Machine Translation for Programming Languages

Nikhil Sharma
CSC 722 (001) – Fall 2017
Adv. ML (solo project)
nsharm12@ncsu.edu

Abstract

Deep Neural Networks (DNNs) have shown great success on a variety of complex tasks. One variant of those, the Long Short-Term Memory (LSTM) networks have shown great success on a broad range of problems that have some sort of temporal component to it. When applied to the field of Natural Language Processing, they have yielded good results on tasks of machine translation and chatbots. The aim of this project is to study their capabilities on a similar task, which is aimed at machine translation for programming languages. It is inspired from two excellent literature works, Sequence to Sequence Learning and Learning to Execute. These works have provided LSTM based models for English to French translation, and Python code snippet evaluation respectively. In this project, I present a LSTM based neural model that learns to convert short code snippets from Python to their equivalent in Lua language. Further, this project extensively comments on the applicability of these models for conversions to other languages, the associated limitations and potential ways to address them. A part of this project also evaluates the results presented in the paper on Learning to Execute, and further comments on those tasks that are similar (on a high-level) to the task this project deals with.

1 Introduction

Sequence to Sequence Learning [1] and Learning to Execute [2] are inspired works that report great capabilities of DNNs on standard challenges in natural language processing. Ref. [1] presents a setup where a LSTM model learns to convert English text to its equivalent in French, and lays foundation for other machine translation tasks for human languages. Ref. [2] presents and gives insight into the capabilities of LSTMs on tasks of computer program evaluation. Specifically, they evaluate the performance of three tasks: simple addition of numbers, execution of nested lines of Python code (which can be executed in linear time, and the result of which is a number), and the memorization task where the network must memorize the entire input sequence and reproduce it in that order. In this project, I developed a LSTM based neural model that learnt to convert such simple Python code snippets to their equivalent in Lua language. To the best of my knowledge, there is no such work in the literature. I believe this is because of the huge limitations it has in being used for practical and scalable purposes, and also by the fact that there exist compliers that can achieve the end result in a different way. It is to note that these compilers achieve the goal by converting one language's code to another machine level code that is understandable by the other language (and its compiler). Although this is what a practical use-case of machine translation would look like, this is not the same as generating the human-readable code itself. I take this project as an opportunity to explore such a task and understand more about the benefits (and capacity) of DNNs applied to language processing tasks. The rest of the report is organized as follows. The next section presents an understanding of the problem statement in technical terms and gives an overview of the relevant literature work. Thereafter, it talks about the problem setup and the proposed architectures for the tasks mentioned in

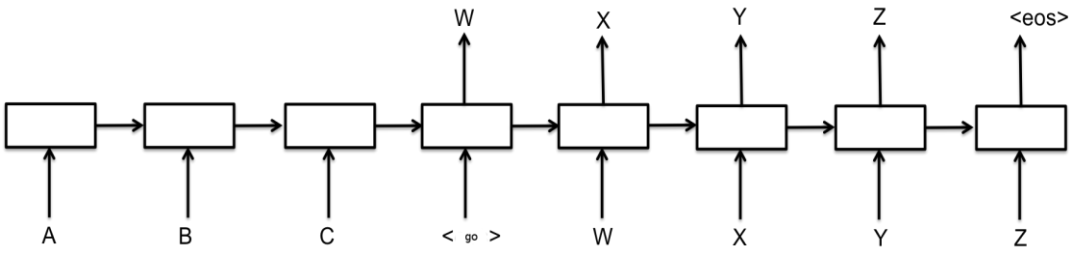

Figure 1: A sequence-to-sequence learning setup. The input sequence that the encoder receives is ABC. To start the decoding process, the decoder is fed a <go> key and the last hidden state from the encoder. Each character that the decoder predicts at a given time-step is fed back as the next input in the sequence. It stops predicting when it outputs a <eos> key which indicates the End of Sequence.

[2], and the program translation task that this project proposes, while intermediately speaking of relevant results/approaches in [1] and [2]. The later sections talk more about the results obtained and their scalabilities.

2 Understanding the Problem

There are some research works that used a variant of Recursive Neural Networks to evaluate symbolic mathematical and logical formulas [3-5], but computer programs are more complex than those mathematical expressions. Maddison and Tarlow [6] presented a language model of program text which learned different structures in a computer program (like *for* loops, conditional *if-else* statements), and is the closest in terms of problem modelling to this project. While they use a parse tree structure based approach to learn the model, this project sees it as a character level language modelling task. Specifically, I have formulated the task as a character level sequence-to-sequence learning problem with a recurrent neural network. Figure 1 gives an example of such a setup.

