[Github Repo] [CodaLab Worksheet]

Task Definition

Motivation and Project Scope

From Amazon’s Alexa to MicroSoft's Cortana, these AI assistants rely on the ability to translate natural language instructions to machine commands, e.g launch itunes, search for the weather. Although the number of commercial AI assistants is growing, they usually help people perform daily tasks. It would be great to develop an AI assistant to help programmers and computer scientists with their work too. Although this project does not aim to surpass Alexa or Cortana, it strives to come up with an AI program that can assist users of Unix systems by translating English instructions to Unix commands. Specifically, the scope of this project will be limited to instructions to navigate directories, finding files, opening files and listing directory content. Given its success, the model and techniques used in this project may be generalized to translating other commands, and it will especially benefit users who are unfamiliar with or unaware of the command line utilities and would like to exploit such utilities.

Input & Output

In the scope of this project, our ***input*** will be English instructions describing tasks of the following category: navigating directories, finding files, opening files and listing directory content. An example of the input could be: *can you find lalala.docx for me*

The ***output*** will be the corresponding Unix commands that could perform the tasks in the instruction. An example of the output could be: *find / -name "lalala.docx"*

Evaluation Metric

There are 3 metrics used to evaluate the system implemented in this project:

PM: On average, the percentage of segments of each command that are correct. E.g:
when the prediction is (find / -name somefile.txt) and oracle/target is (find . –name somefile.txt), PM is 3/4 = 75%

CO: The percentage of output sequences that has the correct ordering. For example,
if the command sequence of the oracle is (A B C D), our output (A B F D) will
be considered as having the right ordering.

CM: Percentage of completely correct output/command

Infrastructure

Dataset Generation

Real people have different ways of expressing the same meaning, or defining the same task. In order to get a comprehensive collection of English instructions, I consulted with Google, my programmer friends, and used a crowdsourcing tool (Upwork) to come up with different paraphrases for each type of instructions and synonyms for action verbs, e.g. ‘find’. For example, I will ask people to help me paraphrase “find something for me”, then I would get many answers, and all of which will map to the same Unix command “find / -name something”. I also substituted the arguments with many different arguments, using synonyms, and combining different phrases to generate more data points.

Training and Validation Set

Training set contains 5035 pairs of instruction and command, while the Validation set contains 298 pairs. The types of instructions are: finding a file/folder, listing what’s in a directory or catting a file, go to certain directory, go certain levels up directory tree, and go back to previous folder.

The validation set contains only instructions that have structures previously **unseen** in the training set to measure how good the model is at learning the underlying structures of this translation problem. Please refer to comments in data_gen.py for details on what unseen structures mean. A simple explanation is that given only “bring somefile.txt” and “give me somefile.txt” in training set, I want to test whether the model will learn that "bring me somefile.txt", a new structure, means the same thing and should map to the same command. Note that the validation set does not use new vocabularies. I also wanted to see if the model learns how to distinguish argument (somefile.txt) from action (bring) by letting the validation set use file names never appeared in the training set.

Approach

The Challenges

The main challenges in this project are:
1. Learning/extracting the same tasks behind different phrasing of an instruction
2. Learning to ignore rhetorics that doesn’t add anything to the task description (e.g. “Can you please”, “Help me”)
3. Learning to distinguish the argument part of instructions and don’t translate them (e.g. “somefile.txt” in “find somefile.txt for me”)
4. Dealing with numerical arguments that give instruction an iterative meaning (e.g. “please go 3 level up”)

Oracle

The sequence of Unix commands I, a computer science student, would write out for English instructions of any complexity.

Baseline

A Linear Regression Model, and we only train the W and b from $Y = W*X + b$, where X is the encoded instruction sequence (input) and Y is the command sequence (output).

The Model

The model chosen to translate English instructions to Unix commands is 2-layered RNN with LSTM Cells for hidden units, a representation of the model is shown below:

