Machine Learning for Code Synthesis and Analysis

Steve Kommrusch

Abstract: Deep learning has been successfully applied to a wide array of problems due to its versatility and the variety of algorithms being developed. Deep learning for natural language processing makes use of recurrent networks, attention layers, copy mechanisms, and analysis of the sentence structure by representing input sentences as graphs. These same techniques can be applied to analyzing and generating computer languages as demonstrated by recent research papers.

This paper provides an overview of machine learning for program repair, code summarization, and program equivalence. Background topics are summarized before presenting an in depth discussion and analysis of five papers published in this area. Recent work that relates to these five papers and possible future research paths in this area are discussed.

1 Introduction

The field of program synthesis dates back to the early days of software design, when the goal of designing code to meet a given set of requirements was first being conceptualized. More recently, advances in hardware capability and software algorithms have enabled deep learning to apply multi-layer neural networks to a variety of problems. In this paper, we will study the research on the background and methodology needed to create a machine learning model that can automatically analyze and generate high-level languages (such as C or Java). This paper is the written portion of the PhD research exam for computer science at Colorado State University.

In order to bring focus to our study, we will center our discussion on five base papers. The papers are listed below along with their 'nicknames' used in the following sections for easy reference.

1. Staged Program Repair with Condition Synthesis [16] (the SPR paper) discusses searching through code transformations to repair bugs, our discussion of this paper will explain the problem of program repair in depth.
2. PHOG: Probabilistic Model for Code [7] (the PHOG paper) presents a probabilistic grammar to predict node types in an AST.
3. Get to the Point: Summarization with Pointer-Generator Networks [22] (the Pointer paper) teaches a method useful in natural language processing for copying rare tokens from an input sentence to an output sentence.
4. Graph-to-Sequence Learning using Gated Graph Neural Networks [6] (the graph2seq paper) presents a method for using a graph neural network to analyze an input sentence and produce an output sentence.
5. code2vec: Learning Distributed Representations of Code [3] (the code2vec paper) teaches a method for creating a useful embedding vector that summarizes a snippet of code.

The rest of this paper is organized as follows. Section 2 will introduce background concepts and published work which will be valuable in our study of the five base papers. Section 3 will provide an overview of the details for each of the base papers. Section 4 will introduce two recent papers and explore their relationship to the ideas and approaches discussed in our base papers. Section 5 will present a possible future research path in this area synthesizing opportunities created based on the base papers.

2 Background

As the focus of this paper is machine learning for code synthesis and analysis, this section provides background in this area for later reference when the base papers are discussed. As we will see in

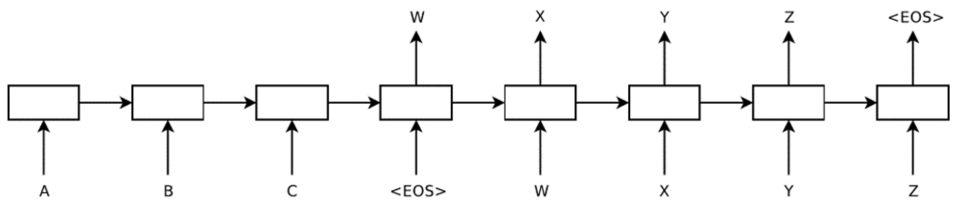


Figure 1: Our model reads an input sentence “ABC” and produces “WXYZ” as the output sentence. The model stops making predictions after outputting the end-of-sentence token. Note that the LSTM reads the input sentence in reverse, because doing so introduces many short term dependencies in the data that make the optimization problem much easier.

Figure 1: Figure from Sequence-to-Sequence paper showing example of early model

our base papers, many successful approaches to applying machine learning to human language have been shown to work well for generating computer languages. Allamanis, *et al.* provide a broad case for this phenomenon in their survey paper [1]. In their survey, the authors note that computer language creates a bridge between humans and computers. Hence, while the syntax of computer languages are more precisely defined that human languages, humans writing code tend to certain coding patterns and styles. This creates meaningful statistical distributions at token, loop, method, and class levels that can be discovered through machine learning techniques.

2.1 Neural machine translation with sequence-to-sequence learning

Neural machine translation (NMT) evolved from statistical machine translation (SMT). SMT made use of smoothed n-gram models to predict the probabilities of words in a destination language given a source language and neighboring words. NMT, by use of examples and back propagation, uses a neural network to learn the most likely translation for a given input [23].

An early example of a sequence-to-sequence network uses an RNN (recurrent neural network) to read in tokens and generate an output sequence, as shown in Figure 1 [23]. In this network, outputs are created from the same neurons that received the inputs. The input tokens are denoted x_t, and after receiving all of the input tokens and a special <EOS> token, the output tokens are fed into the network to aid in proper generation of the next token. The output tokens are denoted y_t. In the following equations, h_t is the hidden state of a recurrent neural network, W^{hx}, W^{hh}, and W^{yh} are weights learnable with supervised learning and backpropagation.

$$\begin{gathered} h_t = \sigma(W^{hx}x_t + W^{hh}h_{t-1}) \\ y_t = W^{yh}h_t \end{gathered}$$

A softmax function is then used to turn the y_t values in the preceding equation into probabilities to choose the most likely token from a learned vocabulary. In this early example, one can see how the weight matrices can mimic the learning of n-gram data used in SMT; after processing the input sequence, the hidden state $h_{<eos>}$ encodes the most likely initial token to begin the output and each subsequent h_t can use the W matrices to predict the most likely next token given the input as well as preceding tokens produced on the output. The W matrices can, thus, encode n-gram-like information, but can also learn when to encode longer range likelihoods based on the information in the training data.

2.2 Gated recurrent unit

Early neural machine translation architectures made use of Long Short Term Memories (LSTMs) [23], but gated recurrent units were found to train more effectively and produce improved accuracy in some cases [10]. A gated recurrent unit is slightly simpler than an LSTM as it has only 2 gates to learn instead of 3. The feedback is shown in figure Figure 2. In the equations below, the notation $[\cdot]_j$ represents the j^{th} element of a vector. x is the input vector to the GRU layer; r is the reset gate;

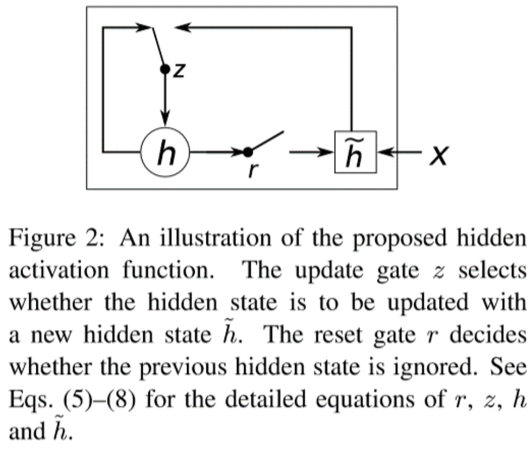


Figure 2: An illustration of the proposed hidden activation function. The update gate z selects whether the hidden state is to be updated with a new hidden state $\tilde{h}$. The reset gate r decides whether the previous hidden state is ignored. See Eqs. (5)–(8) for the detailed equations of r, z, h and $\tilde{h}$.

Figure 2: Figure from initial GRU paper showing GRU functionality

and z is the update gate. W_r, U_r, W_z, U_z, W, and U are all matrices with learnable weights. h_j^t is the hidden state of the j^{th} unit after t iterations of the recurrent equations.

$$r_j = \sigma([W_r x]_j + [U_r h_{t-1}]_j)$$
$$z_j = \sigma([W_z x]_j + [U_z h_{t-1}]_j)$$
$$\tilde{h_j^t} = tanh([Wx]_j + [U(r \odot h_{t-1})]_j)$$
$$h_j^t = z_j h_j^{t-1} + (1 - z_j)\tilde{h_j^t}$$

Papers published recently still may use LSTM or GRU, but the GRU was developed specifically to aid in the problem of learning for neural machine translation.

2.3 Graph neural networks

One of the five primary papers we will be discussing builds on the idea of a graph neural network, so it is useful to introduce the concept. The key initial paper relating to graph neural networks was written in 2009 [20].

In the 2009 paper, the graph neural network is described as an iterative encoding network. The network uses as input labels for nodes and edges, the labels having dimensions d_N and d_E respectively. The labels attached to node n are denoted $l_n \in \mathbb{R}^{d_N}$; the labels attached to edge (n_1, n_2) are denoted $l_{n_1,n_2} \in \mathbb{R}^{d_E}$. Additionally, $l_{co[n]}$ and $l_{ne[n]}$ are the labels for edges connected to node n and labels of nodes connected by edges to node n, respectively.

