PLUR: A Unifying, Graph-Based View of Program Learning, Understanding, and Repair

Zimin Chen*
KTH Royal Institute of Technology
Stockholm, Sweden
zimin@kth.se

Vincent J. Hellendoorn†
Carnegie Mellon University
Pittsburgh, USA
vhellend@cs.cmu.edu

Petros Maniatis
Google Research
Mountain View, USA
maniatis@google.com

Pascal Lamblin, Pierre-Antoine Manzagol, Daniel Tarlow, Subhodeep Moitra
Google Research
Montreal, Canada
lamblinp,manzagop,dtarlow,smoitra@google.com

Abstract

Machine learning for understanding and editing source code has recently attracted significant interest, with many developments in new models, new code representations, and new tasks. This proliferation can appear disparate and disconnected, making each approach seemingly unique and incompatible, thus obscuring the core machine learning challenges and contributions. In this work, we demonstrate that the landscape can be significantly simplified by taking a general approach of mapping a graph to a sequence of tokens and pointers. Our main result is to show that 16 recently published tasks of different shapes can be cast in this form, based on which a single model architecture achieves near or above state-of-the-art results on nearly all tasks, outperforming custom models like code2seq and alternative generic models like Transformers. This unification further enables multi-task learning and a series of cross-cutting experiments about the importance of different modeling choices for code understanding and repair tasks. The full framework, called PLUR, is easily extensible to more tasks, and will be open-sourced (https://github.com/google-research/plur).

1 Introduction

The advent of sequence-to-sequence [Sutskever et al., 2014] and, more recently, text-to-text [Raffel et al., 2019, Radford et al., 2019] abstractions has provided a simplifying and unifying view of much work in natural-language processing. By casting problems in this framework, one can apply a single model architecture to many different problems. This benefits machine learning (ML) researchers by focusing attention on the core modeling problem. It also amplifies ML advances, because a general formulation allows practitioners to frame new problems in terms of a standardized abstraction and then to apply state-of-the-art architectures easily, which also facilitates multi-task and transfer learning.

In this paper, we ask: What is the equivalent unifying abstraction for machine learning for source code (ML4Code)? While many ML4Code tasks have been cast in terms of the sequence-to-sequence abstraction, source code is inherently different from natural language [Hindle et al., 2016]. It is

*Work done during internship at Google.
†Work done during visiting-faculty appointment at Google.

35th Conference on Neural Information Processing Systems (NeurIPS 2021).

syntactically more structured as it is designed to be machine-readable and unambiguously parsable. As such, it lends itself to more sophisticated automated analysis than natural language (e.g., static analysis). Furthermore, code tokens follow a power-law distribution due to many rare identifiers, and exhibit high rate of repetition across files and projects [Allamanis and Sutton, 2013, Casalnuovo et al., 2019]. In addition, the software engineering community recognizes that significant effort goes into *software maintenance* involving small changes, rather than writing code from scratch [Koskinen, 2003]. These considerations have led to a wealth of literature going beyond text-to-text models that make use of graphs, copy mechanisms, pointers, and other custom architectures [Maddison and Tarlow, 2014, Bielik et al., 2016, Allamanis et al., 2016, Mukherjee et al., 2017, Yin and Neubig, 2017, Allamanis et al., 2018, Alon et al., 2018, Dinella et al., 2020, Yasunaga and Liang, 2020, Nye et al., 2020].

In particular, the GRAPH2TOCOPO formulation from Tarlow et al. [2020] pairs a graph encoder with a Transformer-style decoder augmented with pointers and a copy mechanism and has been shown to be effective for certain ML4Code tasks. Our proposal is thus to unify around GRAPH2TOCOPO as a common vernacular for ML4Code. This formulation strictly generalizes text-to-text, by additionally supporting graph structures over the input when available, allowing the generation of both pointers and tokens in the output—pointers to specify the input location to edit and tokens to specify the content of the edit—and it has a built-in copy mechanism to more easily enable the re-use of rare tokens from the input. This flexibility enables us to take 16 tasks and models from the ML4Code literature and convert them to a unified form. Using the same hyperparameter sweep and GRAPH2TOCOPO models, we achieve at or above state-of-the-art results on nearly all tasks, including improving over the Hoppity model [Dinella et al., 2020] for program repair, the GREAT model [Hellendoorn et al., 2020] for variable misuse, and the code2seq model [Alon et al., 2018] for variable naming.

Having unified a number of previous tasks and achieved strong results, we can then answer a number of empirical questions: Can we remove the graph structure as input and just use a Transformer encoder? How important are pointers and copy mechanisms? We can also inspect the differences between our model and previous ones and observe the importance of different design choices. The framework also makes it easy to experiment with alternative graph representations of code, and we demonstrate the effect of five different choices for converting a set of abstract syntax tree (AST) paths into graphs in the code2seq domain. Finally, we demonstrate that due to the unified representation, the framework makes it easy to do multi-task learning across classification and repair tasks, yielding improved performance compared to training the same model separately on each task.

In summary, the resulting framework, called PLUR for *Program Learning, Understanding, and Repair*, provides a powerful and simplifying perspective from which to interpret and build upon many recent ML4Code advances. Open-source code for the full PLUR framework will be available under an Apache 2 license at `https://github.com/google-research/plur`.

2 Background and Related Work

Encoders for Source Code Much work has shown the benefit of structured representations of code based on abstract syntax trees (ASTs) or richer semantic graphs containing control flow and data flow [Allamanis et al., 2018, Brockschmidt et al., 2019, Hellendoorn et al., 2020, Alon et al., 2018, Cvitkovic et al., 2019]. A number of recent models appear to achieve the best of both worlds between graph neural networks (GNNs) and Transformers, by using the graph structure in conjunction with relative attention in Transformers [Hellendoorn et al., 2020, Shaw et al., 2018, Wang et al., 2020].

