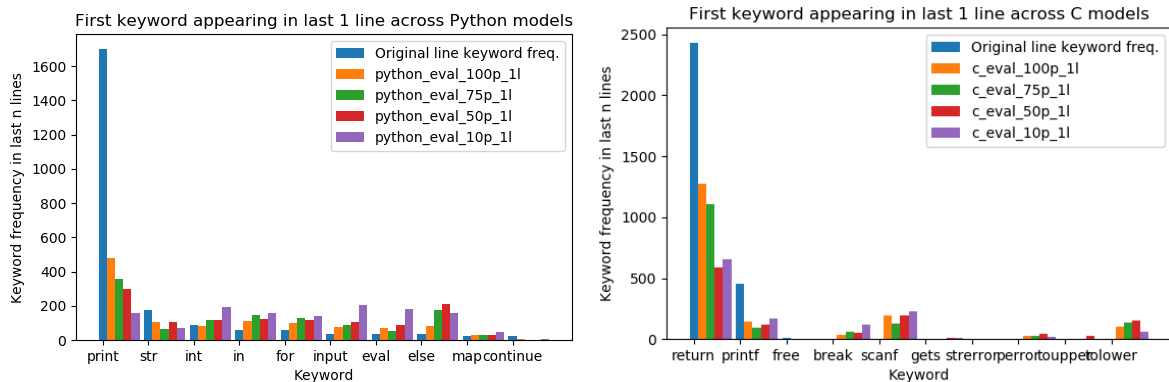


Fig. 8. Accuracy of guessing next keyword and variable for Python models (Left) and C models (Right)

Fig. 9 demonstrates how our trained models perform at guessing keywords vs the actual distribution. From Fig. 9 we can see that both models started to understand the context of where keywords should be used. For example, the first keyword the C model generated when generating the last 3 lines was much more likely to be *if* or *else* than when asked to generate just 1 line. This indicates that the C model may be starting to become aware of the context of the lines it is generating and adjusts its keyword predications accordingly. Whilst this is indicative of a very exciting phenomenon, it is also made apparent from Fig. 9 the limitations of our models. For example, in the top-left graph, the *print* keyword appears ~1700 times in the original line. However, *print* only occurred ~500 times in our best performing Python model. This shows that whilst there is a trend of our models getting closer to the true distribution with more training instances, it is evident there is still a lot more work that could be done.



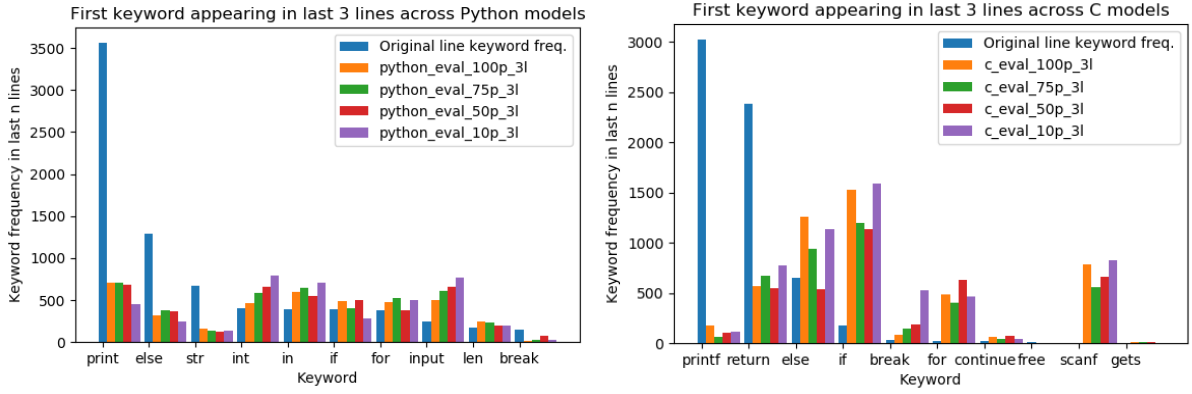


Fig. 9. First appearing keyword frequency in original files and model training splits across Python models tested on generating last 1 line (Top-Left), C models tested on generating last 1 line (Top-Right), Python models tested on generating last 3 lines (Bottom-Right)

The large disparity between variable guessing between the equivalent C and Python models is most likely due to the C model being far more likely to introduce new variables when generating the last line of the program. This phenomenon is best illustrated by Fig. 10. We are unsure why the C model does this. However, we are certain this is the cause of the C models poor performances at variable guessing.