Since there are long term dependencies to be dealt with, I have used the LSTM model, like in [1] and [2]. In terms of program translation, only a certain category of Python code snippets is considered. The most important requirement is for the program to be *O(n)* in terms of evaluation time, and *O(1)* in terms of memory. These restrictions, which are similar to those imposed in [2], come from the structure of the Recurrent Neural Network (RNN) itself, which can only perform a single pass over the program and it has a limited amount of memory to do that.

3 Data

In terms of operations, the following form the sample space: addition, subtraction, multiplication, division, variable assignment, if-else statement, single for-loop. Every sampled input (program) has a combination of one or more of these operations, picked in a random way such that the program has a correct Python syntax and is also consistent. The last line in every program is a print statement, the output of which is an integer (which may or may not be the results of the preceding statements). To generate the data, the script provided by [2] was used. The original script creates Python code snippets as inputs, and the numerical evaluation of those are targets. For my task on translation from Python to Lua, I modified the target generation process in the script, to give equivalent Lua snippets as the required targets. Figure 2 shows some sample scripts.

3.1 Parameterizing the Data

The complexity of the learning problem depends on the complexity of the code snippets that are inputted. Ref. [2] uses two metrics, *nesting* and *length* to govern the complexity of the data generated.

INPUT

```
h=49910;
d=34710
for x in range(20):d+=(h+31879)
print(d)
```

TARGET

```
h=49910;
d=34710
for x=1,20 do d=d+(h+31879); end
print(d)
```

INPUT

```
e=13;
d=3
for x in range(2):d+=(93+(e-(42 if 30<32 else 97)))
print(d)
```

TARGET

```
e=13;
d=3
for x=1,2 do d=d+(93+(e-(if 30<32 then 42 else 97 end))); end
print(d)
```

INPUT

```
g=29330
for x in range(1):g-=1324
print((26032 if g>88371 else 45290))
```

TARGET

```
g=29330
for x=1,1 do g=g-1324; end
print((if g>88371 then 26032 else 45290 end))
```

INPUT

```
print((80290 if 98613>75134 else (19*94506)))
```

TARGET

```
print((if 98613>75134 then 80290 else (19*94506) end))
```

Figure 2: Sample input and target pairs. Inputs are Python code snippets; targets are their equivalent in Lua. Note that the current version of this project does not correctly encode if-else statements in Lua. For instance, in the last input-target pair shown above, the current syntax for the target would be to first assign the if-else value to a variable, and then print that variable. This has a few challenges that are discussed in Section 6.3

Length refers to the maximum number of digits that can be present in any part of the input, while nesting refers to the number of times different operations are combined with each other. The evaluation of the learning capabilities is also done based on the complexity of the input: it is harder for the network to learn inputs that have very long digits in a single number and/or are deeply nested. In this project, the same metrics were used to assess the results. The values of the parameters that were chosen are discussed under the Experiments section.

4 Model and Learning Process

Inspired from [1], I used a character level learning setup. The LSTM takes in the entire input one character at a time, and then produces the output one character at a time. Reserved symbols GO and EOS control the start and stop of the output prediction sequence.

4.1 Input Formatting

In my experiment, "!" was used as a signal for GO, and "." was used to indicate the End of Sequence. The vocabulary size was 52, consisting of the following characters:

"*abcdefghijklmnopqrstuvwxyz <>:.!,;+=-\n*/()1234567890*"

Note that "\n" is just one character that indicates a new-line.

A given input string is converted to a one-hot vector based on the vocabulary index of each character in it, which is then fed to a lookup table to convert it into a N-dimensional vector. In this project, N was chosen as 52 itself, since it is not too large to result in a prohibitive dimension. Every target string is appended with the EOS character.

4.2 Network Architecture

This project used the LSTM architecture, just like in [1] and [2]. LSTMs are good at handling long-term dependencies with the help of memory cells. This feature is crucial, since, in a task of program translation/evaluation, a variable that is used towards the end of the program could have been declared at the start of the program, requiring the network to look all the way back to get the variable's value.