Decoders for ML4Code Tasks The literature on ML4Code proposes a wealth of decoder architectures, targeting a diverse array of tasks. Typical classification tasks (e.g., bug detection) tend to require a simple softmax operation over an output vocabulary. Sequence prediction tasks (e.g., function-name prediction) extend this simple mechanism with an autoregressive decoder [Alon et al., 2018], or a more complex autoregressive structured decoder [Chen et al., 2018]. Other tasks require the ability to point to the input, e.g., to localize a bug [Vasic et al., 2019, Hellendoorn et al., 2020], or point to where an edit must take place [Dinella et al., 2020]. Decoders for such tasks use a pointer network based on attention [Vinyals et al., 2015]. To handle the very large vocabularies of program identifiers and string literals, pointers are also often used to *copy* content from the input, rather than extract it (or reconstruct it) from a limited vocabulary [Allamanis et al., 2016]—this has also been used in some natural-language contexts [Gu et al., 2016]. Unlike using pointer decoders to produce

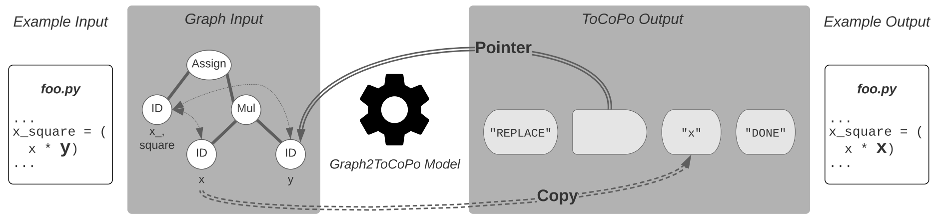

Figure 1: Overview of Graph2ToCoPo abstractions. The raw example input is ingested as a graph; it might be source-code, or another task-specific, pre-computed graph representation. The model produces a ToCoPo output of tokens and pointers, which is compared during training to the ingested ground-truth target of the original task, also re-formulated as a ToCoPo output. Some tokens may be produced as copies from the labels of input nodes. The ToCoPo output can be interpreted by the task to produce the intended raw output, or a task-specific performance metric.

pointers required by the task output, such mechanisms use pointer decoders to identify content in the input from which to copy into the output, e.g., for summarization [See et al., 2017, Fernandes et al., 2018]. Tarlow et al. [2020] proposed combining such decoders into a single decoder that produces ToCoPo, an autoregressive sequence where each element may be a Token, a Copy, or a Pointer.

Source-Code Benchmarks and Frameworks Several recent efforts have created benchmarks of tasks and datasets for ML4Code. CuBERT [Kanade et al., 2020a] released a benchmark of six tasks, for the purpose of evaluating BERT-style pre-trained code embeddings. CodeSearchNet [Husain et al., 2019] created a benchmark and a framework for competing implementations targeting code search and retrieval. CodeXGLUE [Lu et al., 2021] further expanded the benchmark to include tasks from code summarization, repair, code generation, and more. It also provided several baseline model implementations of different characteristics (e.g., CodeBERT [Feng et al., 2020] similar to BERT [Devlin et al., 2019], GraphCodeBERT [Guo et al., 2021] incorporating data-flow edges, etc.), each adapted to a particular subset of task types (e.g., a GPT-based model for sequence generation, a BERT-based model for classification, etc.). In contrast, PLUR seeks a single encoder interface, accepting graphs, and a single decoder interface, producing a mixture of tokens, copies, and pointers, with the thesis that this *single* interface, regardless of model architecture underneath the encoder and decoder, is sufficient to serve a diverse set of ML4Code tasks.

3 A Unifying Framework for Program Understanding and Repair

Our goal from the modeling perspective is specifically *not* to innovate on machine-learning architectures. Our challenge is to package strong existing modeling components in a way that achieves (near) state-of-the-art performance but with maximum simplicity and generality. The novelty in our work comes from making these choices and demonstrating that the result performs strongly across the 16 tasks. Fig. 1 illustrates the approach. See details of the software architecture in the Appendix.

3.1 Models

Based on their strong performance across many tasks in machine learning, we start with Transformers as the base modeling architecture. The encoder is applied to the source-code input, and the decoder generates task-specific sequential outputs. However, we augment the encoder and decoder to incorporate components that have proven useful in recent work on modeling code, most notably relational information that can be used to represent syntax, data flow, and control flow. We adopt the GREAT model as an encoder [Hellendoorn et al., 2020], an instance of a Transformer with relational attention bias. Similarly, we adopt the ToCoPo output formulation from Tarlow et al. [2020] to provide a flexible language that enables outputting arbitrary sequences of tokens, copies, and pointers, along with its modified Transformer decoder architecture. For comparison, we also instantiate this Graph2ToCoPo architecture with a vanilla Transformer encoder, and with a graph neural network encoder (a Gated-Graph Neural Network [Li et al., 2016], or GGNN).

3.2 Input-Output Representations

The next challenge is to adapt a wide variety of recent tasks and datasets into the proposed framework. In some cases, the translation is straightforward. For example, tasks that use GNN encoders and have already released graph representations can be directly used. In other cases, there are custom architectures that do not obviously fit into our proposed unification. However, we show via experiments in Sec. 5 that a light adaptation is able to preserve the favorable inductive biases of the custom approach without the custom modeling. While we use standard representations of inputs and outputs, we describe them explicitly here to make clear what the requirements are for mapping an existing task into the PLUR framework.

