A DEEP DIVE INTO AUTOMATIC CODE GENERATION USING CHARACTER BASED RECURRENT NEURAL NETWORKS

A Thesis
Presented to the
Faculty of
California State Polytechnic University, Pomona

In Partial Fulfillment
Of the Requirements for the Degree
Master of Science
In
Computer Science

By
Renita Priya
2017
SIGNATURE PAGE

THESIS: A DEEP DIVE INTO AUTOMATIC CODE GENERATION USING CHARACTER BASED RECURRENT NEURAL NETWORKS

AUTHOR: Renita Priya

DATE SUBMITTED: Summer 2017

Computer Science Department

Dr. Yu Sun
Thesis Committee Chair
Computer Sciences

Dr. Mohammad Husain
Computer Sciences

Dr. Gilbert Young
Computer Sciences
ACKNOWLEDGEMENTS

I would like to acknowledge my thesis advisor, Dr. Yu Sun. He was always there to encourage me and support me in my thoughts and ideas. He has inspired me to always do the best that I can and taught me to always push forward. Dr. Sun has provided me with so much grace and understanding and I am so grateful for the opportunity to have been his student.

I would also like to thank the other committee members, Dr. Mohammad Husain and Dr. Gilbert Young for being excellent professors and advisors who have given their time and excellent feedback for this research.

Lastly, I would like to give the sincerest acknowledgement to my parents. My parents have constantly supported and encouraged me throughout the completion of my Master Degree. They were the ones who first believed in me that I was even capable of accomplishing this. They have been my support system from day one and have always provided for me. I want to thank all my friends as well. I would like to give special thanks to Brianna who was always there to listen, pray, encourage, and for always having a coffee in hand for me for all those late nights. Thank you Lisa and Ben for all encouragement, prayers and for cheering me on.

Above all, I want to give all glory to God.

Thank you,

Renita Priya
ABSTRACT

Deep Learning is an emerging field in Artificial Intelligence that uses biologically inspired neural networks to recognize patterns in the natural world. These neural networks have an amazing ability to process large amounts of data and learn from them. Recurrent Neural Networks (RNN) are used in applications involving natural language processing like text translations and text generation. This research evaluates the effectiveness of a RNN to be able to automatically generate programming code. Programming languages are different from natural languages in that they have unique structure and syntax. The goal for this research is to conduct experiments on a character RNN model with for three programming languages; Java, Python and C#, and evaluate the results by testing and analyzing the ability for the RNN to automatically produce code that is able to compile.
# TABLE OF CONTENTS

SIGNATURE PAGE .................................................................................................................. ii

ACKNOWLEDGEMENTS ................................................................................................. iii

ABSTRACT ......................................................................................................................... iv

TABLE OF CONTENTS ....................................................................................................... v

LIST OF TABLES ............................................................................................................. vii

LIST OF FIGURES ........................................................................................................... viii

GLOSSARY ......................................................................................................................... xi

INTRODUCTION .................................................................................................................. 1

1.1 Motivation ..................................................................................................................... 1

1.2 Problem Description ................................................................................................... 2

1.3 Research Goals ............................................................................................................ 3

1.3.1 Programming Language ......................................................................................... 3

1.3.2 Dataset Size ............................................................................................................ 3

1.3.3 Training Length ...................................................................................................... 4

1.4 Background and History ............................................................................................. 4

1.4.1 Deep Learning and Neural Networks ..................................................................... 4

1.4.2 Recurrent Neural Networks .................................................................................... 5

1.4.3 Long Short-Term Memory (LSTM) and Gated Recurrent Units (GRU) .............. 7

1.4.4 Language Model ..................................................................................................... 8

v
1.5 Structure of the Experiments ................................................................. 8
RNN MODEL ............................................................................................... 10
2.1 Framework and Resources .................................................................. 10
2.2 Implementation .................................................................................. 10
TRAINING .................................................................................................... 16
ANALYSIS AND EVALUATION .............................................................. 23
FUTURE WORK AND CONCLUSION ......................................................... 48
  5.1 Future Work ....................................................................................... 48
  5.2 Conclusion ......................................................................................... 49
REFERENCES ........................................................................................... 50
APPENDIX A ............................................................................................. 53
APPENDIX B ............................................................................................. 55
APPENDIX C ............................................................................................. 58
APPENDIX D ............................................................................................. 63
APPENDIX E ............................................................................................. 67
LIST OF TABLES

Table 1 Programming language datasets and source .................................................. 12
Table 2 Character codes .............................................................................................. 13
Table 3 Training experiments ..................................................................................... 16
Table 4 Batch loss summary for all training sessions ................................................. 24
Table 5 Batch Accuracy Summary for All Training Sessions ..................................... 25
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>General Structure of a Neural Network</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>Unrolled Recurrent Neural Network</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>Example of a two layer RNN model</td>
<td>7</td>
</tr>
<tr>
<td>4</td>
<td>Example of unrolled character RNN</td>
<td>15</td>
</tr>
<tr>
<td>5</td>
<td>Java large - batch accuracy</td>
<td>17</td>
</tr>
<tr>
<td>6</td>
<td>Java large - batch loss</td>
<td>17</td>
</tr>
<tr>
<td>7</td>
<td>Java small - batch accuracy</td>
<td>18</td>
</tr>
<tr>
<td>8</td>
<td>Java small - batch loss</td>
<td>18</td>
</tr>
<tr>
<td>9</td>
<td>Python large - batch accuracy</td>
<td>19</td>
</tr>
<tr>
<td>10</td>
<td>Python large - batch loss</td>
<td>19</td>
</tr>
<tr>
<td>11</td>
<td>Python small - batch accuracy</td>
<td>20</td>
</tr>
<tr>
<td>12</td>
<td>Python small - batch loss</td>
<td>20</td>
</tr>
<tr>
<td>13</td>
<td>C# large - batch accuracy</td>
<td>21</td>
</tr>
<tr>
<td>14</td>
<td>C# large - batch loss</td>
<td>21</td>
</tr>
<tr>
<td>15</td>
<td>C# small - batch accuracy</td>
<td>22</td>
</tr>
<tr>
<td>16</td>
<td>C# small - batch loss</td>
<td>22</td>
</tr>
<tr>
<td>17</td>
<td>Batch loss summary</td>
<td>24</td>
</tr>
<tr>
<td>18</td>
<td>Batch Accuracy Summary</td>
<td>26</td>
</tr>
<tr>
<td>19</td>
<td>Color coded probabilities</td>
<td>27</td>
</tr>
<tr>
<td>20</td>
<td>Zoomed in probability matrix to show individual characters</td>
<td>28</td>
</tr>
<tr>
<td>21</td>
<td>Java large dataset at epoch 0</td>
<td>29</td>
</tr>
</tbody>
</table>
Figure 22. Java large dataset at epoch 19 ........................................... 29
Figure 23. Java small dataset at epoch 0 ............................................. 30
Figure 24. Java small dataset at epoch 19 .......................................... 30
Figure 25. Python large dataset at epoch 0 ........................................ 31
Figure 26. Python large dataset at epoch 19 ...................................... 31
Figure 27. Python small dataset at epoch 0 ....................................... 32
Figure 28. Python small dataset at epoch 19 ..................................... 32
Figure 29. C# large dataset at epoch 0 ............................................. 33
Figure 30. C# large dataset at epoch 19 ........................................... 33
Figure 31. C# small dataset at epoch 0 ............................................ 34
Figure 32. C# large dataset at epoch 19 ........................................... 34
Figure 33. Full comparison of all 6 trainings after first epoch ................. 35
Figure 34. Full comparison of all 6 trainings after last epoch .................. 36
Figure 35. Java large dataset at epoch 0, 24 errors ............................ 37
Figure 36. Java large dataset at epoch 9, 8 errors .............................. 38
Figure 37. Java large dataset at epoch 19, 7 errors ............................. 38
Figure 38. Java small dataset at epoch 0 ......................................... 39
Figure 39. Java small dataset at epoch 9 .......................................... 39
Figure 40. Java small dataset at epoch 19 ....................................... 40
Figure 41. Python large dataset at epoch 0 ...................................... 40
Figure 42. Python large dataset at epoch 9 ...................................... 41
Figure 43. Python large dataset at epoch 19 .................................... 41
Figure 44. Python small dataset at epoch 0 ..................................... 42
Figure 45. Python small dataset at epoch 9

Figure 46. Python small dataset at epoch 19

Figure 47. C# large dataset at epoch 0

Figure 48. C# large dataset at epoch 9

Figure 49. C# large dataset at epoch 19

Figure 50. C# small dataset at epoch 0

Figure 51. C# small dataset at epoch 9

Figure 52. C# small dataset at epoch 19
GLOSSARY


2. Natural Language Processing (NLP) – a component of artificial intelligence that gives computers the ability to understand natural human speech and language as it is spoken.

3. Recurrent Neural Network (RNN) – a type of neural network where connections form a cycle. These networks are commonly used in NLP applications.

4. Training Dataset – A collection of related sets of information that are fed into a neural network to discover potential relationships.