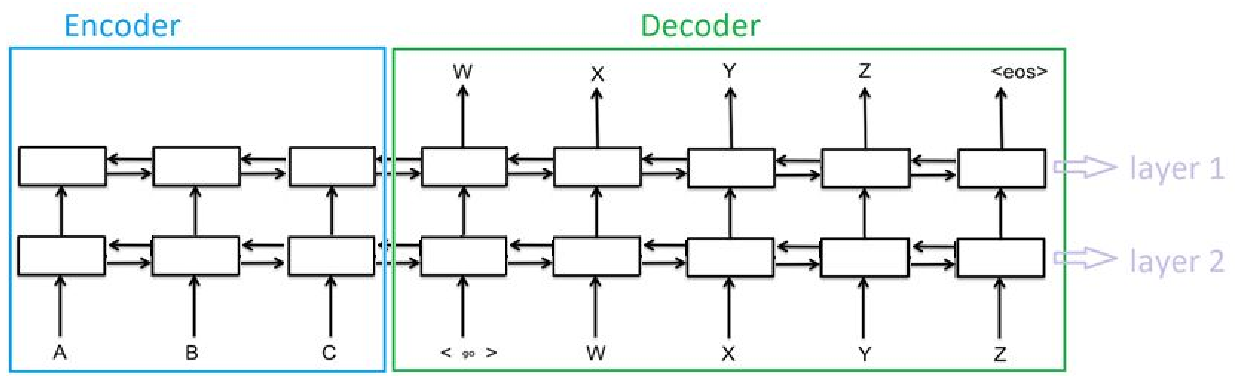


Figure. Model Sketch
*There are also **3 key additions** to the vanilla RNN model, which will be discussed later on.

Each small rectangle represents a hidden state, aka each LSTM cell. Each English Instruction (input sequence) and each Unix command (output/target sequence) need to be encoded into a **sequence of integers**. In my case, both input sequences and output sequences are **padded,** with special character 'PAD'**,** to length 50, since I don't have any instructions or unix commands that can be tokenized to more than 50 tokens. The padding is done because instructions and commands could both come in various lengths, and it's not very practical to construct a RNN model for every pair of (instruction_length, command_length). With padding, I can just build 1 model to handle the (50, 50) case.

For the input sequences, each value are within the range [0, 40,000) which corresponds to 40,000 different english vocabularies. This comes from an existing assumption that there are 40,000 common english words in machine translation[1]. **Each value in the output sequence of the Decoder is within the range [0, 4,000)** which corresponds to 4,000 unique vocabularies/tokens in the Unix commands of my interest, considering the scope of this project.

Continuing the discussion about the Encoder, a hidden state at some time T will see an input representing a special end-of-line character, signalling the end of this input sequence, and the decoder part of the RNN will begin. For each of encoder and decoder, the weights on the edge between any time t-1 and t is the same. Although the encoder and decoder can share weight, I am following the common practice of using different parameters. The model implemented is also bidirectional, since bidirectional model is considered to have more modeling power[2]. The output at time t-1 of the decoder will be fed in as the input to the decoder at time t, e.g. output W at time=t-1 will become input at time=t, shown in the Decoder part of the picture above.

[1] TensorFlow, *"Sequence-to-Sequence Models"*
https://www.tensorflow.org/versions/r1.0/tutorials/seq2seq

[2] Leonardo Araujosantos, *"Machine Translation Using Recurrent Neural Networks"*
https://leonardoaraujosantos.gitbooks.io/artificial-inteligence/content/recurrent_neural_networks/machine-translation-using-rnn.html

A hidden state at time = t, $h_{t,}$ stores weighted combination of the input value at time = t and the information carried forward from the previous hidden state h_{t-1}. Each hidden cell itself is a LSTM which has logistic gates: **write gate, read gate, keep gate**. The write gate is set by the rest of the RNN and when it's set to be one, information/state is stored into it. Similarly, when the read gate is set to be 1, the state stored can come out and influence future states of the RNN. The keep gate will keep the state stored when it's set to 1, and will "forget" when it is set to 0. Choosing LSTM is because it has been proven to be great at remembering information for long periods of time, perform well in machine translation, and back-propagation (the algorithm used to train my RNN) through LSTM cells are also fairly efficient.

The reason behind choosing 2 layers is that I assume there are at least 2 types of information/structures for the RNN to learn in my translation problem. The first layer could learn the order of vocabularies, e.g "come back one level", 'back' should follow 'come'. The second layer, for example, could learn to distinguish between rhetorics and meaningful actions within an instruction: "can you please find somefile.txt" where 'find' is an action and 'can you please' is just rhetoric that adds no value to the translation task.

3 Key Additions to Vanilla RNN
To solve our specific machine translation problem, there needs to be some customization to the vanilla RNN model. During the project, I have discovered 3 additional features that greatly improved the model's performance on both the training set and validation set. They prove to be very effective for the English to Unix Command translation tasks involved in this project.