Figure 3 shows diagramatically how the input information is used in a GNN. x_n is the hidden state for node n, and o_n is the output of node n. These are computed by iterating on the equations below:

$$x_n(t+1) = f_w(l_n, l_{co[n]}, x_{ne[n]}(t), l_{ne[n]})$$
$$\forall n \in N, o_n(t) = g_w(x_n(t), l_n)$$

The outputs o_n of the network allow for training samples to be used for setting the weights in the functions f_w and g_w. Typically $x_n(0)$ is initialized to 0 for all nodes. The 2009 paper discusses the constraints on the learnable function f_W that insures $x_n(t)$ converges over some finite number of steps. The function f_W is implemented as a different matrix for each edge type, so the number of parameters to learn in a GNN grows linearly with the number of edge types. The hidden state of all connected nodes is processed through each edge matrix to create the final output of f_W. As can be surmised, the number of iteration steps is equal to the hop distance to the furthest node in the graph that can affect another node.

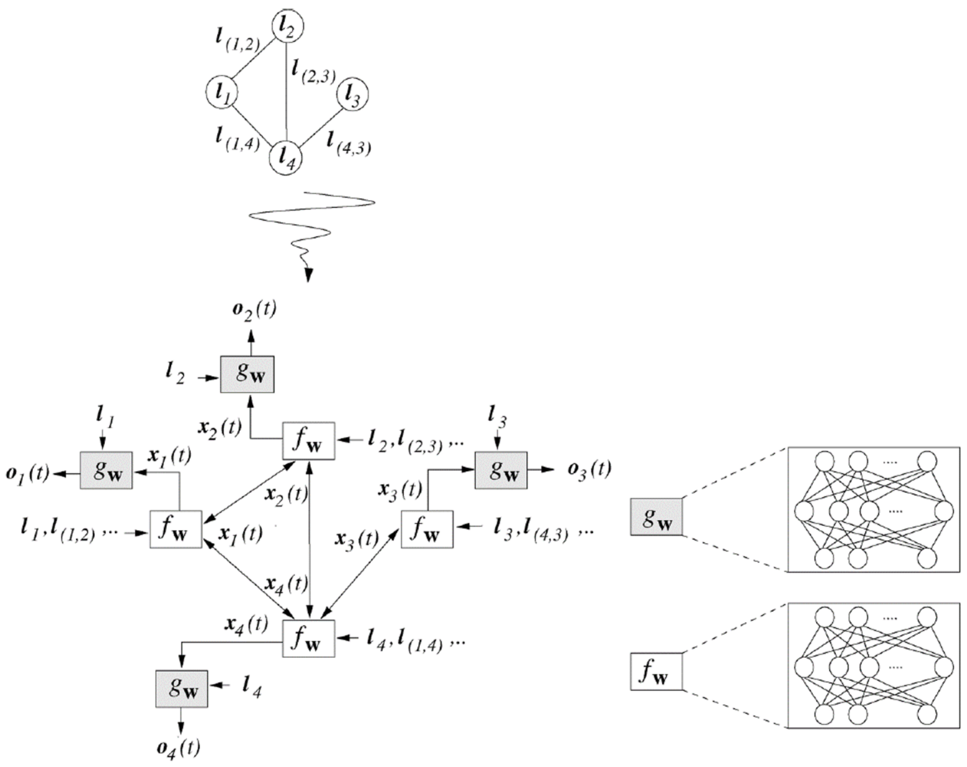


Figure 3: Figure from initial graph neural network paper. Using label values for nodes and edges, a learned function f_W is iterated on at each node and ultimately used to produce $o_n(t)$.

As presented in Figure 3, the GNN is producing one output per node. Another use case for a GNN is to do a softmax function on o_n, which allows a node to be selected as an output. Alternately, summing all o_n together can produce a usable single output for a graph.

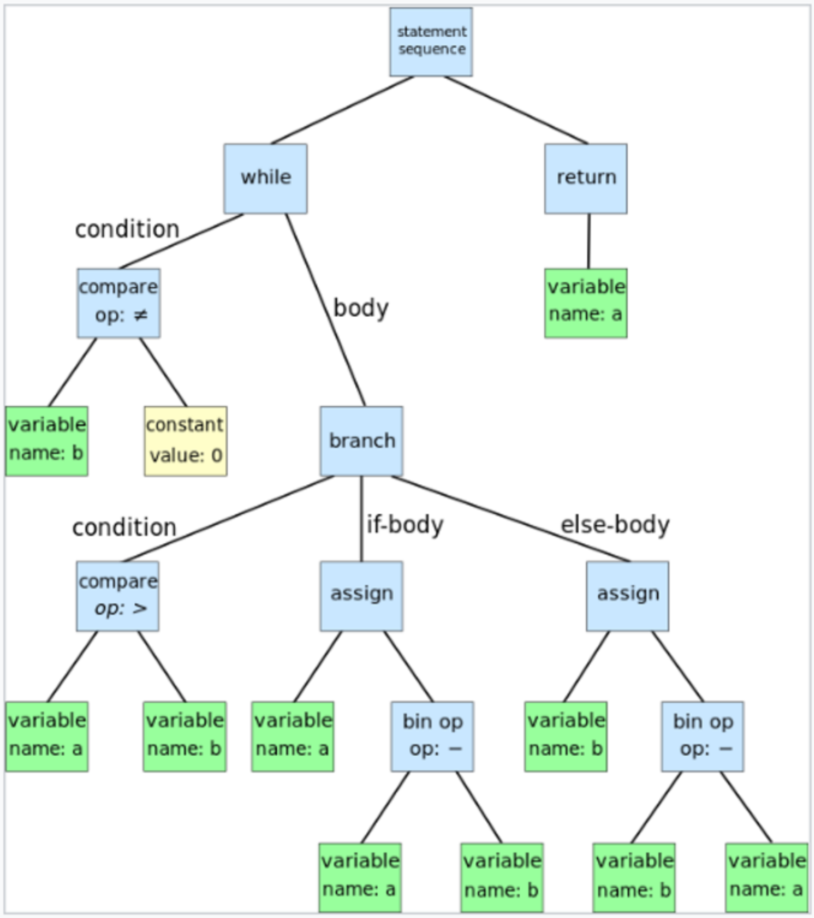
2.4 Abstract Syntax Trees

Human languages can be parsed and broken down into clauses and parts of speech, but this process is not mathematically precise due to exceptions and nuances in human language [12]. Computer languages, on the other hand, are designed to be automatically and predictably parsed.

Figure 4 shows a typical abstract syntax tree (AST) for a short code snippet [24]. The tree is abstract in that certain syntax (such as parenthesis) are not necessary, but given the AST for a code snippet, equivalent code can be reconstructed. The tree provides a structure for identifying the way in which control statements and variables are used in the program, and Section 3 will show how it can be useful as an input to machine learning approaches on code.

2.5 Program equivalence

The problem of proving program equivalence is one of the earliest problems in computer science [15]. The problem is to determine when two programs with different semantics that are given the same inputs will produce the same outputs (and in some formulations, side effects like memory state must also be identical). The applications for program equivalence checking include: 1. validation of any algorithm that does program transformations (such as compilers) 2. plagiarism detection (useful in MOOCs *etc.*) 3. formal verification when refactoring code (for readability, security automation, *etc.*) 4. virus and other malware detection by detecting similar code sequences.