5. Epoch – A single pass of the entire dataset in iterative training of a neural network.
INTRODUCTION

1.1 Motivation

In the past couple of years Deep Learning (DL) has given rise to many amazing applications and technological advances in the field of Artificial Intelligence (AI). As computational resources with Graphic Processing Units (GPUs) become more readily available, researchers can now tackle problems that other machine learning algorithms could not handle (Perez). At the core of DL are Artificial Neural Networks (ANNs) that were inspired by the human brain and its ability to learn and create a vast network of connections through neurons. These biologically inspired neural networks provide computers the ability to process large amounts of data and provide effective solutions for questions researchers have in image recognition, speech recognition and Natural Language Processing (NLP). Natural language processing is the way that computers can learn, analyze and derive meaning and structure from language as it is naturally spoken and written. Large amounts of NLP research have been done in the analysis of literary works and natural language spoken on social media (O’Dell). Computer programming languages have their own complexities that natural language does not have. While code is logical and structured, the ability to write programs and code has become a desirable skill that is treated with the same intensity as knowing another language (Tanz). Just as any other language, the advancements in Neural Networks and Deep Learning has provided researchers resources to study and train computers to be able to understand and produce code. This could have a large impact on how computer programming will look like in the future.
1.2 Problem Description

Unlike training on other literary works, training a neural network on programming languages creates a unique challenge. Training on the works of Shakespeare or on the Harry Potter series by JK Rowling can produce some pretty interesting results. Excerpts of generated texts by these neural networks can almost be confused by the original works and can be fun to see how well the computer learns by example (Karpathy). Training neural networks on computer programming languages are treated the same way as training on the works of Shakespeare; for the computer, it is just another set of training data. When the computer generates random pieces of code, the curiosity of automatic text generation does not just stop at the interesting behavior of this random code, it contributes to the excitement that there could be so much more possibilities. Because of the randomized nature of the automatic generation model, the code might not execute to produce any results or even compile. Speech and literature have a set dictionary of words and general rules for grammar and punctuation. Computer programming languages differ widely within each language and enforce strict rules of punctuation and formatting. Java code, for example, must have semicolons at the end of each line and components of programs must begin and end with brackets. In Python, there are no brackets or semicolons but parenthesis must be open and closed correctly, but more importantly the spacing and tabs contribute greatly to the ability to read and compile code. Individual words in code changes drastically based on the programmer, depending on how they choose to name their variables and methods. A large challenge for deep learning is to be able to accurately predict the locations of parenthesis, semicolons, and spaces in a generated sequence. This paper explores a character based recurrent neural network model and evaluates the effectiveness of the neural network to be
able to generate code and analyzes ability for that automatically generated code to be compiled with the least amount of errors.

1.3 Research Goals
The main research goal is to study the effectiveness of Recurrent Neural Networks (RNN) in their ability to automatically generate code. There are three components to this experiment in order to determine how well a RNN can learn how to code. These three components are programming language, dataset size, and training length. An evaluation of a combination of these components will help decide on what language, dataset size and training length produces the best generated code. While the implementation and internal details of the model will stay constant throughout the different trainings, these three variable components will help evaluate the effectiveness of RNN code generation.

1.3.1 Programming Language
One goal of this research is to be able to identify the ideal programming language to train the RNN. To determine this, three different coding languages will be used for training. RNN model was trained on was Java, Python and C#. Each of these languages vary in syntax and structure. The ability for the neural network to learn around the challenges for these languages will help determine the effectiveness of the model.

1.3.2 Dataset Size
The size of the dataset will also help in determining the effectiveness of the RNN. To determine this, each language will have two different training datasets, one small and one
1.3.3 Training Length

The length of training for each scenario is the third component in determining RNN effectiveness of generating code. The RNN model will train on each dataset for every language for a total of 20 iterations. The analysis and evaluation of the results will determine the threshold iteration for training.

1.4 Background and History

1.4.1 Deep Learning and Neural Networks

Deep Learning is an emerging approach in Artificial Intelligence (AI) that uses Neural Networks to provide computers with the ability to learn and create patterns. Neural networks were first proposed by two MIT researchers, Warren McCullough and Walter Pits in 1944 (Hardesty). Since then, the popularity of neural networks has been up and down. More recently it has gained extreme popularity due to the increased processing power of graphics chips. Neural networks are composed of large, highly interconnected processing elements that work together to create patterns and solve problems, much like a biological neuron. There are different types of deep learning approaches and neural networks.

The most popular types of neural network architectures are Convolutional Neural Networks (CNNs) and Recurrent Neural Networks (RNNs) (Patterson and Gibson). CNNs are used mainly in image recognition and are well suited to be able to process and handle aspects of visual data. RNNs are feed-forward neural networks that have the ability to send information over time steps. In the past these networks were more challenging to train.
compared to CNNs because they require large processing power. The limitations of having small processing units are long gone, with the emergence of GPUs. RNN models can solve problems that involve sequences of vectors such as image captioning, speech synthesis, music generation, language modeling, character level text generation models, and natural language translations (Patterson and Gibson).

1.4.2 Recurrent Neural Networks

Like all other neural networks RNNs are organized in layers each made up of interconnected nodes which contain an activation function. Patterns are inserted into the model through the input layer which communicates to the hidden layers that process the data using a system of weighted connections. Then the hidden layers link to an output layer as the answer. The figure below illustrates this basic architecture.

![Figure 1. General Structure of a Neural Network](image)
Recurrent Neural Networks solve a problem that convolutional neural networks and other neural networks cannot solve, they can understand sequences by the fact that the current state is affected by its previous states. In traditional neural networks, the inputs and outputs are independent of each other. In solving NLP problems, specifically text generation, the network needs to understand the previous word or character in the sequence. The reason these networks are called recurrent is because they perform the same task over for every single element in a sequence. This gives them ‘memory’ to remember what has been calculated so far. The diagram below describes the basic RNN cell and then shows how it can be unrolled for an entire sequence which is what helps the RNN model retain memory.

**Figure 2. Unrolled Recurrent Neural Network**

In figure 2, the left image represents the RNN cell in its entirety. The input is represented as $x$ and output as $y$. The dark blue RNN cell represents the hidden state, this is where the calculations and connections to patterns are made. The arrow that loops around represents the recurrent nature of the network, the input and output for this step is the variable $H$. The hidden state performs calculations based on the previous hidden state, $H_{t-1}$ and input at
that time, $x_t$, and outputs the next hidden state, $H_{t+1}$ and the $y_t$ output. The calculation is usually a tanh or ReLU function. Unlike other neural networks that have different weights and biases across the cells, recurrent neural networks share the same weights and biases across all the cells. This figure only describes an RNN that is only one layer. Deeper networks can have multiple layers of RNN cells as seen in the figure below.

![Figure 3. Example of a two layer RNN model](image)

1.4.3 Long Short-Term Memory (LSTM) and Gated Recurrent Units (GRU)

The RNN cells as seen in figure 2 and 3 can either be a Long Short-Term Memory (LSTM) cell or Gated Recurrent Unit (GRU) cell. Vanilla RNN cells have the ability to remember previous information but they struggle with long term memory. LSTMs help solve this issue by remembering information that was not just in the previous cell but in the ones prior. At their core, LSTMs have the ability to remove or add information to the cell state by the use of regulated structures called gates. The LSTM has three gates to protect and
control the cell state (Olah). A Gated Recurrent Unit (GRU) cell is a variation of an LSTM introduced in 2014 that has two gates, a reset and an update gate (Britz).

1.4.4 Language Model

By creating language models to understand natural language and training neural networks to recognize patterns in text, we can teach computers to be able to generate its own text using the same patterns it trained on. Recurrent Neural Networks (RNNs) are highly suited for recognizing these patterns because of their high-dimensional hidden state that allows them to process and understand past information. Text generation is an emerging topic of research in DL and just by feeding the neural network a large text dataset such as the works of Shakespeare, the computer is able to generate similar text in the same language and structure of Shakespeare (6). Other text generating models have been trained on Harry Potter, music theory, recipes, the Bible and even computer programming languages. These language models are based on predicting the next word based on previous words. Word vectors are a common way of representing words as numbers, and for word based RNNs, the inputs and outputs are in the format of these word vectors. Character based RNNs use characters as the input and outputs. At the core of these language model is \( P(C|T) \) where \( C \) is the next character and \( T \) is the context of all the characters that has come before (Iourovitski).

1.5 Structure of the Experiments

In this paper a character based RNN model is used in training on three different languages, with two different sized datasets and evaluated at different iterations of the model. The
results of the training is analyzed and studied by generating code snippets and analyzing the effectiveness of the RNN by the code it was able to generate. The evaluation will keep in mind the three research goals (programming language, dataset size and training length).
RNN MODEL

2.1 Framework and Resources

There are many deep learning and machine learning libraries and frameworks such as Theano, Keras, Caffe, and Torch currently available for various languages ("Deep Learning Libraries by Language"). One that is gaining more and more popularity in the past couple of years has been a machine learning neural network created by Google called Tensorflow. Tensorflow was released in 2015 as an open sourced stand-alone library for many machine learning and deep neural network applications ("Tensorflow"). The RNN model used in this experiment is built in Python using Tensorflow 1.2 with Python 3.6. Neural networks are very hardware intensive models and use up entire computational resources. Due to the number of experiments to be done, the trainings were done on different machines. Testing the implementation of the RNN model was on a MacBook Pro, the actual training experimentations executed on a 12 core desktop Linux machine and an EC2 g2.2.large computational resource from Amazon Web Services (AWS) ("Deep Learning AMI Ubuntu Version"). The EC2 cloud server was running Amazon’s deep learning Amazon Machine Image (AMI) with NVIDIA graphics card. Tensorflow automatically recognizes the resources available during when compiled and was able to run much faster on a GPU compared to a CPU. The Tensorflow framework on the EC2 was a version of Tensorflow optimized to run on a GPU.

