BUG FIX GENERATION USING GRAPHTRANS

Anonymous authors
Paper under double-blind review

ABSTRACT

Code repair, a task of learning to detect and fix bugs, is an important application of deep learning for source code. Previous work relies on code changes using Transformer models to represent code as sequences. However, code is naturally represented as a graph that encapsulates rich syntactic and semantic dependencies. Hence, generating edits for bug fixes requires both local structural information and global information. Inspired by GraphTrans (Wu et al., 2021), we propose FIXUR, a new architecture for generating bug fixing edits, by complementing graph neural networks with Transformer to encode the code graph. Our experiments show that FIXUR obtains near the state-of-the-art results on the code refinement benchmark, without relying on any large-scale pre-training. FIXUR achieves 20.50% and 11.01% top-1 accuracy on the **small** and **medium** datasets, respectively compared to 19.06% and 10.92% of CodeT5-small, which has a similar size.

1 INTRODUCTION

Advances in deep learning in the past decade have sparked much interest among practitioners in applying the technology on programming languages and big code data. Many exciting applications such as code completion, code synthesis and program repair promise to revolutionize software engineering. Program repair or code refinement is particularly important and intriguing thanks to the availability of code changes in code corpora and to recent successes of pre-trained models. (Feng et al., 2020; Ahmad et al., 2021; Wang et al., 2021) harness the power of the Transformer architecture as well as pre-training and establish state-of-the-art results for the task using the *sequence-to-sequence* formulation (i.e., generating fixed code from the buggy code sequence.)

Besides Transformer, graph neural networks (GNNs) are also widely used on source code and program repair as graph is a natural representation of source code whose rich structures are present in abstract syntax trees and program dependencies (Yin et al., 2018; Allamanis et al., 2017; Hellendoorn et al., 2020; Wang et al., 2020). For example, (Yin et al., 2018; Dinella et al., 2020; Yao et al., 2021) leverage syntactic structures in the underlying abstract syntax tree (AST) to learn bug fixing edits from code changes. However, GNNs have limitations in capturing the long range dependency between nodes, which are prevalent in source code. Increasing GNN depth even worsens its performance on some graph-level prediction tasks, a phenomenon known as *oversmoothing* or *oversquashing* (Alon & Yahav, 2021). On the other hand, Transformer architectures have achieved great success on a wide range of tasks that typically require long range interactions (Chen et al., 2021).

In this work, we seek to complement graph neural networks with Transformer to leverage the node-node interactions in source code for source code representation. Inspired by Wu et al. (2021); Dinella et al. (2020), we introduce FIXUR, a graph transformation model with a tree-based LSTM decoder for code refinement. FIXUR incorporates a Transformer architecture on top of a graph neural network to perform self-attention among nodes on the entire graph rather than on only local neighbors. This allows FIXUR to learn long-range dependencies. Unlike (Wu et al., 2021) which omits positional encodings to ensure permutation invariance, we explore positional encoding to encode the node position since the abstract syntax tree admits a natural ordering of the nodes such as the depth first order. This information is beyond the structural information encoded by the GNN in the node embeddings. FIXUR is learned over tree differencing edits extracted from code changes.

We evaluate FIXUR on the Java code refinement benchmark (Tufano et al., 2019b; Lu et al., 2021). We obtain significant improvements on the benchmark over HOPPITY and achieve near state-of-the-art results compared to large-scale pre-trained models. In summary, our contributions are as follows:

- We show that using Transformer with positional encoding atop a graph neural network improves the ability of FIXUR in capturing long-range interactions between nodes for bug detection and repair. This shows the potential of complementing GNNs with Transformer for learning effective representations of source code.
- We introduce FIXUR that leverages the GraphTrans architecture and positional encoding on the abstract syntax tree for code repair. FIXUR achieves near the state-of-the-art results on the code refinement benchmark, without relying on any large-scale pre-training. FIXUR obtains 20.50% and 11.01% top-1 accuracy on the **small** and **medium** benchmark, respectively, which outperforms all the baselines with comparable sizes without the need of pre-training.