An abstract syntax tree for the following code for the Euclidean algorithm:

```
while b ≠ 0
    if a > b
        a := a − b
    else
        b := b − a
return a
```

Figure 4: Wikipedia image of an abstract syntax tree for a short code snippet

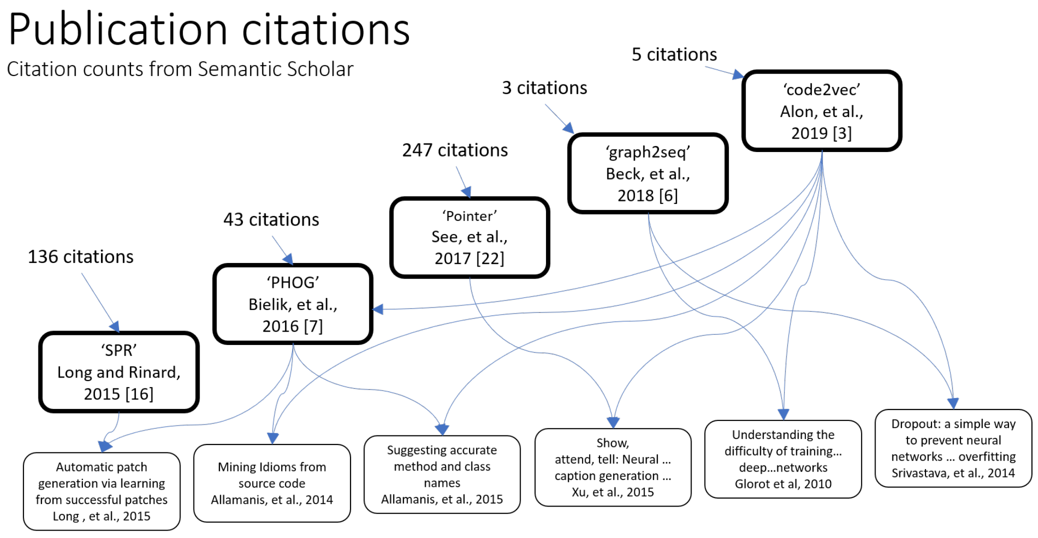

Publication citations
Citation counts from Semantic Scholar
5 citations
'code2vec' Alon, et al., 2019 [3]
3 citations
'graph2seq' Beck, et al., 2018 [6]
247 citations
'Pointer' See, et al., 2017 [22]
43 citations
'PHOG' Bielik, et al., 2016 [7]
136 citations
'SPR' Long and Rinard, 2015 [16]
Automatic patch generation via learning from successful patches Long , et al., 2015
Mining Idioms from source code Allamanis, et al., 2014
Suggesting accurate method and class names Allamanis, et al., 2015
Show, attend, tell: Neural … caption generation … Xu, et al., 2015
Understanding the difficulty of training… deep…networks Glorot et al, 2010
Dropout: a simple way to prevent neural networks … overfitting Srivastava, et al., 2014

Figure 5: Citation counts to the five base papers and selected references shared by the papers.

2.6 Program repair

Program repair is another area that has a rich history related to the five base papers. Program repair is the problem of finding a patch to a buggy input program in order to address the bug. The papers themselves present key aspects of the history of this area, starting with program transformations and moving through machine learning to find embedding vectors for method bodies. So we leave further discussion of this problem to Section 3.

3 Paper overviews

This section summarizes each paper and its relevance to the field given the background covered in Section 2. The inter-relationship between these papers will be discussed in Section 4 and Section 5. Semantic Scholar is a web site that tracks citation counts on scholarly publications. Figure 5 uses data from Semantic Scholar to illustrate how the five base papers have impacted the research community. In order to demonstrate the interconnection of the topics covered by the papers, Figure 5 also shows a small sampling of papers cited by more than one of the base papers.

3.1 Staged Program Repair with Condition Synthesis [16]

This paper introduces staged program repair (SPR), which creates a search space of potential bug fixes in C code. The repair attempts are based on pre-defined parameterized transformation schemas that combine with multiple ways to synthesize changes to conditional statements. The approach to evaluating the schemas allows for a relatively efficient pruning of the search space that improved performance over prior work. There are 6 specific schemas discussed in the paper, which can be useful as a baseline for evaluating machine learning techniques for program repair and synthesis work to see which types of repairs can be found. This paper from 2015 is particularly interesting because it discusses the algorithmic implementation of many program ideas that are today starting to be explored using machine learning techniques.

Before exploring repair schemas, SPR does error localization by finding blocks of code that are often executed for failing test cases but rarely executed for passing test cases. Given the suspected faulty code blocks, SPR will then search for a successful repair by performing code transformations and rerunning the passing and failing tests. The order SPR searches the repair space is a key contribution of this paper as it decreases the search time. For example, the first schema evaluated is to change only a branch condition (*e.g.,* tighten and loosen a condition). Further details are in the paper, but a summary of the 6 schemas that SPR uses to explore code transformations are:

- **Condition Refinement:** Given a target *if* statement, SPR transforms the condition of the *if* statement by conjoining or disjoining an abstract condition to the original *if* condition.
- **Condition Introduction:** Given a target statement, SPR transforms the program so that the statement executes only if an abstract condition is true.
- **Conditional Control Flow Introduction:** SPR inserts a new control flow statement (return, break, or goto an existing label) that executes only if an abstract condition is true.
- **Insert Initialization:** For each identified statement, SPR generates repairs that insert a memory initialization statement before the identified statement.
- **Value Replacement:** For each identified statement, SPR generates repairs that replace either 1) one variable with another, 2) an invoked function with another, or 3) a constant with another constant.
- **Copy and Replace:** For each identified statement, SPR generates repairs that copy an existing statement to the program point before the identified statement and then apply a Value Replacement transformation.

Three of the six transformations involve adding an abstract condition. An abstract condition *abstract_cond()* can be added to an existing *if* statement by adding '*&& abstract_cond()*' or '|| *abstract_cond()*' to the statement. The condition to add is generated by creating traces of passing and failing tests that track values of different variables for each case and a new condition is searched for that causes the failing cases to pass.

Listing 1 is an example of a one-line patch found by SPR. The failure results because the body of the *if* statement did not execute when it should have. An *abstract_cond()* was added to the condition, then concretized to produce the correct fix.

```
− if( isostr_len ) {
+ if( isostr_len  ||  ( isostr  != 0)) {
```

Listing 1: Patch that uses **Condition Refinement** schema to correct *if* statement

A strength of this paper is its discussion on the balance between plausible (passing all tests) and correct patches. Out of 38 plausible patches, 11 are correct, which is a 29% correct/plausible ratio. Other papers have cited lower ratios for correct/plausible; a careful analysis of prior techniques (GenProg, RSRepair, and AE) shows that they have correct/plausible ratios of less than 12% [19]. Given that the authors compare directly against GenProg and AE, their relatively high ratio implies that the repair schemas they use represent reasonable transformations.

Ultimately, the authors compare their results to two other well-cited repair programs (GenProg and AE). On the same benchmark set, SPR was able to fix five times as many defects as the prior art, representing a significant advance.

A weakness of this paper is that it cannot effectively apply more than one tranformation to attempt a patch. As Section 4.1 will show, this is an area that can be addressed by machine learning for patch generation.

3.2 PHOG: Probabilistic Model for Code [7]

This paper creates a statistical model of code that can be used for code generation (including code completion, patch generation, and programming language translation). The model is based on their domain-specific TCOND language which allows for grammar production rules to be context dependent. The probabilities for each contextualized production rule are computed from the training data and evaluated based on how well AST nodes in the test data could be predicted.

The paper builds up ideas based on context-free grammars (CFGs) [4], which include production rules that define how non-terminals can be transformed. Some examples to show the format are:

- $\langle expr \rangle \rightarrow number$
- $\langle expr \rangle \rightarrow \langle expr \rangle + \langle expr \rangle$
- $\langle expr \rangle \rightarrow \langle expr \rangle - \langle expr \rangle$

Model	Error Rate
Non-Terminals	
PCFG	48.5%
3-Gram	30.8%
10-Gram	35.6%
PHOG	25.9%
Terminals	
PCFG	49.9%
3-Gram	28.7%
10-Gram	29.0%
PHOG	18.5%

Table 1: Evaluation of prediction for AST nodes in JavaScript

After discussing CFGs, the concept of a high order grammar (HOG) is introduced, which allows for production rules to be based on contexts such that $\alpha[\gamma] \rightarrow \beta$ represents a rule transforming the non-terminal α with context γ into β, where β can be a terminal or non-terminal of the grammar. A context is created by analyzing the program up to the production rule use point and might include statement types or local variable names. For example: $\langle expr \rangle [return] \rightarrow True$ could represent the rule that $\langle expr \rangle$ expands to $True$ when the expression is in a return statement.

The paper defines a PHOG as: a probabilistic high order grammar is a tuple (G, q) where G is a HOG (high order grammar) and $q : R \rightarrow \mathbb{R}^+$ scores rules such that they form a probability distribution. G includes a set of non-terminals N and a conditioning set C (*i.e.,* the contexts). The probability distribution is computed as:

$$\forall \alpha \in N, \gamma \in C : \sum_{\beta:\alpha[\gamma]\rightarrow\beta\in R} q(\alpha[\gamma] \rightarrow \beta) = 1$$

The function q is learned by counting rule expansions observed in a set of training data, and the authors use smoothing techniques to address sparseness in the training data. This is a straightforward technique to learn probabilities. Machine learning approaches, which can learn similar distributional information, are not as easily understood an the technique used for PHOG.