The input is represented as a graph composed of nodes and edges. Each node has a discrete *label*, a discrete *type*, and an integer *position*. The label is used to represent token values like names of (possibly subtokenized) variables. The type is used to represent more abstract categories, e.g., to distinguish that some nodes represent source code text and others represent internal AST nodes. The positions are used to impose a linear ordering on the nodes. This enables a vanilla Transformer encoder to process the graph like a token sequence. Edges (s, t, e) are directed, linking a source node index s to a target node index t. Additionally, edges are allowed to have a discrete type e that specifies the kind of relationship that the nodes have. Our graph edges are directed, but in preprocessing we always add a corresponding reverse edge $(t, s, e_{reverse})$ with a fresh type $e_{reverse}$.

The TOCOPO output can be intuitively viewed as a *script* that describes the task output in terms of *tokens*, drawn from the output vocabulary, and *pointers* pointing to some input node, concluding with a `DONE` token marking the end of the output. Every task can make its own use of these facilities to express a *grammar* for its output. For example, a classification task can just use token outputs, one per expected class; a sequence-prediction task can produce sequence of tokens; a repair task can use a pointer to point at a particular input node, and a token output to replace that input node. The copy mechanism in the model enables the production of output by copying the label from an input node.

3.3 The PLUR Tasks

We now briefly describe how we brought 16 tasks and datasets into PLUR, introduced in 9 papers in the recent ML4Code literature and available under public-domain licenses. The philosophy of our approach was to approximate the structure of data representation and encoding in the original papers. The goal is not to emulate the original modeling approach precisely; instead, we aim to capture the inductive bias intended by a custom architecture into the representation of the input and TOCOPO output using our standard architecture. See the Appendix for more details about the tasks and datasets.

ManySStuBs4J [Karampatsis and Sutton, 2020] Classification of a Java function to a number of bug types, or as bug-free. We encode the input as a token sequence, and output the class as a token. We measure classification accuracy.

code2seq [Alon et al., 2018] Sequence prediction of a function name from a Java function body. We encode the input as a set of AST paths between identifiers (but see also Sec. 4.2 for a number of variations), and output a sequence of subtokens constituting a function name. The reported metric is the F1 score. The original paper differentiated among *small*, *medium*, and *large* corpora and we keep the distinction here.

funcom [LeClair et al., 2020] Sequence prediction of a method docstring from the method body. We encode the input as a token chain, and the output is a sequence of subtokens constituting the docstring. The reported metrics are the BLEU scores.

VarMisuseH [Hellendoorn et al., 2020] Localization and repair of a Python variable-misuse bug [Allamanis et al., 2018, Vasic et al., 2019]. The input is already represented as a graph, which we use unchanged. We output a special token for a bug-free example, or a pointer to the bug location, and a token of the correct variable that repairs the bug. We report the hardest metric for the task, the localization and repair accuracy of the model.

Hoppity [Dinella et al., 2020] Repair of bugs in a corpus of ASTs. Repairs are modifications of the input AST (node additions, replacements, etc.). The input is already structured as a graph, which we transcribe to our graph format and use unchanged. We output the transformation type (as a token), and the transformation arguments, which include a pointer (the AST node to transform), and other pointers or tokens depending on the transformation type. We report the repair sequence accuracy (i.e., full match of the repair operations and their arguments).

convattn [Allamanis et al., 2016] Sequence prediction of a method name from the method body. We encode the input as a token chain. We output a sequence of subtokens constituting the method name. The reported metric is the F1 score.

ogb-code [Hu et al., 2021] Sequence prediction of a method name from the method body. The input is already represented as a graph, which we transcribe into our graph format with no semantic changes. We output the sequence of method-name tokens. We report the F1 score[1].

CuBERT [Kanade et al., 2020a] CuBERT consists of six Python-based tasks defined on ETH Py150 Open [Kanade et al., 2020b]. *Exception Classification* (CuBERT-EC): predict one of 20 exception types in a `try/except` clause; *Wrong Operator Classification* (CuBERT-WB): predict if a function is using the wrong binary operator (e.g., `<` instead of `>`); *Swapped Operand Classification* (CuBERT-SO): predict if a non-commutative, binary operator's arguments have been swapped (e.g., `a - b` instead of `b - a`); *Function-Docstring Classification* (CuBERT-FD): predict if a documentation string matches a function body; *Variable Misuse Classification* (CuBERT-VM): classify a function as containing a variable misuse bug (as described above); *Variable Misuse Localization and Repair* (CuBERT-VMR): localize and repair a variable misuse bug. We encode input as a sequence of subtokens using the original vocabulary and tokenizer. We output a token for the classification tasks, and a pointer to the bug, along with the correct variable for CuBERT-VMR. We report classification accuracy for the classification tasks, and localization and repair accuracy for the last task.

Retrieve & Edit [Hashimoto et al., 2018] The task is to complete a Python function given the block comment, function name, and arguments. We encode the block comment, function name, and arguments as separate token chains and they are all connected to a root node. The output is a sequence of function tokens. The reported metrics are the correctly predicted average and maximum number of successive tokens.

Our aim with the choice of datasets was to represent a variety of tasks and representational choices. For example, VarMisuseH and Hoppity are graph-based program repair tasks but with differing output representations. CuBERT-VMR is similar to VarMisuseH but does not use a graph-based input representation. Having chosen these three, we did not include other program repair tasks, and instead chose to prioritize other kinds of tasks like variable naming, code completion, and bug classification.