In my experiments, I used the traditional LSTM architecture that is governed by three gates, the input gate, forget gate, output gate, each of which are sigmoidal units. In addition, it has a tanh node to control the parts of input and output that effect the cell and hidden state. In essence, these operations can be defined as follows:

$$LSTM: \{{h_t}^{l-1}, {h_{t-1}}^{l}, {c_{t-1}}^{l}\} \rightarrow \{{h_t}^{l}, {c_t}^{l}\}$$

$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ tanh \end{pmatrix} M_{2n,4n} \begin{pmatrix} {h_t}^{l-1} \\ {h_{t-1}}^{l} \end{pmatrix}$$

$${c_t}^{l} = f \odot {c_{t-1}}^{l} + i \odot g$$

$${h_t}^{l} = o \odot tanh({c_t}^{l})$$

where *h*, *c* represent the hidden and cell states respectively, subscripts represent the sequence time, superscripts represent the layer, *i*,*f*,*o* represent the three gates, and *M* is a linear mapping that the

network learns. Element wise multiplication is represented by ⊙

In my project, I used a 4 layer LSTM structure, with 400 cells per layer. It was unrolled for 100 time steps for the baseline strategy and 160 time steps for the mixed strategy (discussed next). Like in [1], zero masking was incorporated to pad shorter length sequences in each batch with zeros, and to ignore those zeros while doing the forward and backward passes.

At each time step in the decoding process, the output of the decoder is passed via a softmax layer of size equal to the size of the vocabulary, which allows the output prediction to be expressed as a probability distribution over all possible characters in the vocabulary. In accordance, the network is optimized on the negative log-likelihood measure.

4.3 Learning Strategies

As part of my experiments, two of the four learning strategies proposed in [2] were used:

Baseline Strategy: All the training samples have a fixed length and fixed nesting value. For each pair of length and nesting, a suitable model is trained. Due to time consuming training schedules, I tested this strategy for only one pair of (length, nesting) values, namely (5,2). With this trained model, I assess the performance on test data of the same parameters as well as lower and higher parameters.

Mixed Strategy: The training set is composed of all possible sample pairs of length and nesting less than some fixed value, and hence the name *mixed.* The intuition behind this is to present the network with both, easy and tough examples within each batch, which helps the network learn in a step-by-step manner. Such strategies have shown to overcome hard learning problems, which in this case would be to train the network with just one set of large values of (length, nesting). In this project, the mixed strategy was tested with $3 \leq \text{length} \leq 6$ and $\text{nesting} \leq 4$

Note: I used teacher forcing while training the LSTMs. This strategy, which has widely been used by the deep learning community in tasks like this, is such that the decoder at each time step receives the correct input as opposed to feeding the previously generated output (which would happen in real-time decoding). This has proven to improve the learning on a lot of relevant tasks, because it eliminates the case of the output prediction from a decoder at a time step being wrong because the input to that decoder itself was wrong (which is a direct result of the previous step's output being incorrect).

5 Experiments

The entire code was written in Lua, and the models where built from scratch using Torch (a deep learning framework). The code for most of the experiments can be found at

https://github.com/nikhilsharma93/machine_translation_for_programming_language

5.1 Training Setup

The data is setup as mentioned in the earlier sections. Specifically, 500k samples with nesting = 2 and length = 4 were used for the baseline experiment. For the mixed strategy experiment, a total of 500k samples were divided across different lengths and nesting, with 50k samples from nesting = 1, 100k samples from nesting = 2, 150k samples from nesting = 3, and 200k samples from nesting = 4. Even within a given nesting value, the quantity of samples was proportionate to the length parameter (3:4:5:6). The network was trained on a NVIDIA GeForce GTX 980 GPU with 4GB of memory. A batch-size of 100 and initial learning rate of 0.07 were used. Stochastic gradient descent gave the best learning schedules against Adam and rmsprop. I also applied high momentum (0.9). For the baseline strategy, the network was trained for about 10 iterations, beyond which there were no *significant* improvements. It took about 50 hours to train the network. For the mixed strategy, the