To evaluate PHOG, the authors predict JavaScript elements using PHOG and 3 alternate techniques. The prediction test is done by deleting a node from an AST (and its subtree and all nodes to the right) and querying the model to identify the missing node. Their alternate techniques are a PCFG (probabilistic context-free grammar) that only conditions on the parent non-terminal (no context used), and a 3-gram and 10-gram model (an n-gram model conditions on the n-1 previous symbols in the AST traversal as used in Allamanis *et al.* [2]). Table 1 shows their results on predicting both terminal and non-terminal elements. Their results are very strong in this analysis relative to previous techniques; Section 4.1 will contrast these results with a recent machine learning technique for patch generation.

3.3 Get to the Point: Summarization with Pointer-Generator Networks [22]

This paper introduces a new approach to copying information from an input sequence to an output sequence when using a sequence-to-sequence model for natural language processing. A pointer-generator in a neural network model allows the model to 'point' to a specific token on the input sequence that should be copied to the output sequence. In early versions of sequence-to-sequence learning, only tokens that were learned as part of the training vocabulary were available for producing the output [23]. The paper details how a pointer-generator network is useful for text summarization by allowing accurate use in the output of out-of-vocabulary words such as person or place names. In particular, the field of automatically summarizing natural language processing includes both *extractive* approaches where certain key sentences and phrases are copied in full from the source, and *abstractive* approaches which can involve rewording ideas in the input and sound more natural to most readers. The paper notes that abstractive approaches benefit from their copy mechanism, which can use token embedding and encoding information to point to specific tokens that aren't in the language vocabulary but are in the input sequence and should be used at specific points in the output sequence. In addition to improving natural language models, copying tokens from the input directly to the output also has key advantages for program repair. A known challenge to using sequence-to-sequence models for program repair is the issue that the full vocabulary for source code is nearly infinite due to specific identifier names, numbers, strings, *etc.* [13]. Similar to the summarization problem, in program repair new tokens from the computer language vocabulary may be needed for a bug fix, and copying rare tokens can be used to solve the unlimited vocabulary problem.

This paper did not introduce the copy mechanism for NLP, but it has been received by the NLP community as a base on which to build further work. For example, this paper is the basis for the copy mechanism implementation in OpenNMT, a popular open framework for neural machine translation. The main difference between this work and previous work is explicitly computing p_{gen}, the probability for copying a token from the input versus using a token from the vocabulary. The next paragraphs will briefly build up the model to show how p_{gen} is computed and used.

Encoder The encoder is a recurrent neural network using LSTM gates [14] to process the input. It is a bidirectional encoder [21] that allows the encoding for a token to incorporate information from tokens both before and after its occurrence in the input data. The encoder converts the source sequence $X = [x_1, ..., x_n]$ into a sequence of encoder hidden states h_i using a learnable recurrence function f_e. After reading the last token, the last hidden state, h_n^e is used as the context vector c for initializing the decoder [11]:

$$h_i^e = f_e(x_i, h_{i-1}^e); \tag{1}$$

Decoder The decoder is also a recurrent neural network using LSTM gates. When initialized by the encoder, it begins production of the output sequence by receiving the special *start* token as input y_0. For each previous output token y_{j-1}, the decoder updates its hidden state h_j^d using the learnable recurrence function f_d [11]:

$$h_j^d = f_d(y_{j-1}, h_{j-1}^d, c) \tag{2}$$

The decoder states h_j^d are used for token generation by the attention and copy mechanisms in Equation 4 and Equation 5. The model stops updating decoder hidden states and generating new tokens when the last token generated by the model is a special end-of-sequence token.

Attention The attention mechanism provides a way to create a context vector c_j for each decoder output token y_j using a linear combination of the hidden encoder states h_i^e [5]:

$$c_j = \sum_{i=1}^{n} \alpha_i^j h_i^e \tag{3}$$

Where α_i^j represents learnable attention weights. This context vector c_j is used by a learnable function f_a to allow each output token y_j to pay "*attention*" to different encoder hidden states when predicting a token from the vocabulary V:

$$P_V(y_j \mid y_{j-1}, y_{j-2}, ..., y_0, c_j) = f_a(h_j^d, y_{j-1}, c_j) \tag{4}$$

Copy The copy mechanism further adjusts Equation 4 to produce a token candidate by introducing p_{gen}, the probability that the decoder generates a token from its initial vocabulary. Hence, $1-p_{gen}$ is the probability to copy a token from input tokens depending on the attention vector α^j in Equation 3 [22]:

$$p_{gen} = f_c(h_j^d, y_{t-1}, c_j) \tag{5}$$

$$P(y_j) = p_{gen} P_V(y_j) + (1 - p_{gen}) \sum_{i:x_i=y_j} a_i^j \tag{6}$$

f_c in Equation 5 is learnable function. Using Equation 6, the output token y_j for the current decoder state is selected from the set of all tokens that are either: 1. tokens in the training vocabulary (including the <unk> token) or 2. tokens in the input sequence.

In addition to the copy mechanism, this paper introduces a technique for limiting repetition of output sequences. This technique is important for the text summarization use case they envision, but is less relevant for source code. Reusing the same variable multiple times in a line may sometimes be appropriate (i.e. "if (x < 8) && (x > 2)").

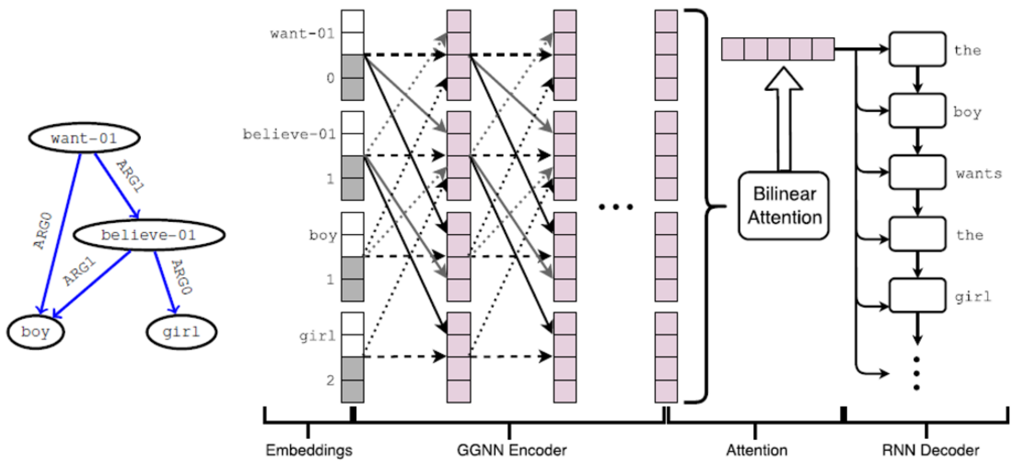


Figure 6: Figure from Graph-to-Sequence paper showing encoder and attention mechanism feeding into sequence generation

3.4 Graph-to-Sequence Learning using Gated Graph Neural Networks [6]

Sequence-to-sequence models for code generation have been augmented to include some AST information [8], but the rich information available statically during program analysis would benefit from a graph neural network. This paper, like the Pointer paper, is discussing natural language processing, and the concept introduced invites use for code analysis. The paper uses a gated graph neural network to analyze an input string after the string is transformed into an Abstract Meaning Representation (AMR) graph. To minimize the number parameters in the graph neural network model due to edge types, they discuss transforming the edges in the graph to be extra vertices through a Levi graph transformation. For code analysis, a Levi transformation could be valuable - it allows for a wider variety of edge types to be represented with a reasonable number of parameters.

Figure 6 shows an overview of their approach. In a sequence-to-sequence model with attention, as described in Section 3.3, the attention is computed using the encoder hidden states for each received token. The encoder state includes information from the embedding of the input token, as well as a function of the encoder states before and after this token. In a graph neural network, the attention is computed using the node hidden states after the network has iterated on the structure. Since the nodes are initialized with the token inputs, the node hidden states can include information from an embedding of the input, as well as any other nodes that can be reached during network iteration. An aggregation of the final node states can be used to initialize the hidden state of the sequence decoder.

The transformation of the input AMR graph into a Levi graph allowed for fewer edge types in the model. The paper shows that for optimal performance, the Levi network had these types of edges: self edges, forward and reverse token sequence edges, and forward and reverse tree connection edges. This results in five edge types, each of which has a weight matrix to compute how the gated recurrent unites (GRUs) update each iteration step.