2.2 Implementation

The RNN model used for the experiments were based on Martin-Gorner’s character RNN and sherjiloizir’s character RNN models in Tensorflow from Github (Gorner and
Sherjilozir). The RNN was also inspired from Andrej Karpathy’s experiments on character RNNs for text generation using the Torch framework (Karpathy). The training datasets were derived from various popular Github open sourced repositories in Java, Python and C#. The experiments were done on a large dataset and a small dataset for each of the programming languages. The Java datasets were downloaded as pre-collected Github source code from Github Java Corpus (Allamanis and Sutton). In the interest of DL and NLP, many researchers make their corpus and datasets available for further learning thus allowing other researchers to spend more time on their deep learning applications instead of cleaning and purifying data for training (“Learning from Big Code”). The Python and C# programming languages datasets the files were scraped manually from the specific Github repositories since it was difficult to find a corpus for Python and C#. After downloading the repositories, the files were filtered by either a .py file or .cs file to determine if it was a python or C# source code. The table below describes the languages and which Github projects were used in each of the datasets
Table 1

Programming languages datasets and sources

<table>
<thead>
<tr>
<th>Language</th>
<th>Dataset size</th>
<th>Github Repositories</th>
</tr>
</thead>
<tbody>
<tr>
<td>Java</td>
<td>small (4.06 MB)</td>
<td>libgdx</td>
</tr>
<tr>
<td>Java</td>
<td>large (8.05MB)</td>
<td>spring-framework</td>
</tr>
<tr>
<td>Python</td>
<td>small (4.04 MB)</td>
<td>keras, pyorithm, pytorch, tensorflow</td>
</tr>
<tr>
<td>Python</td>
<td>large (8.08MB)</td>
<td>pandas</td>
</tr>
<tr>
<td>C#</td>
<td>small (4.02 MB)</td>
<td>Dapper, Hangfire, reactive native, Shadowsock</td>
</tr>
<tr>
<td>C#</td>
<td>large (8.05MB)</td>
<td>PowerShell</td>
</tr>
</tbody>
</table>

In order for the RNN model to train on the programming language datasets the data needs to be prepared first. The RNN for this research is a character language model so the inputs and outputs are characters. To do this each character in the Java, Python or C# file was appended together into one list so at each epoch, the model would iterate through this list. Before being placed in this list, each character was encoded into an integer between 0 and 97. Table 2 illustrates the numeric values for each character. The character is first converted into an ASCII code and then assigned a number. Since this RNN is a character based language model instead of a word based model, each character needs to be converted into a number. If it was a word based model, then each word would have needed to be converted into a word vector. Once all the characters in each file are converted into this integer code, it is then ready to be fed into the RNN.
Table 2

Character codes

<table>
<thead>
<tr>
<th>Code</th>
<th>Character</th>
<th>Code</th>
<th>Character</th>
<th>Code</th>
<th>Character</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>null byte</td>
<td>34</td>
<td>@</td>
<td>68</td>
<td>b</td>
</tr>
<tr>
<td>1</td>
<td>tab</td>
<td>35</td>
<td>A</td>
<td>69</td>
<td>c</td>
</tr>
<tr>
<td>2</td>
<td>space</td>
<td>36</td>
<td>B</td>
<td>70</td>
<td>d</td>
</tr>
<tr>
<td>3</td>
<td>!</td>
<td>37</td>
<td>C</td>
<td>71</td>
<td>e</td>
</tr>
<tr>
<td>4</td>
<td>&quot;</td>
<td>38</td>
<td>D</td>
<td>72</td>
<td>f</td>
</tr>
<tr>
<td>5</td>
<td>#</td>
<td>39</td>
<td>E</td>
<td>73</td>
<td>g</td>
</tr>
<tr>
<td>6</td>
<td>$</td>
<td>40</td>
<td>F</td>
<td>74</td>
<td>h</td>
</tr>
<tr>
<td>7</td>
<td>%</td>
<td>41</td>
<td>G</td>
<td>75</td>
<td>i</td>
</tr>
<tr>
<td>8</td>
<td>&amp;</td>
<td>42</td>
<td>H</td>
<td>76</td>
<td>j</td>
</tr>
<tr>
<td>9</td>
<td>'</td>
<td>43</td>
<td>I</td>
<td>77</td>
<td>k</td>
</tr>
<tr>
<td>10</td>
<td>(</td>
<td>44</td>
<td>J</td>
<td>78</td>
<td>l</td>
</tr>
<tr>
<td>11</td>
<td>)</td>
<td>45</td>
<td>K</td>
<td>79</td>
<td>m</td>
</tr>
<tr>
<td>12</td>
<td>*</td>
<td>46</td>
<td>L</td>
<td>80</td>
<td>n</td>
</tr>
<tr>
<td>13</td>
<td>+</td>
<td>47</td>
<td>M</td>
<td>81</td>
<td>o</td>
</tr>
<tr>
<td>14</td>
<td>,</td>
<td>48</td>
<td>N</td>
<td>82</td>
<td>p</td>
</tr>
<tr>
<td>15</td>
<td>-</td>
<td>49</td>
<td>O</td>
<td>83</td>
<td>q</td>
</tr>
<tr>
<td>16</td>
<td>.</td>
<td>50</td>
<td>P</td>
<td>84</td>
<td>r</td>
</tr>
<tr>
<td>17</td>
<td>/</td>
<td>51</td>
<td>Q</td>
<td>85</td>
<td>s</td>
</tr>
<tr>
<td>18</td>
<td>0</td>
<td>52</td>
<td>R</td>
<td>86</td>
<td>t</td>
</tr>
<tr>
<td>19</td>
<td>1</td>
<td>53</td>
<td>S</td>
<td>87</td>
<td>u</td>
</tr>
<tr>
<td>20</td>
<td>2</td>
<td>54</td>
<td>T</td>
<td>88</td>
<td>v</td>
</tr>
<tr>
<td>21</td>
<td>3</td>
<td>55</td>
<td>U</td>
<td>89</td>
<td>w</td>
</tr>
<tr>
<td>22</td>
<td>4</td>
<td>56</td>
<td>V</td>
<td>90</td>
<td>x</td>
</tr>
<tr>
<td>23</td>
<td>5</td>
<td>57</td>
<td>W</td>
<td>91</td>
<td>y</td>
</tr>
<tr>
<td>24</td>
<td>6</td>
<td>58</td>
<td>X</td>
<td>92</td>
<td>z</td>
</tr>
<tr>
<td>25</td>
<td>7</td>
<td>59</td>
<td>Y</td>
<td>93</td>
<td>{</td>
</tr>
<tr>
<td>26</td>
<td>8</td>
<td>60</td>
<td>Z</td>
<td>94</td>
<td></td>
</tr>
<tr>
<td>27</td>
<td>9</td>
<td>61</td>
<td>[</td>
<td>95</td>
<td>}</td>
</tr>
<tr>
<td>28</td>
<td>:</td>
<td>62</td>
<td>\</td>
<td>96</td>
<td>~</td>
</tr>
<tr>
<td>29</td>
<td>;</td>
<td>63</td>
<td>]</td>
<td>97</td>
<td>new line</td>
</tr>
<tr>
<td>30</td>
<td>&lt;</td>
<td>64</td>
<td>^</td>
<td></td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>=</td>
<td>65</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>&gt;</td>
<td>66</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>33</td>
<td>?</td>
<td>67</td>
<td>a</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

During training, the RNN model receives different parameters that will help in determining the effectiveness of the model. Some parameters of the model will stay as constants for all the trainings. The characteristics of the RNN that remain the same are that it consists of
three layers of GRU cells with an internal hidden state size of 512, batch size of 200 and a sequence unrolling length of 30 characters. The vocabulary set of 98 characters was also maintained. The training dataset only had java, python or C# files in which all the characters were able to be converted in the code as described in table 2 above. For the evaluation the variables that changed across the training was the datasets. There were 3 different languages (Java, Python and C#) and 2 different sizes per language (small ~4MB and large ~8MB). While each model was trained for a maximum of 20 epochs or cycles through the entire dataset, the model was evaluated at different states during the 20 epochs. The number of batches at each epoch varies due to the size of the dataset. This amount is calculated by the length of the dataset, the batch size and sequence length. Each epoch iterates through the entire dataset using minibatches. The minibatch size is at a fixed length of the batch size (200) multiplied by the sequence length of (30), so each iteration of the RNN will process through 600 characters. The number of minibatches per epoch is calculated based on the total character sequence divided by 600. During the training a checkpoint of the training set is saved every 500 batches into a checkpoints folder. Tensorboard, the graphical visualization tool part of tensorflow is also initiated and tracks the training. After the training is done, we can evaluate the results based on the saved checkpoints. The RNN model learns patterns based on the data fed to it. The hidden state is able to calculate probabilities and form patterns by being forced both the input and output layers. Figure 4 illustrates how this works. The RNN’s input is just the sequential order of the data, the output is just the next character in the sequence. After training, characters are randomly generated based on the probability of the previous character
**Figure 4.** Example of unrolled character RNN
During this research there were a total of 6 different training sessions. Statistics on the trainings are described in the tables and figures below.