2 RELATED WORK

Deep learning for code repair. Based on the lexical similarity between source code and natural language, previous work has successfully applied Transformers models and large-scale pre-training for code-related tasks, including code repair. Models such as CodeBERT (Feng et al., 2020), PLBART (Ahmad et al., 2021) and CodeT5 (Wang et al., 2021) achieve state-of-the-art results on a standard code refinement benchmark (Tufano et al., 2019b). Unlike these works, we build FIXUR on the inherent abstract syntax structure of code instead of its token-based representation.

Graph neural networks. GNNs are an important architecture for graph-structured data, particularly source code which has rich structures in abstract syntax trees and program dependencies. Yin et al. (2018); Dinella et al. (2020); Yasunaga & Liang (2020); Yao et al. (2021) use graph neural networks to model the program using the abstract syntax representation for downstream tasks, such as program repair. One of the well-known approaches to incorporate global information is increasing the depth. Alon & Yahav (2021) shows that increasing depth results in *over-squashing*. To combat this issue, several work has attempted to apply Transformer architectures to graphs. Models like GREAT (Hellendoorn et al., 2020) leverage the attention mechanism of Transformer models to encode relational information in graph representation of the input. GraphTrans Wu et al. (2021) passes the output of GNN to a Transformer module and significantly boosts the performance on several open graph benchmarks.

3 FIXUR

Our goal is to learn to generate bug fixes for a given buggy AST. We use a graph neural network module to encode the program and an autoregressive decoder to generate bug fixing edits. Inspired by GraphTrans (Wu et al., 2021), we introduce FIXUR that combines GNNs with Transformer for graph encoding and use LSTM-based decoding for the code refinement. FIXUR consists of three main modules: a GNN encoder followed by a Transformer module, and a tree-based decoder. Figure 1 illustrates the high-level architecture. We will describe each of the components below and reuse some formulas of (Wu et al., 2021).

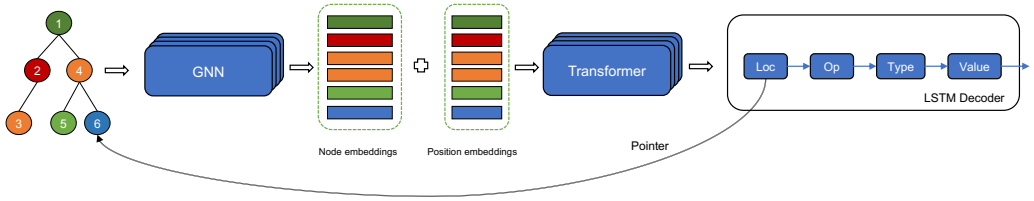

Figure 1: The high-level architecture of FIXUR for one edit step.

Multi-edge GNN module. We aim to learn graph representations of programs to leverage their rich syntax and dependency structures. Given a program, we construct a directed graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ where the nodes and initial edges come from the abstract syntax tree. Similar to (Allamanis et al., 2017; Yin et al., 2018; Brockschmidt et al., 2018; Hellendoorn et al., 2020; Dinella et al., 2020), we

augment the tree with additional data flow by connecting edges between adjacent leaf nodes as well as between nodes of the same values, and all the connections are bidirectional.

Our goal is to compute node embeddings of $\mathcal{G}$ using graph neural networks. We initialize for each node $v \in \mathcal{V}$ a feature vector $\boldsymbol{h}_v^0 \in \mathbb{R}^{d_0}$ as the sum of the label and value embeddings assigned to the node. Then, a generic GNN can be used to compute for each graph with to the edge type $k = 1, \ldots, K$ the following embeddings:

$$\boldsymbol{h}_v^{(\ell,k)} = f_\ell \left(\boldsymbol{h}_v^{(\ell-1,k)}, \{\boldsymbol{h}_u^{(\ell-1,k)} | u \in \mathcal{N}^k(v)\} \right), \quad \ell = 1, \ldots, L_{\text{GNN}} \tag{1}$$

