Project Report: Analyzing Software Using Deep Learning

Marvin Muñoz Barón
University of Stuttgart
st141535@stud.uni-stuttgart.de

***Abstract*—Supporting computer programmers in writing code free of syntax errors has been a research goal since the dawn of programming. Modern text editors and integrated development environments (IDE) offer sophisticated tools to assist programmers in achieving such goals. In this work, we present an approach using multiple recurrent neural networks (RNN) to find and correct syntax errors. Our dataset consisted of 40000 Python code snippets, each with a single syntax error, which was used to train the models and learn how to locate and categorize errors and implement fixes. In the 400 previously unseen code snippets used for evaluation, our model was able to fix 89 % of syntax errors. Overall, in the case of this study, deep learning proved to be an effective tool for automating the location and correction of syntax errors in Python source code.**

I. Introduction

Modern computer programs are created with ever-improving processes and sophisticated software engineering methods. Yet still, on a basic level, many programming languages require programmers to adhere to a strict syntax for programs to run correctly. Nevertheless, both novice and experienced programmers often make syntax errors when writing code [1].

Text editors and integrated development environments (IDE) offer functionality to automatically detect syntax errors and even correct them. In order to offer such functionality, these tools need to rely on complex underlying algorithms for error detection and correction. In recent years, deep learning approaches have found success in using neural networks to perform these tasks [2], [3].

As part of this work, we introduce an approach to use deep learning with recurrent neural networks (RNN) to find and fix syntax errors in Python source code snippets. Using a dataset of 40000 code snippets with syntax errors, we train four RNNs which, together, are able to consume a faulty code snippet and produce a corrected version. To achieve this, the combined application is capable of taking a single action such as deleting, modifying (i.e. replacing), or inserting a single source code token to implement the fix.

Structure: Section II provides an overview of the overall approach and neural network architecture. In section III, we discuss the capabilities and limitations of the completed application. Finally, a summary and conclusion follow in section IV.

II. Methods

In the following section, we describe the individual stages of our approach, starting with a faulty code snippet and ending with the fixed version of the snippet.

A. Preprocessing

The first step for each code snippet is to transform the code string into a list of tokens. For this task, we use the standard Python library `tokenize` [4]. The tokenizer from the library reads the code string and splits it into a list of tokens based on the code syntax. Specifically, program elements such as operators, identifiers, newlines, and others are separated into individual tokens. At this point, each token is still in its original string representation as it was in the original snippet.

To make this input understandable for a neural network, it must be encoded into numbers or vectors. In our approach, we use index-based encoding, where each unique token is mapped to a number, building a vocabulary of 99 different words. For example, the token or word ”ID” may be mapped to the number two and then, in turn, each occurrence of ”ID” in the token sequence is replaced with the number two. This vocabulary is fixed in length and contains all unique tokens in the training dataset. At this point, we have a token sequence in the form of multiple encoded word tokens as token indexes, following the structure and length of the original program. Additionally, the token sequence is padded with zeros to get each sample to the same length of 50 tokens. For a prediction, all tokens from a sample are individually fed into each model, with the output of the final token being used as the output of the prediction.

B. Model Architecture

To implement the neural networks, we use the `PyTorch` [5] library which offers functionality to build, train and predict with neural networks. Figure 1 shows the architecture of a single network in our application. Arrows are annotated with the size of the input and output vectors for each layer.