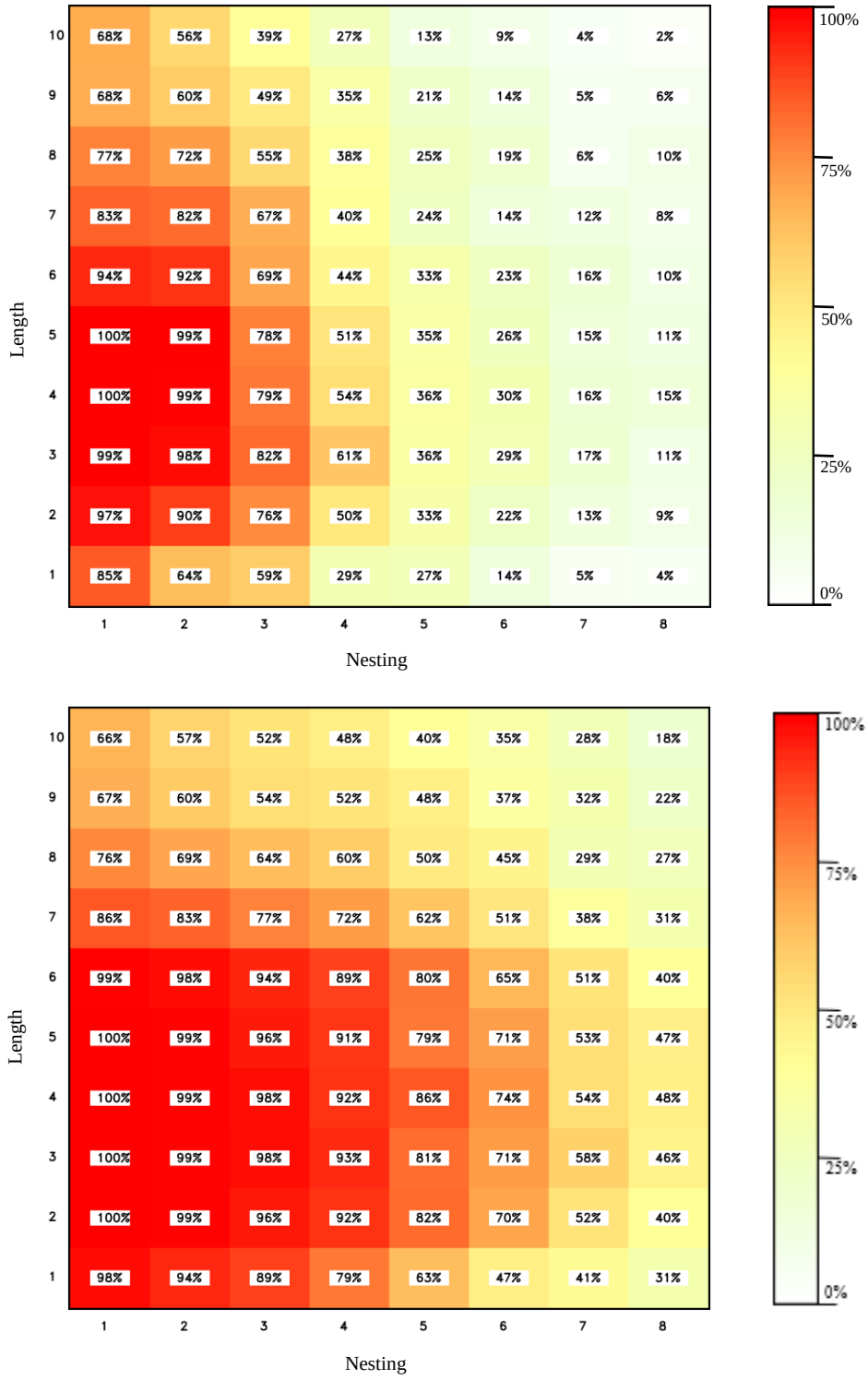

Figure 3: Accuracies obtained with baseline and mixed strategies (top and bottom). It has been color coded into a heat map for better visualization.

network was trained for about 18 epochs, beyond which the network's learning had become too slow, it seemed to have converged, and I had to cut it off due to time constraints. It took about 110 hours to train the network.

While LSTMs solve the vanishing gradient problem, they are still susceptible to exploding gradients, and hence this project clipped gradients above ± 5. The learning rate was decreased once, by a factor of 2, after the 8^{th} epoch in baseline training, and after 12^{th} epoch in mixed strategy training.

5.2 Results

5.2.1 Metrics

The raw metrics for training the network were the NLL scores. However, these measures are not intuitive to understand. The best option would be to visually see the code outputs and comment on the results. Although this would give a good estimate, the need for a general arithmetic metric exists. To address this, the best metric I could come up with was to compare the output and target on a pattern by pattern basis, where each pattern is one entire chunk of text that exists between two *spaces.* The logic behind choosing this metric was to let it estimate the similarity measure (in terms of the number of characters that are not the same between output and target), and mainly account for some room of error. For instance, if the output had "556" instead of "55" in the middle of the entire code, and if we were comparing characters by their exact spatial locations, every character after 556 in the output would be misaligned only because of the one incorrect digit "6". The proposed metric, hence, corrects for the alignment by first dividing the entire output and target into smaller blocks (based on the *space* character) and then checks for spatial similarity within each block. Note that this still has issues like the inability to account for misalignments due to extra space between characters (like "5 5 6"), but such issues are complicated to handle, and more significantly, are rare. They are beyond the scope of this project.