where L_{GNN} is the total number of GNN layers, $\mathcal{N}^k(v)$ denotes the set of nodes that connects to or from v with an edge of type k, and $f_\ell(\cdot)$ is some propagation and aggregation function parameterized by a neural network. Then, we gather the node embeddings from each graph and produce give the final node embeddings $\boldsymbol{n} = \{\boldsymbol{h}_v^{L_{\text{GNN}}}\}_{v \in \mathcal{V}}$ for each node $v \in \mathcal{V}$ using another neural network g:

$$\boldsymbol{h}_v^{L_{\text{GNN}}} = g([\boldsymbol{h}_v^{(L_{\text{GNN}},1)}, \ldots, \boldsymbol{h}_v^{(L_{\text{GNN}},K)}]).$$

Node Transformer module. The node embeddings $\boldsymbol{h}_v^{L_{\text{GNN}}} \in \mathbb{R}^{d_{L_{\text{GNN}}}}$ are then passed to the Transformer module to learn attention-based representations, with a series of forward propagation steps: First, it transforms $\boldsymbol{h}_v^{L_{\text{GNN}}}$'s to the Transformer dimension and normalize the embedding with a layer normalization:

$$\boldsymbol{h}_v^0 = \text{LayerNorm}(\boldsymbol{W}^{\text{Proj}} \boldsymbol{h}_v^{L_{\text{GNN}}}) + \text{PE}(v) \tag{2}$$

where $\boldsymbol{W}^{\text{Proj}} \in \mathbb{R}^{d_{\text{TF}} \times d_{L_{\text{GNN}}}}$ is a learnable projection matrix, d_{TF} is the Transformer dimension and $\text{PE}(v)$ is the positional embedding of v. Next, it computes the attention-based node encodings, for a single attention head in layer ℓ:

$$\begin{aligned} a_{v,w}^\ell &= (\boldsymbol{W}_\ell^Q \boldsymbol{h}_w^{\ell-1})^\top (\boldsymbol{W}_\ell^K \boldsymbol{h}_u^{\ell-1}) / \sqrt{d_{\text{TF}}} \quad \text{for } w \in \mathcal{V}, \\ \alpha_{v,u}^\ell &= \text{softmax}_{w \in \mathcal{V}}(a_{v,w}^\ell), \qquad \boldsymbol{h}_v'^\ell = \sum_{w \in \mathcal{V}} \alpha_{v,w}^\ell \boldsymbol{W}_\ell^V \boldsymbol{h}_w^{\ell-1}, \end{aligned} \tag{3}$$

where $\boldsymbol{W}_\ell^Q, \boldsymbol{W}_\ell^K, \boldsymbol{W}_\ell^V \in \mathbb{R}^{d_{\text{TF}}/n_{\text{head}} \times d_{\text{TF}}/n_{\text{head}}}$ are the learnable query, key, and value matrices, respectively. We use n_{head} attention heads and concatenate the resulting per-head encodings $\boldsymbol{h}_v'^\ell$, as done in Transformer. The combined encodings are then passed to a final neural network, with standard dropout, normalization and residual layers.

Postional Embeddings. Unlike (Wu et al., 2021), which directly feeds the transformed node embeddings $\boldsymbol{h}_v^0$ into the Transformer stack without positional encoding, we instead explore positional encoding to encode the node position since the abstract syntax tree admits a natural node ordering such as the depth first order. The encoding is then added to $\boldsymbol{h}_v^0$. This information is beyond the structural information encoded by the GNN in the node embeddings. We find the positional encoding helpful in our experiments.

Finally, we need a single encoding vector that summarizes the entire graph's local and global structures. An interesting aspect of the Transformer architecture is a global readout in which one can employ a special-token embedding for downstream tasks by adding the `<CLS>` token to the input sequence and take its corresponding output embedding to represent the input. With the Transformer module and the `<CLS>` token, the information of the whole input with node-node interactions is aggregated into that embedding via the attention module. In our experiments, however, we found that sum pooling works better for the bug detection and refinement tasks.

