Code that Writes Itself: Seq2Seq for the Description2Code Problem

Pooja Sethi Ishan Ranade

March 2017

1 Abstract

Sequence to sequence learning is a deep learning technique that has demonstrated state-of-the-art results for machine translation tasks. For our final project, we investigated the use of sequence to sequence learning for translating descriptions to code. We trained a variety of small neural nets and though we were not able to produce real, runnable code, we found that the neural nets were able to learn some interesting relationships between the description and code data. We discuss our model setup, data set, preliminary results, and areas for future work.

2 Introduction

Programming is a highly specialized skill that requires months to years of training to do well. But what if anyone could code, simply by giving a computer a set of directions written in natural language? Being able to translate a description directly into code (Description2Code) could be an extremely powerful and useful ability, making programming more accessible and efficient. Though the Description2Code problem is a difficult one and called "extremely ambitious" by organizations such as OpenAI [1], we were interested in understanding how well today's NLP and deep learning techniques perform when put up to the challenge.

In a blog post called the *Unreasonable Effectiveness of Recurrent Neural Networks*, Andrej Karpathy describes how he produced Linux-like source code using 3-layer Long Short-Term Memory (LSTM) networks [2]. Though the source code produced by his model was not real code, in the sense that most of it cannot compile or perform a meaningful task, from a first glance it has an uncanny resemblance to what you would see in the Linux kernel. Karpathy's technique, however, did not take in any English input. It simply generated code character-by-character by using Linux source code on GitHub as training data.

On a related note, the past few years have seen a huge improvement in Machine Translation systems due to LSTMs. Particularly, Sequence to Sequence

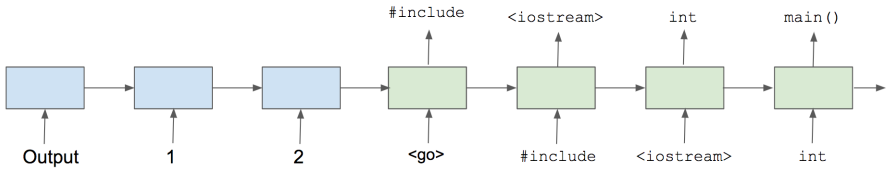
Learning (Seq2Seq) initially introduced by researchers at Google has proven to be extremely influential [3]. The Seq2Seq model allows translation between a sequence of variable length to another sequence of variable length while using a fixed size model. The Seq2Seq paper evaluates the performance of the model on an English to French translation task. However, the scope of Seq2Seq is beyond translation between one human language and another. For example, researchers have found applications of Seq2Seq for creating a neural conversational model i.e. chatbots [4] and parsing [5].

We investigated the use of Seq2Seq for translating between problem statements written in English (similar to those used for coding challenges on sites such as HackerRank, l33tCode, etc.) to code that solves the given problem. Though we did not necessarily expect that the code produced by our model would be capable of solving an unseen problem statement or for that matter even be able to compile, we were interested in knowing what kinds of results we would obtain by using known deep learning techniques and how close the code produced is to a real solution.

3 Model

3.1 Encoding and Decoding

Our sequence to sequence learning model consists of two RNNs: one encoder RNN that takes in the English description as input, and one decoder RNN that generates the code output. A simplified view of our model is shown below.



During training, each token of the description is fed into an encoding Gated Recurrent Unit (GRU) cell and each token of the corresponding code is fed into a decoding GRU cell. In reality, the above model is more complex than the above picture because we also use attention mechanisms. This allows the model to search the description for relevant words while producing the code output.

The conditional probability that we are trying to maximize is:

$$p(c_1, ..., c_N | e_1, ..., e_n) = \prod_{i=1}^{N'} p(c_i | e_1, ..., e_N, c_1, ..., c_{i-1}) \tag{1}$$

where the sequence $e_1, ..., e_N$ is the English problem statement and $c_1, ..., c'_N$ is the code solution that goes with it. Note that the length of the sequences do

not necessarily need to be equal. Each sentence in our model is required to end with the special $\langle stop \rangle$ symbol in order to form a valid probability distribution.