A similarity score is obtained using the metric discussed above, which is converted into a percentage score by dividing with the length of the target sequence. Such percentage values were calculated for about 100 test samples, and were averaged to give the overall percentage accuracy.

5.2.2 Baseline Strategy

The training approximately converged with training NLL of 0.71124 and test NLL of 0.7346. Figure 3 (top) shows the accuracies with a heat-map. While the model was trained with (length, nesting) = (5, 2), its performance was evaluated on all possible pairs between 1 to 8 for length and 1 to 4 for nesting.

5.2.3 Mixed Strategy

The training slowed down at training NLL of 12.75 and test NLL of 12.11. However, these scores were dominated by a few tough examples, and the training converged well on an overall basis (however, it could get better with different hyperparameters). Due to time constraints, I had to stop the training with the corresponding trained model. Figure 3 (bottom) shows the accuracies on the same test set that was used to test the baseline strategy.

Results on a few test sets that show the input, output, and target triplets are reported in the appendix section of this report. They have not been included in this section because of their length.

Note: While the memorization task in [2] and sequence to sequence learning problem in [1] reported to have significantly boosted the model's performance by reversing the input sequences, that technique did not give good results with my experiments. My best guess as to the logical reason behind that, if any, is that computer syntax would lose its intrinsic structure when reversed (this is still a weak guess).

6 Comments, Scalability and Limitations

This section is intended to give detailed comments on the results obtained, the scalability of the proposed method in terms of longer codes and other languages, and possible improvements to that.

6.1 General Comments

Based on the results obtained, the mixed strategy appears to perform significantly better on a wider range of parameters (of length and nesting). As the length and/or nesting values increase, the network's predictive power decreases. This is in-line with the results obtained by [2] on the memorization tasks (discussed in Section 7). It is interesting to note that even though the accuracies for these high values are not decent, the network appeared to have learnt the basic structure, and most of the errors come from its inabilities to correctly output the numerical values rather than the syntaxes. This is discussed further under 6.4.

6.2 The Obvious Limitations

Some of the limitations of this method come from the very nature of the way codes are structured. The first of them is with the class of programming languages that cannot work at all. If the input is in an interpreted language like Python that does not require variable type declarations, and if the output is a complied language like C which requires every variable to be explicitly declared, then there is no way for the network to predict a *type* for every *variable* it converts, since that is usually inferable only at run-time. Another limitation occurs with the length of the program being converted. If there are 200 lines of code, in which a variable is declared (or defined) in the first line, and is only used in the last line, then this is a very long-term dependency even for LSTMs (or other similar variants like GRUs).

6.3 Other Limitations

There are some other limitations that occur due to limited computational power and/or expressing capability of LSTMs. One of them occurs with the length of the digits that are passed in. As seen in the results, as the length of the numbers increases, it becomes harder to even memorize them (discussed in section 7), leave alone performing computations on them. Another issue comes with teaching the network to generate new variables on its own, as and when required. As noted in figure 2, the correct syntax for the Lua code in the last example would require an assignment of the result of the if-else clause to a new variable in the first place, followed by printing that variable. Figure 4 (a) compares these syntaxes. The problem here comes because the network must now generate a new variable "x" (like in figure 4 (a)) on its own. It should make sure that the variable name "x" was never used for any other values (or in case it was, it should come up with a new one). This becomes a problem while training the network because if different targets provided to the network had different variable names for such cases, the network might jointly learn those names with the examples, which is incorrect. Figure 4 (b) illustrates this problem. To my knowledge, it does not seem possible to make the network understand what those variables mean, and make it differentiate between the variable name "x" and any other variable name that has "x" as part of it.

6.4 Potential Improvements

The problems mentioned in section 6.2 have a central component: the LSTMs ability to distinguish between those substrings that will change in between input and output, against those that will stay intact, and against those that will have to be generated. I believe these problems can be approached by introducing priors for each longest sub-string that falls in one of those categories, and learning a conditional LSTM. For instance, if we have a 20 digit long number, it is going to remain intact in the output as well, so we could add a one-hot encoding with that 20 digit number that acts as a prior.

