Modifying a Graph-based, Self-Supervised Neural Network for Program Repair

Aaron Wu, Henry Tang, Ruijie Fang[1]

[1]This is a final project for COS484, Spring 21 at Princeton University. We thank Professor Danqi Chen, Professor Karthik Narasimhan, and Zexuan Zhong for their support

Introduction

DrRepair is a graph-based neural architecture for automatically fixing syntax errors of C programs using semi-supervised learning. Given a program as input, DrRepair constructs a **program feedback graph** $G = (V, E)$ with

- V = tokens in the diagnostic arguments
- Every edge in E links two vertices that share the same token

The model then employs a Seq2Seq model with **graph attention mechanism** and LSTM-based encoder/decoders to solve the repair task.

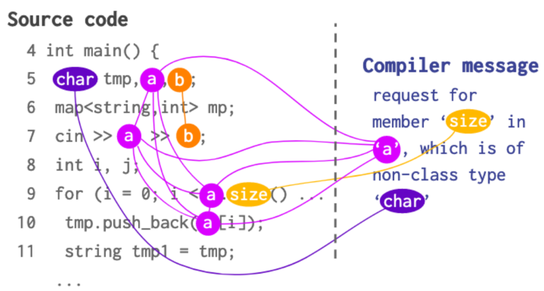


Figure 1:Illustration of program feedback graph.

The **DrRepair** model focuses on fixing four types of compile-time errors, shown below:

Our auto-corruption module	Example

Syntax (deletion, insertion, replacement of operator/ punctuator ,.;(){}'"++, etc.)	`return 0; }` → `return 0; } }`
	`cout << "YES";` → `cout << YES;`
	`min(s.size(), n)` → `min(s.size()), n)`
	`tmp = *a;` → `tmp = &a`

ID-type (deletion, insertion, replacement of type)	`for (int i=0; i<n;)` → `for (i=0; i<n;)`
	`k = k + 1;` → `int k = k + 1;`
	`string tmp;` → `char tmp;`

ID-typo (deletion, insertion, replacement of IDentifier)	`int a, b=0, m, n;` → `int a, m, n;`
	`string x,y,z;` → `string x,y,z,z;`
	`for (i=0; i<n;)` → `for (j=0; i<n;)`

Keyword (deletion, insertion, replacement of keyword/call)	`if (n >= 0)` → `while (n >= 0)` `l = s.length();` → `l = s.;`

Figure 2:The four categories of error fixed by DrRepair.

Modification 1: Uniform Data Distribution

Problem. The distribution of faulty programs is uneven in each of the four categories: Of the four error types (operator, ID conflict, undeclared variable, other) DrRepair can fix, the proportion of each type are 48%, 5%, 33%, 14% on the DeepFix data set.

Solution. We implemented a new tool RepairAssistant, which automatically generates new samples first using **CSmith** [1], a fuzzer that generates random C programs, and then automatically corrupts them to populate each of the four error categories. This approach also enables us to control the complexity of the programs generated, such as the number of statement blocks, and the complexity of expressions, in our dataset.

Modification 2: GRU

GRU is an alternative type of recurrent neural network. However, its architecture is simpler, in that it does not contain a cell state [2].

1. **Fewer parameters**. This is a consequence of the removal of the cell state. Furthermore, the initialization of the decoder network only relies on the hidden states.
2. **Similar Performance**. GRUs provide similar performance to LSTMs.

Model Architecture

Input. Source code $\mathbf{x}_{1:L}$ and feedback $f = (i, m)$, where i denotes the line number and m denotes the message of error.

Encoder. The input is encoded using two bidirectional LSTMs: $\mathbf{LSTM}^{(1)}_{\text{code}}$ and $\mathbf{LSTM}^{(1)}_{\text{msg}}$. Positional encoding is used to associate the error line number i_{err} with the current number i. This layer outputs the encoding of code $\mathbf{h}_{x_{ij}}$ and error message $\mathbf{h}_{m_i}$.

Graph attention layer. An N-layer graph attention network is used on the program feedback graph G. The encoded tokens $\mathbf{h}$. $\mathbf{h}^{i-1}, \mathbf{h}^{i}$ denote the input/outputs of the n-th layer. $\mathbf{h}_{x_{ij}}$ or $\mathbf{h}_{m_i}$ is fed in as $\mathbf{h}^0$. $\mathbf{Attention}_G(\mathbf{h}_t)$ computes the attention weights over neighbors of token t and takes the weighted average of the token representations among its neighbors. Each layer in the program feedback graph computes contextualized representations of tokens via

$$\mathbf{c}^n = \mathbf{Attention}_G(\mathbf{h}^{n-1})$$

and a feedforward network **MLP**,

$$\mathbf{h}^n = \mathbf{MLP}([\mathbf{h}^{n-1}; \mathbf{c}^n])$$

Recontextualization layer. We propagate the information in the graph attention network to the local context in this layer by passing token representations $\mathbf{g}$ to two additional LSTMs denoted by $\mathbf{LSTM}^{(2)}_{\text{code}}$. An embedding for each line i is acquired by concatenating the final hidden states of the LSTMs:

$$\mathbf{r}_i = [\mathbf{LSTM}^{(2)}_{\text{code}}(\mathbf{g}_{x_i}); \mathbf{LSTM}^{(2)}_{\text{msg}}(\mathbf{g}_m)]$$

