Project Report: NN MOOCS

Yewen Pu (SID: 928-171-210)

Abstract

We tackle the problem of program correction as an editing problem in the domain of MOOCS. Given an incorrect student submission program **P**, find a minimal edits **ed** such that the correct program **Q** is obtained by applying the edits **ed** to **P**. Furthermore, the edits **ed** may be thought of being composed of many single-edits, which can be applied in series to the same effect as **ed**. In this project, we develop a neural network model that, given an incorrect program **P**, generates one correct single-edit **e** that is part of **ed**.

Introduction

Providing personalized feedbacks to individual students is a challenge for MOOC, as there are thousands of students, it is improbable for instructors to address each students individually. Any form of automated, personalized feedbacks would be very valuable.

In this project, we focus on one particular automatic feedback problem: automatic program correction. Given an incorrect student submission program **P**, find a correct program **Q** that require the fewest number of edits to **P**. The rationale is that a student is more likely to understand the program **Q**, which more similar to the student's incorrect solution, than to the correct reference solution.

Formally, we seek a solution **Q** such that:

Q = ArgMin_{P'' s.t. P'' is correct} dist(P, P'')

Rather than solving the above problem by searching over the space of correct programs, we find an edit **ed** which, when applied to the incorrect program **P** alters it into a correct program **Q.**

Q = apply_edits(ed, P)

To further simplifies the problem, we think of the edits **ed** as a composition of several single-edits **e**, which consists of the following:

e = (pos, move)

where **pos** is a location within the incorrect program **P** that one should apply the single-edit, and **move** is a single-edit move consisting of skip, insert, delete, or replace operations.

We build a model that can predict the one single edit **e** that's part of a full edit **ed,** of an incorrect program **P**. The model consists of a bi-directional LSTM unit for encoding, and several feed-forward neural network for decoding into the single-edit **e**.

Data Statistics and collection

Before diving into the model details, it is helpful to analyze some salient aspects of the data we have. The data is all the student submissions to an edx course taught at MIT to a simple python exercise problem called **oddTuples**, this program takes in a tuple, and returns a tuple whose element is every other element of the input tuple. For example:

oddTuples((1,2,3,4,5)) = (1,3,5)

The data consists of total ~28000 anonymous submissions from the students. Each student submits multiple solutions to the problem until she reaches a correct solution.

The data is time-stamped, thus we assumed that submissions that are temporally close are likely to be from the same student. Thus, for each correct submission, we look-back with a fixed window size of 20 attempting to find an incorrect solution from the same student.

We deem two solutions are from the same student if the edit-distance between the two solutions are reasonably close. The edit distance is measured by interpreting both programs as a sequence of tokens. The two solutions are “close enough” if the number of edits is no more than 25% of the length of each program.

Using the above metrics, we’re able to recover ~15400 pairs of incorrect to correct programs, whose edits between the two are sufficiently small.

There are duplications in the data as many students share the same incorrect program and correct programs. Of the 15400 pairs of programs, there are 2554 unique incorrect programs, and 608 correct programs.

We limit ourselves to short programs, where the length of the program is no more than 20 tokens. With this restriction, we are left with 3089 pairs of programs, with 114 unique incorrect programs and 16 unique correct programs.

Below is an example pair for incorrect short program and its correction:

incorrect program:

```
def oddTuples(x0):
  return x0[1:-1:2]
```

corrected:

```
def oddTuples(x0):
  return x0[::2]
```

Correction as Edits

One can attempt to solve the problem of program correction as a translation task, given an incorrect program **P**, directly generate the correct program **Q**. We decided against this approach in favor for predicting the single-edits for the following reasons:

1. The correct program **Q** is very similar to the incorrect program **P**, a trivial translator that copies every token is likely to achieve a locally-maximal high-score
2. The full-edits **ed** that completely transform the program **P** to **Q** must have variable length, as there might be multiple edits that transforms **P** into **Q**. Thus, to generate **ed** we need to address the generation for the <stop> symbol.
3. The single-edit **e**, when applied to an incorrect program **P** can partially transform it to an incorrect program **P'** that is closer to **Q**. We can run our model on the program **P'** again to correct it further (in a single step). The chaining of single-edit corrections can be thought of as a sequence of actions, which lends naturally to reinforcement learning, which is the ultimate goal for this project. This decomposition creates shareable sub-problems between different student submissions.

Formally, a single-edit consists of 2 parts, a position and a move. **e = (pos, move)**. The position marks a token in the program where a change is to be made, the move dictates what change it should be.

A move is further decomposed into 2 parts. **move = (decision, insertion)**. The decision is a boolean value dictating if the current token should be **kept** or **deleted**. The **insertion** is a token symbol denoting if a token should be inserted afterward, it can also be null, denoting that nothing should be inserted. It is easy to show this definition is equivalent to the more traditional edit distance formulation that has insert/delete/replace/skip operators.

Let's use a simple example, instead of programs, let's edit the sequence of numbers [4,5,6,7,8] to the sequence [4,5,1,8]. This is possible with 2 single-edits:

(3, (delete, 1))
(4, (delete, null))

The first edit selects the third token "6", and says it should be deleted and a token "1" should be inserted afterwards. The second edit selects the fourth token "7", and says it should be deleted with nothing inserted afterwards.

As we can see, when these 2 edits are applied, the sequence [4,5,6,7,8] is changed into the sequence [4,5,1,8].