To partly overcome the problem of the LSTM generating new variable names, we can teach it to generate a specific symbol (which is decided and fixed at the start, just like EOS and GO) whenever a variable name, that has to be generated, is flexible. Those symbols than then be replaced with some other algorithms.

```
print((if 98613>75134 then 80290 else (19*94506) end))
```

```
if 98613>75134 then x = 80290; else x = (19*94506) end
print(x)
```

(a)

```
a = 245;
b = (13*12);
c = (a+b);
if 98613>75134 then x = 80290; else x = (19*94506) end
print((x+c))
```

(b)

Figure 4: (a): The top line is an example of the syntax the network was trained on, and the bottom line shows the correct syntax in Lua. Note that the correct syntax would require the network to generate the variable name "x". (b): An example showing the difficulty teaching the network to generate a correct variable name. If this example is given as a target to the network, then it might learn the name "x" to occur with other constituents of the code. However, any variable name other than "a", "b", and "c" should be an equally likely output from the network since they are all unused variables.

7 Comparison with Learning to Execute

The work done in this project and the one done in Learning to Execute [2] has many similar problem formulations, and it is interesting to compare those results. Since the inputs in both the cases are mostly the same Python code snippets, the comparison gives good insights. Ref. [2] formulates two main tasks: program evaluation and input memorization. In program evaluation, an LSTM based network receives Python code snippets as inputs and is trained to output the numerical value that the snippet evaluates to. In the memorization task, the input is a sequence of random numbers, and the LSTM has to output the same sequence. Figures 5 and 6 give examples of these problem definitions.

The results on the program evaluation task in [2] show the difficulty for LSTMs to represent a highly nested structure. Although (for the model) understanding the exact nesting protocol is more critical for an evaluation task, it is only slightly less important for the conversion task, which requires the model to be able to generate the numerical conditions in the correct place in the nesting. As seen in the results in this project, while the network is able to recreate the equivalent nested structure by correctly rearranging all the "if-s", "else-s", "for-s", and so forth, it is not so efficient with the numerical operators and the numerical values themselves.

In terms of memorizing the structure and values in the input, the memorization task in [2] provides some outside insights. Though this was catered to memorize only numbers (thereby reducing the input space to only 10 digits, making it harder to differentiate between the possible output values), it still gives an understanding of the performance as the length increases. In this project, the structures with high nesting values (~ 5-8) and high lengths (~ 8-10), which had "for" and "if-else" blocks, had variables being used about 100 characters apart from where they were declared. This appears to be a tough long-term dependency for the LSTM.

8 Closing Comments

The work done in this project presents some abilities of LSTMs in machine translation tasks where the long-term dependencies are crucial. Although these architectures are nowhere near being able to convert complete codes, they are promising when restricted to small code snippets. While the results

Input:

```
j=8584
for x in range(8):
  j+=920
b=(1500+j)
print((b+7567))
```

Target: 25011.

Input:

```
i=8827
c=(i-5347)
print((c+8704) if 2641<8500 else 5308)
```

Target: 12184.

Figure 5: Examples for the program evaluation task in [2]. (Source: Ref. [2])

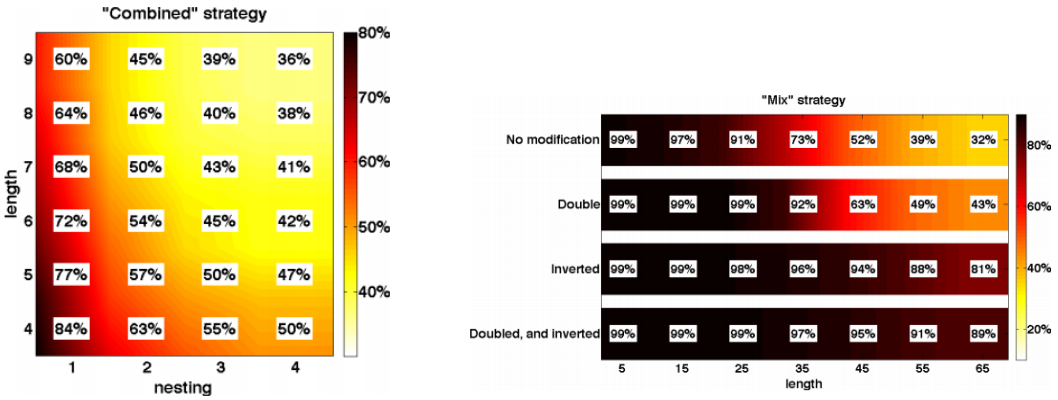

*Figure 6: **Left:** Results obtained by [2] on program evaluation. "Combined" strategy is similar to the "mix" strategy, and the technical details of that are ignored in this comparison. **Right:** Results obtained by [2] on the memorization task. "No modification" is the case of providing the input as it is, "double" refers to providing the same input twice (horizontally repeating the input sequence two times), "inverted" refers to reversing the input. (Source: Ref. [2])*

show that they are not efficient over parameters larger than what they were trained with, the improvements suggested in the later sections of this report are, in my opinion, very promising. Further, the results obtained in this project might not be the optimal ones with the described architecture. I did not have time and resources to experiment with varying the hyper-parameters like batch-sizes, learning schedules, number of layers, etc. This project used only the LSTM architecture, and to my understanding there might be other mechanisms/improvements that can better handle the tasks presented herein.