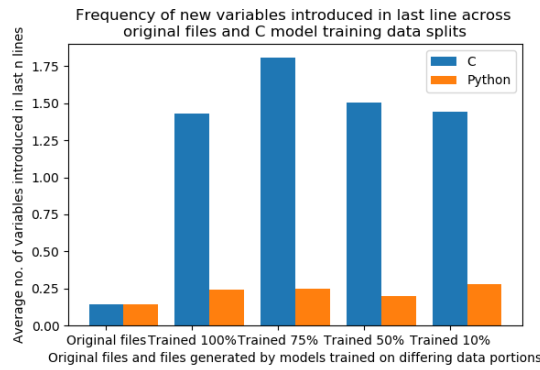


Fig. 10. Frequency of new variable introduction in last line of program across Python and C models

Table 3 shows how the best performing models' variable / keyword guessing compares to random guessing. Random guessing for keywords was determined by the chance of randomly selecting any keyword from the language. Variable guessing for a given program was determined by the number of distinct variables occurring before the last n lines, with 1 added to account for a new variable appearing on the last line. This measure is useful as it shows us how well CBT has learnt the keyword and variable usage for each language. Both models appear to significantly outperform random guessing for keyword generation. On the other hand, the Python

model was slightly better than random at guessing variables whilst the C model was worse than random guessing. This indicates the models were starting to grasp the usage patterns of more universal constructs such as keyword usage, however struggled with the local usage patterns such as variable usage.

Table 3: Keyword and variable stats compared with random guessing for C and Python models

	Python Model			C Model		
	Random guessing	Generated	Δ	Random guessing	Generated	Δ
Keywords	0.93%	22.00%	21.07%	1.69%	18.72%	17.03%
Variables	9.72%	14.93%	5.21%	7.06%	4.62%	-2.44%

5.2.3. Empirical evaluation

We carried out an empirical evaluation on the C model trained on 100,000 examples and the Python model trained on 10,000 examples. Three non-whitespace, non-bracket lines were removed from five programs from each language. Three lines were then generated and evaluated empirically. The programs used were simple algorithms such as calculating an average and finding the maximum number in an array. These lines were evaluated across four different criteria, 1) appropriate use of new variables, 2) quality of the syntactic style, 3) use of variable types in the correct location, and 4) the overall semantic quality. Each criterion was rated either *poor*, *average* or *good* based on my project partner and I's programming experience.

The results of this evaluation can be seen in Table 4. From this evaluation it appears that neither model significantly outperforms the other. One common issue we found was that models tended to introduce new variables without first declaring them. This was most common in programs which had few variables declared before the generated lines. We think this might have occurred due to our models learning that programs typically have several variables and assuming one was declared beforehand. Both models used functions and variables interchangeably. Whilst this is possible in Python as it is dynamically typed, it is not possible in C due to its stricter type system. Neither model shows much higher-level understanding of coding semantics. The lines that are generated appear to be more random than deliberate and not help the overall goal of the program. In summary, our models appear to generate reasonably correct code, but fail to properly use variables or to understand important program semantics.

Table 4: Quality scores for models across code examples

Program	Python Model					C Model				
	GCD	LCM	Factorial	Gnome sort	Merge sort	Is Prime	Recurse	Array max	Transpose	Average
New Variables	poor	poor	poor	good	good	poor	average	poor	poor	poor
Syntax/Styling	good	poor	average	poor	good	good	good	good	average	poor
Var. type use	average	average	average	good	average	good	average	good	poor	average
Semantics	average	average	poor	average	average	poor	average	poor	poor	poor

5.3. Discussion

From the results mentioned in the previous sections it is evident that there are significant limitations to our models and a lot of room for improvement. The perceived improvement of performance as the number of training examples increases indicates more data would likely lead to improved models. We believe this could show significant improvements with little changes to the approach. It is also made apparent that our models are okay at learning syntax and show a lot of promise at improving their understanding as more data is provided. On the other hand, our models show significant limitations in understanding variable scope, typing, and program semantics. Our models perform better on dynamically typed languages; however, this is likely due to less strict language rules and not due to proper usage. Our models poor handling of scope, typing, and semantics can be attributed to short comings in our tokenization. Their performance could hence likely be improved with better tokenizing of variables and functions. For example, if we could encode more information regarding context and type our models might be able to begin to better understand a variable/function's context.

