CS230

Robot Apocalypse: Generating Code Snippets from Natural Language Descriptions

Jinhie Skarda
jskarda@stanford.edu

Janette Cheng
jancheng@stanford.edu

Priyanka Sekhar
psekhar@stanford.edu

Abstract

Automated source code generation from high level natural language descriptions could significantly lower the barrier to building software and increase engineering efficiency. In this project, we investigate how natural language translation tools can be extended to the task of python code generation. We use an LSTM with Luong attention, and our results are comparable to state of the art Seq2Seq models specifically designed for code generation tasks. Our word-level model generally outperformed our character-level model, achieving BLEU scores of 46.20/33.53/31.53 on the train/dev/test sets. We found that structure is important for code generation and syntax knowledge could be further exploited in future work to improve performance.

1 Introduction

Successful source code generation from natural language descriptions has the potential to lower the barrier to entry for software engineering, enhance IDE autocomplete features, reduce simple engineering errors, remove redundancy, and in the long run, reduce the time it takes to create complex software. There is already a large body of existing research on human language translation tasks (e.g English to French). If human language translation paradigms can be extended to code generation, which is fundamentally a "translation" between high level descriptions and computer language, then the task of code generation may be able to advance quickly by leveraging parallel advancements in human language translation.

Our project investigates whether human language translation paradigms can be extended to the task of python source code generation from high-level pseudocode descriptions. We use an open source Sequence to Sequence (Seq2Seq) model originally developed for natural language translations, with pseudocode snippets as the inputs and and python code snippets as the outputs. We use BLEU scores to compare models.

2 Related work

Traditional human language translation often relies on Seq2Seq models such as the Long Short-Term Memory models used by Sutskever et al.[1] The LSTM shows improvement over other Statistical Machine Translation approaches and other RNNs. Consequently, we decided to use the Seq2Seq LSTM for our task with Luong [2] attention - which shows a significant BLEU score gain over non-attention models and offers significant vectorization speed up over Bahdanau [3] attention.

While we investigated Reinforcement Learning for this task as described by Nguyen et al. [4], we ultimately decided against this approach. Despite modest gains using RL, because we must translate line-by-line, code may not compile even if it is correct. Thus an automated compile check would not provide useful feedback for the reward states in this method.

Oda et al. [5] released the dataset used for their task of python code to psuedocode summarization, and their model achieves a relatively high BLEU score. The task of summarization is more lenient than code generation, so we do not expect to match their performance, but their public dataset was key to our project.

A shortcoming of the methods above lies in their inability to take advantage of code structure. Unlike natural language, code must adhere to strict syntactical patterns. Models that are built specifically for the task of code generation rely on the Abstract Syntax Trees (ASTs) [6], [7] used by compilers. Because we wish to investigate the generalizability of human language translation to computer code generation, we decided to use techniques that did not enforce code specific characteristics.

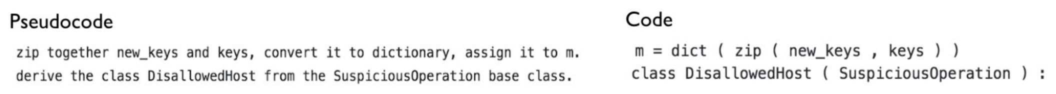
FIG. 1 shows a summary of code generation BLEU scores in existing literature [6],[8].

Retrieval	RL	SEQ2SEQ Code Gen	SEQ2TREE
18.6	24.94	35.9	44.6

Figure 1: Comparison of results for code generation task

3 Data and Preprocessing

We are using the dataset released by Oda et al.[5] - which contains 18,805 pairs of python code and corresponding English pseudocode-like annotations. FIG. 2 shows examples from the dataset. For our code generation task, We use the annotation files as the inputs and the code files as the targets, where each pseudocode line corresponds to a python code line.



Pseudocode	Code
zip together new_keys and keys, convert it to dictionary, assign it to m.	m = dict (zip (new_keys , keys))
derive the class DisallowedHost from the SuspiciousOperation base class.	class DisallowedHost (SuspiciousOperation) :

Figure 2: Two examples of pseudocode and its corresponding python code from the dataset.