References

[1] Sutskever, I., Vinyals, O., and Le, Q. (2014). Sequence to sequence learning with neural networks. arXiv:1409.3215

[2] Wojciech Zaremba, Ilya Sutskever (2014) Learning to execute. arXiv:1410.4615

[3] Zaremba, Wojciech, Kurach, Karol, and Fergus, Rob (2014). Learning to discover efficient mathematical identities. arXiv:1406.1584

[4] Bowman, Samuel R, Potts, Christopher, and Manning, Christopher D (2014). Recursive neural networks for learning logical semantics. arXiv:1406.1827

[5] Bowman, Samuel R (2013). Can recursive neural tensor networks learn logical reasoning? arXiv:1312.6192

[6] Maddison, Chris J and Tarlow, Daniel (2014). Structured generative models of natural source code. arXiv:1401.0514

Code written for this project

GitHub link: https://github.com/nikhilsharma93/machine_translation_for_programming_language

Appendix

This section presents the results of the experiments that were done. Some of the test cases are commented if there is something noteworthy about them.

Baseline Strategy with Length = 5 and Nesting = 2

Input

```
f=48578
for x in range(13):f-=(20135 if 64841<71065 else 82526)
print(f)
```

Output

```
f=48578
for x=1,13 do f=f-(if 64841<71065 then 20135 else 82526 end); end
print(f)
```

Expected Output

```
f=48578
for x=1,13 do f=f-(if 64841<71065 then 20135 else 82526 end); end
print(f)
```

Input

```
print((1622 if 10735<(70848-59906) else 2827))
```

Output

```
print((if 10735<(70848-59906) then 1622 else 2827 end))
```

Expected Output

```
print((if 10735<(70848-59906) then 1622 else 2827 end))
```

Input

```
print((18055*9))
```

Output

```
print((18055*9))
```

Expected Output

```
print((18055*9))
```

Input

```
c=61943;
print((56416 if 31396<(c+72064) else 98259))
```

Output

```
c=61943;
print((if 31396<(c+72064) then 56416 else 98259 end))
```

Expected Output

```
c=61943;
print((if 31396<(c+72064) then 56416 else 98259 end))
```

Input

```
d=36161
for x in range(6):d-=56571
print(d)
```

Output

```
d=36161
for x=1,6 do d=d-56571; end
print(d)
```

Expected Output

```
d=36161
for x=1,6 do d=d-56571; end
print(d)
```

Baseline Strategy with Random Lengths from 1 to 5 and Random Nesting from 1 to 4

Input

```
d=9561034
for x in range(60):d-=1083468245
print(d)
```

Output

```
d=95610
for x=1,6 do d=d-108346; end
print((82435+d))
```

Expected Output

```
d=9561034
for x=1,60 do d=d-1083468245; end
print(d)
```

Input

```
e=65500;
j=(65477 if (e-47161)<85023 else 29517);
print((j-69467))
```

Output

```
e=65500;
print((if 47161<850235 then (e-654771) else 69467 end)-8748))
```

Expected Output

```
e=65500;
j=(if (e-47161)<85023 then 65477 else 29517 end);
print((j-69467))
```

Input

```
print((113257 if (778769 if 534695>644151 else (129921+943124))<464517 else 230578))
```

Output

```
print((if (77876 if 53469>64541 else 12992)<24955 then 113291 else (238101*199) end))
```

Expected Output

```
print((if (778769 if 534695>644151 else (129921+943124))<464517 then 113257 else 230578 end))
```

Input

```
e=606
for x in range(4):e+=422
b=819;
c=(5*(b+e));
print((c-378))
```

Output

```
e=606
for x=1,4 do e=e+422; end
f=e
for x=1,1 do f=f-378; end
print(f)
```

Expected Output

```
e=606
for x=1,4 do e=e+422; end
b=819;
c=(5*(b+e));
print((c-378))
```

Input

```
print((6*(227659*18)))
```

Output

```
print((6*(227659*18)))
```

Expected Output

```
print((6*(227659*18)))
```

These results show that despite the network does not perform well on large nesting and/or length values, it still does retain most of the structure.