Table 3

Training experiments

<table>
<thead>
<tr>
<th>Session</th>
<th>Dataset</th>
<th>Number of Epochs</th>
<th>Number of Batches per epoch</th>
<th>Computational resource</th>
<th>Time it took to train</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Java Large</td>
<td>20</td>
<td>1406</td>
<td>AWS ec2</td>
<td>2 hrs 48 min</td>
</tr>
<tr>
<td>B</td>
<td>Java Small</td>
<td>20</td>
<td>709</td>
<td>Local machine</td>
<td>5 hrs. 53 min</td>
</tr>
<tr>
<td>C</td>
<td>Python Large</td>
<td>20</td>
<td>1412</td>
<td>AWS ec2</td>
<td>2 hrs 48 min</td>
</tr>
<tr>
<td>D</td>
<td>Python Small</td>
<td>20</td>
<td>705</td>
<td>Local machine</td>
<td>5 hrs. 57 min</td>
</tr>
<tr>
<td>E</td>
<td>C# Large</td>
<td>20</td>
<td>1406</td>
<td>AWS ec2</td>
<td>2 hrs 48 min</td>
</tr>
<tr>
<td>F</td>
<td>C# Small</td>
<td>20</td>
<td>102</td>
<td>Local machine</td>
<td>5 hrs. 51 min</td>
</tr>
</tbody>
</table>

Tensorboard is a part of Tensorflow that charts the progress of batch accuracy and batch loss during training. The tensorboard log is saved into a folder while training that can be viewed later. The figures below share the tensorboard logs for each training session.
**Figure 5.** Java large - batch accuracy

**Figure 6.** Java large - batch loss
**Figure 7.** Java small - batch accuracy

**Figure 8.** Java small - batch loss
Figure 9. Python large - batch accuracy

Figure 10. Python large - batch loss
Figure 11. Python small - batch accuracy

Figure 12. Python small - batch loss
Figure 13. C# large - batch accuracy

Figure 14. C# large - batch loss
Figure 15. C# small - batch accuracy

Figure 16. C# small - batch loss
ANALYSIS AND EVALUATION

Once all the trainings completed, we can evaluate and analyze the effectiveness of the models by generating code from the model. During the training, batch accuracy and batch loss was recorded at different epochs and was plotted using tensorboard. The tensorboard logs are very helpful in visualizing the trainings, but they do not show the comparisons between different models since it only can access the current saved model at a time. Tables 4 and 5 below summarizes the loss and accuracy of each of the 6 training session at the end of every epoch, in order to compare the trainings with each other. Figures 17 and 18 is a graphical representation of the data from the tables.
Table 4
Batch loss summary for all training sessions

<table>
<thead>
<tr>
<th>Epoch</th>
<th>Java Large</th>
<th>Java Small</th>
<th>Python Large</th>
<th>Python Small</th>
<th>C# Large</th>
<th>C# Small</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start</td>
<td>4.51</td>
<td>4.49678</td>
<td>4.37293</td>
<td>4.41688</td>
<td>4.25490</td>
<td>4.32310</td>
</tr>
<tr>
<td>0</td>
<td>0.772</td>
<td>1.06545</td>
<td>0.91763</td>
<td>1.22841</td>
<td>0.72429</td>
<td>0.84215</td>
</tr>
<tr>
<td>1</td>
<td>0.685</td>
<td>0.84425</td>
<td>0.78813</td>
<td>0.95005</td>
<td>0.61355</td>
<td>0.73176</td>
</tr>
<tr>
<td>2</td>
<td>0.546</td>
<td>0.68981</td>
<td>0.80628</td>
<td>0.87477</td>
<td>0.52401</td>
<td>0.68870</td>
</tr>
<tr>
<td>3</td>
<td>0.592</td>
<td>0.63700</td>
<td>0.74797</td>
<td>0.87743</td>
<td>0.58099</td>
<td>0.61555</td>
</tr>
<tr>
<td>4</td>
<td>0.535</td>
<td>0.62757</td>
<td>0.77052</td>
<td>0.83717</td>
<td>0.53939</td>
<td>0.53586</td>
</tr>
<tr>
<td>5</td>
<td>0.559</td>
<td>0.58016</td>
<td>0.77977</td>
<td>0.76136</td>
<td>0.51382</td>
<td>0.52587</td>
</tr>
<tr>
<td>6</td>
<td>0.517</td>
<td>0.60927</td>
<td>0.73285</td>
<td>0.74546</td>
<td>0.50796</td>
<td>0.46528</td>
</tr>
<tr>
<td>7</td>
<td>0.531</td>
<td>0.57777</td>
<td>0.64736</td>
<td>0.74567</td>
<td>0.52213</td>
<td>0.47390</td>
</tr>
<tr>
<td>8</td>
<td>0.535</td>
<td>0.49653</td>
<td>0.72951</td>
<td>0.73310</td>
<td>0.49364</td>
<td>0.51497</td>
</tr>
<tr>
<td>9</td>
<td>0.502</td>
<td>0.52024</td>
<td>0.71509</td>
<td>0.69547</td>
<td>0.47115</td>
<td>0.46882</td>
</tr>
<tr>
<td>10</td>
<td>0.470</td>
<td>0.49797</td>
<td>0.66690</td>
<td>0.73697</td>
<td>0.46078</td>
<td>0.50323</td>
</tr>
<tr>
<td>11</td>
<td>0.469</td>
<td>0.53708</td>
<td>0.70858</td>
<td>0.69472</td>
<td>0.47727</td>
<td>0.40127</td>
</tr>
<tr>
<td>12</td>
<td>0.55770</td>
<td>0.48325</td>
<td>0.74692</td>
<td>0.70609</td>
<td>0.48603</td>
<td>0.43742</td>
</tr>
<tr>
<td>13</td>
<td>0.50317</td>
<td>0.46395</td>
<td>0.77704</td>
<td>0.68498</td>
<td>0.44946</td>
<td>0.43079</td>
</tr>
<tr>
<td>14</td>
<td>0.50046</td>
<td>0.49079</td>
<td>0.85047</td>
<td>0.68451</td>
<td>0.49535</td>
<td>0.44398</td>
</tr>
<tr>
<td>15</td>
<td>0.49613</td>
<td>0.47014</td>
<td>0.84611</td>
<td>0.66400</td>
<td>0.47850</td>
<td>0.42843</td>
</tr>
<tr>
<td>16</td>
<td>0.50447</td>
<td>0.50499</td>
<td>0.90589</td>
<td>0.65964</td>
<td>0.47061</td>
<td>0.41944</td>
</tr>
<tr>
<td>17</td>
<td>0.47549</td>
<td>0.48510</td>
<td>0.96728</td>
<td>0.69549</td>
<td>0.47810</td>
<td>0.43202</td>
</tr>
<tr>
<td>18</td>
<td>0.46991</td>
<td>0.42856</td>
<td>1.07797</td>
<td>0.61658</td>
<td>0.44490</td>
<td>0.42208</td>
</tr>
<tr>
<td>19</td>
<td>0.45545</td>
<td>0.44939</td>
<td>1.11552</td>
<td>0.62770</td>
<td>0.44443</td>
<td>0.40515</td>
</tr>
</tbody>
</table>

Figure 17. Batch loss summary
Table 5
Batch Accuracy Summary for All Training Sessions

