Understanding Neural Code Intelligence Through Program Simplification

Md Rafiqul Islam Rabin
mrabin@uh.edu
University of Houston
Houston, TX, USA

Vincent J. Hellendoorn
vhellendoorn@cmu.edu
Carnegie Melon University
Pittsburgh, PA, USA

Mohammad Amin Alipour
maalipou@central.uh.edu
University of Houston
Houston, TX, USA

ABSTRACT

A wide range of code intelligence (CI) tools, powered by deep neural networks, have been developed recently to improve programming productivity and perform program analysis. To reliably use such tools, developers often need to reason about the behavior of the underlying models and the factors that affect them. This is especially challenging for tools backed by deep neural networks. Various methods have tried to reduce this opacity in the vein of "transparent/interpretable-AI". However, these approaches are often specific to a particular set of network architectures, even requiring access to the network's parameters. This makes them difficult to use for the average programmer, which hinders the reliable adoption of neural CI systems. In this paper, we propose a simple, model-agnostic approach to identify critical input features for models in CI systems, by drawing on software debugging research, specifically delta debugging. Our approach, SIVAND, uses simplification techniques that reduce the size of input programs of a CI model while preserving the predictions of the model. We show that this approach yields remarkably small outputs and is broadly applicable across many model architectures and problem domains. We find that the models in our experiments often rely heavily on just a few syntactic features in input programs. We believe that SIVAND's extracted features may help understand neural CI systems' predictions and more generally learned behavior. The code and data of our prediction-preserving simplification approach are available at https://github.com/mdrafiqulrabin/SIVAND.

KEYWORDS

Models of Code, Interpretable AI, Program Simplification

1 INTRODUCTION

Deep learning has increasingly found its use in state-of-the-art tools for code intelligence (CI), thriving in defect detection, code completion, type annotation prediction, and many more [13, 14, 37]. This growing body of work benefits from the remarkable generality of deep learning models: they are *universal function approximators* [17]; *i.e.*, they can represent any function, not just linear ones. In practice, deep neural network methods appear to learn rich representations of raw data through a series of transforming layers [22]. This substantially reduces the burden of feature engineering in complex domains, including vision, natural languages, and now software.

Given this capacity, learned models seem capable of discovering many non-trivial properties about the source code, even ones that are beyond the reach of traditional, sound static analyzers. As such, they can uncover bugs in code that is not syntactically invalid, but rather "unnatural" [28]. Although this may be reminiscent of a software developer's ability to intuit properties about programs, there is a sharp contrast in interpretability: developers can explain their deductions and formulate falsifiable hypotheses about the behavior of their code. Deep neural models offer no such capability. Rather, they remain stubbornly opaque "black boxes", even after years of research on interpreting their behavior [23].

This opacity is already a concern in non-critical applications, where the lack of explainability frustrates efforts to build useful tools. It is substantially more problematic in safety- and security-critical applications, where deep learners could play a key role in preventing defects and adversarial attacks that are hard to detect for traditional analyzers. At present, deep neural models can change their predictions based on seemingly insignificant changes, even semantic-preserving ones [24, 26, 27], and fail to provide any *traceability* between those predictions and the code.

In this paper, we propose a simple, yet effective methodology to better analyze the input (over)reliance of neural models in software engineering applications. Our approach is *model-agnostic*: rather than study the network itself, our approach relies on the input reductions using delta debugging [39]. The main insight is that by removing irrelevant parts to a prediction from the input programs, we may better understand important features in the model inference, consequently.