During training, the objective we wanted to maximize was:

$$1/|S| \sum_{(E,C) \in S} logP(C|E) \tag{2}$$

where S is the training set and (E, C) are the English-code pairs in the training set. At the decoding stage, we picked the most likely code string for a given English problem such that $\hat{C} = \arg\max_C p(C|E)$.

4 Data Set

For developing and evaluating our model we used a publicly available data set from OpenAI compiled by Ethan Caballero [1]. This data set contained 5000 input-output examples of problem statements in English and code pairs. The problems provided are similar to what is commonly seen in programming competitions. The problems were sorted by various degrees of difficulty level, from easiest to hardest. Solutions to the problems were provided in two languages, Python and C++. An example problem statement and solution pair is given below.

Description

```
Write a program to find the remainder when two given numbers
are divided.
Input The first line contains an integer T, total number of test
cases. Then follow T lines, each line contains two Integers A and B.
Output Find remainder when A is divided by  B.
Constraints  1 <=  T <= 1000 1 <= A,B <= 10000
Example  Input 3 1 2 100 200 10 40  Output 1 100 10
```

Code

```
#include<iostream>
#include<math.h>
using namespace std;

int main() {
 // your code goes here
 int T;

 cin>>T;
 while(T--){
     int A,B;
     cin>>A>>B;
     cout<<A%B<<endl;
 }
 return 0;
```

Actually using this data set to train our model was not as straightforward as it initially seemed, as we had to make many important design decisions that sanitized the inputs and outputs before they were ready to use in our model. We describe some of the dimensions on which we considered sanitizing the data.

4.1 Newlines and Indentation

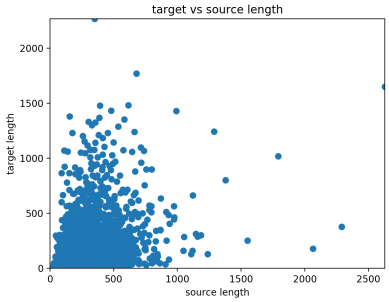
Unlike in spoken human languages like English and French, newlines and indentation can make a semantic difference in programming languages. This is especially true for programs written in languages like Python, which cannot be interpreted if the indentation is incorrect. Therefore, we needed to treat newlines and indentations as if they were actually tokens in our vocabulary. Initially, we created a script that took the Python code and replaced all newlines in our training set with a special $\langle newline \rangle$ token and all indents with a special $\langle indent \rangle$ token. In practice, we found that this did not lead to good results. During decoding, the model would predict sentences with all $\langle newline \rangle$ or all $\langle indent \rangle$ tokens due to over-representation in the training set. Because of this, we decided to remove these special tokens from our training sentences. However, in the future we would like to find a better way of representing them. Learning more complex models of larger sizes may also help reduce this problem.

4.2 Programming Language Choice

Initially, we had planned on training our model on Python code rather than C++ code due to Python's readability and simpler syntax. However, due to the indentation and newline issue described above, we decided to go with C++ instead since it has more lax requirements about how code should be indented and spaced.

4.3 Vocabulary Size, Bucketing, and Padding

We used a vocabulary size of 10,000 for both the input English tokens and for the code tokens. We also created buckets to allow our model to handle description and code sequences of different lengths. On average, we found that the description sentences were 110 tokens longer than the code sentences. We also plotted code sequence length versus description sequence length to help inform our decision for the bucket sizes.



The buckets we created were of size [(110, 1), (220, 110), (400, 220), (550, 400), (770, 550)].

In retrospect, our first bucket was too small as there are no code sequences of length 1. In the future, we would make this longer.

4.4 Size of Training, Dev, and Test Sets

Our training set contained approximately 1500 description and code pairs, our development set 750 pairs, and our training set 750 pairs. In total, the size of our training set was around 4 MB. Though we did try to use the full data set, it was too large to fit in the memory of the CPU we were using.