<table>
<thead>
<tr>
<th>Epoch</th>
<th>Java Large</th>
<th>Java Small</th>
<th>Python Large</th>
<th>Python Small</th>
<th>C# Large</th>
<th>C# Small</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start</td>
<td>0.09433</td>
<td>0.12917</td>
<td>0.28967</td>
<td>0.24700</td>
<td>0.36150</td>
<td>0.31883</td>
</tr>
<tr>
<td>0</td>
<td>0.78417</td>
<td>0.72433</td>
<td>0.74900</td>
<td>0.66600</td>
<td>0.79817</td>
<td>0.77283</td>
</tr>
<tr>
<td>1</td>
<td>0.81167</td>
<td>0.76517</td>
<td>0.77800</td>
<td>0.73800</td>
<td>0.83167</td>
<td>0.79650</td>
</tr>
<tr>
<td>2</td>
<td>0.83917</td>
<td>0.80183</td>
<td>0.77300</td>
<td>0.75633</td>
<td>0.85567</td>
<td>0.80750</td>
</tr>
<tr>
<td>3</td>
<td>0.82867</td>
<td>0.81733</td>
<td>0.78817</td>
<td>0.75183</td>
<td>0.83250</td>
<td>0.82367</td>
</tr>
<tr>
<td>4</td>
<td>0.84783</td>
<td>0.81733</td>
<td>0.77950</td>
<td>0.76500</td>
<td>0.84283</td>
<td>0.84983</td>
</tr>
<tr>
<td>5</td>
<td>0.84217</td>
<td>0.83750</td>
<td>0.77633</td>
<td>0.78550</td>
<td>0.85517</td>
<td>0.84717</td>
</tr>
<tr>
<td>6</td>
<td>0.88067</td>
<td>0.82567</td>
<td>0.78717</td>
<td>0.79433</td>
<td>0.85600</td>
<td>0.86533</td>
</tr>
<tr>
<td>7</td>
<td>0.84483</td>
<td>0.83300</td>
<td>0.81167</td>
<td>0.78733</td>
<td>0.85200</td>
<td>0.86900</td>
</tr>
<tr>
<td>8</td>
<td>0.84000</td>
<td>0.85567</td>
<td>0.78667</td>
<td>0.79150</td>
<td>0.85917</td>
<td>0.85233</td>
</tr>
<tr>
<td>9</td>
<td>0.85400</td>
<td>0.84767</td>
<td>0.79100</td>
<td>0.80317</td>
<td>0.86400</td>
<td>0.86550</td>
</tr>
<tr>
<td>10</td>
<td>0.85400</td>
<td>0.85717</td>
<td>0.80617</td>
<td>0.79300</td>
<td>0.86833</td>
<td>0.85567</td>
</tr>
<tr>
<td>11</td>
<td>0.8483</td>
<td>0.84400</td>
<td>0.79667</td>
<td>0.80200</td>
<td>0.86150</td>
<td>0.88533</td>
</tr>
<tr>
<td>12</td>
<td>0.84417</td>
<td>0.85700</td>
<td>0.79000</td>
<td>0.79817</td>
<td>0.86067</td>
<td>0.87617</td>
</tr>
<tr>
<td>13</td>
<td>0.85050</td>
<td>0.86250</td>
<td>0.78250</td>
<td>0.80633</td>
<td>0.86950</td>
<td>0.87583</td>
</tr>
<tr>
<td>14</td>
<td>0.85433</td>
<td>0.85783</td>
<td>0.77117</td>
<td>0.80317</td>
<td>0.85933</td>
<td>0.87467</td>
</tr>
<tr>
<td>15</td>
<td>0.85433</td>
<td>0.86417</td>
<td>0.78050</td>
<td>0.80950</td>
<td>0.86250</td>
<td>0.87400</td>
</tr>
<tr>
<td>16</td>
<td>0.85350</td>
<td>0.85050</td>
<td>0.76067</td>
<td>0.81167</td>
<td>0.86300</td>
<td>0.88000</td>
</tr>
<tr>
<td>17</td>
<td>0.86400</td>
<td>0.85950</td>
<td>0.74267</td>
<td>0.79783</td>
<td>0.86317</td>
<td>0.87650</td>
</tr>
<tr>
<td>18</td>
<td>0.86383</td>
<td>0.87217</td>
<td>0.71533</td>
<td>0.82433</td>
<td>0.86800</td>
<td>0.87400</td>
</tr>
<tr>
<td>19</td>
<td>0.86467</td>
<td>0.87033</td>
<td>0.70117</td>
<td>0.81983</td>
<td>0.86850</td>
<td>0.88700</td>
</tr>
</tbody>
</table>
From the loss and accuracy data collected the training model, the C# small dataset had the strongest performance, and the python large dataset was the weakest. Overall, Java and C# programming languages did better in comparison to Python. The differences between the large and small datasets were minimal, and did not seem to make much of a difference to train on a larger dataset. Similar results could have been achieved using a smaller dataset and would have saved a lot of time and resources. Another way to observe the effectiveness of the RNN model other than the information collected during training is by analyzing the results of the training. The RNN model is able to generate text by printing character by character. It decides on which character to pick based on the probability of the next character occurring given the current character. These probabilities were computed during the training.
The figures below illustrate a color coded mapping of the probabilities. This is a 98 x 98 matrix of all the possible characters. The lighter the block, the lower the probability of that character combination of occurring and the darker the block the higher the probability. The color legend for the probabilities is in figure 19. Since these matrices are so large and difficult to see on a character level, figure 20 shows an example of a zoomed in section of the matrix for the large Java training. This probability matrix is a visual representation of how the model determine what character to choose next. As expected, after the first epoch, the probabilities are low and pretty similar for the entire matrix and at the end of the training, the matrix has darker spots representing the higher probabilities. Figure 20 takes a snapshot of the large java model’s probabilities at epoch 19. The characters on the y axis represent the current character and the ones on the x axis represent the probability of the next character occurring next. For example, if the current character is ‘I’ there is a much higher probability that ‘n’ will follow compared to a ‘q’

![Figure 19. Color coded probabilities](image)

Figure 19. Color coded probabilities
Figure 20. Zoomed in probability matrix to show individual characters

When zoomed out, the individual characters are difficult to see, but it provides a better snapshot and overall idea of how well the model trained. Figures 21 through 32 show the probability matrices for every training dataset after the first epoch and at the end of the training.
Figure 21. Java large dataset at epoch 0

Figure 22. Java large dataset at epoch 19
Figure 23. Java small dataset at epoch 0

Figure 24. Java small dataset at epoch 19
Figure 25. Python large dataset at epoch 0

Figure 26. Python large dataset at epoch 19
Figure 27. Python small dataset at epoch 0

Figure 28. Python small dataset at epoch 19
Figure 29. C# large dataset at epoch 0

Figure 30. C# large dataset at epoch 19
Figure 31. C# small dataset at epoch 0

Figure 32. C# large dataset at epoch 19
While it is certainly interesting to view the probability matrices for all the trainings, viewing all the images together will help provide insight in the similarities and differences of the trainings. Figures 33 and 34 show these comparisons.

**Figure 33.** Full comparison of all 6 trainings after first epoch
Even though the training statistics of batch loss and accuracy produced similar results for both the small datasets and large datasets, these probability matrices show a significant difference in the models. The larger datasets have more dark spots, illustrating that the larger the dataset the stronger the connections between the characters were.

**Figure 34.** Full comparison of all 6 trainings after last epoch
The final way to analyze the effectiveness of the character RNN is to see the results of the automated code. Code is generated by reestablishing a session from a checkpoint that was saved during training. When a session is restored the RNN generates code by selecting the next character based on the highest probability. Below there are examples of some generated code for each training dataset at the end of epochs 0, 9 and 19. The code snippets were limited to 1000 characters and after being generated they were copied into sublime text editor to highlight recognizable syntax and key words. Examples of the generated code can be seen in figures 35 through 52.

![Code Snippet](image)

**Figure 35.** Java large dataset at epoch 0, 24 errors
Figure 36. Java large dataset at epoch 9, 8 errors

Figure 37. Java large dataset at epoch 19, 7 errors
Figure 38. Java small dataset at epoch 0

Figure 39. Java small dataset at epoch 9
internal image shader shader is
    static final int ETTY = 0x9070;
    public static int STAGE_SIZE = 0x80E000;
    public static int GL_FRAMEBUFFER = 0;
    public static int ST_TE_CODE_CONTACT_SIZE = 0;
    public static final int GL_TRANSFORM_FEEDBACK = 0xBDEE;
    public final int GL_COMPRESSED_SIZE_E = 1x9977;
    public static int FT_STRIDES_LENGTH = 0x8EE4;
    public final int GL_COMPRESSED_SIZE = 0x8E43;
    public final int GL_TRANSFORM_FEEDBACK = 0x800000;
    public static final int EGL11 = 0x9002;
    public static final int GL_COMPRESSED_STENCIL = 0x9170;

    public float simpleName;
    private float maxAdvance;
    private float max[] maxAdvances;
    private int mask;
    private int copyFrom = 0, mask = 0, i < 1; i++) {
        if (maximumCapacity < 0) throw new IllegalArgumentException("Couldn't be >= size : " + index);
        if (maximumCapacity < maximumCapacity) {
            map.copyStream(matcher);
        } else {
            if (matcher != null) {
                if (match == null) {
                    if (matcher) {
                        if (part != maximumCapacity) {
                            S
                        }
                    }
                }
            }
        }
    }

Figure 40. Java small dataset at epoch 19

Python('day', 'man')

    index = Index(['A', 'B']).mean()
    result = df_resalpes.comapt.index(idx.index)

    def test_decode(self, data, datetime(2016, 7, 3),
        datetime(2014, 1, 1, 0, 1, 1):
        tm.assert_frame_equal(date_index(data, df, date_time64_as_index_index)

    def test_default_index_name(self):
        self.df = DataFrame([[1, 1, 1], [1.0, 2.0, 1.0, 1.2, 1.0, 2.0, 1.0], [1, 1], [1, 1, 1, 1],
            [1, 1], [1, 1, 1, 1, 1, 1, 1, 3]]
            'B': [1, 2, 3, 3, 3]), index_idx, expected_day=

1, dtype='full',

    'date':
        'timedelta64': datetime(2014, 1, 1, 1, 0)),
        date_range(10, 2, 3, 1, 0),

Figure 41. Python large dataset at epoch 0
Pectors, `pandas_tests_io_formats.python.org/pandas_tests.parser`) and not hasattr(df, 'display.max_columns'), ('date', 'date'), 'datetime', 'timedelta64[ns]')
result = values.to_timestamp(how='end')
expected = Series([Timestamp('20110101'), tz='US/Eastern'],
                  index=[('20110101'), tz='US/Eastern'])
assert_series_equal(result, expected)
result = to_timestamp('2016-01-01', tz='US/Eastern')
assert_series_equal(result, exp)
dates_a = pd.to_datetime(data, copy=False)
tm.assert_series_equal(rs, rs)
df = DataFrame({'A': [1, 2], 'B': [1, 1]})
result = self.read_csv(StringIO(data), skiprows=['sum'])
tm.assert_frame_equal(result.columns, expected_columns)
def test_set_axis_number(self):
    # see gh-16747
    df = DataFrame({'A': [1, 2],
                    'B': ['A', 'B']})
    result = self.read_csv(StringIO(data), skiprows=['sum'])
tm.assert_frame_equal(result.columns, expected_columns)
def test_sum_axes(self):
    df = DataFrame({'A': [1, 2],
                    'B': ['a', 'b', 'c']})
    df = self.read_csv(StringIO(data), skiprows=['sum'])
tm.assert_frame_equal(result, expected)
def _check_colors(af, bins, colors):
    ax = df.plot.bar(stacked=False, ax=af,
                     sines='loc')
    self._check_bool_result(ax, sines='loc')
    self._check_colors(ax.get_lines(), linecolors=['joe', 'joe'])
    ax = df.plot.scatter(x='x', y='y', c='b')
    self._check_colors(ax.patches[:10], ax=ser,
                        ax=ax)
axes = df