Tree-based decoder module. Once we have the graph representation, we use an auto-regressive tree decoder to perform one bug fixing edit step. Given the graph representation $\boldsymbol{g}_t$ of the program at step t and the edit history $\boldsymbol{s}_{t-1}$ from the previous step, an LSTM decoder is used to update the state $\boldsymbol{s}_t = \text{LSTM}(\boldsymbol{g}_t | \boldsymbol{s}_{t-1})$ after each prediction call, based on the following probability decomposition of an edit action:

$$p(a_t | \boldsymbol{s}_t) = p(loc_t | \boldsymbol{s}_t) p(op_t | \boldsymbol{s}_t, loc_t) p(type_t | \boldsymbol{s}_t, op_t, loc_t) p(value_t | \boldsymbol{s}_t, op_t, loc_t, type_t). \tag{4}$$

More specifically, based on the updated state $\boldsymbol{s}_t$, the model predicts edit actions where each node edit includes a pointer network that acts as the node selector, an operator prediction layer as

well as type and value prediction layer. The node selector selects a node loc_t, then the operator predictor decides which operator op_t to apply the edit on loc_t among the possible operators $\{\texttt{DELETE}, \texttt{ADD}, \texttt{COPY}, \texttt{REPLACE}, \texttt{NO\_OP}\}$. Depending on the selected operation, the type predictor further determines the node type/label $type_t$ for the newly added node. For terminal nodes, the value predictor computes the node value $value_t$. For COPY, the node selector also finds a pointer to where on the original AST the model should copy from. We maintain the single LSTM to encode the global state after each of the location, operation, value and type predictions. For more details into this grammar-based decoding, we refer the reader to (Yao et al., 2021).

4 Experimental Results

We conduct experiments on the *Patches in the Wild* Java benchmark (Tufano et al., 2019b), which is also known as the code refinement benchmark (Lu et al., 2021). We compare FIXUR against a wide range of baselines to show its efficacy. We also analyze the performance of GraphTrans in representing source code for bug localization, a task that aims to detect the buggy location in the code, and demonstrate the challenge of this task.

4.1 Dataset

We use the code refinement benchmark, which consists of 123,804 bug-fix pairs and is divided into two subsets — **small** and **medium**. The **small** set has 58,350 methods/functions with less than 50 tokens whereas each of the 65,454 **medium** function is between 50 and 100 tokens. Both datasets come with an 8:1:1 split for training, validation and testing. A test example of buggy and fixed code is shown in Listing 1. We parse the source code into ASTs and filter out a few un-parsable samples. The average numbers of AST nodes on the two datasets are 65 and 142, respectively.

Buggy code:

```
public static boolean METHOD_1(TYPE_1 VAR_1) {
        return TYPE_2.METHOD_2(VAR_2);
}
```

Fixed code:

```
public static boolean METHOD_1(TYPE_1 VAR_1) {
        return TYPE_2.METHOD_2(VAR_1);
}
```

FIXUR **edit:**

```
1. "edit": REPLACE:::8:::VAR_1
```

Listing 1: A variable-misuse bug and the ground-truth fixing edit in the **small** dataset.

Ground-truth edits. Unlike sequence-to-sequence models, which predict the target fixed code given the buggy code sequence, FIXUR tries to learn bug fixing edits and apply them on the buggy input. As such, we need to extract ground-truth edit sequences between buggy and fixed code and compute the objective function. There are multiple ways to generate edits given a pair of buggy and fixed ASTs. HOPPITY (Dinella et al., 2020) uses JSON differencing algorithm, which produces long and uninterpretable sequences. Instead, we implement a grammar-constrained and shortest distance tree differencing algorithm based on a dynamic program of Yao et al. (2021) (Algorithm 3). The resulting edit sequences are 3-4X shorter than JSON differencing edits in both datasets. Listing 1 displays the fixing edit derived from the buggy and fixed code in which an edit includes the edit location, the edit operation type, the node type and value (when applicable).