Before training our models, we perform a series of NLP preprocessing steps. We first normalize punctuation, which removes the additional spaces from differing indentations in the python code. We then tokenize to split each line into a series of chunks. We experimented with tokenizing the python code by word and by individual character in order to train and compare word-level and character-level models. Next, we learn and apply Byte-Pair Encoding (BPE), which replaces consecutive character pairs with a new single character in order to reversibly compress the data and remove unnecessary complexity. Then, we create a vocabulary for both the inputs and outputs from their entire respective preprocessed datasets. Finally, we shuffle the preprocessed datasets and split the data 80/10/10 to form our train, dev, and test sets.

4 Methods

We use an open source LSTM model designed for English-to-French translation [9] and modify it to take a line of psuedocode as input and output a line of python code. Our Github can be found at `https://github.com/janettec/cs230-robot-apocalypse`. We use Tensorflow and Tensorboard for model modification and loss visualization [10].

FIG 3 shows the encoder/decoder framework of our model. The encoder learns a vector representation of the psuedocode snippet which the decoder then takes as input to learn the code output. The encoder and decoder are trained simultaneously, and the model can have variable length input and output. Because of the separate forget and input gates, LSTMs have plenty of versatility for both long and short input sequences and prove a strong framework for this task.

$$L(y, \hat{y}) = -\sum^{M} y_j log(\hat{y}_j)$$

To train the model, we used Adam optimization and the softmax cross entropy loss above, where M is the number of classes (or vocabulary size), and $y_j \hat{y}_j$ are the label/probability for the jth class for example y.

We use Luong attention, which calculates global attention by using the output from the encoder and decoder for the current time step. The generated context vector and the target decode are concatenated and transferred using the tanh function [2].

Finally, we use the global Bilingual Evaluation Understudy (BLEU) [11] score to compare the decoded outputs to the actual outputs, where 100 indicates a perfect translation and 0 indicates no correlation.

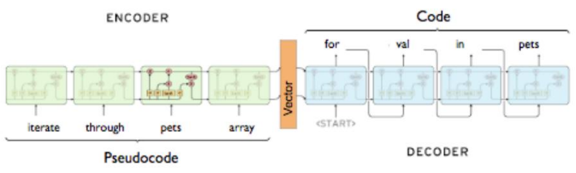


Figure 3: LSTM Architecture [12].

5 Experiments, Results and Discussion

We conducted a coarse hyperparameter sweep on a baseline word-level model to inform fine tuning. We then preprocessed data as described above and trained a character-level model, a weighted loss character-level model, and a word-level model. The insights from the hyperparameter sweep and the details of each model are discussed below.

5.1 Hyperparameter Sweep

We used a subset of the shuffled data for hyperparameter tuning with a train set of 1000 examples and a dev set of 100 examples to quickly assess a wide range of hyperparameters. We swept learning rates over the range [0.0001,1], depths over the range [1,8], number of hidden units over the range [500,1024], dropout rate over the range [0,0.95], and decoder beam width in the range [1,5]. Figure 5 shows a subset of results illustrating the overall trends we observed in the sweep.

hidden_units=1024; embedding_size=500; num_encoder_symbols=6909;
num_decoder_symbols=4617; batch_size=128; lr_decay=exponential

Depth	Dropout	Learning Rate	Train Loss	Train BLEU	Dev BLEU
8	0.26	0.47943	2.36E+07	0.0	0.0
1	0.78	0.00568	21.19	0.0	0.0
7	0.42	0.0015	2.439	6.86	7.88
3	0.09	0.00022	0.3668	9.70	6.09

Figure 4: Results of coarse hyperparameter sweep

The learning rate was the most impactful hyperparameter. As evident in the table above, learning rates close to 0.0002 produced the best train losses and BLEU scores, and learning rate above 0.005 tended to cause the loss to blow up. It is also worth noting that the loss and BLEU score correlate well: when the loss is large, BLEU score is low and vice versa.

Using our insights from this coarse sweep, we fixed the learning rate to be on the order of 10^{-4} and performed a finer sweep for both the character and word-level models. For the word-level model, we found that networks with a moderate depth of 4-6 layers, 1024 hidden units, and dropout rates above 0.6 gave the best, most generalizable models. For the character-level model, we found that deepers networks (8 or 9 layers) with 512 hidden units and dropout rates around 20-50% gave the best performance on train and dev. The final hyperparameters used for the character and word-level models are given in table 5, and the performance of these models is discussed in the following two sections.