Figure 42. Python large dataset at epoch 9

Params([[1], [1], ['B', 'B']])
assert_frame_equal(expected, result)

def test_set_axis_number(self):
    # see gh-16747
    df = DataFrame({'A': [1, 2],
                    'B': [1, 1]})
    result = self.read_csv(StringIO(data), skiprows=['sum'])
tm.assert_frame_equal(result.columns, expected_columns)
def test_sum_axes(self):
    df = DataFrame({'A': [1, 2],
                    'B': ['a', 'b', 'c']})
    df = self.read_csv(StringIO(data), skiprows=['sum'])
tm.assert_frame_equal(result, expected)
def _check_colors(af, bins, colors):
    ax = df.plot.bar(stacked=False, ax=af,
                     sines='loc')
    self._check_bool_result(ax, sines='loc')
    self._check_colors(ax.get_lines(), linecolors=['joe', 'joe'])
    ax = df.plot.scatter(x='x', y='y', c='b')
    self._check_colors(ax.patches[:10], ax=ser,
                        ax=ax)
axes = df

Figure 43. Python large dataset at epoch 19
Figure 44. Python small dataset at epoch 0

Figure 45. Python small dataset at epoch 9
def test_all_reduce_add(self):
    self._test_rosenbrock(
        lambda params: optim.Adagrad(self.data_type, [batch_size]),
    )

def test_baddcmul_self):
    self._test_basic_cases(
        self._test_reduce_backward_hooks(self._test_reduce_helper(
            callbacks.cpdn(),
            callbacks.ones_like(constants))
    )

def test_add_arguments(self):
    self._test_rosenbrock(
        lambda params: optim.Adagrad, self._test_rosenbrock_
        lambda params: optim.Adagrad, lambda t: [], 'lr', 'bilinear'),
        'recurrent_dropout': [10, 10],
        lambda params: aras, learningRate=1e-3, true, lr=1e-2),
    )

def test_max_pool_dir(self):
    self._test_rodensime()
    x = torch.randn(5, 5).cuda(0)
    x = Variable(torch.randn(5, 5), requires_grad=True)

Figure 46. Python small dataset at epoch 19

Figure 47. C# large dataset at epoch 0
private static readonly Dictionary<string, List<CmdletInfo>> lockedCommandExceptedCmdAndHand
lers = new CommandInfo(string modulePath, string currentBlock, string childName, CommandOrigin comma
ndOrigin)
{
    {
        return completionCompleter.GetCommandArguments();
    };

    return baseCommandInputHashCode t
} // IsPathParametervendor

internal void RemoveActivityHostConfiguration(ParserImplementationContext context)
{
    return selectorset;
}

/// <summary>

public PSVariable ConvertToCmdlet(
    string name,
    bool inSupported,
    Obsectable true)
{
    while (null == cmdletParameters)
    {
        throw new ArgumentNullException("sourcePaths");
    }

    if (context == null)
    {
        throw PSTraceSource.NewArgumentNullException("paths");
    }

    ProviderInfo provider = null;
    List<ProviderInfo> helpFileIdentifier = null;

try
{
    _providerInstance = value;
}
// Check if there is not have been valid. )
this.ScenarifyArray = propertyValue;

Figure 48. C# large dataset at epoch 9

Figure 49. C# large dataset at epoch 19
Figure 50. C# small dataset at epoch 0

Figure 51. C# small dataset at epoch 9
When comparing the automatically generated code there are many observations to be noted. Keeping the original goals of this research in mind, we can analyze the effectiveness of the RNN model. In relation to programming languages, all the languages performed well with automatic code generation, however the automatically generated Java and C# code produced better code. These examples were more clear and the neural network learned how to open and close brackets and parenthesis, they also identified key words such as “var”, “String”, “public” and “void”. The python model also was able to produce definitions of functions. None of the languages were able to produce code that had no compile errors. In comparison to dataset lengths, there was not much differences between the large and smaller dataset. This is interesting considering the larger dataset had created deeper connections between characters from the probabilities matrices. From a practical standpoint there was not much difference in performance of the large dataset models and
small dataset models. The final unit of measure was observing the model at different times in the training. For all languages and dataset sizes, the generated code at epoch 20 was much better. The models learn how to open and close parenthesis, return variables in its own method definition
5.1 Future Work

Recurrent neural networks are extremely powerful and amazing deep learning algorithms to learn probabilities and generate text. Further research in automatic code generation using RNNs would be to expand on this model and fine tune the model to produce a higher batch accuracy and lower batch loss. One way to do this would be to train for longer epochs given the proper computational resources. Other ways to create better generated outputs would be to run through the training datasets and select training code that would produce the optimal results. For example, the training datasets that were used in this experiment had both source code as well as unit test code, thus the generated code had a combination of unit test syntax and source code syntax. Further research would be to limit the training datasets to only source code or unit test code to create better generated code. Java and C# languages performed the best for the RNN model, so further research would be to continue to use one of these languages to produce better results or to see if there are changes to be made in the model to allow Python to achieve similar results as the other two languages.

Another emerging field in deep learning is reinforcement learning. Currently researchers in the field of reinforcement learning have been able to train deep learning programs to play complex games and these neural network computers have been able to beat their human competitors (Shaikh, Sunil and Kumar. Reinforcement learning is based on a reward system, a positive or negative reward is given after an action is taken, the model or agent then begins to learn the patterns and choose the option with the highest reward. This type of learning could be paired with the RNN in code generation. For example, once the
model is trained on the character probabilities, it would run through a RL network and receive positive rewards when it generates a sequence of code that is able to compile. This would provide a little more structure for the RNN and allow for less random and more thoughtful code generation.

In other future work the character RNN model could also be combined with other NLP deep learning networks to be able to generate code based on the understanding of what type of code to generate. The possibilities of research in this field is so vast and these RNN can be used to create many interesting applications to further technological growth.

5.2 Conclusion

Deep learning is an emerging field with many different possibilities for computers to understand and process large amounts of data. In NLP, recurrent neural networks are able to learn sequences of words and characters to be able to quite accurately generate its own text sequences. Using character based language models in an RNN to generate programming code we were able to evaluate the actual effectiveness of these networks and provide insight and analysis of how these RNNs can create code. Java and C# programming languages performed much better than Python. While the dataset size did contribute to the level of complexities and patterns observed in the model, ultimately the dataset size did not contribute significantly to be able to produce better code. The longer the model was able to train the better it was at generating code. The combination of a Java or C# smaller dataset for a longer amount of epochs would be able to generate the best code. We were able to solve the problem of what is really needed to train a neural network to be able to produce sequences of automatically generated code.
REFERENCES


Tanz, Jason. "Soon We Won't Program Computers. We'll Train Them Like Dogs." *Wired.* Conde Nast, 03 June 2017. Web.


Source code for model.py

```python
import tensorflow as tf
from tensorflow.contrib import rnn
from tensorflow.contrib import layers

class Model:
    def __init__(self, args):
        self.args = args

        tf.set_random_seed(0)
        self.learning_rate = tf.placeholder(tf.float32, name='learning_rate')
        self.batchsize = tf.placeholder(tf.int32, name='batchsize')
        self.pkeep = tf.placeholder(tf.float32, name='pkeep')

        # GRUcell initialization
        self.grucells = [rnn.GRUCell(args.internal_size) for _ in range(args.num_layers)]
        self.dropcells = [rnn.DropoutWrapper(cell, input_keep_prob=self.pkeep) for cell in self.grucells]
        self.multicell = rnn.MultiRNNCell(self.dropcells, state_is_tuple=False)
        self.multicell = rnn.DropoutWrapper(self.multicell, output_keep_prob=self.pkeep)

        # input, outputs and initial state size
        self.X = tf.placeholder(tf.uint8, [None, None], name='X')
        self.Y_ = tf.placeholder(tf.uint8, [None, None], name='Y_')
        self.initial_state = tf.placeholder(tf.float32, [None, args.internal_size * args.num_layers], name='initial_state')

        self.Xoh = tf.one_hot(self.X, args.vocab_size, 1.0, 0.0)
        self.Yoh_ = tf.one_hot(self.Y_, args.vocab_size, 1.0, 0.0)

        self.Yr, self.H = tf.nn.dynamic_rnn(self.multicell, self.Xoh, dtype=tf.float32,
                                             initial_state=self.initial_state)
        self.H = tf.identity(self.H, name='H')

    # Softmax layer
```

APPENDIX A
self.flatten_Y = tf.reshape(self.Yr, [-1, 
args.internal_size])
self.logits_Y = layers.linear(self.flatten_Y, 
args.vocab_size)
self.flatten_Y_ = tf.reshape(self.Yoh_, [-1, 
args.vocab_size])
self.loss = 
tf.nn.softmax_cross_entropy_with_logits(logits=self.logits_Y, 
labels=self.flatten_Y_)
self.loss = tf.reshape(self.loss, [self.batchsize, 
-1])
self.Yout = tf.nn.softmax(self.logits_Y, name='Yout')
self.Y = tf.argmax(self.Yout, 
1)
self.Y = tf.reshape(self.Y, [self.batchsize, 
-1], 
name='Y')
self.train_step = 
tf.train.AdamOptimizer(self.learning_rate).minimize(self.loss)