Given a prediction, correct or not, we show that model inputs can often be reduced tremendously, even by 90% or more, while returning the same prediction. Importantly, our work is the first to show that these reductions in no way require the inputs to remain natural, or, depending on the task, in any way valid. This allows us to generate significantly simpler explanations than related work [36]. Our results hold on four neural architectures across two

Original

```
@Override
public void onCreate(Bundle savedInstanceState){
    super.onCreate(savedInstanceState);
    setContentView(R.layout.fragmentdetails);
}
```

Minimized

```
d onCreate(u ve){s.onCreate();}
```

Figure 1: Example of an original and minimized method in which the target is to predict onCreate as the method name. The minimized example clearly shows that the model (i.e. Code2Vec) has learned to take short-cuts, in this case looking for the name in the function's body.

popular tasks for neural CI systems: code summarization, using Code2Vec and Code2Seq [4, 5], and variable misuse detection, with RNNs and Transformers [15]. We show that our minimal programs are related to, but not fully explained by the "attention" [35] in models that use it. Overall, our findings suggest that current models, when trained on a single task such as ours, have very little care for the readability of functions as a whole; rather, they overwhelmingly focus on very small, simple syntactic patterns that provide salient clues to the required output.

2 MOTIVATING EXAMPLE

We first present an example to illustrate our approach and the insights that it can provide about the prediction of neural code intelligence models. Figure 1 shows a code snippet from the METHOD-NAME dataset. Here, the goal is to infer the method name from the method body. The Code2Vec model predicts the method name onCreate, which is indeed correct. Unfortunately, it is hardly clear why; Code2Vec considers dozens to hundreds of "paths" in the methods' AST (abstract syntax tree) and struggles to identify which ones are most informative – indeed, in our results, we find its built-in "attention" based explanations to only poorly correlate with essential tokens.

Of course, a developer looking at this program could provide several explanations why this is intuitively correct. For one, the method invocation super.onCreate strongly suggests that this method overrides a method with the same name in the parent class, as such calls are usually made to the overridden method. This guess does not need to be informed by just that call; if the developer is familiar with Android development, they might recognize this code as working with Android APIs, and perhaps even know that the Activity[1] in Android systems provides onCreate method to initialize a user interface in which developers can place their UI objects on the interface window using the setContentView method.

Turning back to the Code2Vec model, it does not offer any of these explanations, but rather an output based on a complex mixture of its inputs and millions of parameters. This lack of transparency is problematic for practitioners and researchers alike, who are unlikely to accept recommendations without *evidence*, especially in cases that are not so trivial. For instance, does Code2Vec note and use the

---

[1]https://developer.android.com/reference/android/app/Activity

.onCreate invocation? If so, does it specifically leverage the inheritance relation? And/or, does it use any of the more Android-specific reasoning about the presence of the setContentView invocation?

Our approach, Sivand, can better provide compelling answers to these questions. The second half of Figure 1 shows the smallest possible version of this program that is still syntactically valid (a necessity for Code2Vec) and yields the same prediction. Any evidence of this method relating to Android development has been thoroughly scrubbed; all that remains is the single onCreate method call – even the mention of super has been minified to a single character 's'.

Table 1 shows some steps that yielded this reduced program in greater detail. Sivand works by iteratively reducing the size of the input program by the Code2Vec model while preserving the prediction output at every step. We continue this reduction until the program is either fully reduced (to its minimal components, depending on the task) or any further reduction would corrupt the prediction. Each row in this table shows the intermediate, most reduced program that still yields the *same* prediction as the original input program in the first row, as well as the probability ("score") of that prediction for reference. A close examination of the intermediate results suggests that the savedInstanceState parameter and the setContentView call, indicative of Android development, is reduced with almost no penalty to the score. Other such cues follow soon after. In step 10, the super call, which represents cues of inheritance, is similarly pruned, and here too with virtually no penalty to the score. The subsequent steps largely serve to remove a few miscellaneous characters; these compromise the prediction score somewhat, perhaps because the method is increasingly irregular. Regardless, as long as onCreate is present, the score remains high, ample to sustain the prediction. Evidently, even if the model noticed the Android-related features, it certainly did not need them, nor even the mention of super; all that remains is the presence of another method call onCreate.