4 Using the PLUR Framework

4.1 Model Development & Evaluating Cross-Cutting Model Choices

Perhaps the most natural use of PLUR is as an evaluation suite for new model development; following the PLUR GRAPH2TOCOPO abstraction enables a new model to be evaluated against all 16 tasks. Each of these has baseline performance reported in the corresponding 9 papers, and this work provides additional, PLUR-based model implementations and corresponding performance metrics, to establish strong, reproducible baselines for future research. We provide these experimental results in Sec. 5.2.

A related use for PLUR is to perform controlled experiments on modeling choices. For example: How do Transformers compare to GNNs? Are pointer and copy mechanisms helpful? We study several cross-cutting empirical questions in Sec. 5.3. PLUR provides significant variety across tasks and some questions have different answers depending on the task. This shows that PLUR allows for more nuanced conclusions than would be possible from evaluating on a less diverse set of tasks.

There are also a number of interesting conclusions to be drawn by comparing the unified models against the custom models from the literature. Perhaps surprisingly, we often find that the more general model works better than the model designed for the particular task. This provides additional guidance about which modeling choices are important and which are less so. We discuss these results in Sec. 5.2, alongside the comparisons of the standardized PLUR approach to the original approaches.

[1]This task has recently been updated. We will release results for the updated version of the dataset on the open-source repository accompanying the paper.

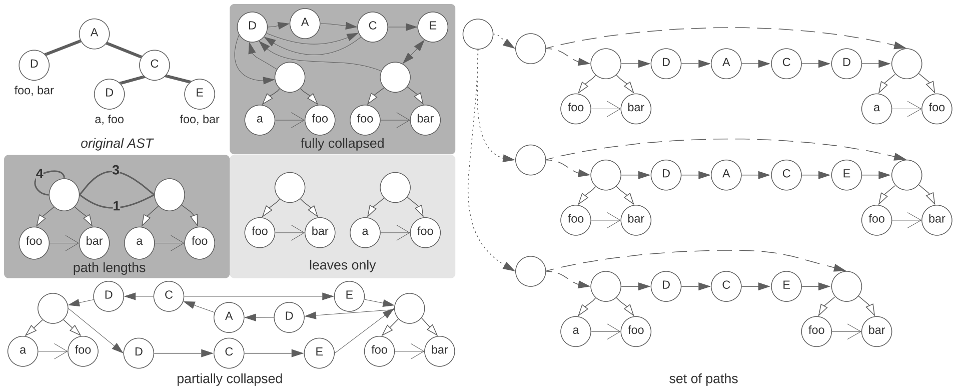

Figure 2: Representation alternatives for code2seq. The original AST is at the top left. Different arrow types represent distinct edge types.

4.2 Task Exploration

Conversely, a practitioner exploring a new task and dataset can cast their existing data to the GRAPH2TOCOPO abstraction and use PLUR's standard model implementations. Without the need for implementing a new model or even connecting a new data-generation pipeline to a training framework, a practitioner gains a rapid and powerful path for obtaining results on new tasks.

We demonstrate this process in the context of the code2seq dataset and task. code2seq uses a custom architecture that represents code as a set of AST paths, and argues that a path-based representation provides a desirable inductive bias, outperforming alternative methods [Alon et al., 2018]. While we cannot replicate the custom architecture exactly as a GRAPH2TOCOPO model, PLUR enables graph representations that capture similar favorable inductive biases, as well as experimentation with alternative, more economical representations that result in fewer graph vertices and model parameters (see Fig. 2). The original graph representation of a Java function is as a set of AST paths between identifier leaf nodes. Each leaf contains a sequence of subtokens representing the identifier, and each path is a sequence of discrete AST node types. Perhaps the most straightforward representation of this original input is as a graph composed of a set of paths. Every path is a chain of nodes (source identifier node, followed by all AST nodes along the path, followed by the destination identifier node), connected via edges. Identifier nodes are further connected to their subtokens, represented also as subtoken-node chains. We call this the *set-of-paths* graph representation.

We also develop four alternatives: *leaves only*: just keep the identifier AST nodes and their subtokens; *path lengths*: keep one instance of the identifier AST nodes and their subtokens, and connect them with edges whose type carries the length of the path between them; *partially collapsed*: keep one instance of each identifier node and its subtokens, and connect them via the path chains, collapsing equivalent path prefixes; *fully collapsed*: keep one instance of each AST node type, and create the multigraph that maps the original path edges to those unique node types. We emphasize that the only change to experiment with these alternatives was in the mapping from the original dataset to the graph representation used as an input. We show in Sec. 5.4 that this leads to improved performance over the custom code2seq model, which gives confidence that PLUR allows one to experiment with alternative input representations without having to simultaneously search for a custom ML architecture.

4.3 Multi-task Learning

The unified view makes it easy to experiment with multi-task learning. The common input representation enables the model to improve its input embedding, by seeing more input graphs, and the common decoder improves its (shared) ability to attend to relevant features of the input.

One benefit of multi-task learning is the ability to train a single model for multiple tasks, which reduces storage and serving costs. However, even when the savings in storage and serving are not the goal, a multi-task model can improve the performance of each task. In principle, similar tasks would benefit from this joint training regime. As an example, CuBERT-WB and CuBERT-SO have the same input representation: subtokenized functions. Since those are binary classification tasks, they share negative classes but have different positive classes (i.e., bug types). This means that the model is exposed to twice as many useful negative examples, and also benefits from the discrimination between positive examples of each class. Especially for small datasets, e.g., CuBERT-EC with only 18K training samples compared to CuBERT-VM's 700K, a shared encoder and decoder gives the smaller task a benefit similar to unsupervised pre-training [Kanade et al., 2020a, Guo et al., 2021]. An additional benefit of the PLUR formulation is that multitask learning is not limited to tasks with the same output format. Because PLUR uses a flexible decoder even for classification, the same parameters are shared between classification tasks and localization & repair tasks.