4.5 Spaces and Programmer's Style

An interesting observation we had is that the programmer's use of spaces affects what tokens our model sees. For example, one programmer might put a space before a while loop, for example

```
while (T--
```

. Another programmer might not include the space, for example

```
while(T--
```

. In the first case, the model learns two different vocabulary words because we split on spaces. In the second, case, it only learns one vocabulary word. Yet semantically, this makes no difference in C++. Ideally, we would be able to sanitize our training data so that we could make the style consistent across training examples.

4.6 Multiple Solutions Per Problem

One problem in trying to translate from English to code is that there are infinitely many possible ways to write a solution for a problem. Different programmers may use different variable names, style, and even algorithms resulting in very different-looking code but the same behavior during execution.

The data set provided by Caballero [1] contains multiple solutions for each problem statement. In our data set, we only used the first solution given for every problem. However, it would be interesting to see what would happen if we were to train the model with multiple possible "translations" for the same description.

5 Implementation

We used the TensorFlow sequence to sequence library for implementing our translation model. We decided to use TensorFlow because it nicely abstracts away much of the low-level implementation of Seq2Seq, which allowed us to focus on the design decisions surrounding our data set and hyper-parameters It has also previously shown success in creating and English-to-French translation model.

5.1 Model Size

Sequence to sequence learning works best on very large models. For an English-to-French translation model, the TensorFlow documentation suggests using a model with 3 layers and 1024 neurons per layer. The data set used for training is 20 GB. After much experimentation, we realized that training such a large model would be infeasible for us due to the limited computational resources allocated to University of Washington CSE undergrads and our lack of GPU processing. Because of this, we had to stick with much smaller sizes ranging from 2 to 5 layers and 8 to 32 neurons per layer.

5.2 GRUs vs. LSTMs

An important design decision we had to make was whether or not to use Gated Recurrent Unit (GRU) or Long Short-Term Memory (LSTM) cells in our model. We ended up using GRU cells because they have fewer parameters and have been shown to be easier to train in many cases.

5.3 Attention

We used a neural net with attention-based decoding, as this has shown to improve performance on long sentences. This was especially helpful for our use case, as the vast majority of the descriptions and solutions in our training data had over 100 tokens.

5.4 Reversing Inputs

We reversed the input description text, as this has shown to be beneficial for the English-to-French language model.

6 Results

6.1 Quantitative Analysis

The output produced by the models was extremely different than previously expected. The metrics we had previously considered were not longer effective because of the randomness and repetition of the output and more subjective evaluative measures had to be developed. One measure used was to find the edit distance or Levenshtein distance between output and input. This measure is a simplified version of the TER score and is useful because it indicates the amount of post processing that would be required to change the predicted output to the actual output. This score was calculated by finding the edit distance of each input and output pairing and averaging these. Another measure was to find the trend in the size of predicted outputs versus real outputs. This measure is useful because it would indicate to use if our model was inserting extraneous code into its solution and that information can be used to further improve the model in the future. Below is a table of some of the data collected:

Layers	Neurons per Layer	Avg. Edit Distance (100 data points)	Avg. Length Proportion (Size of Predicted / Size of Actual)
2	8	2439	1.17
2	16	2478	1.16
2	32	2468	1.49
3	8	3108	1.00
3	16	2211	3.05
5	8	2916	1.53

Looking at the edit distance portion of the table, we saw that increases in the number of layers and size did not seem to decrease the edit distance. In fact, a neural net with five layers and sixteen neurons had a distance of 2572 while a neural net with two layers and eight neurons had a distance of 2439, which was counterintuitive. There did not seem to be a consistent trend despite increasing layers and neurons. In addition, the length portion of the table showed interesting results. Code produced by our model tended to be much larger than the real output, up to a factor of 3. It seemed out model was choosing very long matches to the input sequences. In all test cases, our model never produced an average predicted output length that was smaller than the average actual output length. In the future, we could possibly prevent this by adding more bucket sizes during training. The average size of the input descriptions was about 1500 words and the edit distances calculated were around 2500-3000. This further indicated that to transform the predicted output to the actual output we would not only have to delete almost all the characters but then replace them with new, correct characters, effectively rewriting the entire

solution. This was another indication that our model was not performing very well at this task.