As described previously, the input of this first layer is a single token in its encoded integer representation. This first layer is an embedding layer that takes the token and returns a corresponding vector embedding. While using a pre-trained embedding model for source code has been shown to be effective in classification tasks [6], such approaches often rely on the context information included in the identifier names. As our model uses a simplified dataset where all identifiers are replaced with a shorthand representation, we opt to not use a pre-trained embedding layer. Instead, the embedding layer is trained with the rest of the model and optimized throughout the training process. This ensures that the embeddings are tuned

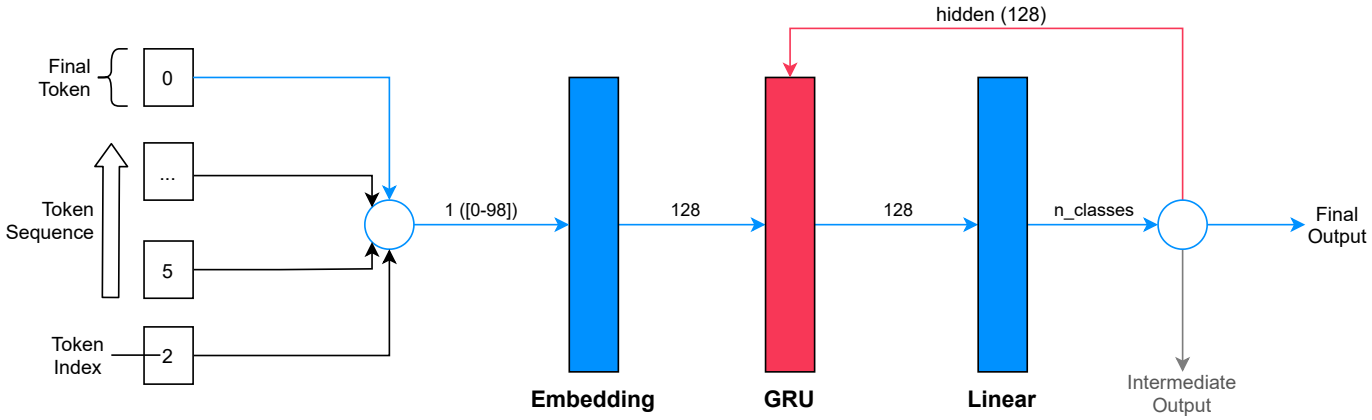

Fig. 1. The architecture of the neural network. Blue arrows represent the path of the final token in the sequence.

to our specific task. The output of the embedding layer is a vector of size 128.

The second layer in our network is a gated recurrent unit (GRU). GRUs are similar to LSTMs in their ability to remember information from the input in longer sequences. Overall, GRUs seem to perform similarly well, or in some tasks better than LSTMs and traditional RNN units [7]. In initial testing, a GRU showed promising results, leading us to choose it as the RNN layer in our network. The number of hidden units in this layer is 128, which proved to be sufficiently complex to learn the task.

Finally, the network concludes with a standard linear layer that maps the GRU output to an output vector of a fixed size. The size of this output vector is directly determined by the number of target classes for predictions resulting in a different output vector for each individual model.

We divide the overall task of finding and fixing syntax errors into four subtasks, each of them solved by a single neural network. All four networks use the same, previously described architecture, only differing in the size of the final linear layer. Figure 2 shows the setup of these networks and the prediction process for a single code snippet. First, we determine the location of the syntax error with the **Fixlocation** model. This model takes the token sequence as input and returns the index of the token with the syntax error in the sequence. Second, we determine the type of fix required to correct the syntax error with the **Fixtype** model. Again, the model takes the token sequence as input but this time returns one of three fix types: *Delete*, *Insert*, or *Modify*.

At this point, the approach differs depending on the determined fix type:

- For *Delete*, the location and type of fix is sufficient to correct the syntax error.
- For *Insert*, the token sequence is then given to the **Fixinsert** model, which returns a token from the vocabulary which should be inserted to correct the error.
- For *Modify*, the token sequence is given to the **Fixmodify** model, which in turn returns a token from the vocabulary which should replace the token at the fix location.