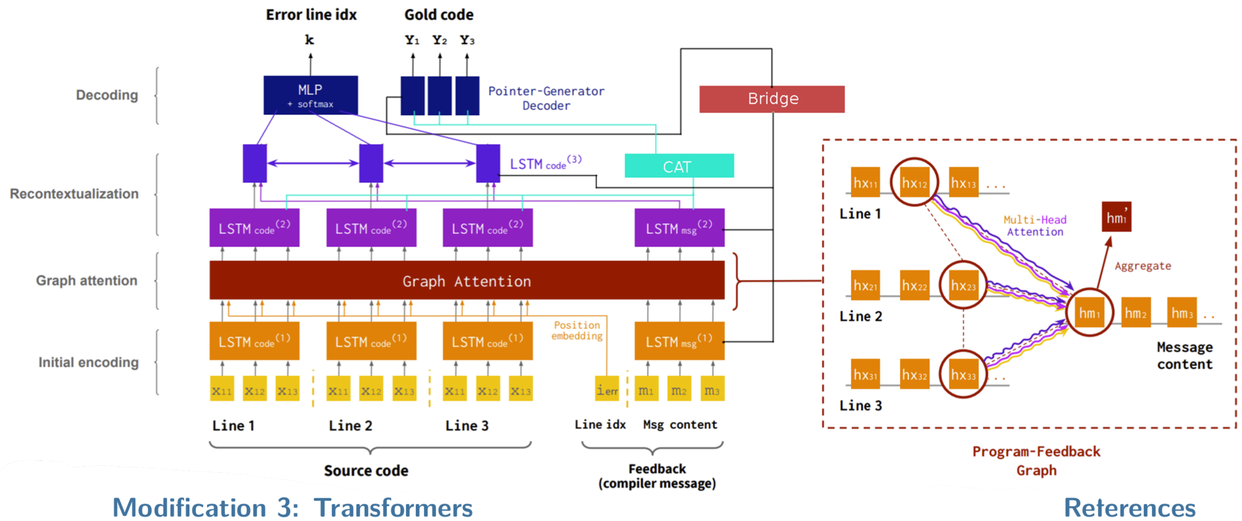
The final line embedding is given by

$$\mathbf{s}_{1:L} = \mathbf{LSTM}^{(3)}_{\text{code}}(\mathbf{r}_{1:L})$$

Decoder. A feedforward network models the probability $p(k)$ that line k is erroneous. The corresponding repair y_k is modelled via a pointer-generator decoder [3]:

$$p(k|\mathbf{s}_{1:L} = \text{Softmax}(\mathbf{MLP}(\mathbf{s}_{1:L}))$$
$$p(y_k|\mathbf{s}_{1:L}) = \text{PtrGen}(\mathbf{s_k})$$

Loss. Standard negative log-likelihood: $-\log p(k, y_k|x, f)$



Modification 3: Transformers

Transformers utilize an attention mechanism instead of passing hidden states [4].

1. **Better Context Understanding**. As a consequence of the attention mechanism, transformers are able to capture long term context better than LSTMs.
2. **Resizing of Hidden States**. The last output from each Transformer Encoder layer was collected, fed through a linear layer to reshape it to the appropriate dimension, and passed as initial input into the decoder.
3. **Better Parallelization** Empirically, our models trained 1.8x faster on 8 Intel Skylake CPU cores and a NVIDIA V100 GPU.

Results

Our GRU models performed similarly and achieved a 1.8x speedup as compared to LSTMs. The Transformer models also exhibited a 1.8x speedup as compared to LSTMs. However, the development accuracy for both localization and repair tasks are lower than the corresponding LSTM and GRU models.

Model	Dev. Localize	Dev. Repair	Test Repair
Code only LSTM	95.40%	71.70%	39.9%
No graph LSTM	97.65%	71.05%	64.7%
Base LSTM	98.20%	75.35%	71.7%
Base+pretrain+finetune LSTM	98.65%	78.95%	70.7%
Code only GRU	95.95%	70.95%	40.3%
No graph GRU	97.45%	70.45%	65.5%
Base GRU	97.10%	74.35%	67.9%
Base+pretrain+finetune GRU	98.55%	78.10%	70.5%
Code only Transformer	41.25%	9.80%	N/A
Base Transformer	90.15%	41.40%	N/A

Table 1:Results for LSTM and GRU ablations.

Analysis

Our GRUs performed slightly better for simpler models trained using smaller datasets. On the other hand, LSTMs perform slightly better when pretraining is performed. This is expected, as GRUs have fewer parameters, and thus overfit less with smaller data, but can't capture as much information on larger datasets.

Transformers, interestingly, performed worse in all metrics compared to their corresponding LSTM and GRU counterparts. Since Transformers produced similar results on the training data, but performed worse during development and test time, we believe the model may have overfit. Another possibility is that the original model's configuration parameters do not translate to optimal learning on the Transformer.

References

[1] Xuejun Yang et al.
Finding and understanding bugs in c compilers., 2011.

[2] Kyunghyun Cho et al.
Learning phrase representations using rnn encoder-decoder for statistical machine translation, 2014.

[3] Abigail See, Peter J. Liu, and Christopher D. Manning.
Get to the point: Summarization with pointer-generator networks, 2017.

[4] Ashish Vaswani et al.
Attention is all you need, 2017.

[5] Michihiro Yasunaga and Percy Liang.
Graph-based, self-supervised program repair from diagnostic feedback, 2020.

[6] Petar Veličković et al.
Graph attention networks, 2018.

[7] Junyoung Chung et al.
Empirical evaluation of gated recurrent neural networks on sequence modeling, 2014.