# display statistics
self.seq_loss = tf.reduce_mean(self.loss, 1)
self.batch_loss = tf.reduce_mean(self.seq_loss)
self.accuracy = tf.reduce_mean(tf.cast(tf.equal(self.Y_, 
tf.cast(self.Y, tf.uint8)), tf.float32))
self.loss_summary = tf.summary.scalar("batch_loss", 
self.batch_loss)
self.accuracy_summary = 
tf.summary.scalar("batch_accuracy", self.accuracy)
self.summaries = tf.summary.merge([self.loss_summary, 
self.accuracy_summary])
Source code for train.py

```
#!/usr/bin/env python3
import tensorflow as tf
import argparse
import os
import time
import math
import numpy as np
from utils import Utils
from utils import Progress
from model import Model

def main():
    parser = argparse.ArgumentParser(
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)
    # data
    parser.add_argument('--data', type=str, default='csharp-large',
                        help='data containing code text files')
    # save checkpoints directory
    parser.add_argument('--save_dir', type=str,
                        default='checkpoints/checkpoints-csharp-large',
                        help='directory to store checkpointed models')
    # log directory
    parser.add_argument('--log_dir', type=str,
                        default='logs/logs-csharp-large/log',
                        help='directory to store tensorboard logs')
    # rnn internal size
    parser.add_argument('--internal_size', type=int, default=512,
                        help='size of RNN hidden state')
    # number of layers
    parser.add_argument('--num_layers', type=int, default=3,
                        help='number of layers in the RNN')
    # batch size
    parser.add_argument('--batch_size', type=int, default=200,
                        help='minibatch size')
    # sequence length
    parser.add_argument('--seq_length', type=int, default=30,
                        help='RNN sequence length')
    # number of epochs
    parser.add_argument('--num_epochs', type=int, default=20,
                        help='number of epochs')
    # learning rate
    parser.add_argument('--learning_rate', type=float, default=0.001,
```
# learning rate
parser.add_argument('--vocab_size', type=int, default=98,
                    help='learning rate')

# dropout probability
parser.add_argument('--keep_prob', type=float, default=0.8,
                    help='probability of keeping weights in
the hidden layer')

args = parser.parse_args()
train(args)

def train(args):
    data = Utils(args)

    rnn = Model(args)

    # save to log
    timestamp = str(math.trunc(time.time()))
    summary_writer = tf.summary.FileWriter(args.log_dir +
                                           timestamp)

    # save models
    if not os.path.exists(args.save_dir):
        os.mkdir(args.save_dir)
    saver = tf.train.Saver(max_to_keep=1000)

    display_freq = 50
    _50_batches = display_freq * args.batch_size *
    args.seq_length
    progress = Progress(display_freq, size=111 + 2, msg="Training
on next" + str(display_freq) + " batches")

    input_state = np.zeros([args.batch_size, args.internal_size *
                            args.num_layers])
    init = tf.global_variables_initializer()
    session = tf.Session()
    session.run(init)
    step = 0

    # training loop
    for x, y_, epoch in data.minibatch_sequencer(data.codetext,
                                                  args.batch_size,
                                                  args.num_epochs):
        feed_dict = {rnn.X: x, rnn.Y_: y_,
                     rnn.initial_state: input_state,
                     rnn.learning_rate: args.learning_rate,
                     rnn.pkeep: args.keep_prob,
                     rnn.batchsize: args.batch_size}
_, y, output_state = session.run([rnn.train_step, rnn.Y, rnn.H], feed_dict=feed_dict)

if step % _50_batches == 0:
    feed_dict = {rnn.X: x, rnn.Y_: y_, rnn.initial_state: input_state, rnn.pkeep: 1.0,
                 rnn.batchsize: args.batch_size}
    y, l, bl, acc, summ = session.run([rnn.Y, rnn.seq_loss, rnn.batch_loss, rnn.accuracy, rnn.summaries],
                                       feed_dict=feed_dict)
    data.print_learning_stats(x, bl, acc, data.batches_in_epoch, step, epoch)
    summary_writer.add_summary(summ, step)

    # display a short text generated with the current weights and biases (every 150 batches)
    if step // 3 % _50_batches == 0:
        data.print_code_gen_header()
        ry = np.array([[data.convert_from_alphabet(ord("K"))]])
        rh = np.zeros([1, args.internal_size * args.num_layers])
        for k in range(1000):
            feed_dict = {rnn.X: ry, rnn.pkeep: 1.0, rnn.initial_state: rh, rnn.batchsize: 1}
            ryo, rh = session.run([rnn.Yout, rnn.H], feed_dict=feed_dict)
        rc = data.sample_from_probabilities(ryo, topn=10
        if epoch <= 1 else 2)
            print(chr(data.convert_to_alphabet(rc)), end="")
        ry = np.array([[rc]])
        data.print_code_gen_footer()

    # save a checkpoint (every 500 batches)
    if step // 10 % _50_batches == 0:
        saved_file = saver.save(session, args.save_dir + '/rnn_train_' + timestamp, global_step=step)
        print("Saved file: " + saved_file)

    # display progress bar
    progress.step(reset=step % _50_batches == 0)

    # loop state around
    input_state = output_state
    step += args.batch_size * args.seq_length

if __name__ == '__main__':
    main()
APPENDIX C

Source code for utils.py

```python
import numpy as np
import glob
import sys

class Utils:
    def __init__(self, args):
        data = args.data
        batch_size = args.batch_size
        seq_length = args.seq_length

        if data == 'java-small':
            self.directory = "data/java-small/**/*.java"
        elif data == 'java-large':
            self.directory = "data/java-large/**/*.java"
        elif data == 'python-small':
            self.directory = "data/python-small/**/*.py"
        elif data == 'python-large':
            self.directory = "data/python-large/**/*.py"
        elif data == 'csharp-small':
            self.directory = "data/csharp-small/**/*.cs"
        elif data == 'csharp-large':
            self.directory = "data/csharp-large/**/*.cs"
        else:
            raise Exception("data type not supported: {}".format(data))

        self.codetext, self.coderanges = self.read_data_files(self.directory)
        # display some stats on the data
        self.batches_in_epoch = len(self.codetext) // (batch_size * seq_length)
        self.print_data_stats(len(self.codetext), self.batches_in_epoch)

        self.vocab_size = args.vocab_size

    def read_data_files(self, dir):
        codetext = []
        coderanges = []
        datasetlist = glob.glob(dir, recursive=True)
        for codefile in datasetlist:
            code = open(codefile, "r")
            print("Loading file: " + codefile)
            start = len(codetext)
            codetext.extend(self.encode_text(code.read()))
```

58
end = len(codetext)
coderanges.append({'start': start, 'end': end,  
"name": codefile.rsplit("/", 1)[-1]})

code.close()
if len(coderanges) == 0:
sys.exit("No training data has been found.")
return codetext, coderanges

def minibatch_sequencer(self, raw_data, batch_size,  
sequence_size, nb_epochs):
    data = np.array(raw_data)
data_len = data.shape[0]
    # using (data_len-1) because we must provide for the
sequence shifted by 1 too
    nb_batches = (data_len - 1) // (batch_size *  
sequence_size)
    assert nb_batches > 0, "Not enough data, even for a
single batch. Try using a smaller batch_size."
    rounded_data_len = nb_batches * batch_size *  
sequence_size
    xdata = np.reshape(data[0:rounded_data_len], [batch_size,  
nb_batches * sequence_size])
ydata = np.reshape(data[1:rounded_data_len + 1],  
[batch_size, nb_batches * sequence_size])

    for epoch in range(nb_epochs):
        for batch in range(nb_batches):
            x = xdata[:, batch * sequence_size:(batch + 1) *  
sequence_size]
y = ydata[:, batch * sequence_size:(batch + 1) *  
sequence_size]
x = np.roll(x, -epoch, axis=0)  # to continue the
    text from epoch to epoch (do not reset rnn state!)
y = np.roll(y, -epoch, axis=0)
yield x, y, epoch

def print_learning_stats(self, X, batch_loss, batch_accuracy,  
epoch_size, index, epoch):
    """Display utility for printing learning statistics""
    print()
    # epoch_size in number of batches
    batch_size = X.shape[0]  # batch_size in number of
sequences
    sequence_len = X.shape[1]  # sequence_len in number of
characters
    start_index_in_epoch = index % (epoch_size * batch_size *  
sequence_len)

    # print statistics
    batch_index = start_index_in_epoch // (batch_size *
sequence_len
    batch_string = "batch {}/{} in epoch {}\n", format(batch_index, epoch_size, epoch)
stats = "{: <28} batch loss: {:.5f}, batch accuracy: {:.5f}".format(batch_string,
    batch_loss, batch_accuracy)

print()
print("TRAINING STATS: {}".format(stats))

def print_data_stats(self, datalen, epoch_size):
datalen_mb = datalen / 1024.0 / 1024.0
print("Training text size is {:.2f}MB".format(datalen_mb) +
      " There will be {} batches per epoch".format(epoch_size))

def sample_from_probabilities(self, probabilities, topn):
    size = self.vocab_size
    p = np.squeeze(probabilities)
    p[np.argsort(p)[-topn]] = 0
    p = p / np.sum(p)
    return np.random.choice(size, 1, p=p)[0]

def get_probabilities(probabilities, size, topn):
    p = np.squeeze(probabilities)
    p[np.argsort(p)[-topn]] = 0
    p = p / np.sum(p)
    return np.random.choice(size, 1, p=p)[0]