We saw significant limitation in quantitatively analyzing the semantic correctness of the code we generated. This is mostly due to our data having no I/O examples alongside them. This meant we could only compare to other syntactic and empirical evaluations. We found with regards to syntactic correctness a lot of better performing studies had either access to a lot more data or a much smaller scope. This allowed their approaches to be a lot less general and generally better than our 27% correct for Python and 32% for C. For example, in [8] they plugged missing statements into templated code and saw 100% syntactic correctness. Furthermore, [5] were able to reach 90% syntactic and semantic correctness. [15] used statistical machine translation to generate pseudo code from code. They saw about 50% semantic acceptability; however, their criteria was generally less strict as they were generated pseudo code. If we use Table 4 for reference, we see our models

had about 20% semantic acceptability meaning [15] also outperforms our models semantically. We saw similar results to [4] where their LSTMs learnt the structure of basic music patterns but struggled to learn longer drum scores. Our models were also outperformed by [2] where they found their generated rap lyrics to be analogous to the training data. Our generated code on the other hand failed to be analogous to the training data. Whilst our approach seemingly performed quite poorly compared to other literature it is important to note that our approach is very novel, hence has a lot of potential to be further developed.

6. Conclusions & Future Directions

To conclude, we have developed and evaluated a novel approach to code generation using machine learning called CBT. CBT uses LSTMs and tokenization to generate code in both Python and C. We have demonstrated that CBT shows promise at understanding a programming languages syntax with 27% of generated Python code being interpretable and 32% of C code being compilable. Furthermore, our models' ability to guess the next keyword in a program reaches as high as 40% and its ability to guess the next variable is as high as 15%. A major shortcoming of our approach is its ability to understand context and semantics of a program leading to a lot of incorrect usages of named constructs. Our approach performed poorly when compared with other approaches to code generation. This is mostly due to our expanded scope of generated code and a lack of data. Furthermore, our model performed averagely against other LSTM text generation approaches. This is mostly due to the high ceremony of programming languages compared to music notation or natural language. The solution we present in this report and the results of our evaluation should be considered as a bench mark for purely LSTM based code generation approaches. We have identified and evaluated a novel machine learning approach to code generation which shows promise for further research. Furthermore, we have demonstrated how LSTM text generation techniques can possibly be used to generate more complex sequences.

There are several future directions which could be taken to expand on the research presented in this report. The main developments we would like to see is the further development of our tokenization approach. For example, the scope of a variable could be somehow included in the token. This might make it easier for the LSTM model to recognize when a variable can and cannot be used. Furthermore, even more information might be able to be encoded in tokens. For example, the type of variable might be able to be encoded. This might

help the LSTM recognize when certain types of variables can be used. For example, only variables that are iterable could appear after an “in” keyword in Python.

Another direction that could be taken with CBT is training and evaluating its performances on other programming languages which use different programming paradigms than Python and C. For example, a functional programming language such as Prolog or a high ceremony object-oriented language like Java. Performing analysis of CBT’s performance on different programs would give us further enlightenment on how LSTMs perform on languages with different structures.

Lastly, we would like to see our evaluation framework be extended to include an evaluation of code style. This was something we initially intended to include in our evaluation but unfortunately did not have time to implement. We believe this would provide a lot of extra knowledge about the performance of CBT. Whilst our current evaluation framework tells us a lot, evaluating whether CBT can generate stylistically correct code and where its weak points are would be a valuable addition to assessing its performance.

Acknowledgements

I would like to thank the following people for their contributions to CBT and this report: Buster Major for being a great project partner throughout the process of developing and evaluating CBT, Jing Sun for being a supportive project supervisor and providing invaluable guidance throughout this project and Chenghao Cai for providing great explanations and advice with regard to various machine learning techniques.

References

- [1] F. Huang, B. Liu, & B. Huang, “A taxonomy system to identify human error causes for software defects”, July 2012, doi: 10.13140/2.1.4528.5445.
- [2] P. Potash & A. Romanov, & A. Rumshisky, GhostWriter: Using an LSTM for Automatic Rap Lyric Generation, 2015, doi: 1919-1924. 10.18653/v1/D15-1221.
- [3] Y. Yang et al., "Video Captioning by Adversarial LSTM," in IEEE Transactions on Image Processing, vol. 27, no. 11, pp. 5600-5611, Nov. 2018. doi: 10.1109/TIP.2018.2855422
- [4] K. Choi & G. Fazekas & M. Sandler, “Text-based LSTM networks for Automatic Music Composition”, 2016.
- [5] J. Devlin, J. Uesato, S. Bhupatiraju, R. Singh, A. Mohamed, & P. Kohli, “RobustFill: Neural Program Learning under Noisy I/O”, ICML, 2017
- [6] B. Matej, G. Alexander L., B. Marc, N. Sebastian, & T. Daniel, “DeepCoder: Learning to Write Programs”, ArXiv, Nov. 2016, abs/1611.01989.
- [7] R. Singh, & P. Kohli, “AP: Artificial Programming”, SNAPL, 2017, doi: 10.4230/LIPIcs.SNAPL.2017.16.