That Sivand can elucidate these insights is a mixed blessing and curse: its model-agnostic approach effectively lets us bypass interpreting the millions of parameters and complex inner workings of the studied models, and yields remarkably short programs that make the original predictions easier to comprehend. The use of delta debugging [39] is a novel approach to such model interpretation in general, and its intermediate steps evidently provide useful insights into the model's "thinking". At the same time, our approach is a notable departure from more common neural attribution methods (*e.g.* [33]), which typically try to find the part of the input that was most informative, but do not necessarily assume that all other parts could be removed entirely. Sivand, on the other hand, frequently corrupts its input programs almost beyond recognition. That the assessed models continue to perform so well during this process (Section 5), across multiple code intelligence tasks, is indicative of these models' overreliance on small features and lack of holistic interpretation of their input programs. We discuss various interpretations and implications of this phenomenon in Section 6.

3 SIVAND METHODOLOGY

This section describes Sivand, a methodology for better understanding and evaluating CI models. This methodology is model-agnostic

Table 1: Reduction of a program while preserving the predicted method name `OnCreate` by the CODE2VEC model.

Step(s)	Score	Code

1	1.0	`@Override`
		`public void onCreate(Bundle savedInstanceState){`
		`super.onCreate(savedInstanceState);`
		`setContentView(R.layout.fragmentdetails);`
		`}`

-	-	`...`
4	1.0	`void onCreate(Bundle savceState) { super.onCreate(snceState); setContentView(R.layout.fragmentdetails); }`
7	1.0	`void onCreate(Bundle savceState) { super.onCreate(snceState); setContentView(s); }`
10	0.99	`void onCreate(Bu savate) { s.onCreate(snceState); setContentView(s); }`
13	0.98	`void onCreate(Bu savate) { s.onCreate(snte); View(s); }`
15	0.62	`void onCreate(Bu savate) { s.onCreate(snte);(s); }`
18	0.46	`id onCreate(Bu save) { s.onCreate(snte);; }`
27	0.86	`d onCreate(u ve){s.onCreate(e); }`
-	-	`...`
29	0.79	`d onCreate(u ve){s.onCreate();}`

and can be used for any CI model, however, in this paper, we will focus on the CI models that use neural networks for training, as their lack of transparency warrants more attention.

3.1 Notation

We use p_o to denote the original input to the neural CI model and p_r to denote the simplified input. Let $\mathcal{M}$ be an arbitrary CI model, and p an arbitrary input program, then $Prediction(\mathcal{M}, p)$ denotes the prediction result of $\mathcal{M}$ given p, and $|p|$ the resultant size of p. Conceptually, we define size as the number of atomic parts that p has. In the evaluation of this approach, we use token as the atomic part, hence $|p|$ denotes the number of tokens that are returned by the lexer of the language.

The high-level goal of SIVAND is to produce a reduced input program p_r that is smaller than the size of p_o, i.e., $|p_r| < |p_o|$ (and ideally $|p_r| \ll |p_o|$), such that if $Prediction(\mathcal{M}, p_o) = r$, p_r still retains the same prediction r: $Prediction(M, p_r) = r$. While such a reduction process can in principle terminate under many kinds of conditions (and even return the original input program), we are interested in finding the so-called "1-minimal" input program [39], where no single part of p_r can be removed without losing some desired property of p_o.

3.2 Workflow in SIVAND

Figure 2 depicts a high-level view of the workflow in the proposed methodology, *i.e.* SIVAND. Given a input program, SIVAND uses the DD module to reduce the size of the program. The DD module uses delta debugging to produce various candidate programs by removing various parts of the original input program and iteratively searches for the 1-minimal input program that produces the same prediction as the original input program. Some of the generated candidates might be invalid programs; that is, they do not follow the syntax of the programming language that the program is written in. Therefore, since some CI models require inputs to be

syntactically valid; to enhance the efficiency, SIVAND filters out the invalid candidates that do not parse only for those CIs.

3.3 Reduction Algorithm

Algorithm 1 High level algorithm for *ddmin* delta debugging. The algorithm is initiated by $ddmin(\mathcal{M}, p_o, 2)$.

Input: $\mathcal{M}$, CI model; p, input program; and n, number of partitions.