The following example is another interesting case. The network was provided with an incorrect input: there is no variable that is being decremented. A possible correct input is also shown, where the variable would have some value, which will be decremented. The output of the network shows that the network was able to rectify this mistake by assigning a variable ("f") and using the same variable to decrement it.

Incorrect Input

```
for x in range(4):=-18652
```

Sample Correct Input

```
g=4322
for x in range(4):g=-18652;
print(g)
```

Output

```
f=(148-26893)
for x=1,3 do f=f-6592; end
print(f)
```

Mixed Strategy with Length < 8 and Nesting < 5

INPUT

```
c=(4*514706)
for x in range(15):c+=827640
print(c)
```

OUTPUT

```
c=(4*514706)
for x=1,15 do c=c+827640; end
print(c)
```

TARGET

```
c=(4*514706)
for x=1,15 do c=c+827640; end
print(c)
```

INPUT

```
a=33759;
print(((a+(242716 if 542419<928910 else 463951))*16))
```

OUTPUT

```
a=33759;
print(((a+(if 542419<928010 then 242716 else 463951 end))*16))
```

TARGET

```
a=33759;
print(((a+(if 542419<928910 then 242716 else 463951 end))*16))
```

INPUT

```
c=590
for x in range(4):c+=586
print((395 if 275>626 else (885+c)))
```

OUTPUT

```
c=590
for x=1,4 do c=c+586; end
print((if 255>626 then 395 else (885+c) end))
```

TARGET

```
c=590
for x=1,4 do c=c+586; end
print((if 275>626 then 395 else (885+c) end))
```

INPUT

```
print((80290 if 98613>75134 else (19*94506)))
```

OUTPUT

```
print((if 98613>75134 then 80290 else (19*94506) end))
```

TARGET

```
print((if 98613>75134 then 80290 else (19*94506) end))
```

INPUT

```
f=86726
for x in range(15):f-=(90217*11)
print(f)
```

OUTPUT

```
f=86726
for x=1,15 do f=f-(90017*11); end
print(f)
```

TARGET

```
f=86726
for x=1,15 do f=f-(90217*11); end
print(f)
```

INPUT

```
b=6523;
h=(b+4966)
for x in range(7):h-=3951
print((8*(h*10)))
```

OUTPUT

```
b=6523;
h=(b+4966)
for x=1,7 do h=h-3951; end
print((8*(h*10)))
```

TARGET

```
b=6523;
h=(b+4966)
for x=1,7 do h=h-3951; end
print((8*(h*10)))
```

INPUT

```
print((11*(85924-(4756-99082))))
```

OUTPUT

```
print((11*(85924-(4756-99082))))
```

TARGET

```
print((11*(85924-(4756-99082))))
```

INPUT

```
print(643503)
```

OUTPUT

```
print(643503)
```

TARGET

```
print(643503)
```

Tougher Examples with Mixed Strategy (Note the ability to retain the syntax vs the ability to retain the numerical values)

INPUT

```
i=836448;
e=893113
for x in range(6):e+=(i-(415587-925524))
print((7*e))
```

OUTPUT

```
i=836448;
e=893113
for x=1,6 do e=e+(i-(62295--925564)); end
print((7*e))
```

TARGET

```
i=836448;
e=893113
for x=1,6 do e=e+(i-(415587-925524)); end
print((7*e))
```

INPUT

```
e=13;
d=3
for x in range(2):d+=(93+(e-(42 if 30<32 else 97)))
print(d)
```

OUTPUT

```
e=13;
d=3
for x=1,2 do d=d+(684-(e+(if 72<32 then 260 else 97 end))); end
print(d)
```

TARGET

```
e=13;
d=3
for x=1,2 do d=d+(93+(e-(if 30<32 then 42 else 97 end))); end
print(d)
```

INPUT

```
f=282;
print((4*((5970+(f+2683)) if 6226<2888 else 5854)))
```

OUTPUT

```
f=282;
print((2*(if 6226<2888 then (5950+(f+2820)) else 5854 end)))
```

TARGET

```
f=282;
print((4*(if 6226<2888 then (5970+(f+2683)) else 5854 end)))
```

INPUT

```
i=((70-71)*6);
print((i-79))
```

OUTPUT

```
i=((70-713)*6);
print((i-7))
```

TARGET

```
i=((70-71)*6);
print((i-79))
```