6.2 Qualitative Analysis

We then took a deeper look at the output produced by the model. After training a neural net with 3 layers and 32 neurons, the output it produced for a test question about finding common substrings between two strings A and B was:

```
brr[0][m-0] brr[0][m-0] brr[0][m-0] cin>>a>>m cin>>a>>m co co
cin>>a>>m strcmp strcmp strcmp ...
(strcmp repeats approx. 50 more times)
```

It seemed that the code produced was mainly gibberish with a few key words thrown in, such as pulling in "strcmp" for a string comparison question. In most cases key phrases were repeated many times in a row and none of the output was actually compilable and runnable. The last method of our planned evaluation, in which we would find the number of ifs, whiles, and other programming constructs in the output was also rendered invalid because almost all of the output contained a few repeated words and phrases, and these were almost always words pulled from comments blocks or a random variable that had appeared in the code. Overall, after attempting to use sequence to sequence RNNs to solve the description to code problem, it seems that a straightforward deep learning approach may not be appropriate. This problem seems to extend beyond the scope of a simple sequence to sequence matching, and more advanced methods will have to be used.

7 Conclusion and Future Work

The evaluation on our results showed that the sequence-to-sequence matching approach is not the best for this description to code problem. Despite this, there are a number of changes we can make to our approach to perhaps boost the performance of our model. First, the use of larger model architectures and GPU-assisted computation would be very beneficial as it would provide more computation to analyze larger data sets. With the time constraints of this project and the computational resources available to us, we were only able to analyze smaller data sets, of around 3000 data points, which is not enough data for a deep neural net. Second, better data sanitization policies may boost performance. Some issues we ran into were tab vs spacing, the variety of variable names, spacing, and indentation in the sample solutions. Third, since descriptions to code have much longer sequence lengths than sentence-to-sentence natural language translations, the use of globally coherent attention mechanisms for keeping track of syntactical information could boost performance. Finally, an increased data size would be hugely beneficial, since neural nets tend to perform better when provided with as much sample data as possible. This project gave us an interesting look at the application of neural nets to a difficult open

problem, and showed us that neural nets were not the silver bullet solution that we originally believed them to be. Though increased sample size, increased computational power, and data sanitization may increase our model's performance, the ultimate solution may very well lie in a different natural language processing model.

8 References

[1] "Requests for Research." OpenAI. N.p., n.d. Web. 17 Mar. 2017. https://openai.com/requests-for-research/description2code.

[2] Karpathy, Andrej. "The Unreasonable Effectiveness of Recurrent Neural Networks." N.p., 21 May 2015. Web. 17 Mar. 2017. http://karpathy.github.io/2015/05/21/rnn-effectiveness/.

[3] Sutskever, Ilya, et. al. "Sequence to Sequence Learning with Neural Networks." (n.d.): n. pag. Web. 17 Mar. 2017. https://papers.nips.cc/paper/5346-sequence-to-sequence -learning-with-neural-

[4] Vinyals, Oriol, et. al. "Grammar as a Foreign Language." (n.d.): n. pag. 9 June 2015. Web. 17 Mar. 2017. https://www.cs.toronto.edu/ hinton/absps/grammar.pdf.

[5] Le, Quoc V, et. al. "A Neural Conversational Model." (n.d.): n. pag. 22 July 2015. Web. 17 Mar. 2017. https://arxiv.org/pdf/1506.05869.pdf.

[6] "Sequence-to-Sequence Models." TensorFlow. Google, 8 Mar. 2017. Web. 17 Mar. 2017. https://www.tensorflow.org/tutorials/seq2seq.

[7] Lavie, Alon. "Evaluating the Output of Machine Translation Systems." N.p., 31 Oct. 2010. Web. 17 Mar. 2017. https://amta2010.amtaweb.org/AMTA/papers/6-04- LavieMTEvaluation.pdf.

[8] See, Abi. "Exploiting the Redundancy in Neural Machine Translation." (n.d.): n. pag. Web. 17 Mar. 2017. https://nlp.stanford.edu/courses/cs224n/2015/reports/26.pdf.