As is common for reporting machine translation results, the authors evaluate results using BLEU scores (BiLingual Evaluation Understudy). The BLEU score was proposed by Kishore Papineni *et al.* in their 2002 paper “BLEU: a Method for Automatic Evaluation of Machine Translation“ [18] and scores candidate translations based on a reference translation. Their test set contains both english-to-german and english-to-czeck translation tasks. In their results section they show the highest BLEU scores for the models tested, including the same test set tested with recent state-of-the-art approaches. Interestingly, they also include results for the ChrF++ scoring method, for which their approach does not score the best. They note that ChrF++ scores have been found to align better with human translation scoring than BLEU and leave the challenge of improving ChrF++ scores to future work. The merits of BLUE versus ChrF++ methods for language output scoring are not directly related to machine learning for code sequence generation. The scores can help alert code generation researchers to new improvements in sequence generators, but the methods tend to relate best to human languages.

3.5 CODE2VEC: LEARNING DISTRIBUTED REPRESENTATIONS OF CODE [3]

This paper describes an approach to create an embedding vector for entire Java methods in a way similar to the widely successful word2vec approach used in NLP [17]. The example use case is to predict method names, but the paper aims to produce an embedding that can be used for a variety of cases. Indeed, this approach can be directly applied to the problem of program equivalence. The paper presents cases where similar methods have similar embeddings and 'adding' the embedding from one program can have meaningful results in the method name predicted. In this way, a possible future application for code2vec would be test two programs for equivalence by subtracting their embeddings and the resulting vector could be used to seed a sequence decoder and create a description of the program differences.

The paper builds up d-dimensional embedding for a piece of code (in their example use case they are looking at methods). The embedding is built up using weighted summations of embeddings for path-contexts. A path-context is a sample from the AST for the code that includes a start terminal, the non-terminals from the AST, and the end terminal. In theory, a snippet of code with n terminals in the AST has n^2 path-contexts, but the authors set AST distance constraints on the path-contexts allowed and also have found that an upper bound of 200 path-contexts is sufficient to represent most code correctly. Like the popular word2vec, embeddings are learned during model training for the terminals and AST paths. The embeddings for the set of all terminal symbols are collected into $value\_vocab$, and the embeddings for all of the paths seen in the training set are collected into $path\_vocab$. Hence, given a path-context that starts at terminal x_s and terminates at x_t following a path p_j through the AST, the path-context is mathematically represented as:

$$c_i = embedding(\langle x_s, p_j, x_t \rangle) = [value\_vocab_s; path\_vocab_j; value\_vocab_t] \in \mathbb{R}^{3d}$$

Figure 7 shows an example of path-contexts. For example, the path-context for the red path labeled ① in the figure is the embedding for:

$\langle$elements,(Name $\uparrow$ FieldAccess $\uparrow$ Foreach $\downarrow$ Block $\downarrow$ IfStmt $\downarrow$ Block $\downarrow$ Return $\downarrow$ BooleanExpr),true$\rangle$

Using the path-context c_i and a trainable weight matrix $W \in \mathbb{R}^{d \times 3d}$, the *combined context vector* $\tilde{c_i}$ is:

$$\tilde{c_i} = tanh(W \cdot c_i)$$

A trainable global attention vector $a \in \mathbb{R}^d$ is used in a softmax function to compute attention weights α_i based on the path context embeddings:

$$\alpha_i = \frac{\exp(\tilde{c_i}^T \cdot a)}{\sum_{j=1}^{n} \exp(\tilde{c_j}^T \cdot a)}$$

Ultimately, the whole code snippet is represented by an aggreggated code vector $v \in \mathbb{R}^d$:

$$v = \sum_{i=1}^{n} \alpha_i \tilde{c_i}$$

A strength of this paper is that it teaches a way to visualize the parts of the program that are being used to determine the method name, which is a helpful way to improve understanding of the neural network. Since understanding what a network has learned is a known problem in machine learning, this attention visualization is valuable. Figure 7 shows the various weights of the top 4 paths used in computing v for the snippet based on the thickness of lines. Figure 8 diagrams the full use model of this paper. The original code, with '?' for the method name is shown, along with the top 4 weighted paths used to predict the method name 'count'.

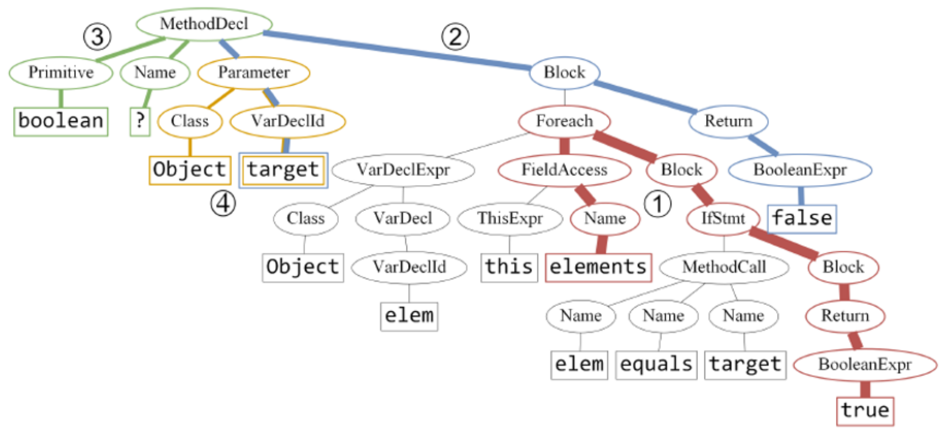
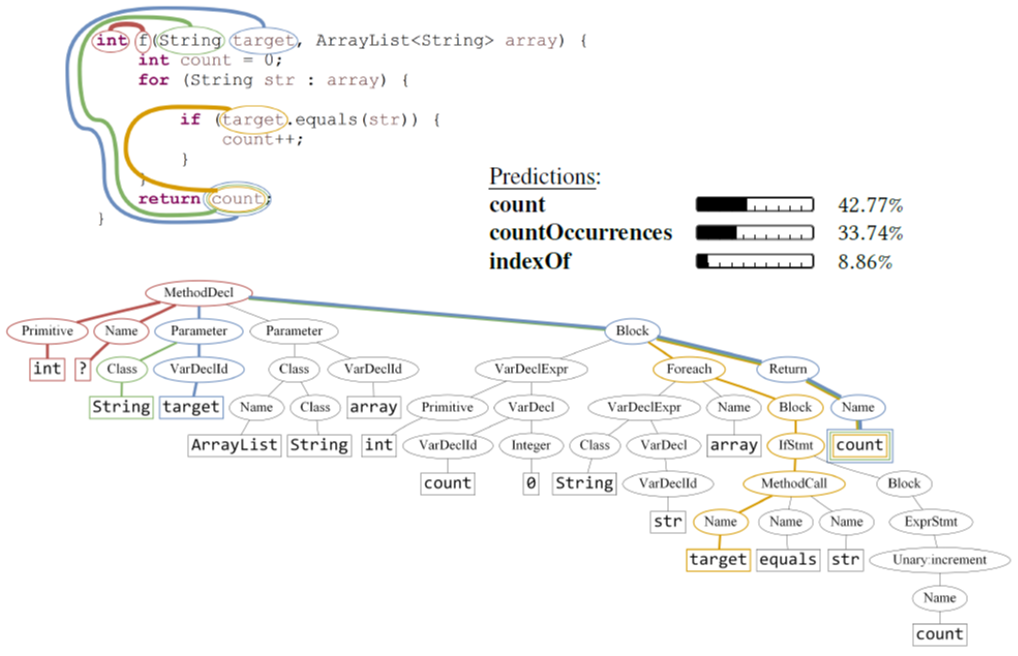


Fig. 3. The top-4 attended paths of Figure 2a, as were learned by the model, shown on the AST of the same snippet. The width of each colored path is proportional to the attention it was given (red ①: **0.23**, blue ②: **0.14**, green ③: **0.09**, orange ④: **0.07**).

Figure 7: Figure from code2vec paper showing different paths through AST



```
int f(String target, ArrayList<String> array) {
    int count = 0;
    for (String str : array) {
        if (target.equals(str)) {
            count++;
        }
    }
    return count;
}
```

Predictions:

count		42.77%
countOccurrences		33.74%
indexOf		8.86%

Fig. 7. An example for a method name prediction, portrayed on the AST. The top-four path-contexts were given a similar attention, which is higher than the rest of the path-contexts.