Metrics. We report the standard top-1 exact match (EM) accuracy for FIXUR and the baselines. Given the buggy code, a generated fix is considered correct if it exactly matches the ground-truth. For graph-based approaches, we compare the corresponding AST via the depth first order. In addition, we use the bug localization accuracy of predicting the bug locations in an additional analysis.

Implementation and training details. We implement FIXUR with PyTorch and train the model with various configurations on a machine that has 8 V100 GPUs and 32GB each. We use the Adam optimizer and a linear learning rate scheduler with a default learning rate 0.0001 and warmup in the

first 10% of the total training iterations. We train all FIXUR models for 20 epochs using 16 edit steps and batch size from 32 to 128, depending on the model size. The training takes about 0.5 hour per epoch on the **small** subset and about 1.5 hours per epoch on the **medium** set. We choose the best model based on the validation loss and report the top-1 accuracy on the test set. During the testing phase, we use beam search of size 5 and select the candidate with highest score normalized by the edit sequence length as the final fix. We did not perform any exhaustive hyper-parameter search, so we might be able to achieve better results with appropriate tuning. For HOPPITY, we use a larger learning rate of 0.001 that allows the model to learn longer edits faster and perform better.

4.2 RESULTS

Table 1: Top-1 exact match (EM) accuracy of FIXUR and baselines without pre-training. For FIXUR, we use graph isomorphism networks (GIN) with $L_{\text{GNN}} = 8$ and $d_{\text{GNN}} = 512$. The Transformer module has the same latent dimension $d_{\text{TF}} = 512$, $n_{\text{head}} = 8$ heads and $L_{\text{TF}} = 6$ encoder blocks.

Model	Size	Small	Medium
LSTM (Tufano et al., 2019b)	10M	9.22%	3.22%
GRU + Token Copy (Panthaplackel et al., 2020)	250K	14.80%	7.00%
Transformer (Drain et al., 2021)	60M	11.10%	2.70%
GRU + Span Copy (Panthaplackel et al., 2020)	250K	17.70%	8.00%
HOPPITY	73M	14.48%	5.42%
FIXUR without Transformer	43M	<u>18.96</u>%	<u>9.26</u>%
FIXUR	53M	**20.50%**	**11.01%**

First, we compare FIXUR to baselines that are all trained from scratch. The results are shown in Table 1. We can see that FIXUR significantly outperforms the traditional LSTM (Tufano et al., 2019b), and a sequence-to-sequence model based on Transformer with the comparable number of parameters. Next, we include edit-based approaches (Panthaplackel et al., 2020) and HOPPITY (Dinella et al., 2020) for comparison. We run two configurations of FIXUR with and without the Transformer module, which have 53M and 43M parameters respectively. Overall, FIXUR outperforms both models by good margins. In particular, it is worth noting that the GRU + Span Copy model in (Panthaplackel et al., 2020) works as competitively as FIXUR despite using much fewer parameters. It would be interesting to see if such a model could scale and improve the performance when the model size increases.

Compared to HOPPITY, which also uses GINs for the graph encoding, FIXUR benefits from the shorter edit sequences, and the initialization of node embeddings as the sum of type and value embeddings, instead of one-hot encodings. We can see the improvement of FIXUR over HOPPITY without Transformer. Adding the Transformer on top of the graph encoder boosts the performance further, giving about 2% gain in accuracy on each dataset. We observe that increasing the GIN's depth and latent dimension did not help, so the performance gain given by the Transformer component is not merely in terms of increasing the number of parameters.

Table 2: Top-1 exact match (EM) accuracy of FIXUR and to larger-scale pre-trained models. For FIXUR, we use GINs with $L_{\text{GNN}} = 8$ and $d_{\text{GNN}} = 512$. The Transformer module has the same latent dimension $d_{\text{TF}} = 512$, $n_{\text{head}} = 8$ heads and $L_{\text{TF}} = 6$ encoder blocks.