To study multi-task learning, we created a new task, CuBERT-MT, by adding an extra node to each example with a new type and the task as a label, keeping the output unchanged, and combining all examples into one dataset. We evaluate the resulting model on the per-task test datasets and metrics (see Sec. 5.5 for experiments and Appendix for further task details).

5 Experiments

In the experiments we ask the following research questions:

- **RQ1: How does the general PLUR approach compare to approaches like GREAT [Hellendoorn et al., 2020], Hoppity [Dinella et al., 2020], and code2seq [Alon et al., 2018]?** To evaluate this, we compare the PLUR family of models to the approaches and metrics used by the original papers across the 16 tasks.
- **RQ2: How do encoders based on the Transformer, GREAT [Hellendoorn et al., 2020], and GGNN [Li et al., 2016] compare across the PLUR benchmark?** To answer this, we train each kind of model on each task, and then we evaluate the results.
- **RQ3: How important are copy mechanisms? How important are pointers to specify locations?** Does the Transformer effectively have a copy mechanism built in due to the use of attention? Similarly, can pointers be replaced by special tokens representing locations? We study these questions by disabling the copy and pointer mechanisms in the PLUR decoder for the Variable Misuse localization and repair tasks, and compare results against the default decoder.
- **RQ4: What is the effect of different graph representations?** We study the effect of the graph representations from Sec. 4.2, and evaluate if we can achieve favorable inductive bias using the generic GRAPH2TOCOPO architecture.
- **RQ5: Does multi-task learning provide improvements?** PLUR makes it easy to do multi-task learning even across tasks with different output kinds (e.g., across classification and program repair tasks). Is this direction promising?

5.1 Experimental Details

We use a common protocol across our experiments and follow the descriptions in Sec. 3.3 to ingest the 16 tasks. Dataset sizes are in the Appendix. We trained each model variant (GREAT2TOCOPO, TRANSFORMER2TOCOPO, and GGNN2TOCOPO) on each of the tasks using 8-core TPU-v2s for acceleration. For each task and model variant, we perform a grid search over hyperparameters and minor implementation variations (see Appendix). Typical training and evaluation time is 3 days per task, although larger tasks such as CuBERT-MT took slightly longer. We swept 12 hyperparameter combinations per task, across 3 models and across the 16 tasks, spending around 14,000 hours of accelerator time overall. We choose the best checkpoint for each run according to full-sequence accuracy on a subset of validation data, and we also choose the best hyperparameter setting according to full-sequence accuracy on the full validation data. We generate test predictions from the best checkpoint and hyperparameter setting, then we evaluate the predictions against ground truth outputs using task-specific evaluation metrics, e.g., F1 score, BLEU, etc. We report 95% Bernoulli confidence intervals for accuracy metrics. For non-accuracy metrics, e.g., F1 score, we compute a 95% bootstrap estimate by resampling the validation and test sets with replacement 20 times.

5.2 Main Results

Our first results address **RQ1** and appear in Tab. 1. The first observation is that PLUR is able to match the reported performance across nearly all tasks, showing that the generality of the PLUR

Table 1: Task-specific test metrics, comparing to the reporting papers in comparable settings. All results reported with 95% confidence intervals. Significant results are **bold**.

Task	PLUR	Original Paper	Metric
VarMisuseH	**82.8%** (±0.10)	80.4%	Loc.+Repair Acc.
CuBERT-VMR	**78.6%** (±0.13)	56.9%	Loc.+Repair Acc.
Hoppity	**26.6%** (±0.32)	14.2%	Sequence Acc.
funcom	**42/31/25/21** (±0.19)	39/22/15/11	BLEU-1/2/3/4
convattn	**52.5** (±0.21)	44.7	F1
code2seq small	**48.7** (±0.43)	43.0	F1
code2seq med	**54.2** (±0.13)	53.2	F1
code2seq large	**62.1** (±0.09)	59.2	F1
ogb-code	**35.3** (±0.54)	32.6	F1
CuBERT-EC	**53.0%** (±0.96)	49.6%	Classification Acc.
CuBERT-FD	86.6% (±0.15)	**91.0%**	Classification Acc.
CuBERT-SO	87.7% (±0.18)	87.8%	Classification Acc.
CuBERT-VM	75.0% (±0.14)	**78.3%**	Classification Acc.
CuBERT-WB	**80.9%** (±0.15)	76.6%	Classification Acc.
ManySStuBs4J	64.1% (±2.09)	—	Classification Acc.
Retrieve&Edit	2.5/10.3 (±0.24)	**5.8/17.6**	Avg/Longest Length

formulation does not come at the cost of accuracy. In fact, the PLUR version outperforms strong models like those from Hellendoorn et al. [2020], Dinella et al. [2020] and Alon et al. [2018]. In the Alon et al. [2018] setting, it is notable that it provides improvement for all scales of dataset, even on the "small" dataset, where previously code2seq was shown to have a large advantage over Transformer models.

To understand this, it is informative to compare modeling choices. For instance, the PLUR approach is similar to the model from Hellendoorn et al. [2020], but has a different decoder: whereas the original predicted the locations of bug and repair using independent output heads, the ToCoPo decoder uses autoregressive connections within the decoder, conditioning on bug locations when predicting repairs, which likely improves performance. Another interesting result is the strong performance of the vanilla Transformer encoder on this task, even though there are many informative edge types in the data. Leveraging this information with graph-based Transformer encoders to improve even further could be an interesting modeling challenge.