  # returns a list of code points from a string
  def encode_text(self, string):
      return list(map(lambda a: self.convert_from_alphabet(ord(a)), string))

  # encode
  def convert_from_alphabet(self, a):
      if a == 9:
          return 1
      if a == 10:
          return 127 - 30
      elif 32 <= a <= 126:
          return a - 30
      else:
          return 0

  def convert(a):
      if a == 9:
          return 1
      if a == 10:
return 127 - 30
elif 32 <= a <= 126:
    return a - 30
else:
    return 0

# decodes
def convert_to_alphabet(self, char, avoid_tab_and_lf=False):
    if char == 1:
        return 32 if avoid_tab_and_lf else 9
    if char == 127 - 30:
        return 92 if avoid_tab_and_lf else 10
    if 32 <= char + 30 <= 126:
        return char + 30
    else:
        return 0

def decode(char, avoid_tab_and_lf=False):
    if char == 1:
        return 32 if avoid_tab_and_lf else 9
    if char == 127 - 30:
        return 92 if avoid_tab_and_lf else 10
    if 32 <= char + 30 <= 126:
        return char + 30
    else:
        return 0

def print_code_gen_header(self):
    print()
    print("{:-^111}".format('Generating random text from learned state'))

def print_code_gen_footer(self):
    print()
    print("{:-^111}".format('End of generation'))

class Progress:
    def __init__(self, maxi, size=100, msg=""):  
        self.maxi = maxi
        self.p = self.__start_progress(maxi)()  # () to get the iterator from the generator
        self.header_printed = False
        self.msg = msg
        self.size = size

    def step(self, reset=False):
        if reset:
            self.__init__(self.maxi, self.size, self.msg)
        if not self.header_printed:
def __print_header(self):
    print()
    format_string = "0{%: ^{}100%"
    print(format_string.format(self.msg))
    self.header_printed = True

# Bresenham's algorithm to display progress
def __start_progress(self, maxi):
    def print_progress():
        dx = maxi
        dy = self.size
        d = dy - dx
        for x in range(maxi):
            k = 0
            while d >= 0:
                print('=', end='', flush=True)
                k += 1
                d -= dx
            d += dy
            yield k

    return print_progress
Source code for generate_code.py

# !/usr/bin/env python3

import tensorflow as tf
import argparse
import numpy as np
import subprocess

...  

java-large:
  checkpoints/checkpoints-java-large/rnn_train_1502340505-6000000
  checkpoints/checkpoints-java-large/rnn_train_1502340505-8400000
  checkpoints/checkpoints-java-large/rnn_train_1502340505-16800000

java-small:
  checkpoints/checkpoints-java-small/rnn_train_1502748756-3000000
  checkpoints/checkpoints-java-small/rnn_train_1502748756-4200000
  checkpoints/checkpoints-java-small/rnn_train_1502748756-8400000

python-large:
  checkpoints/checkpoints-python-large/rnn_train_1502404122-6000000
  checkpoints/checkpoints-python-large/rnn_train_1502404122-8400000
  checkpoints/checkpoints-python-large/rnn_train_1502404122-16800000

python-small
  checkpoints/checkpoints-python-small/rnn_train_1502333713-3000000
  checkpoints/checkpoints-python-small/rnn_train_1502333713-4200000
  checkpoints/checkpoints-python-small/rnn_train_1502333713-8400000

csharp-large:
  checkpoints/checkpoints-csharp-large/rnn_train_1502415447-6000000
  checkpoints/checkpoints-csharp-large/rnn_train_1502415447-8400000
  checkpoints/checkpoints-csharp-large/rnn_train_1502415447-16800000

csharp-small:
  checkpoints/checkpoints-csharp-small/rnn_train_1502391634-3000000
  checkpoints/checkpoints-csharp-small/rnn_train_1502391634-4200000
def main():
    parser = argparse.ArgumentParser(
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)
    # data
    parser.add_argument('--data', type=str, default='java-large',
                        help='data containing code text files')
    # save checkpoints directory
    parser.add_argument('--checkpoint', type=str,
                        default='checkpoints/checkpoints-java-large/rnn_train_1502340505-168000000',
                        help='directory to get checkpointed models')
    # rnn internal size
    parser.add_argument('--internal_size', type=int, default=512,
                        help='size of RNN hidden state')
    # number of layers
    parser.add_argument('--num_layers', type=int, default=3,
                        help='number of layers in the RNN')
    # sequence length
    parser.add_argument('--seq_length', type=int, default=30,
                        help='RNN sequence length')
    # learning rate
    parser.add_argument('--vocab_size', type=int, default=98,
                        help='learning rate')
    # characters for generated code
    parser.add_argument('--play_len', type=int, default=1000,
                        help='learning rate')

    args = parser.parse_args()
    play(args)

    # compile('java', 'random.java')
    # compile('python', 'random.py')

def play(args):
    # f = open('random.java', 'w')
    # f = open('random.py', 'w')
    count = 0
    with tf.Session() as session:
        saver = tf.train.import_meta_graph(args.checkpoint + '.meta')
        saver.restore(session, args.checkpoint)
y = np.array([[encode(ord("P")),]])
h = np.zeros([1, args.internal_size * args.num_layers])

print_code_gen_header()
print('P', end=''
for i in range(args.play_len):
    feed_dict = {'X:0': y, 'pkeep:0': 1.0,
                'initial_state:0': h, 'batchsize:0': 1}
yo, h = session.run(['Yout:0', 'H:0'],
                    feed_dict=feed_dict)
    c = get_probabilities(yo, size=args.vocab_size,
                          topn=2)
y = np.array([[c]])
c = chr(decode(c))
print(c, end=''
if c == '\n':
    count = 0
else:
    count += 1
if count == 100:
    print(''
count = 0
print_code_gen_footer()

# Utils for generating code

def compile(language, file):
    if language == 'java':
        command = 'javac ' + file
    elif language == 'python':
        command = 'python3 ' + file

e = open('errors.txt', 'w')
proc = subprocess.Popen(command, shell=True,
                        stdout=subprocess.PIPE, stderr=subprocess.STDOUT,
                        universal_newlines=True)
proc.wait()
for line in proc.stdout:
    print(line)
e.write(line)

def print_code_gen_header():
    print()
    print("r{:-^111}".format('Generating random code'))
def print_code_gen_footer():
    print()
    print("└{:^111}".format('End of generation'))

def encode(a):
    if a == 9:
        return 1
    if a == 10:
        return 127 - 30
    elif 32 <= a <= 126:
        return a - 30
    else:
        return 0

def decode(char, avoid_tab_and_lf=False):
    if char == 1:
        return 32 if avoid_tab_and_lf else 9
    if char == 127 - 30:
        return 92 if avoid_tab_and_lf else 10
    if 32 <= char + 30 <= 126:
        return char + 30
    else:
        return 0

def get_probabilities(probabilities, size, topn):
    p = np.squeeze(probabilities)
    p[np.argsort(p)[::-topn]] = 0
    p = p / np.sum(p)
    return np.random.choice(size, 1, p=p)[0]

# run main program
if __name__ == '__main__':
    main()
APPENDIX E

Source code for probabilities.py

#!/usr/bin/env python3

import tensorflow as tf
import argparse
import numpy as np
from tabulate import tabulate

...
checkpoints/checkpoints-csharp-small/rnn_train_1502391634-
84000000

def main():
    parser = argparse.ArgumentParser(
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    # saved checkpoints directory
    parser.add_argument('--checkpoint', type=str,
                        default='checkpoints/checkpoints-
java-large/rnn_train_1502340505-168000000',
                        help='directory to get checkpointed
models')

    # rnn internal size
    parser.add_argument('--internal_size', type=int, default=512,
                        help='size of RNN hidden state')

    # number of layers
    parser.add_argument('--num_layers', type=int, default=3,
                        help='number of layers in the RNN')

    args = parser.parse_args()
    play(args)

def play(args):
    with tf.Session() as session:
        saver = tf.train.import_meta_graph(args.checkpoint +
'.meta')
        saver.restore(session, args.checkpoint)

        num = []
        for n in range(98):
            num.append(n)

        characters = []
        for n in num:
            characters.append(chr(decode(n)))

        char_prob = []
        for c in characters:
            y = np.array([encode(ord(c))])
            h = np.zeros([1, args.internal_size *
args.num_layers])

            for i in range(1):
                feed_dict={'X:0': y, 'pkeep:0': 1.0,
'initial_state:0': h, 'batchsize:0': 1}
                yo, h = session.run(['Yout:0', 'H:0'], feed_dict=}
feed_dict)
    p = get_prob_list(yo, size=98)
    char_prob.append(p)

    print(tabulate(char_prob))

def encode(a):
    if a == 9:
        return 1
    if a == 10:
        return 127 - 30
    elif 32 <= a <= 126:
        return a - 30
    else:
        return 0

def decode(char, avoid_tab_and_lf=False):
    if char == 1:
        return 32 if avoid_tab_and_lf else 9
    if char == 127 - 30:
        return 92 if avoid_tab_and_lf else 10
    if 32 <= char + 30 <= 126:
        return char + 30
    else:
        return 0

def get_prob_list(probabilities, size):
    p = np.squeeze(probabilities)
    p[np.argsort(p)[:-size]] = 0
    p = p / np.sum(p)
    return p

if __name__ == '__main__':
    main()