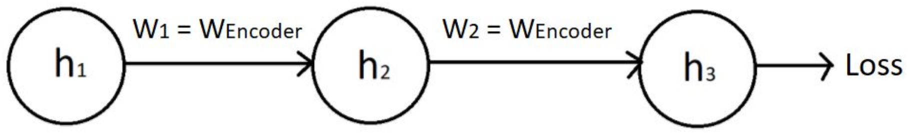
1. Argument Detection
During the pre-processing of input data, the model uses a linear classifier with feature extraction to detect whether a segment in instructions/commands is an argument or not. Examples of an argument could be "somefile.txt" in "can you please fine somefile.txt for me"; or "somefolder" in "can you open somefolder". Note that we are only considering the case where each instruction only contains 1 argument, due to the scope of this project and the commands of interest.

2. Semantic Equivalence
The model tries to uncover meaning of numbers in English instructions, e.g. "go up 3 levels" suggests a loop structure. During pre-processing of data, the "loop structure" is mapped to an equivalent flattened expressions, e.g. "go up 3 levels" => "go up levels levels levels", this is for better Sequence to Sequence translation performance, and it's proven to be effective.

3. Early Stopping
Since we want the model to generalize to different English expressions, I need a way to stop the model from overfitting to training data. Early stopping prevents overfitting to training data and allows better validation set performance. In the model, I stop when the current evaluation/dev loss is larger than all past 10 values.

The Training Algorithm - Back Propagation

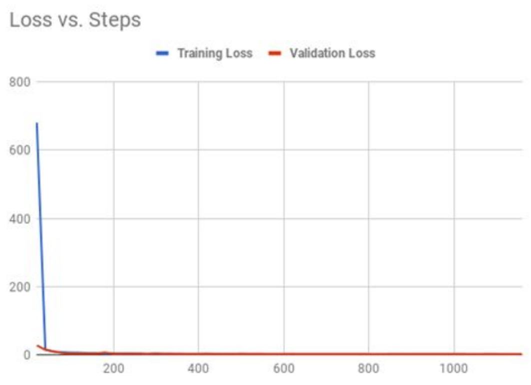
The algorithm used to train my RNN is Back Propagation, it is chosen because it is the best algorithm available to me. The Tensorflow code specifically uses Truncated Backpropagation to only consider propagation up to a specific number of steps for tractable computation. Bearing similarities to feedforward nets, back propagation on RNN is quite straight forward, a simplified example is shown below (removed some parts from model):



To constraint $W_1 = W_2 = W_{Encoder}$:
We need $\Delta W_1 = \Delta W_2$ for every update to the weights
So We compute $\frac{\partial Loss}{\partial W2} = \frac{\partial Loss}{\partial h3}\frac{\partial h3}{\partial W2}$, and $\frac{\partial Loss}{\partial W1} = \frac{\partial Loss}{\partial h3}\frac{\partial h3}{\partial h2}\frac{\partial h2}{\partial W1}$
And our $\Delta W_1 = \Delta W_2 = \eta\,\frac{1}{2}\left(\frac{\partial Loss}{\partial W2} + \frac{\partial Loss}{\partial W1}\right)$, where η is our learning rate
$W_1 \leftarrow W_1 - \Delta W_1$
$W_2 \leftarrow W_2 - \Delta W_2$

Results and Analysis

Besides measuring the 3 metrics defined earlier, the loss function, sampled softmax loss, should serve as a good general indicator of how the model is performing as it approximates how many parts/tokens of a translated command matches with the target command.



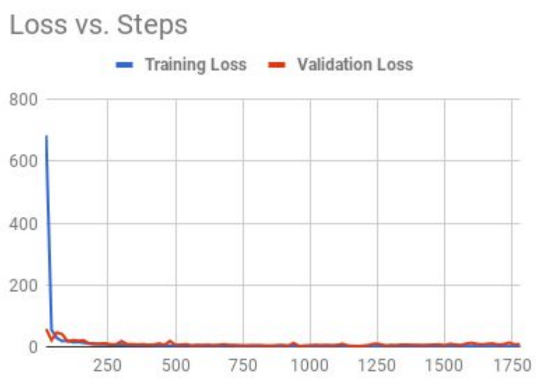


Figure. Loss with 3 key additions *Figure. Loss with Vanilla 2-layer RNN*

From the above graph, we can see that the validation loss drops to a smooth horizontal line with 3 key additions. Moreover, the validation loss after incorporating the 3 key additions reduces significantly, which is hard to show on the graph but we have the tabulated data below:

	PM	CO	CM
Training	99.6%	100%	97.7%
Validation	99.6%	100%	97%

3 Metrics evaluated with 3 Key additions
*Please refer to page 1 - Evaluation for definition of metrics

	PM	CO	CM
Training	65.85%	84.47%	20.48%
Validation	48.34%	88.59%	5.03%

3 Metric evaluated without any additions

The 2 layered RNN is great at learning different phrasing and their underlying meanings. It especially produced good results with the "Key Additions". After thorough examination, the model is able to detect arguments with high accuracy, and "flattening" iterative instructions also greatly improved the translation accuracy of them. Comparing to the baseline, whose performance is very poor, the result of our model looks very promising.

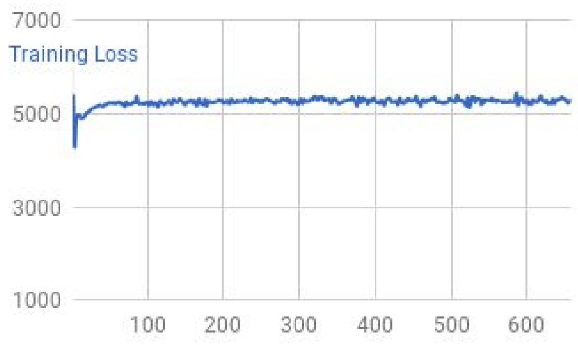


Figure. L2 Training Loss vs Training Epoch - Linear Regression Model

I also evaluated the 3 metrics on the **baseline**:
PM: Overall parts matched per command in percentage: 9.78% in Training Set

CO: Percentage of correct ordered command: 95.81% in Training Set (Meaningless when M1 rate is so low)
CM: Number of complete matches: 8 out of 5030 in Training Set, which is 0.16%

Obviously, the performance of our system is way better than that of the baseline.

Literature Review

There have been other attempts to build systems for Natural Language to Commands translation. During my research, the system that is most comparable to my project is described in a paper from 2008 by Anders Pedersen from the Technical University of Denmark: "Natural Language Interface to Command Line File Operations"[3].

Anders' approach is drastically different from mine. Instead of resorting to RNN for machine translation. He first translates both natural language (English) phrases and functionality of commands to logic by analyzing sentence/command structures with linguistic analysis and knowledge about command line utilities. During translation, his system then maps natural language sentences to commands that describe the same logic.

Since one of the purpose of using RNN is so that us humans don't have to analyze natural language structures and translation principles ourselves, and we expect the model to learn these things by training them; I would consider Anders' approach to be **orthogonal** to mine. However, his approach does bare some similarity to the first 2 key additions to my system, as I also tried to supplement my system with some human analysis and ingenuity.

Since he is not testing outside of his system's knowledge base and vocabularies, perfect accuracy is expected (although not explicitly stated in his paper). However, his approach might become unrealistic if we try to expand the pool of commands that we try to translate to. This is because writing out logic representations of all the functionalities of a very large pool of commands is just intractable amount of work. It's also very difficult to have a comprehensive English language model, analyzing all of those different expressions and their corresponding logic.

In conclusion, I believe that my system offers both competitive translation accuracy and more possibility to expand to a larger pool of commands. That's not saying that Anders' system is worse than mine. I believe it would be beneficial to find balance between human ingenuity and machine learning. Maybe there is a sweet spot, the right amount of human analysis added to machine learning, and could make full-scale natural language to commands translation both accurate and achievable.

---

[3] Anders Pedersen, Natural Language Interface to Command Line File Operations
http://etd.dtu.dk/thesis/221233/bac08_23_net.pdf

Error Analysis

With the 3 key additions, the 2 layered RNN model should be able to:

1. Correctly translate English instructions with 1 argument embedded in it
2. Map different paraphrases to the same command
3. Accurately translate instructions with iterative meaning

To verify these abilities (pros) of the system, I designed 3 experiment each corresponding to one of the 3 aspects respectively. `python translate.py --decode --experiment`

Experiment 1
In this experiment, we test on the translation of 177 instructions (out of the validation set) that contains one argument, e.g. “bring me angular.js thanks”. The results of evaluating the 3 metrics are below:

PM	CO	CM
100%	100%	100%

The results indicate that the system is indeed very good at distinguishing arguments in the input and bringing them verbatim to the output.