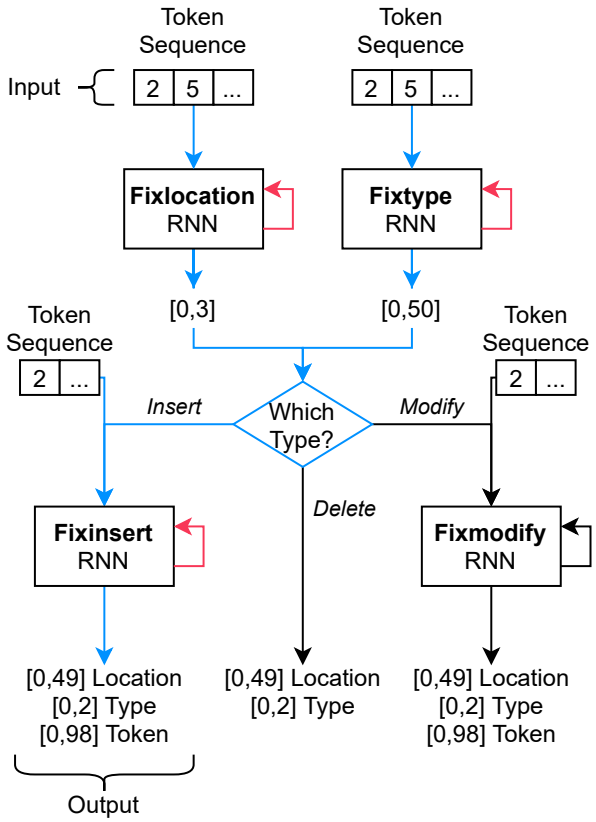

Fig. 2. The overall model consisting of four submodels.

At this point, we have all the necessary information to correct the original code string. The output of the combined models is the fix location, fix type, and in case of *Insert* of *Modify*, a fix token.

C. Training

For training the models, we used a dataset of 39600 code snippets, each with a single syntax error. 400 samples were excluded from training to be used later for evaluation. As the dataset contained target labels for each of the four subtasks, we

were able to train the models separately, each using the code snippet transformed into a list of 50 tokens as the input. While the **Fixlocation** and **Fixtype** models were trained on all 39600 samples, the **Fixinsert** model was trained on 9769 samples and the **Fixmodify** model was trained on 12936 samples. These lower numbers were the two subsets of samples that had the fix type target labels of *Insert* and *Modify*, respectively.

Each model was trained over 30 epochs, using Adamax as an optimizer and batches of the size 64. Adamax was used with the standard parameters given in PyTorch, only the learning rate was increased to 0.005. To calculate the loss for each prediction, the CrossEntropyLoss function was used as a criterion.

D. Postprocessing

The output of the models is a set of tuples for each code snippet, either (location, type) or (location, type, token). These tuples are then used to implement the code fix and produce a corrected code string. Specifically, the original code string is manipulated depending on the fix type. First, the location, or the index of the token with the syntax error in the input sequence is mapped to the exact character in the code string. Then, according to the fix type, the token following the determined character location is either deleted, replaced with a different token, or a new token is added before it. This altered code string serves as the final output of our application, completing the process of prediction for a single code snippet.

III. Results

Once the training is complete, the models are evaluated on a set of 400 code snippets that were excluded from training and have previously not been seen by the models. In the following section, we evaluate each model's performance on the individual subtasks and the combined application's performance on the overall task.

A. Individual Model Performance

Table I shows the performance of the four individual models on samples of each fix type. The fix type here refers to the correct fix type of the sample in the dataset, regardless of the model's prediction. Each model performed with an accuracy of over 92 %, with the **Fixmodify** model being the most accurate with 98.70 % when predicting the right token out of 99 different classes. However, as not all of the 400 code snippets used for evaluation had the fix type *Insert* or *Modify*, the **Fixinsert** model was evaluated on only 133 samples and the **Fixmodify** on 154 samples.

TABLE I
Performance of the individual models on subtasks.