Split p into n partitions to build $\Delta_1, ..., \Delta_n$
if $\exists \Delta_i$ such that $Prediction(\mathcal{M}, p_o) == Prediction(\mathcal{M}, \Delta_i)$
then
  $ddmin(\mathcal{M}, \Delta_i, 2)$
else if $\exists \Delta_i$ s.t. $Prediction(\mathcal{M}, p_o) == Prediction(\mathcal{M}, p - \Delta_i)$
then
  $ddmin(\mathcal{M}, p\text{-}\Delta_i, max(n-1, 2))$
else if $n < |p|$ **then**
  $ddmin(\mathcal{M}, p, min(2n, |p|))$
else
  return p
end if

Algorithm 1 describes the delta debugging algorithm proposed by Zeller and Hildebrandt [39] and later extended by Groce *et al.* [11, 12], that is adapted to our task for finding minimal input programs. At a high level, the delta debugging algorithm iteratively splits a input program into multiple candidate programs by removing parts of the input program. It uses n to specify the granularity of parts. That is, for an input program p and granularity level n, it generates $2n$ candidates: n candidates by splitting p into n partitions $\Delta_1, \ldots, \Delta_n$, and another n candidates by computing $p - \Delta_1, \ldots, p - \Delta_n$. At each of these steps, the algorithm checks if any resulting candidate program p satisfies the desired property, which here means preserving the prediction of the model on the original input
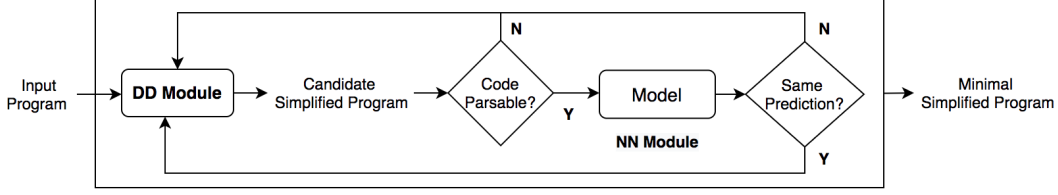
Md Rafiqul Islam Rabin, Vincent J. Hellendoorn, and Mohammad Amin Alipour

Figure 2: Workflow of Sivand.

program, i.e., $Prediction(\mathcal{M}, p_o) = Prediction(\mathcal{M}, \Delta)$, where p_o denotes the original input program. When the algorithm finds a candidate satisfying the property, it uses the candidate as the new base p to be reduced further; otherwise, the algorithm increases the granularity, i.e., n for splitting, until the algorithm determines that the p is 1-minimal. The time complexity of the delta debugging algorithm is quadratic, i.e. $O(n^2)$ in the size of the input program.

4 EXPERIMENTAL SETTINGS

Our proposed methodology is task- and program agnostic. We assess these properties by studying two tasks, and two models on each of these. This section outlines all of these.

4.1 Subject Tasks

We study two popular code intelligence tasks that have gained interest recently: method name prediction (METHODNAME) [3–5], and variable misuse detection VARMISUSE [2, 15, 34].

4.1.1 METHODNAME. In the method name prediction task, the model attempts to predict the name of a method, given its body. We study two commonly used, and similar, neural approaches for this task: CODE2VEC [5], and CODE2SEQ [4]. Both these approaches rely on extracting "paths" in the method's AST, that connect one terminal (token) to another, which are mapped to vector embeddings. These paths are enumerated exhaustively and used by the two models in various ways. The premise is that these paths consolidate both lexical and syntactic information, thus providing more benefit than strictly token-based models, such as RNNs.

In CODE2VEC [5], each path, along with its source and destination terminals, is mapped into a vector embedding that is learned jointly with other network parameters during training. The separate vectors of each path-context are then concatenated to a single context vector using a fully connected layer. An attention vector is also learned with the network, which is used to aggregate the path-context representations into a single code vector representing a method body. The model then predicts the probability of each target method name given the code vector of the method body via a softmax-normalization between the code vector and each of the embeddings of a large vocabulary of possible method names.