Model	Learning Rate	Depth	Hidden Units	Dropout
Character-level	0.00012	8	512	0.25
Word-level	.0002	4	1024	0.66

Figure 5: Parameters used to train models discussed in sections 5.2 and 5.3

5.2 Character-level Model

Single characters in code often represent what would be word- or phrase-level concepts in natural language. For example, given a pseudocode string like 'define the function foo with parameter bar,' the parentheses in the target output '`def foo(bar):`' might map to the phrase 'with parameter.' With this intuition, we decided to train a model using character-level output.

After training some initial models, we noticed the output was dominated by the most common characters. To ameliorate this tendency, we implemented and re-trained our models using an inverse-character-frequency-weighted softmax cross entropy loss. Our weight w_{c_i} for each class/character c_i with frequency f_{c_i} was calculated by taking the inverse of frequency and then normalizing such that all weights added to 1: $w_{c_i} = \frac{\frac{1}{f_{c_i}}}{\sum_j \frac{1}{f_{c_j}}}$.

Multiplying the cross-entropy loss with these weights forces the model to pay more attention during training to rarer characters. The weighted loss did help for specific cases with relatively rare characters. Example 1 in FIG. 6 shows how the same model trained using the weighted loss correctly includes the '`**`' commonly found before '`kwargs`.' Models trained with the weighted loss also tend to include more capital characters and digits in the output (Example 2 in FIG. 6).

Example 1:
Pseudocode: define the function close_caches with dictionary pair of elements kwargs as argument.
Target code: `def close_caches(** kwargs):`
Unweighted output: `def clear():`
Weighted output: `def che(**kw):`

Example 2:
Pseudocode: CRITICAL is integer 50.
Target code: `CRITICAL = 50`
Unweighted output: `self._cache()`
Weighted output: `INGUG = 250000`

Figure 6: Comparison of outputs of same model trained using weighted and unweighted loss

Unfotunately, degenerate outputs were also more common from models trained with the weighted loss (e.g. outputting 50 `!`'s because exclamation points are relatively rare). Thus, our best character-level model was still trained using normal softmax cross entropy and the hyperparameters in table 5, achieving **BLEU scores of 38.84, 32.52, and 29.83** on train, dev, and test, respectively.

An overarching issue with the character-level models we trained was an inability to accurately capture the structure of code. To address this issue, we also tried using a word-level model (see next section).

5.3 Word-level Model

Since there is a lot of word-level structure in code that is not captured with the character-level model, we also trained a word-level model using the hyperparameters given in table 5. To assess how well this word-level model captures the code structure, we visualized the attention weights to study the pseudocode-to-code mappings the model was learning. FIG.7 shows two examples of the attention matrices for similar pseudocode inputs. These examples illustrate the learned mapping trends in our data. For example, the model consistently learns that 'convert' in pseudocode should map to '.' for a function call or '=' for a variable assignment in python pseudocode, and 'to' in pseudocode tends to indicate a change of letter-case (uppercase or lowercase).

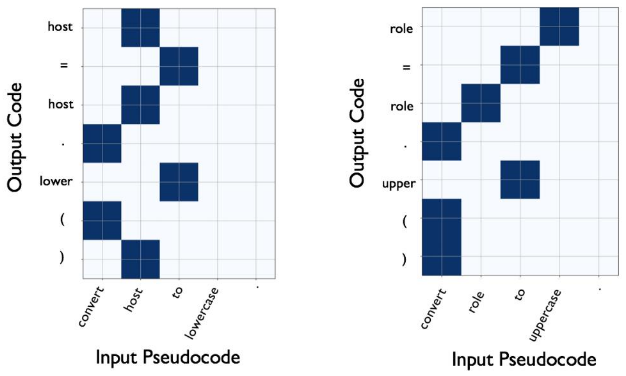


Figure 7: Visualization of attention weights for two examples of pseudocode inputs and the generated python code output.

We obtained the best results for the word-level model using Greedy search, likely because this model does well interpreting complex pseudocode inputs. This consistently observed trend is illustrated in the first example in FIG.8, where the model correctly generates the target python code containing variable assignments, function calls, and function arguments all at once. However, the weakness of our Greedy search model is that, although it does very well with individual code clauses, it cannot handle multiple code clauses and simply terminates after the first clause. As illustrated in Example 2 in FIG.8, when we increase the beam width to 5, the model is able to generate code related to all the multiple code clauses but the generated code for each clause is significantly less accurate.