Model	*Delete*	*Insert*	*Modify*	Overall
Fixlocation	91.08 %	92.06 %	95.45 %	92.70 %
Fixtype	96.76 %	95.97 %	96.73 %	96.55 %
Fixinsert	–	96.99 %	–	96.99 %
Fixmodify	–	–	98.70 %	98.70 %

The **Fixlocation** model had the lowest accuracy overall and the most trouble with predicting the location of the syntax error where a token had to be deleted. There, the accuracy was only 91.08 %. Figure 3 shows the distance between the predicted token location of the syntax error compared to the correct location. The y-axis determines the distance in the number of tokens from the correct token location. As the number of errors is low, we can display every single error made in the evaluation. All zero values, which correspond to correct predictions, were omitted for clarity. Most erroneous predictions fall within 1 token of the correct location. This is especially apparent for *Delete* predictions, where most errors occurred when the model predicted a location one token after the correct one. Unfortunately, we were unable to determine the cause for this.

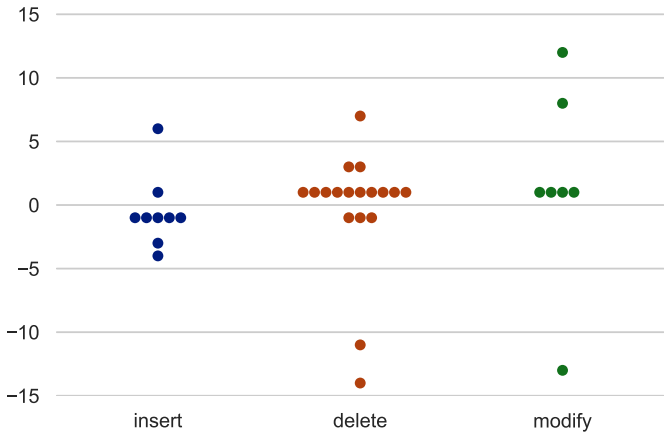

Fig. 3. Swarmplot showing the distance from the correct token location.

B. Overall Task Performance

Table 2 shows the performance of the model on the overall task. Here, the first measure is the percentage of exact matches of a prediction on a code sample with the target labels, meaning the prediction has the correct fix location, fix type and fix token (for *Insert* and *Modify*). This measure is strict, meaning that even if just one of the three values does not match the target label, then the whole sample is not counted. The dataset also contained a syntactically correct version of each code string. The second and third measures show whether the fixed code string produced by the model matches that string, with and without space characters. Lastly, the code string produced by the model is run through the parser of the Python `libcst` [8] library which is only able to parse syntactically correct code. The result of this is shown in the last measure, which serves as the most accurate representation of the model performance on the overall task: The percentage of syntactically correct code strings produced by the model.

There is a large discrepancy with the values for perfect predictions and exact code string matches. This is likely due to how our application fixes the code strings compared to what was used in the original dataset. When removing all spaces from the produced string, this discrepancy almost completely

TABLE II
PERFORMANCE OF THE COMBINED MODEL ON THE OVERALL TASK.

Fix Type	Perfect Prediction	Exact Code String Match	Exact Code String Match (Ignore Spaces)	Correct Syntax Fix
Delete	87.96 %	77.78 %	87.50 %	90.74 %
Insert	85.48 %	78.23 %	86.29 %	86.29 %
Modify	91.50 %	52.29 %	90.85 %	91.50 %
Overall	88.44 %	69.98 %	88.24 %	89.86 %

vanishes. The code fixing method in our work likely adds slightly more or fewer spaces in some cases compared to the one used in the dataset. The end result, however, still leads to syntactically correct code, as indicated by the last measure.

The model has the most struggle with *Insert* fixes, as it produced around 4 % less syntactically correct code compared to the other two fix types. This mostly matches the perfect prediction values, except for *Delete* samples, where the percentage of syntactically correct samples jumps up 2 to 3 % compared to the perfect prediction counterpart. This shows another point of interest, which is the fact that the percentage of correct syntax fixes is higher than the percentage of perfect predictions. This means that in some cases, even if the model did not predict the expected target labels from the dataset, the produced code string still compiled. Furthermore, this means that for each code sample, there are multiple ways to solve the syntax error, even with the limited information of just the location, fix type, and fix token.