In the code2seq task, we use the same AST path data as in Alon et al. [2018], and both approaches have an attention-based decoder. The main difference is that code2seq independently encodes each path, whereas our formulation uses a relation-aware Transformer on a graph representation of the paths. We suspect that the increased connectivity across paths is favorable, while the encoder is still able to make use of some relational information in the paths. Compared to Hoppity, the PLUR formulation is a bit simpler, e.g., not applying graph-edit operations during decoding, and it treats the output sequence more generically, sharing the same attention-based decoder to generate each pointer or token in the output script.

On the CuBERT-based tasks, there are some cases where the PLUR results are better and some where the BERT Transformer implementation (reported in the original paper) is better. We report comparisons to the CuBERT results that do not use pretraining, so these variations are likely due to differences in Transformer implementations and hyperparameter searches (CuBERT searched over thousands of Transformer hyperparameter combinations in contrast to our 12). However, the fact that PLUR is competitive shows that we are working from a strong Transformer implementation.

The one case where PLUR significantly under-performs results from the original paper is the Retrieve & Edit task. This is perhaps unsurprising because the PLUR formulation does not use the retrieval mechanism from the paper, which is shown to produce large improvements in results. However, the PLUR result is in-line with performance of the Seq2Seq baseline reported in Hashimoto et al. [2018] for the main metrics of average and maximum completion length, although we observed higher BLEU scores than reported for the Seq2Seq baseline.

In total, these results demonstrate that the default PLUR models produce almost uniformly strong results, without per-task model customization.

Table 2: (a) Comparing PLUR Encoders: GNN vs. Transformer vs. Relation-aware Transformer (validation sequence accuracies). (b) Multi-task over the CuBERT tasks (test metrics). (c) Impact of alternative graph representations on the test metric of the code2seq med task, using GREAT. (d) Validation sequence accuracy for different output DSL ablations. Validation sequence accuracy results are within an error interval of 0.4 at 95% confidence. Significant results are **bold**.

(a) Comparing encoders.

Task	GGNN	Transformer	GREAT
VarMisuseH	84.6%	**87.2%**	86.7%
CuBERT-VMR	61.5%	85.6%	85.7%
Hoppity	31.4%	29.1%	31.1%
funcom	12.8%	13.4%	13.3%
convattn	37.3%	37.4%	**38.5%**
code2seq small	30.0%	23.8%	**31.6%**
code2seq med	36.9%	35.5%	**39.1%**
code2seq large	38.7%	36.6%	**39.7%**
ogb-code	22.9%	22.7%	**24.1%**
CuBERT-EC	52.0%	56.2%	56.5%
CuBERT-FD	84.9%	88.8%	88.6%
CuBERT-SO	80.5%	**89.5%**	88.0%
CuBERT-VM	64.8%	**76.3%**	74.7%
CuBERT-WB	78.2%	**83.2%**	81.7%
ManySStuBs4J	65.7%	65.1%	65.6%
Retrieve&Edit	3.8%	4.1%	4.1%

(b) Multi-task setting.

Task	Original paper	PLUR		
		Uni-task	Multi-task	Per Task
CuBERT-EC	49.6%	53.0%	62.7% (±0.93)	**64.8**% (±0.92)
CuBERT-FD	**91.0**%	86.6%	80.8% (±0.18)	84.3% (±0.16)
CuBERT-SO	87.8%	87.7%	90.3% (±0.16)	**90.7**% (±0.16)
CuBERT-VM	78.3%	75.0%	89.6% (±0.10)	**90.1**% (±0.10)
CuBERT-WB	76.6%	80.9%	86.1% (±0.14)	**86.4**% (±0.13)
CuBERT-VMR	56.9%	78.6%	79.7% (±0.13)	**81.0**% (±0.12)

(c) Graph representations.

Representation	Test F1
leaves only	51.2 (±0.14)
path lengths	53.8 (±0.16)
fully collapsed	52.2 (±0.10)
partially collapsed	54.2 (±0.13)
set of paths	**56.4** (±0.12)

(d) Pointer/Copy ablations.

Output DSL	VarMisuseH			CuBERT-VMR		
	GREAT	Transformer	GGNN	GREAT	Transformer	GGNN
Full	**87.1**%	85.4%	**84.6**%	**85.7**%	85.6%	61.5%
No pointer	85.0%	85.1%	84.0%	83.9%	**86.2**%	60.9%
No copy	79.1%	79.2%	77.8%	83.5%	83.0%	58.4%
No copy, no pointer	76.3%	78.8%	78.0%	84.0%	84.5%	61.8%

5.3 Ablations & Model Choices

Next, we address **RQ2** and **RQ3**. To evaluate **RQ2**, we break down results by model variant (Tab. 2a), and we report validation full-sequence accuracy. This provides uniformity and is also the criterion used for model selection, which may reduce some possible confounding of results.

A first interesting comparison is the GGNN encoder against the Transformer encoder. There are some tasks where the GGNN is superior (code2seq all sizes and Hoppity), some where the Transformer is superior (VarMisuseH, all CuBERT tasks), and several where they perform similarly. When GGNN is superior, we interpret this to mean that the graph structure provides a strongly useful signal that the Transformer is not able to infer implicitly through its attention computations. When the Transformer is superior, we interpret this to mean that the sequential structure in the problem, long-distance all-to-all communication, and perhaps overall model capacity are more important than the graph structure. Second, we can compare the GREAT encoder to both the alternatives. In cases where the GGNN encoder is superior to the Transformer encoder, GREAT is also superior to the Transformer encoder. However, the GREAT encoder is much stronger than the GGNN encoder in cases where Transformer encoder outperforms the GGNN encoder. In cases where there is not useful signal in the graph representation, we attribute differences between Transformer and GREAT encoders to minor implementation details. In total, results show that the GREAT encoder provides good performance across the full range of tasks.