The CODE2SEQ [4] model uses an encoder-decoder architecture to encode paths node-by-node and generate labels as sequences at each step. In CODE2SEQ, the encoder represents a method body as a set of paths in AST where individual paths are compressed to a fixed-length vector using a bi-directional LSTM, which encodes paths node-by-node with splitting tokens into sub-tokens. The

decoder again uses attention to select relevant paths while decoding, and predicts sub-tokens of a target sequence at each step when generating the method name.

4.1.2 VARMISUSE. A variable misuse bug [2] occurs when the intended variable used at a particular location of the program differs from the actual variable used. These mistakes are commonly found in real software development [21], and naturally occur as "copy-paste bugs". We specifically experiment with the joint bug localization and repair task of VARMISUSE [34]: given a tokenized sample (buggy or correct), the task is to predict two pointers into these tokens: a) localization: one pointer to indicate the location of the token that contains the wrong variable, or some default token if no bug exists, and b) repair: another pointer to indicate the location of any token that contains the correct variable (ignored for bug-free samples).

We specifically use the dataset released by Hellendoorn *et al.* [15], whose repository also includes a number of models that be applied directly to this dataset.[2] From these, we use the following two generic neural approaches: RNN, and TRANSFORMER. The RNN model here is a simple bi-directional recurrent architecture that uses GRU as the recurrent cell, and has 2 layers, and 512 hidden dimensions. The TRANSFORMER model is based entirely on attention, in which the representations of a snippet's tokens are repeatedly improved through combination with those of all others in the functions. We use the parameters from the original Transformer [35], with 6 layers, 8 heads, and 512 attention dimensions.

4.2 Datasets and Models

Table 2 summarizes the performance characteristics of the CI models that we use in our experiments which is on par with the ones reported in the original studies.

For the METHODNAME task, we use the JAVA-LARGE dataset[3] to train both the CODE2VEC and CODE2SEQ models. This dataset contains a total of $9,500$ Java projects from GitHub, partitioned into $9,000$ projects as training data, 200 projects as validation data, and 300 projects to test on. Overall, it contains about 16M methods.

For the VARMISUSE task, we use the PY150 corpus [4] to train both the RNN and TRANSFORMER models. This dataset contains functions from a total of 150K Python files from GitHub, and are already partitioned into 90K files as training set, 10K files as validation set, and 50K files as testing set. Each function is included both as

---

Table 2: Characteristics of the trained models.

Model	Dataset	Precision	Recall	F_1-Score
Code2Vec	Java-Large	45.17	32.28	37.65
Code2Seq	Java-Large	64.03	55.02	59.19

Model	Dataset	Localization Accuracy	Repair Accuracy	Joint Accuracy
RNN	Py150	63.56	63.22	52.18
Transformer	Py150	73.39	76.79	66.05

a bug-free sample, and with up to three synthetically introduced bugs, yielding about 2 million samples in total.

4.3 Sample Inputs

To evaluate the effectiveness of Sivand, we choose both correctly and incorrectly predicted samples from the test set of these datasets.
a) Correctly predicted samples:

- For MethodName task, we choose 1000 correctly predicted samples for token-level reduction, where the smallest unit of reduction is a token, and around 500 correctly predicted samples for character-level reduction, where the smallest unit is a character. Running character-based reduction was slow as the search space of possible reductions is much larger, hence the lower total. We use the same randomly selected samples for both Code2Vec and Code2Seq models.
- For VarMisuse task, we choose 2000 correctly predicted samples: 1000 from buggy samples, and 1000 bug-free ones. For the selected buggy samples, models correctly predicted both the location pointer and repair targets. For the selected non-buggy samples, models correctly identify as no-bug by prediction special 0-index. We use the same randomly selected samples for both RNN and Transformer models, thus ensuring that we selected only samples on which their predictions were both correct.

b) Wrongly predicted samples:

- For MethodName task, we choose around 500 samples where the predicted method name is wrong. We use token-level reduction only, and randomly select different wrong samples for Code2Vec and Code2Seq, separately.
- For VarMisuse task, we choose 1000 buggy samples where the predicted location pointer is wrong but models correctly predict the repair targets. We randomly select different wrong samples for RNN and Transformer, separately.