- [8] W. Ling, E. Grefenstette, K. M. Hermann, T. Kočiský, A. Senior, F. Wang, & P. Blunsom, “Latent Predictor Networks for Code Generation”, ArXiv, 2016, doi: 10.18653/v1/p16-1057.
- [9] R. Sutton, “Reinforcement Learning”, 1992.
- [10] B. Satchuthanan R., C. Harr, Z. Luke S., & B. Regina, “Reinforcement Learning for Mapping Instructions to Actions”, Proceedings of the Joint Conference of the 47th Annual Meeting of the ACL and the 4th International Joint Conference on Natural Language Processing of the AFNLP: Volume 1 - Volume 1 (ACL '09), Vol. 1, 2009, ISBN: 978-1-932432-45-9.
- [11] A. Solar-Lezama, L. Tancau, R. Bodik, S. Seshia, & V. Saraswat, “Combinatorial sketching for finite programs”, SIGPLAN Not. 41, 11, pp 404-415, Oct. 2006, doi: <https://doi.org/10.1145/1168918.1168907>
- [12] J. K. Feser, S. Chaudhuri and I. Dillig, "Synthesizing data structure transformations from input-output examples," ACM SIGPLAN Notices, vol. 50, 2015, pp. 229-239, doi: 10.1145/2813885.2737977.
- [13] S. Minton and S. R. Wolfe, "Using machine learning to synthesize search programs," Proceedings KBSE '94. Ninth Knowledge-Based Software Engineering Conference, Monterey, CA, USA, 1994, pp. 31-38, doi: 10.1109/KBSE.1994.342680
- [14] T. Poibeau, “Machine Translation”, MIT Press, 2017.
- [15] Y. Oda, H. Fudaba, G. Neubig, H. Hata, S. Sakti, T. Toda, & S. Nakamura, "Learning to Generate Pseudo-Code from Source Code Using Statistical Machine Translation (T)," 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), Lincoln, NE, 2015, pp. 574-584, doi: 10.1109/ASE.2015.36
- [16] T. Gvero and V. Kuncak, "Synthesizing Java expressions from free-form queries," ACM SIGPLAN Notices, vol. 50, (10), pp. 416-432, 2015. . DOI: 10.1145/2858965.2814295.
- [17] A. Cozzie and S. T. King, “Macho: Writing Programs with Natural Language and Examples,” Urbana-Champaign, 2012.
- [18] J. Y. Xu and Y. Wang, "Towards a Methodology for RTPA-MATLAB Code Generation Based on Machine Learning Rules," 2018 IEEE 17th International Conference on Cognitive Informatics & Cognitive Computing (ICCI*CC), Berkeley, CA, 2018, pp. 117-123, doi: 10.1109/ICCI-CC.2018.8482093.
- [19] A. Sethi, A. Sankaran, N. Panwar, S. Khare, M. Kumarasamy, & S. Kumar, “DLPaper2Code: Auto-generation of Code from Deep Learning Research Papers”, CoRR abs/1711.03543, 2017.
- [20] B. Hammer, “Learning with Recurrent Neural Networks”, Springer, London, 2000, doi: <https://doi-org.ezproxy.auckland.ac.nz/10.1007/BFb0110016>
- [21] F. A. Gers and E. Schmidhuber, "LSTM recurrent networks learn simple context-free and context-sensitive languages," in IEEE Transactions on Neural Networks, vol. 12, no. 6, pp. 1333-1340, Nov. 2001. doi: 10.1109/72.963769
- [22] I. Sutskever, O. Vinyals, Q. V. Le, “Sequence to Sequence Learning with Neural Networks”, in Proc. Adv. Neural Inf. Process. Syst., 2014, pp. 3104–3112.
- [23] “150k Python Dataset”, Zürich: ETH Zürich, 2018, available: <https://eth-sri.github.io/py150>
- [24] A. Sharma, “Codechef Competitive Programming”, ver. 5, Kaggle, available: <https://www.kaggle.com/arjoonn/codechef-competitive-programming>