Figure 8: Figure from code2vec paper showing full use model from code snippet to method name prediction

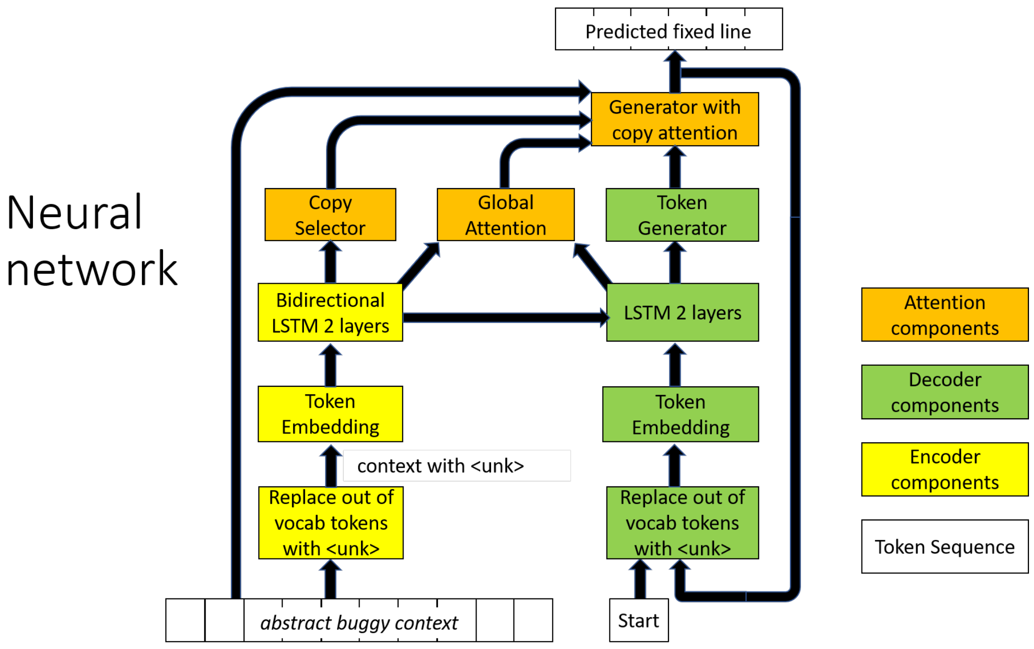


Figure 9: Figure from Graph-to-Sequence paper showing encoder and attention mechanism feeding into sequence generation

A weakness of the code2vec paper is that it learns on full tokens (not subtokens) and has a vocabulary limitation. It cannot learn embeddings for novel variable names. Also, the embeddings it does learn are based on the training set provided, hence the embedding may not be appropriate for how a variable is used in a given method. For example, a method that iterates over an array using i and accumulates the sum of array elements into sum is semantically equivalent to a method that uses j for the iterator and i for the accumulation. But the somewhat odd use of i for accumulation would result in a shifted code vector for the method.

4 Recent work in program repair and equivalence

4.1 SequenceR: Sequence-to-sequence learning for end-to-end program repair[9]

I shared first author credit on this paper, which describes a system to create patches for buggy programs based on the sequence-to-sequence model with copy mechanism from the Pointer paper discussed in section Section 3.3. SequenceR relies on a fault localization step (similar to the approach discussed in the SPR paper) to prepare input to the model. For use as a code repair model, we first tokenize an input program and remove the body of methods that are not identified as containing the bug needing repair. As shown in Figure 9, SequenceR receives as input this *abstract buggy class* and generates as output a patch proposal. The model uses a beam search to track the 50 most likely patches and relies on passing a test suite to predict plausible patches.

One of the research questions in the SequenceR paper relates to which types of repairs SequenceR can learn. While the answer to the question was not trying to align with the transformation schemas from the SPR paper, there is some overlap. Of the 6 schemas used in SPR, examples are shown in the SequenceR paper for 'Condition Refinement' and 'Value Replacement'. The 4 other schemas from SPR involve adding new lines of code, which is not supported by the one-line fix model for SequenceR. Unlike SPR, SequenceR can perform multiple transformations on a single line, and it can learn transformations that are not listed in SPR (such as changing a field access into a method call). The plausible/correct ratio for SequenceR on the tested dataset was 30%, which is slightly higher than SPR achieved and indicates that SequenceR was learning reasonable code transformations without hand-crafted transformation rules.

In relation to PHOG, SequenceR is using machine learning in a way that can approximate that work. The token generation and copy mechanism are being trained to predict a conditional probability about what the correct next token is given the *abstract buggy context* and the tokens created so far. Figures in the SequenceR paper show validation accuracies of 88%, which is an error rate of 12% and is lower than the PHOG prediction rates in Table 1.

Another research question covered by the SequenceR paper relates to the effectiveness of the copy mechanism for overcoming the unlimited vocabulary problem. Even more so than natural language (which can have specific names for people and places), code often has rare symbol names for identifiers. Since the Pointer paper was the basis of the copy mechanism implemented in OpenNMT, SequenceR made use of this technique. The SequenceR paper found that using the copy mechanism along with a with a 1,000 token vocabulary was 4 times more accurate than a 50,000 word vocabulary without the copy mechanism. The copy mechanism is very useful in processing rare tokens. Effectively, the copy mechanism is 'reasoning' about the out-of-vocab tokens with reference to the encoder hidden state related to the token, which is using the context of the token to inform the patch generation step. This information is more valuable than the embedding that is produced for rarely seen token names.

SequenceR uses a sequence-to-sequence model with copy mechanism. Section 5 discusses how the graph-to-seq paper can be leveraged to replace the bidirectional encoder used in SequenceR with a graph neural network to more completely analyze the available code and test coverage information.

In relation to code2vec, the SequenceR paper is producing a form of buggy code embedding with the connection between the encoder and decoder LSTM blocks shown in Figure 9. This connection is indicating the standard encoder/decoder structure where the encoder state is used to initialize the decoder state. This state is 256x2=512 dimensions in SequenceR; which corresponds to the 128 dimensions that code2vec creates for summarizing a code snippet. We tested SequenceR with and without an extra learnable layer between the encoder and decoder and found that this 'bridge' layer was useful; indicating using a learnable function between the components of the encoder and decoder improved performance. This learnable function is similar to the global attention vector used in the code2vec paper, but it is not as easy to comprehend the mapping. The attention used in SequenceR is somewhat interesting to visualize - as tokens are generated one can visualize the attention given to tokens from the input.

4.2 Neural Network-based Graph Embedding for Cross-Platform Binary Code Similarity Detection [25]

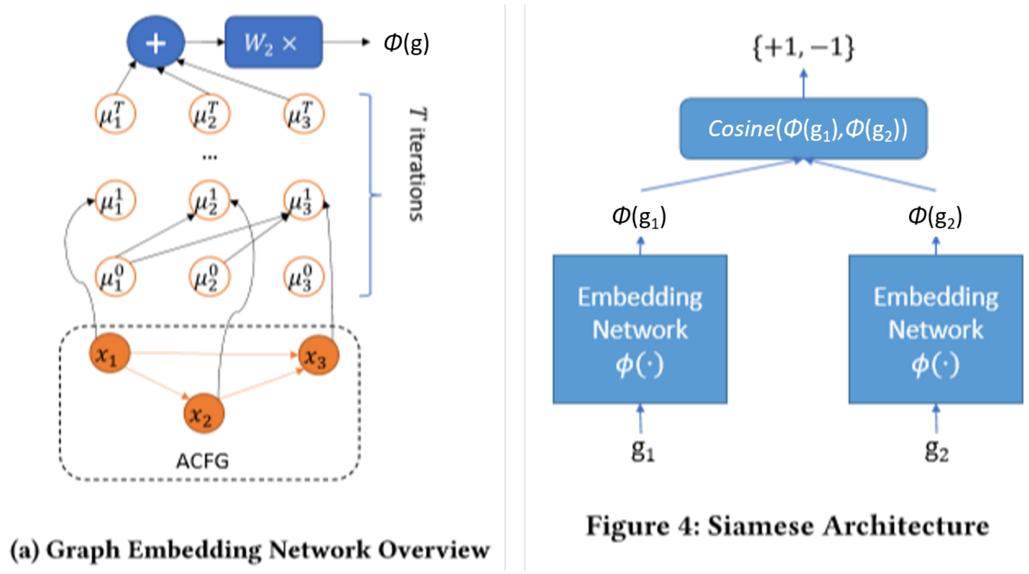
This is an interesting paper using graph neural networks to detect binary code similarity. Like the code2vec paper (which is more recent), this paper learns and embedding function on code and the use model is to compare embedding functions to detect code similarity for vulnerability detection. They refer to their approach as Structure2vec.