Example 1:
Pseudocode: call the method tuple . _ _ getitem _ _ with 2 arguments : self and key , substitute the result for value .
Generated code: `value = tuple .  _ _ getitem _ _ ( self , key )`

Example 2:
Pseudocode: if y and m and d are all true
Target code: `if y and m and d :`
Generated code (Greedy search): `if y and d :`
Generated code (Beam search): `if y and isinstance ( y , m ) :`

Figure 8: Examples of the python code outputs our word-level model generates for the given pseudocode inputs.

Our word-level model achieved **BLEU scores of 46.20, 33.53, and 31.53** on the train, dev, and test, respectively. By comparing our performance with the performance of the models discussed in the Related Work section in FIG. 1, we find that our model outperforms probabilistic NLP approaches like retrieval, achieves comparable performance to Seq2Seq code generation methods, and is outperformed by sequence-to-tree methods.

6 Conclusion and Future Work

This investigation demonstrates that human language translation tools can be effective in the task of source code generation from high level natural language descriptions. Using an open source Seq2Seq framework developed for natural language translation, we achieved results comparable to Seq2Seq models specifically designed for code generation [6]. The word-level model (BLEU 31.53) outperformed the character-level model (BLEU 29.83) likely due to inherent structure preservation. The best models tend to use ASTs, and thus incorporating tree representations could provide a significant increase in performance. The predicted python code from the AST-based model in Yin and Neubig achieves a BLEU score of 84.5 on the Oda et al. python dataset [6] and is considered state of the art for this specific task. Future work would focus on incorporating code-specific structure into the human language translation tools such as the LSTM.

7 Contributions

All three of us trained, tuned, and implemented LSTMs on GCP. We each decoded and preprocessed data and assisted each other in model modifications and parsing tasks. In addition, the group members made the following major individual contributions:

Janette Cheng: Character-level model training and tuning, weighted loss implementation, GPU setup.
Jinhie Skarda: Word-level model training, coarse sweeping, AWS and GCP setup, attention matrix visualization.
Priyanka Sekhar: Coarse sweeping, baseline model setup, hyperparameter summarization, BLEU score summarization, Tensorboard/average loss setup and visualization.

References

[1] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pages 3104–3112, 2014.

[2] Minh-Thang Luong, Hieu Pham, and Christopher D Manning. Effective approaches to attention-based neural machine translation. *arXiv preprint arXiv:1508.04025*, 2015.

[3] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

[4] Khanh Nguyen, Hal Daumé III, and Jordan Boyd-Graber. Reinforcement learning for bandit neural machine translation with simulated human feedback. *arXiv preprint arXiv:1707.07402*, 2017.

[5] Yusuke Oda, Hiroyuki Fudaba, Graham Neubig, Hideaki Hata, Sakriani Sakti, Tomoki Toda, and Satoshi Nakamura. Learning to generate pseudo-code from source code using statistical machine translation (t). In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*, pages 574–584. IEEE, 2015.

[6] Khanh Nguyen, Hal Daumé III, and Jordan Boyd-Graber. Reinforcement learning for bandit neural machine translation with simulated human feedback. *arXiv preprint arXiv:1707.07402*, 2017.

[7] Li Dong and Mirella Lapata. Language to logical form with neural attention. *Proceedings of ACL.*, 2016.

[8] Wang Ling, Edward Grefenstette, Karl Moritz Hermann, Tomáš Kočiskỳ, Andrew Senior, Fumin Wang, and Phil Blunsom. Latent predictor networks for code generation. *arXiv preprint arXiv:1603.06744*, 2016.

[9] Jay Parks. tf-seq2seq. *https://github.com/JayParks/tf-seq2seq*, 2018.

[10] Tensorflow. Neural machine translation (seq2seq) tutorial. *https://www.tensorflow.org/tutorials/seq2seq*, 2018.

[11] moses smt. Bleu score script. *https://github.com/moses-smt/mosesdecoder/blob/master/scripts/generic/multi-bleu.perl*, 2016.

[12] Denny Britz. Deep learning for chatbots, part 1 – introduction. *http://www.wildml.com/2016/04/deep-learning-for-chatbots-part-1-introduction/*, 2016.