Overall, the model produced around 90 % syntactically correct code strings.

C. Limitations

One limitation of the model used in this work is that each of the four models uses the same input. Better results may be achieved by using the output of one network as the input of the next. For example, the **Fixtype** model may produce better results if it receives both the token sequence as well as the location of the syntax error.

The approach could be further refined by combining all four models into a hierarchical neural network (HNN). This way, the models could be trained on the overall task instead of individually. Model parameters would be optimized based on the loss calculated from the overall task, meaning that even if one of the subtasks is solved correctly, the corresponding model may still be adjusted to conform to the overall task. Furthermore, the approach used in this work and an HNN approach could be combined by first pre-training the models individually and then adding a fine-tuning step where the models are further tuned to the overall task by combining them into an HNN.

Another limitation of the approach used in this work is that the training data is simplified before feeding it into the network. For example, every identifier name is simplified into the ”ID” shorthand, possibly losing valuable information in the process. This, however, significantly decreased the training effort and complexity of the final models. The vocabulary, for example, may increase almost infinitely as the number of possible identifiers in a program is extremely large. This could be prevented by only using the top identifiers that occur most in the dataset and mapping all others into a shorthand.

Lastly, in some instances, the final code string did not compile even though all individual predictions were correct. This is likely due to a shortcoming in the post-processing part of the application, as the code is fixed by directly manipulating the original code string. In these rare cases, an approach that manipulated the token sequence and performed fixes based on the token type may be better than straight string manipulation.

IV. CONCLUSION

In this paper, we proposed an approach using multiple RNNs to locate and correct syntax fixes in Python source code. A code snippet is first tokenized and encoded and then fed into four individual models. Each model performs a separate subtask such as locating the error, categorizing the necessary fix type, identifying the token be inserted or the token to be used when replacing an existing token. The outputs of the individual models are then used to produce a new code snippet, free of syntax errors. We then evaluated the individual models and the combined application on a set of 400 unseen code snippets.

Overall, the outcome of this study was **positive**. We were able to correct the syntax fixes in around **90 %** of test samples using an approach based on predictions with a set of four RNN models with similar architectures. Future works may build on this foundation through further optimization such as combining the models into a larger hierarchical network or improving the input-output behavior of the application.

REFERENCES

[1] S. K. Kummerfeld and J. Kay, “The neglected battle fields of syntax errors,” in *Proceedings of the fifth Australasian conference on Computing education-Volume 20*. Citeseer, 2003, pp. 105–111.

[2] S. Bhatia and R. Singh, “Automated correction for syntax errors in programming assignments using recurrent neural networks,” *arXiv preprint arXiv:1603.06129*, 2016.

[3] E. A. Santos, J. C. Campbell, A. Hindle, and J. N. Amaral, “Finding and correcting syntax errors using recurrent neural networks,” *PeerJ PrePrints*, vol. 5, p. e3123v1, 2017.

[4] Python Software Foundation, “tokenize — tokenizer for python source,” 2021. [Online]. Available: https://docs.python.org/3.8/library/tokenize.html

[5] PyTorch, “PyTorch,” 2021. [Online]. Available: https://pytorch.org/

[6] A. Kanade, P. Maniatis, G. Balakrishnan, and K. Shi, “Learning and Evaluating Contextual Embedding of Source Code,” *arXiv e-prints*, p. arXiv:2001.00059, Dec. 2019.

[7] J. Chung, Ç. Gülçehre, K. Cho, and Y. Bengio, “Empirical evaluation of gated recurrent neural networks on sequence modeling,” *CoRR*, vol. abs/1412.3555, 2014. [Online]. Available: http://arxiv.org/abs/1412.3555

[8] Instagram, “LibCST,” 2021. [Online]. Available: https://github.com/Instagram/LibCST