4.4 Metrics

Here, we define the metrics that we measure in the experiments as follows.

We use *size reduction ratio* of input programs to evaluate the effectiveness of Sivand in reducing the size of the original input programs. For the input program p_o, and its reduced counterpart p_r, size reduction ratio (or Reduction%) is calculated as $100 * \frac{|p_o|-|p_r|}{|p_o|}$.

All neural network models in our work are capable of making probabilistic predictions, in which a probability distribution is inferred over either tokens (for VarMisuse) or method name (parts) (for MethodName). We can leverage these prediction probabilities to compute a "prediction score", that indicates the confidence of
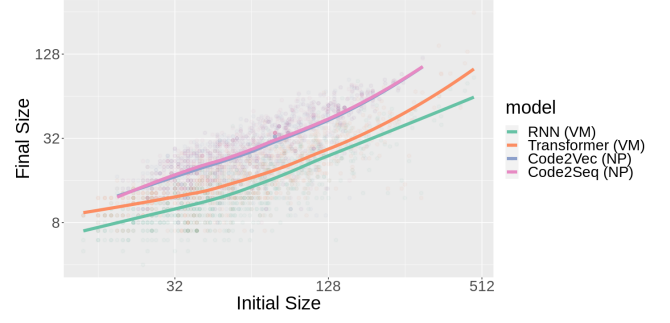

Figure 3: Initial vs. final size of reduced programs in our dataset, measured in tokens. Note the log-scaling on both axes.

the model in a particular prediction. When we reduce the inputs, we track changes to these scores on the original targets; the reduction is stopped once the model ceases to predict the correct output, which generally comes with a drop in the score (probability) of that target. To assess whether our models lose certainty during this reduction phase, we study the changes in the distribution of these scores relative to program reductions.

Hardware. We used a server with an Intel(R) Xeon(R) 2.30GHz CPU and a single NVIDIA Tesla P100 GPU with 12GB of memory to run the experiments in this study.

5 RESULTS

In this section, we present the results of our experiments where we seek to answer the following research questions.

RQ1 How much can typical input programs be reduced?
RQ2 What factors influence reduction potential?
RQ3 Do reduced programs preserve the tokens most used by attention mechanisms?
RQ4 What is the computational cost of Sivand?

Note that for brevity, in this section, unless otherwise noted, the results belong to token-level reduction of correct predictions, and in VarMisuse models, buggy programs.

5.1 RQ1: Length Reduction

The goal of Sivand is to reduce the size of programs as much as possible to help uncover features that highly impact the prediction. Figure 3 shows its capacity to do so in terms of the size of the original programs versus the reduced programs, measured in tokens. This plot aggregates the results of 1,000 such reductions in which each program is reduced until just before its prediction changes. In this, and similar figures, we show the LOESS trend to amortize that large volume of data points (also shown with low opacity). The confidence intervals on these trends are generally very tight and mostly omitted, except where informative.

We find this relation to be mostly linear across all our datasets and models, with the final program containing 2 to 5 times fewer tokens than the initial one. Figure 4 provides an alternative view, showing the achieved reduction as a percentage of the input program size; evidently, program reduction is somewhat easier on larger programs, perhaps because those are more redundant.
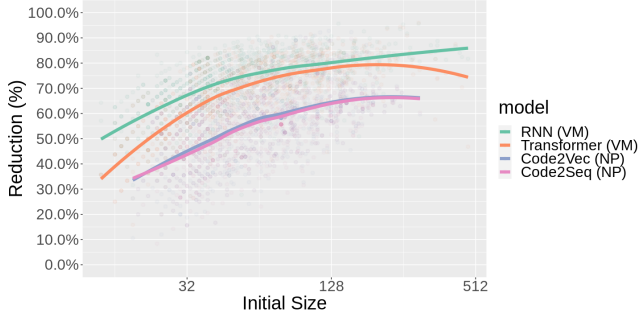
Figure 4: Percentage of tokens reduced vs. initial program size in the various models studied. Note the log-scaling on the x-axis.
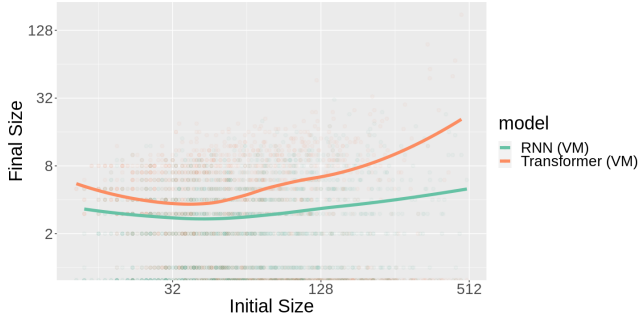