Model	Pre-trained	Size	Small	Medium
CodeBERT (Feng et al., 2020),	✓	180M	16.40%	5.20%
GraphCodeBERT (Guo et al., 2020)	✓	110M	17.30%	9.10%
BART (Drain et al., 2021)	✓	400M	16.70%	6.70%
PL-BART (Ahmad et al., 2021)	✓	140M	19.21%	8.98%
CodeT5-small (Wang et al., 2021)	✓	60M	19.06%	10.92%
CodeT5-base (Wang et al., 2021)	✓	220M	**21.61**%	**13.96**%
FIXUR	✗	53M	<u>20.50</u>%	<u>11.01</u>%

Finally, we benchmark FIXUR against several large-scale models that are based on the Transformer architecture and pre-trained on unlabeled code corpuses. Without pre-training, FIXUR beats most of these baselines. For example, it outperforms CodeBERT by 4.10% (in absolute terms) on the **small** subset and 5.50% on the **medium** subset. We also see similar accuracy gains of FIXUR over

BART and PL-BART. With comparable model sizes, FIXUR is better than CodeT5-small on both datasets. CodeT5-base with 220M parameters is currently the state-of-the-art performance on both datasets. *Compared to CodeT5-base,* FIXUR *has 4X fewer parameters and is trained from scratch.* In other words, FIXUR achieves competitive results without pre-training. Using careful hyper-parameter tuning and pre-training strategies, we may be able to beat the state-of-the-art CodeT5-base models.

4.3 BUG LOCALIZATION ANALYSIS

We aim to reason about the overall performance on the bug fixing task and understand the accuracy gap between the two **small** and **medium** datasets in FIXUR. To that end, we run FIXUR for bug localization, which is a simplified task. Specifically, we train the models to predict the location of the first bug location without attempting to generate the fix. We do so by discarding the two fully connected layers for the type and value predictions in the decoder while keeping the LSTM decoder as is. The number of parameters are therefore slightly smaller than the full models presented in the previous section. We run each of the models with several hyper-parameter and readout variations on the **small** and **medium** datasets. We report the location accuracy on the test set.

Table 3: Experiments of FIXUR with different readouts and model sizes. Here, we use GINs with $L_{\text{GNN}} = 8$ and $d_{\text{GNN}} = 512$, and the Transformer module with the same latent dimension d_{TF}, $n_{\text{head}} = 8$ heads and $L_{\text{TF}} = 6$ encoder blocks. Without the LSTM layer, FIXUR has 47M parameters.

Model	Model Size	Readout	Positional Encoding	Avg. Graph Size		Small	Medium
				Small	Medium		
FIXUR	47M	<CLS>	✗	65	142	29.89%	16.52%
FIXUR	47M	<CLS>	✓	65	142	30.11%	16.14%
FIXUR	47M	sum	✓	65	142	**30.69%**	**17.28%**

The results are shown in Table 3. First, we can see that the model with positional encoding and sum pooling works better than the ones using the <CLS>-style readout. Second, the location accuracy reported in this Table gives an upper bound on the overall accuracy of the full corresponding models presented earlier. Improving the decoder, which has only one LSTM layer, may bridge the accuracy gap between the localization and the bug fix performance. Also, we include the average number of nodes of the graph in both test sets. We can see that there is a significant difference in the location accuracy between two sets because buggy code in medium set has significant more nodes than that in the small set. Finally, we also observe when the first bug location is predicted correctly, then the fix is often correct. Hence, improving the bug localization is crucial for solving the code refinement task.

5 CONCLUSION

Program repair or code refinement is an important application of deep learning on source code. In this work, we build on the recent GraphTrans model and propose FIXUR, a model for the bug fix prediction task, which requires modeling both the graph structure and long-range relational information. We show the efficacy of FIXUR on the code refinement benchmark that achieves near state-of-the-art performance despite smaller size and without pre-training. We find that positional encodings are helpful for the bug fix prediction task and that <CLS>-based pooling does not exploit the structure present in AST-based representation.

REFERENCES

Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. Unified pre-training for program understanding and generation. *arXiv preprint arXiv:2103.06333*, 2021.

Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. Learning to represent programs with graphs. *arXiv preprint arXiv:1711.00740*, 2017.

Uri Alon and Eran Yahav. On the bottleneck of graph neural networks and its practical implications. In *ICLR*, 2021.

Marc Brockschmidt, Miltiadis Allamanis, Alexander L Gaunt, and Oleksandr Polozov. Generative code modeling with graphs. *arXiv preprint arXiv:1805.08490*, 2018.

Saikat Chakraborty, Yangruibo Ding, Miltiadis Allamanis, and Baishakhi Ray. Codit: Code editing with tree-based neural models. *IEEE Transactions on Software Engineering (TSE)*, 2020.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

Zimin Chen, Steve James Kommrusch, Michele Tufano, Louis-Noël Pouchet, Denys Poshyvanyk, and Martin Monperrus. Sequencer: Sequence-to-sequence learning for end-to-end program repair. *IEEE Transactions on Software Engineering (TSE)*, 2019.

Elizabeth Dinella, Hanjun Dai, Ziyang Li, Mayur Naik, Le Song, and Ke Wang. Hoppity: Learning graph transformations to detect and fix bugs in programs. In *International Conference on Learning Representations (ICLR)*, 2020.

Dawn Drain, Chen Wu, Alexey Svyatkovskiy, and Neel Sundaresan. Generating bug-fixes using pretrained transformers. *arXiv preprint arXiv:2104.07896*, 2021.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*, 2020.

Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. Graphcodebert: Pre-training code representations with data flow. *arXiv preprint arXiv:2009.08366*, 2020.

Hideaki Hata, Emad Shihab, and Graham Neubig. Learning to generate corrective patches using neural machine translation. *arXiv preprint arXiv:1812.07170*, 2018.

Vincent J Hellendoorn, Charles Sutton, Rishabh Singh, Petros Maniatis, and David Bieber. Global relational models of source code. In *International conference on learning representations*, 2020.

Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.

Yi Li, Shaohua Wang, and Tien N Nguyen. Dlfix: Context-based code transformation learning for automated program repair. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pp. 602–614, 2020.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. Codexglue: A machine learning benchmark dataset for code understanding and generation. *arXiv preprint arXiv:2102.04664*, 2021.

Sheena Panthaplackel, Miltiadis Allamanis, and Marc Brockschmidt. Copy that! editing sequences by copying spans. *arXiv preprint arXiv:2006.04771*, 2020.

Michele Tufano, Jevgenija Pantiuchina, Cody Watson, Gabriele Bavota, and Denys Poshyvanyk. On learning meaningful code changes via neural machine translation. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pp. 25–36. IEEE, 2019a.

Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. An empirical study on learning bug-fixing patches in the wild via neural machine translation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 28(4):1–29, 2019b.

Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. Graph attention networks. *arXiv preprint arXiv:1710.10903*, 2017.

Yu Wang, Ke Wang, Fengjuan Gao, and Linzhang Wang. Learning semantic program embeddings with graph interval neural network. 2020.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859*, 2021.

Zhanghao Wu, Paras Jain, Matthew A. Wright, Azalia Mirhoseini, Joseph E. Gonzalez, and Ion Stoica. Representing long-range context for graph neural networks with global attention. In *NeurIPS*, 2021.

Ziyu Yao, Frank F. Xu, Pengcheng Yin, Huan Sun, and Graham Neubig. Learning structural edits via incremental tree transformations. In *International Conference on Learning Representations (ICLR)*, 2021. URL https://openreview.net/forum?id=v9hAX77--cZ.

Michihiro Yasunaga and Percy Liang. Graph-based, self-supervised program repair from diagnostic feedback. In *International Conference on Machine Learning*, pp. 10799–10808. PMLR, 2020.

Pengcheng Yin, Graham Neubig, Miltiadis Allamanis, Marc Brockschmidt, and Alexander L Gaunt. Learning to represent edits. *arXiv preprint arXiv:1810.13337*, 2018.