The Structure2vec approach starts with an ACFG (*attributed control flow graph*), in which each node represents a basic block and includes attributes such as 'number of calls', 'numeric constants', 'number of offspring', and five other easily computed attributes. After initializing and iterating the graph neural network for T iterations, the embedding for the graph $\phi(g)$ is computed using a learned matrix W to combine the final hidden state $\mu_v^{(T)}$ of each vertex:

$$\phi(g) = W \cdot \sum_{v \in V} \mu_v^{(T)}$$

Figure 10 diagrams the approach for detecting code similarity. Subfigure (a) shows how after T iterations of the graph neural network, $\phi(g)$ is created. The Siamese architecture shows how the difference between two code embeddings is computed using the cosine of two multidimensional vectors.

A disadvantage of this approach is that the embedding itself cannot have any component that distinguishes similar but non-equal components. If the cosine function were replaced by a 2-layer neural network, then mappings from the program embedding to the equivalence value could account for disjoint but similar areas in the embedding spaces.



$W_2 \times$
$\Phi(g)$
μ_1^T μ_2^T μ_3^T
...
μ_1^1 μ_2^1 μ_3^1
μ_1^0 μ_2^0 μ_3^0
T iterations
x_1 x_2 x_3
ACFG
(a) Graph Embedding Network Overview
{+1, −1}
Cosine(Φ(g1),Φ(g2))
Φ(g1)
Φ(g2)
Embedding Network φ(·)
Embedding Network φ(·)
g1
g2
Figure 4: Siamese Architecture

Figure 10: Figures from Code Similarity paper showing generation and use of $\phi(g)$

5 Future Research

This collection of five base papers can be used to construct a new approach to creating a code embedding that could be used for both program repair as well as program equivalence checking. The approach would leverage static analysis of the code to create an annotated AST graph that can be the input for a graph neural network. In the equivalence embodiment of the approach, two ASTs could be processed and the output could be a binary equivalence output value. In the repair embodiment of the approach, a copy mechanism could be added to the approach in the graph2seq paper to allow a repair to be produced in a manner similar to the SequenceR paper.

Figure 11 adds line numbers and new edges to the AST graph from Figure 4. As an implementation for program repair, the idea here is that the new line numbers can be used to include coverage data from a fault localization run and can also be used by a copy mechanism during program repair. The figure does not include all of the edge types that would likely be beneficial. Given that we have a variety of node types in a graph such as this, here are the initial values that nodes could include:

- **Non-terminal type:** This would identify the node as, for example, an 'assign' or 'while' node in the AST. There might be 30-40 one-hot dimensions in the initial vector.
- **Terminal:** This would identify the value of the terminal as another set of one-hot dimensions. Like the SequenceR paper, this may include a few hundred common terminals, and an <unk> terminal for out-of-vocab terminals (but the copy mechanism would allow the actual input value to be copied for use in an output sequence).
- **Line:** This would be a <line_unk> token and take a single one-hot dimension to identify. However, these nodes would include two dimensions that contain test coverage information: normalized coverage for the line on both failing and passing test cases.
- **Compare:** For program equivalence, the top nodes for the two programs to compare could be connected to this node, which would provide a mechanism for the GNN to compute the similarity between the two graphs, and a potential conduit for information needed to evaluate equivalence.

The number of node types and other information will constrain the initial label size for nodes. The number of edge types is a different matter. In a typical graph neural network, each edge type implies another trainable weight matrix, which is why the graph2seq paper used a Levi graph to transform edges into nodes for the GNN to process. The list of edge types we would want in an AST graph augmented for this problem includes the edge types shown in Figure 11 and some data/control edges:

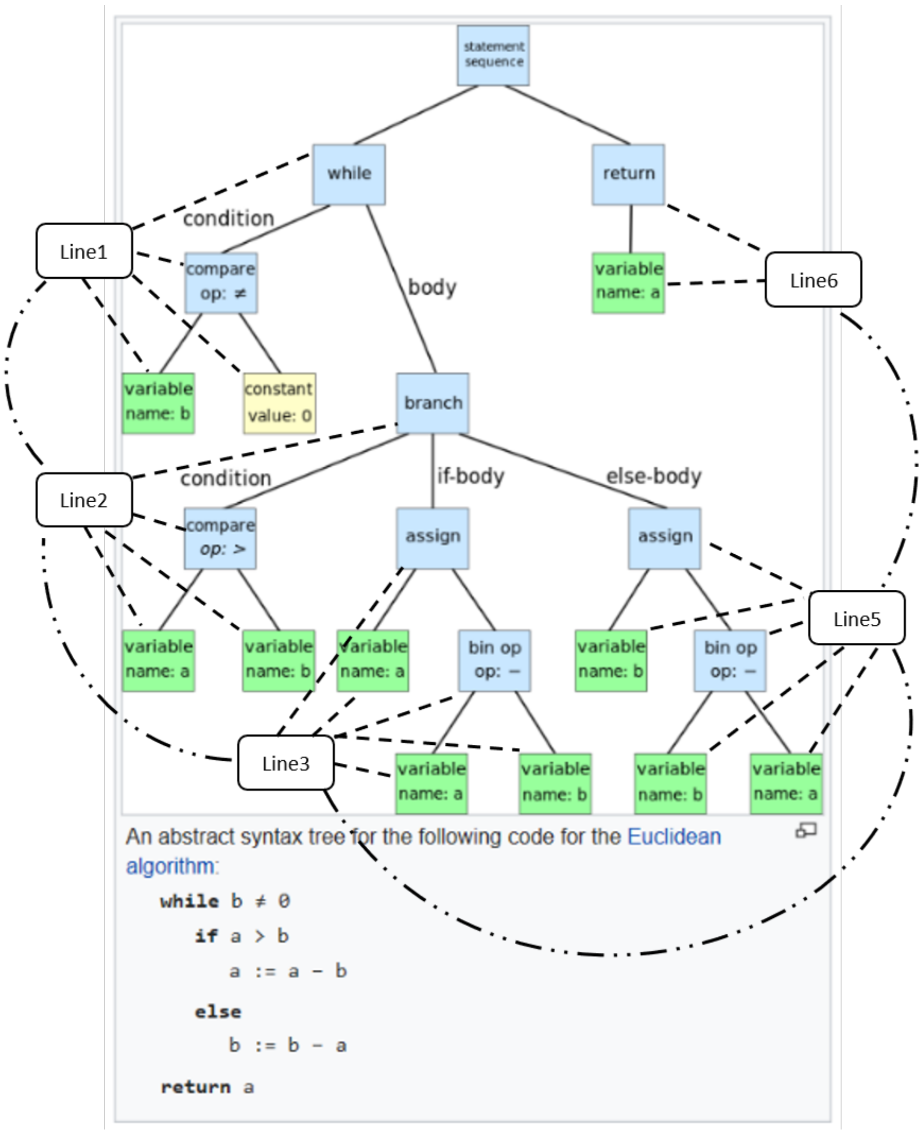
- **self:** an edge pointing to the current node; implies a weight matrix to include a node's current state in next state computations
- **condition:** an edge for a loop or if condition
- **body:** an edge for a loop body
- **operand1:** edge points to 1st operand node
- **operand2:** edge points to 2nd operand node
- **other AST edge types:** some edge types could be collapsed, experiments may help choose optimal set (i.e., 'body' could also be an edge to 'if-body').
- **last write:** an edge from a terminal node that refers to a variable to the node where the variable was last written in the AST.
- **method call:** an edge from a method call to the top non-terminal node representing the method.
- **type:** an edge from a variable node to its type declaration node in the AST.
- **other data and control edge types:** experiments may justify more static analysis edge types
- **line contains:** an edge from a line node to a node which is part of that line of code (dashed lines in Figure 11).
- **next line:** an edge from a line node to the next line node for the program (dash and 2 dots lines in Figure 11).
- **Reverse edges:** there should be different weight matrices for moving information back and forth along an edge, so all of the edges above except the self edge would have a reverse edge.

One approach to node setup for code analysis is to have the label for each node, as discussed in Section 2.3, include a one-hot vector for node type. But this forces at least the 'self' weight matrix to learn an optimal embedding for each token. It may be preferable to add a single layer for embedding between the one-hot token an the node label, which would learn a vocabulary embedding similar to the code2vec paper.