Figure 5: Tokens remaining relative to the minimum possible reduction, in the VarMisuse task. Note the log-scaling on both axes.

These figures alone suggest that the Sivand can reduce a large portion of the input programs, but we found that the true reduction is more substantial: in both datasets, the *maximum possible reduction* is limited, in the VarMisuse case by the need to preserve all variable occurrences (which are error and repair targets), and in the MethodName case by requiring the program to be syntactically valid (which requires keeping at least some syntactic tokens). The true minimum is less tractable for the latter dataset, but when we subtract the irreducible portion from the former, we found the relation shown in Figure 5. Interestingly, for most programs, this trend was *nearly constant*, especially for the RNN, which on average preserves just a few "extra" tokens. The Transformer, meanwhile, appears to be especially reliant on additional syntax on larger programs, although still just a fraction of the original (some 5% of non-essential tokens). In other words, both models require nearly no additional syntax than just the variable names to make their predictions.

The "Reduction(%)" columns in Table 3 provides details of the ratio of reduced tokens to the number of maximum allowed tokens in the models. It shows that our Sivand could remove more than 10% of tokens from each input programs, on average, it removes 62.39%, 61.22%, 93.61%, and 89.43% of the maximum allowed tokens of the original inputs in Code2Vec, Code2Seq, RNN, and Transformer models, respectively. Also in some cases, Sivand was
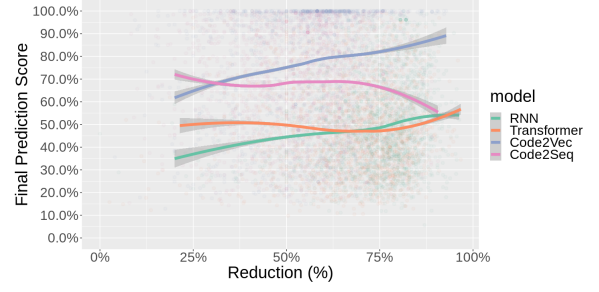

Figure 6: The final score of maximally reduced programs (immediately before changing predictions) vs. the degree of reduction.

able to remove *all* or almost all of the maximum allowed tokens and isolate the important features. This substantial reduction allows to better understand and pinpoint important features in the prediction of the models.

> **Observation 1:** Sivand can reduce the input programs significantly: on average, more than 61% in MethodName models, and more than 89% in VarMisuse models. Different neural architectures show slightly different behaviors, with Transformer permitting less reduction than RNN.

5.2 RQ2: Factors Impacting Reduction

We next study a range of factors that impact, and are impacted by our program reduction approach, to better understand its effect.

Impact on prediction score: Figure 6 tracks the final prediction score against the fraction of the program that was reduced, with inter-quartile ranges shaded ("buggy" samples only for VarMisuse). Recall that the reduction process halts when any further reduction would change its prediction, so it is expected that scores remain modestly high (although not necessarily above 50%). The Code2Vec and Code2Seq models, and to a lesser extent the Transformer model, display pronounced "tipping point" behavior, in which the final reduction step still preserved relatively high probabilities while the immediate next step would have to drop at least below 50%. Note that most samples started out with a score of almost exactly 100%, regardless of models; thus, the difference between the initial and final score is not especially informative. The degree of reduction does not appear to impact the prediction scores much; only for the RNN model on the VarMisuse task do we see a slight downwards trend among input programs that could not be reduced by much, usually smaller ones. We found similarly little correlation with the input and final reduced program sizes. This further reinforces that just one or a few tokens make the difference between a confident prediction and a misprediction.