In the pair of programs example given in the last section, the incorrect program:

```
def oddTuples(x0):
  return x0[1:-1:2]
```

can be changed to the correct program:

```
def oddTuples(x0):
  return x0[::2]
```

by the following set of single edits:

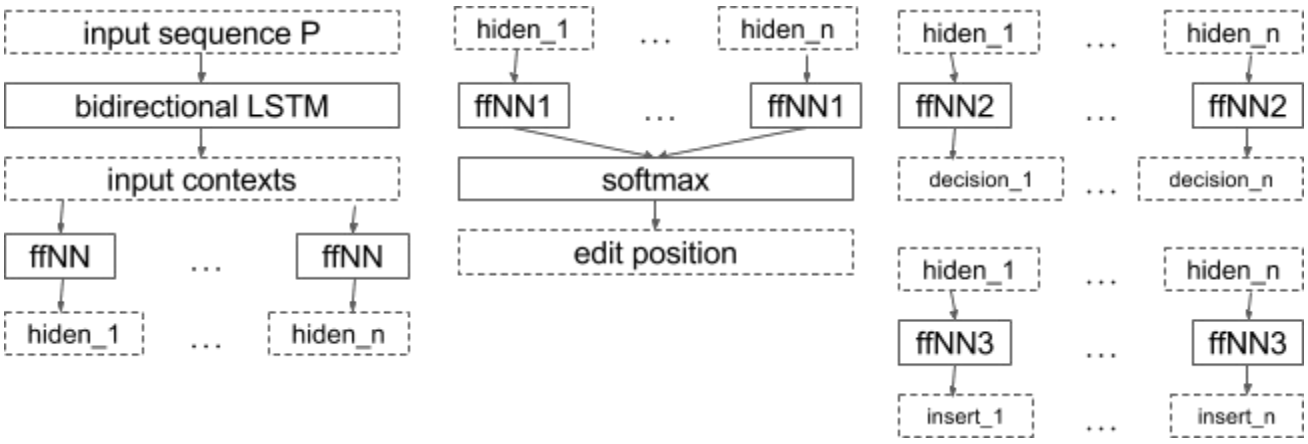
```
(12, (delete, null))
(14, (delete, null))
(15, (delete, null))
```

Neural Network Model

To predict a single edit **e**, we develop the following neural network model:

1. First, bidirectional LSTM layer that process the input program **P** of length **n** into **n** contexts, each token **i** in P has a **context_i** associated with it.
2. Then, a single layer of relu is applied to each **context_i**, mapping it into a hidden representation **hidden_i**
3. From the hidden representation, 3 additional feed forward neural networks are applied to it. The first network outputs a voting weight, proposing how likely the current token, **token_i** should be changed. The second network outputs a **decision**, deciding if the current token should be **kept** or **deleted**. The third network outputs an insertion, indicating if any token should be inserted after the current token.
4. All the voting weights across the n contexts are put together through a softmax layer, and the output predicts the most likely position in the program to perform the single-edit.

The network architecture is shown in the figure below:



Evaluation

Given a program **P** and the set of single-edits **ed** that can transform **P** into **Q**, we consider a single-edit **e** an accurate prediction if **e** is in **ed**

We measure accuracies for predicting the position, decision, and insertions separately.

Predicting Position:
For predicting the location of a token to be edited, the baseline predictor choose randomly among the 20 possible locations. This achieves an accuracy of 0.13. This is expected because there are on-average 2 edits per program.
Our model can predict the location with an accuracy of 0.2

Predicting Decision:
For predicting the decision whether a token should be kept of deleted, the baseline predictor choose randomly among the 2 options, which achieves an accuracy of 0.5.
Our model can predict the decision with an accuracy of 0.90

Predicting Insertion:
For predicting which token (if any) is to be added after the current token, the baseline predictor choose randomly among all 66 possible tokens, which achieves an accuracy of 0.015.
Our model can predict the insertion with an accuracy of 0.95

Improving prediction rate for position is possible when the model is trained to only output the edit positions without also having to predict the decisions and the insertions. It is likely that the hidden layers are too restrictive and have to compromise between generating correct positions and generating the correct moves. When trained to only predict the position, our model achieves an accuracy of 0.3

Conclusion and Future Works

In conclusion, we developed a reasonable model to predict a single-edit toward a correct program given an incorrect program. This model beats the baseline model slightly in predicting the location of the edits, and beats the baseline soundly in predicting the correct move. The trained network can serve as a bootstrap starting point for the reinforcement learning phase, where it will learn to correctly compose the single-edits together into a full edit.

One exciting addition that can be easily added is to encode compiler errors into the input. When attempting to execute a python program and the program fails to run, the compiler will issue an error message stating possible syntax errors or type-mismatch. Such information can greatly help predicting if certain tokens are responsible for causing the error, and should be incorporated into the model as an additional input. I am working on this now.

Links:
The (out dated) project presentation slides:
https://docs.google.com/presentation/d/1gZYVy_9wNGye48JVYudulEKm-UBVJsJ_qHWpH_9sf9Q/edit?usp=sharing

The github repository for the code:
https://github.mit.edu/yewenpu/nnmooc

Acknowledgements
Thanks to Karthik for insights and fantastic inputs
Thanks to Regina and Armando for proposing this great project
Thanks to Zenna, Bolei, Elena for helpful discussions and brainstorming