For **RQ3**, we perform ablation studies that disable the copy mechanism, the use of pointers in the output scripts, and both. We run these experiments on VarMisuseH and CuBERT-VMR because these tasks make use of both pointers (to specify the bug location) and the copy mechanism (to generate the replacement variable name). We further split out results by model type to see if, e.g., Transformers have less use for a copy mechanism than GGNNs. Results appear in Tab. 2d. The first conclusion is that including pointers and the copy mechanism both lead to improved performance across models and tasks, with the copy mechanism alone being more useful than the pointer mechanism alone. There does not appear to be significant model-specific conclusions to be drawn, aside from the fact that CuBERT-VMR results are poor for GGNN because there is no graph structure to make use of.

5.4 Alternative Graph Representations

In **RQ4**, we explore the effect of the alternative ways of converting AST paths to graph representations. We run training sweeps for each dataset variation described in Sec. 4.2. Test results appear in Tab. 2c, and more details appear in the Appendix.

There is significant variation in performance as the graph representation changes. As a general trend, the relative benefit of structure increases as representations become compact, and overall performance of Transformer encoders increases as graph representations become less compact. "Path lengths" performs relatively well and is reminiscent of a path-length positional encoding representation [Zügner et al., 2021]. The results demonstrate how the inductive biases from a custom model can be imported into PLUR, and how PLUR can be used to explore alternative representations of code.

5.5 Multi-task Learning

Tab. 2b shows the results of multi-task training. Each cell shows the metric computed on the test examples of the corresponding task drawn from the model chosen under a different model-selection strategy. "Original paper" and "Uni-task" show again the best test values from Tab. 1 for comparison. "Multi-task" selects a single checkpoint via our model-selection protocol over the entire set of validation examples. This emulates the scenario in which we seek a single model for all tasks, and we select one based on a task-agnostic metric, the validation sequence accuracy. It outperforms 5 of the 6 uni-task models. The benefit is particularly pronounced in CuBERT-VM. It is also significant for the smallest of the tasks, CuBERT-EC, which has very few training examples, and therefore benefits from exposure to examples from different tasks. "Per task" chooses one multi-task checkpoint per task; for each task, it evaluates the validation examples only of that task, and it chooses the multi-task model with the highest validation sequence accuracy. Here a task picks a multi-task checkpoint that suits it, resulting in further improvements. This is akin to augmentation or pre-training, eschewing reduced cost in favor of a stronger per-task model.

6 Conclusion

PLUR provides a powerful and simplifying perspective from which the ML community can study the unique challenges in ML4Code problems. We have demonstrated that the unified framework achieves state-of-the-art results on several ML4Code tasks, showing that our carefully chosen general approach can outperform many task-specific models. PLUR is easily extensible and could grow in the future to include more tasks and more models that are compatible with the core abstractions. Experimentally, we have explored and answered several interesting questions about what models work best, the effect of graph structure design, the importance of copies and pointers, and provided initial evidence that multi-task training in PLUR is promising.

One limitation is that PLUR is cast within a fairly straightforward framework of supervised learning with encoder-decoder models. It is less suitable for approaches that go beyond this formulation, as we see with the Retrieve & Edit task. However, it would be interesting future work to explore using PLUR as a component in other paradigms. For example, we might use the retrieval method of Hashimoto et al. [2018] in conjunction with the graph construction and learning of PLUR. From a broader impacts perspective, one risk is that the benchmarks may not be representative of the full diversity of potential use-cases of the technology, and the particular choices may emphasize use-cases that disproportionately favor a subset of the population. While we believe it to be understudied in ML4Code, it would be interesting in future work to study how ML4Code benchmarks could be oriented to provide more uniform benefits (e.g., perhaps including more emphasis on new programmers). Finally, we have not yet incorporated pretraining into PLUR. Another direction for future work is to explore pretraining that can be used generically within PLUR and leveraged across the full suite of tasks.

7 Acknowledgements

We would like to thank David Bieber for carefully reading our paper and providing valuable feedback and research guidance. Additionally, we are thankful to researchers in the learning for code effort at Google Research as well as members of Google Brain Montreal for their comments and advice.

References

Miltiadis Allamanis and Charles Sutton. Mining source code repositories at massive scale using language modeling. In *2013 10th Working Conference on Mining Software Repositories (MSR)*, pages 207–216, 2013. doi: 10.1109/MSR.2013.6624029.

Miltiadis Allamanis, Hao Peng, and Charles Sutton. A convolutional attention network for extreme summarization of source code. In *International conference on machine learning*, pages 2091–2100. PMLR, 2016.

Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. Learning to represent programs with graphs. In *International Conference on Learning Representations*, 2018.

Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. code2seq: Generating sequences from structured representations of code. In *International Conference on Learning Representations*, 2018.

Pavol Bielik, Veselin Raychev, and Martin Vechev. PHOG: Probabilistic model for code. In *International Conference on Machine Learning*, pages 2933–2942. PMLR, 2016.

Marc Brockschmidt, Miltiadis Allamanis, Alexander Gaunt, and Oleksandr Polozov. Generative code modeling with graphs. In *International Conference on Learning Representations*, 2019.

Casey Casalnuovo, Kenji Sagae, and Prem Devanbu. Studying the difference between natural and programming language corpora. *Empirical Software Engineering*, 24(4):1823–1868, 2019.

Xinyun Chen, Chang Liu, and Dawn Song. Tree-to-tree neural networks for program translation. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, NIPS'18, page 2552–2562, Red Hook, NY, USA, 2018. Curran Associates Inc.