Experiment 2
This experiment focuses on translating instructions that are merely paraphrases of each other and expect the same output command. 30 different instructions with the same underlying task, “find asdnoiwn.txt”, is used as a representative dataset for this experiment; and the results are very good, which indicates that my system performs well in the second aspect listed above.

PM	CO	CM
100%	100%	100%

Experiment 3
This experiment aims to find how well the system achieves aspect 3: Accurately translate instructions with iterative meaning. The dataset used in this experiment consists of instructions to navigate certain numbers of directories up, e.g. “please go up 6 directory”, and the findings are very interesting.

PM	CO	CM
99.3%	100%	92%

Upon close inspection on the instructions that are translated incorrectly, I discovered something interesting. Here are the instructions that failed to translate correctly:

1. can you move 10 level up
2. please go 10 directory up
3. please go up 10 directory
4. please go up 10 level
5. please go 10 level up
6. please go 10 steps up
7. can you go out 10 level
8. can you move 10 steps up

I noticed that they all represent the same task, and all have a loop structure of length 10. Moreover, all of them incorrectly translated to “cd ../../../../../../../../../”, which is moving 9 levels up.

However, 2 of their equivalence was translated correctly:

1. can you go out 10 directory
2. can you move 10 directory up

This led me to suspect 2 things:

1. Maybe the system is not doing a great job at aspect 2: Map different paraphrases to the same command
2. Maybe the nature of the data in my training set caused this issue. Since 10 is the maximum number of levels used in the training set, the model is very likely just under-trained to handle a “flattened” input size that big, remember that a *flattened* input is “go up level level level” for “go up 3 level”. The smaller number of iterations are remembered better by the model because the bigger iterations will have flattened input sequence that include the smaller iterations, e.g. “go up 10 level” will produce a flattened sequence that includes the sequence of “go up 9 level”.

To dig deeper and verify my hypothesis, I conducted an additional experiment, see details below:

Experiment: 3-extra
In this experiment, I trained a model using dataset that was almost exactly like the ones I had before, except that now the maximum number of iterations is 11 (basically I added instructions like “go up 11 level”). My goal is to verify hypothesis number 2, and the results seem to partially confirm my theory.

With the new setup, the instructions that failed to translate correctly are:

1. please go up 10 level
2. can you move 10 level up
3. can you go out 1 level

4. can you go out 1 directory

So there is a reduce in the number of wrong translations for instructions equivalent to “move up 10 levels”, since 10 is no longer the maximum iterations seen during training. Yet instruction 1 and 2 still translated to “cd ../../../../../../../../../”, aka “move up 9 levels”.

Moreover; instruction 3 and 4, which should be translated to “cd ../”, were incorrectly translated to “cd -”. This led me to suspect that the ambiguous nature of the English language and the fact that we use similar words to mean different things also contributed to these incorrect translations. For example, although “got out” is always related to “cd ../”, “go back” could mean “cd -” and “cd ../” if appearing as “go back to parent” in the training set. This ambiguity probably led the model to be stuck at a local minima instead of fully learning the principles of this machine translation problem.

So hypothesis 1 probably also bares some truth, in the sense that our model possibly isn’t trained enough or isn’t expressive enough to capture all the principles of this translation problem. In the future, I will explore increasing the model’s expressiveness, e.g. increasing number of layers. But for the time frame of this project and considering the tradeoff between training time and expressiveness, I will stay with the current configuration.

Summary

Given the promising results of this project, it is possible that some of the techniques used can be extrapolated to other types of machine translations. There are still areas for improvement, such as detecting arguments considering the context rather than just analyzing the features of the argument alone, detecting multiple arguments, expanding vocabularies, plural handling, etc. All in all, I would love to continue exploring after this course.

[Github Repo] [CodaLab Worksheet]

References:
[1] TensorFlow, *“Sequence-to-Sequence Models”*
https://www.tensorflow.org/versions/r1.0/tutorials/seq2seq
[2] Leonardo Araujosantos, *“Machine Translation Using Recurrent Neural Networks”*
https://leonardoaraujosantos.gitbooks.io/artificial-inteligence/content/recurrent_neural_networks/machine-translation-using-rnn.html
[3] Anders Pedersen, *“Natural Language Interface to Command Line File Operations”*, Technical University of Denmark
http://etd.dtu.dk/thesis/221233/bac08_23_net.pdf