Use model for code repair: For code repair, the graph neural network is using a learnable function to create annotations for each node. By combining the ideas from the graph2seq and Pointer papers, the node annotations can be used as inputs to a copy mechanism, which allows the potential strength of a graph network analysis to combine with the benefit of a copy mechanism for patch generation. The SequenceR paper relied on a fault localization process to identify possible locations for one-line code repairs. For this new approach, the line nodes can include code coverage data and the network can be expected to identify the line (by copying the line token from the GNN input) before creating output tokens for the repair. When a line should be deleted, no tokens would be output after the line token that identifies the line to delete. Multiple lines could be fixed by allowing an output sequence to multiple line tokens, each of which is followed by the tokens needed by the line. For Java, newline characters are not part of the syntax, so patches that add a line could be done by identifying the location with a line token and having an output sequence generated with new tokens before or after the existing tokens. Also, a weakness of the SequenceR approach is that it could not learn to make fixes outside of a method body (as fault localization is limited to such areas). An AST analysis with line coverage data might be usable to predict when, for example, a package name should be changed as a proposed patch.

Using this network, it could be possible to cover all 6 of the transformation schemas identified in the SPR paper. A study of the attention given to AST nodes during the production of the patch could reference the PHOG paper as one would expect the GNN to learn a statistical model similar to but more flexible than the model learned in PHOG.

Use model for program equivalence: As noted in the node descriptions, on approach to using the GNN for program equivalence is to have a top-level **Compare** node that can aggregate a vector for use in identifying program equivalence. In this case, the ASTs for each program are input into the network along with the **Compare** node connecting them. In addition to such a node, using graph attention in the analysis of the two ASTs could help to identify which nodes are most important for the determination of equivalence or non-equivalence.



An abstract syntax tree for the following code for the Euclidean algorithm:

```
while b ≠ 0
  if a > b
    a := a − b
  else
    b := b − a
return a
```

Figure 11: AST with line number tokens and dashed edges identifying the AST elements in a line and and edges with dashes and dots connecting lines in sequence.

While a standard GNN output value could be used for training a program equivalence classifier, one could also train a graph2seq model that could output transformation rules that are involved in transforming one program to another (i.e., $(a+b)*c \rightarrow a*c+b*c$). Such a sequence generator could provide a path for proving the two models equivalent, instead of relying on the statistical output of an equivalence classifier.

6 Conclusion

Deep learning has been expanding into diverse fields as hardware advances and novel algorithms allow approaches to succeed in new areas. For the problems of program repair and program equivalence, machine learning techniques that have roots in natural language processing have been successfully applied to computer languages. Historic techniques for searching for program repairs using transformation schemas can be mimicked and improved on by training a network to 'translate' buggy code to fixed code, resulting in similar transformations being learned. As machine learning uses probabilistic sequence generation, it is capable of mimicking probabilistic models of language grammars.

New techniques allowing machine learning models to process an AST for code show promising paths forward for future research. As the field continues to evolve, code generation and analysis using machine learning will likely separate from natural language approaches and develop a robust approach based on the specific problems found in that problem space.

References

[1] Miltiadis Allamanis et al. "A Survey of Machine Learning for Big Code and Naturalness". In: *arXiv e-prints*, arXiv:1709.06182 (Sept. 2017), arXiv:1709.06182. arXiv: `1709.06182 [cs.SE]`.

[2] Miltos Allamanis et al. "Bimodal Modelling of Source Code and Natural Language". In: *Proceedings of the 32nd International Conference on Machine Learning*. Ed. by Francis Bach and David Blei. Vol. 37. Proceedings of Machine Learning Research. Lille, France: PMLR, July 2015, pp. 2123–2132.

[3] Uri Alon et al. "Code2Vec: Learning Distributed Representations of Code". In: *Proc. ACM Program. Lang.* 3.POPL (Jan. 2019), 40:1–40:29. ISSN: 2475-1421.

[4] Andrew W. Appel and Jens Palsberg. *Modern Compiler Implementation in Java*. 2nd. New York, NY, USA: Cambridge University Press, 2003. ISBN: 052182060X.

[5] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. "Neural machine translation by jointly learning to align and translate". In: *arXiv preprint arXiv:1409.0473* (2014).

[6] Daniel Beck, Gholamreza Haffari, and Trevor Cohn. "Graph-to-Sequence Learning using Gated Graph Neural Networks". In: *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Melbourne, Australia: Association for Computational Linguistics, 2018, pp. 273–283.

[7] Pavol Bielik, Veselin Raychev, and Martin Vechev. "PHOG: Probabilistic Model for Code". In: *Proceedings of The 33rd International Conference on Machine Learning*. Ed. by Maria Florina Balcan and Kilian Q. Weinberger. Vol. 48. Proceedings of Machine Learning Research. New York, New York, USA: PMLR, June 2016, pp. 2933–2942.

[8] Saikat Chakraborty, Miltiadis Allamanis, and Baishakhi Ray. "Tree2Tree Neural Translation Model for Learning Source Code Changes". In: *arXiv* abs/1810.00314 (2018).

[9] Zimin Chen et al. "SequenceR: Sequence-to-Sequence Learning for End-to-End Program Repair". In: *CoRR* abs/1901.01808 (2019).

[10] Kyunghyun Cho et al. "Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation". In: *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Doha, Qatar: Association for Computational Linguistics, 2014, pp. 1724–1734.

[11] Kyunghyun Cho et al. "Learning phrase representations using RNN encoder-decoder for statistical machine translation". In: *arXiv preprint arXiv:1406.1078* (2014).

[12] David Denison. “Parts of speech: Solid citizens or slippery customers?” In: *Journal of the British Academy*. The British Academy, 2013, pp. 151–185.

[13] Vincent J Hellendoorn and Premkumar Devanbu. “Are deep neural networks the best choice for modeling source code?” In: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM. 2017, pp. 763–773.

[14] Sepp Hochreiter and Jürgen Schmidhuber. “Long short-term memory”. In: *Neural computation* 9.8 (1997), pp. 1735–1780.

[15] Iu I. Ianov. “On the Equivalence and Transformation of Program Schemes”. In: *Commun. ACM* 1.10 (Oct. 1958), pp. 8–12. ISSN: 0001-0782.

[16] Fan Long and Martin Rinard. “Staged Program Repair with Condition Synthesis”. In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 2015. Bergamo, Italy: ACM, 2015, pp. 166–178. ISBN: 978-1-4503-3675-8.

[17] Tomas Mikolov et al. “Efficient Estimation of Word Representations in Vector Space”. In: *arXiv e-prints*, arXiv:1301.3781 (Jan. 2013), arXiv:1301.3781. arXiv: `1301.3781 [cs.CL]`.

[18] Kishore Papineni et al. “BLEU: A Method for Automatic Evaluation of Machine Translation”. In: *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*. ACL ’02. Philadelphia, Pennsylvania: Association for Computational Linguistics, 2002, pp. 311–318.

[19] Zichao Qi et al. “An Analysis of Patch Plausibility and Correctness for Generate-and-validate Patch Generation Systems”. In: *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. ISSTA 2015. Baltimore, MD, USA: ACM, 2015, pp. 24–36. ISBN: 978-1-4503-3620-8.

[20] Franco Scarselli et al. “The Graph Neural Network Model”. In: *IEEE Transactions on Neural Networks* 20 (2009), pp. 61–80.

[21] Mike Schuster and Kuldip K Paliwal. “Bidirectional recurrent neural networks”. In: *IEEE Transactions on Signal Processing* 45.11 (1997), pp. 2673–2681.

[22] Abigail See, Peter J. Liu, and Christopher D. Manning. “Get To The Point: Summarization with Pointer-Generator Networks”. In: *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Vancouver, Canada: Association for Computational Linguistics, 2017, pp. 1073–1083.

[23] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. “Sequence to Sequence Learning with Neural Networks”. In: *CoRR* abs/1409.3215 (2014).

[24] Wikipedia contributors. *Abstract syntax tree — Wikipedia, The Free Encyclopedia*. [Online; accessed 3-March-2019]. 2019.

[25] Xiaojun Xu et al. “Neural Network-based Graph Embedding for Cross-Platform Binary Code Similarity Detection”. In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. CCS ’17. Dallas, Texas, USA: ACM, 2017, pp. 363–376. ISBN: 978-1-4503-4946-8.