Milan Cvitkovic, Badal Singh, and Animashree Anandkumar. Open vocabulary learning on source code with a graph-structured cache. In *International Conference on Machine Learning*, pages 1475–1485. PMLR, 2019.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, Minneapolis, Minnesota, June 2019. Association for Computational Linguistics. doi: 10.18653/v1/N19-1423. URL `https://www.aclweb.org/anthology/N19-1423`.

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.

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.

Patrick Fernandes, Miltiadis Allamanis, and Marc Brockschmidt. Structured neural summarization. In *International Conference on Learning Representations*, 2018.

Jiatao Gu, Zhengdong Lu, Hang Li, and Victor OK Li. Incorporating copying mechanism in sequence-to-sequence learning. *arXiv preprint arXiv:1603.06393*, 2016.

Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie LIU, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. GraphCodeBERT: Pre-training code representations with data flow. In *International Conference on Learning Representations*, 2021. URL `https://openreview.net/forum?id=jLoC4ez43PZ`.

Tatsunori B. Hashimoto, Kelvin Guu, Yonatan Oren, and Percy Liang. A retrieve-and-edit framework for predicting structured outputs. In Samy Bengio, Hanna M. Wallach, Hugo Larochelle, Kristen Grauman, Nicolò Cesa-Bianchi, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*, pages 10073–10083, 2018. URL `https://proceedings.neurips.cc/paper/2018/hash/cd17d3ce3b64f227987cd92cd701cc58-Abstract.html`.

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. URL `https://openreview.net/forum?id=B1lnbRNtwr`.

Abram Hindle, Earl T Barr, Mark Gabel, Zhendong Su, and Premkumar Devanbu. On the naturalness of software. *Communications of the ACM*, 59(5):122–131, 2016.

Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, and Jure Leskovec. Open graph benchmark: Datasets for machine learning on graphs, 2021.

Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. CodeSearchNet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436*, 2019.

Aditya Kanade, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi. Learning and evaluating contextual embedding of source code. In *Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 12-18 July 2020*, Proceedings of Machine Learning Research. PMLR, 2020a.

Aditya Kanade, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi. A redistributable subset of the eth py150 corpus. `https://github.com/google-research-datasets/eth_py150_open`, 2020b.

Rafael-Michael Karampatsis and Charles Sutton. How often do single-statement bugs occur? the manysstubs4j dataset. In *Proceedings of the 17th International Conference on Mining Software Repositories*, pages 573–577, 2020.

Jussi Koskinen. Software maintenance costs. *Information Technology Research Institute, ELTIS-Project University of Jyväskylä*, page 16, 2003.

Alexander LeClair, Sakib Haque, Lingfei Wu, and Collin McMillan. Improved code summarization via a graph neural network. In *Proceedings of the 28th International Conference on Program Comprehension*, pages 184–195, 2020.

Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard S. Zemel. Gated graph sequence neural networks. In *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, 2016. URL `http://arxiv.org/abs/1511.05493`.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin B. Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. CodeXGLUE: A machine learning benchmark dataset for code understanding and generation. *CoRR*, abs/2102.04664, 2021.

Chris Maddison and Daniel Tarlow. Structured generative models of natural source code. In *International Conference on Machine Learning*, pages 649–657. PMLR, 2014.

Rohan Mukherjee, Dipak Chaudhari, Matthew Amodio, Thomas Reps, Swarat Chaudhuri, and Chris Jermaine. Neural attribute grammars for semantics-guided program generation. *arXiv e-prints*, pages arXiv–1705, 2017.

Maxwell Nye, Yewen Pu, Matthew Bowers, Jacob Andreas, Joshua B Tenenbaum, and Armando Solar-Lezama. Representing partial programs with blended abstract semantics. *arXiv preprint arXiv:2012.12964*, 2020.

Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. *OpenAI Blog*, 1(8), 2019.

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *arXiv preprint arXiv:1910.10683*, 2019.

Abigail See, Peter Liu, and Christopher Manning. Get to the point: Summarization with pointer-generator networks. In *Association for Computational Linguistics*, 2017. URL `https://arxiv.org/abs/1704.04368`.

Peter Shaw, Jakob Uszkoreit, and Ashish Vaswani. Self-attention with relative position representations. *arXiv preprint arXiv:1803.02155*, 2018.

Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. *arXiv preprint arXiv:1409.3215*, 2014.

Daniel Tarlow, Subhodeep Moitra, Andrew Rice, Zimin Chen, Pierre-Antoine Manzagol, Charles Sutton, and Edward Aftandilian. Learning to fix build errors with Graph2Diff neural networks. In *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*, pages 19–20, 2020.

Marko Vasic, Aditya Kanade, Petros Maniatis, David Bieber, and Rishabh Singh. Neural program repair by jointly learning to localize and repair. *CoRR*, abs/1904.01720, 2019. URL `http://arxiv.org/abs/1904.01720`.

Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. Pointer networks. *arXiv preprint arXiv:1506.03134*, 2015.

Bailin Wang, Richard Shin, Xiaodong Liu, Alex Polozov, and Matthew Richardson. RAT-SQL: Relation-aware schema encoding and linking for text-to-sql parsers. In *ACL 2020*, June 2020. URL `https://www.microsoft.com/en-us/research/publication/rat-sql-relation-aware-schema-encoding-and-linking-for-text-to-sql-parsers/`.

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

Pengcheng Yin and Graham Neubig. A syntactic neural model for general-purpose code generation. *arXiv preprint arXiv:1704.01696*, 2017.

Daniel Zügner, Tobias Kirschstein, Michele Catasta, Jure Leskovec, and Stephan Günnemann. Language-agnostic representation learning of source code from structure and context. *arXiv preprint arXiv:2103.11318*, 2021.