Character vs. token level: On the MethodName dataset, we additionally studied character-level delta debugging (besides the token-level default). This has the potential to reduce inputs beyond what is possible with token level models, and Figure 7 confirms that it frequently does: this based approach is able to remove another 10-20% of the characters in the input program, thus yielding shorter,
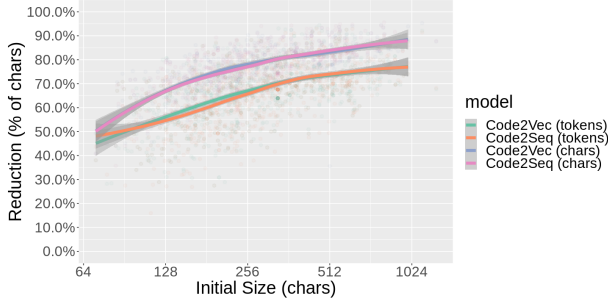
Figure 7: Character vs. token-based model reduction potential on the METHODNAME task, both compared in terms of total characters reduced. Note the log-scaling on the x-axis.

and potentially more informative reductions (such as the one in Figure 1).

Incorrect predictions: Finally, we analyzed whether models could reduce evidence for *incorrect* predictions as well: so far, all our analysis focused on correctly predicted samples, but of course, in practice all these models regularly make mistakes. We can extract the reduced programs for these mispredictions in much the same way, by setting the goal for SIVAND to preserve the incorrect target while reducing the program. Here, we found basically no effect: the correct and incorrect predictions could on average (across all models) be reduced by 62.9% and 64.8% respectively. Similarly, this did not seem to correlate much with the initial program size for any of our models. Thus, it appears that the models do not differentiate in terms of the evidence required to mispredict: correct or not, the models base their prediction on small parts of the input program.

5.3 RQ3: Similarity Between Attention & Reduction

Attention mechanism [35] in neural networks attempts to capture the important features in the input and directly connect them to the output layer to increase the impact in the prediction and improve the performance of neural networks and accelerate training. Intuitively, attention captures the important features in the input programs; therefore, the number of features shared between SIVAND and attention can indicate the effectiveness of the approach for retaining important features.

To evaluate the similarity between tokens in reductions and attention, we apply the SIVAND to reduce the input program p_o to p_r. Let T_{dd} denotes the set of tokens in p_r, and k is the length of T_{dd}. To evaluate how much the set of features retained by SIVAND overlap with the features used in the attention mechanism, for each p_o, we then collect the set of tokens, T_{attn} in the original input programs that receives high attention score. For a fair comparison, we select almost the same size of attention tokens as in the reduced program. From TRANSFORMER model, we get the attention score for each token, hence, we select top-k tokens as T_{attn}. However, for the CODE2VEC and CODE2SEQ models, we get an attention score for each path between two terminal nodes in the program's AST. Therefore, we instead repeatedly choose high attention paths and collect all nodes until we found k distinct nodes. Finally, we calculate the
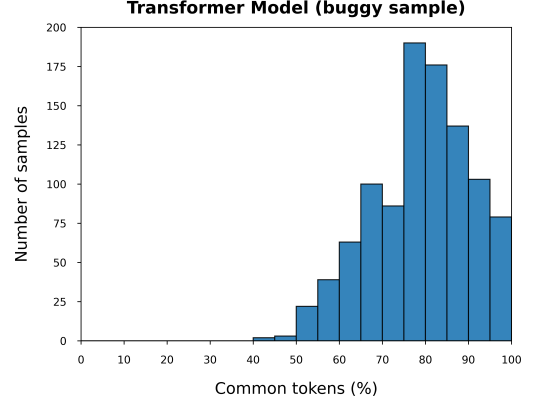

Figure 8: Percentage of common tokens between attention and reduced input programs in TRANSFORMER for buggy inputs.
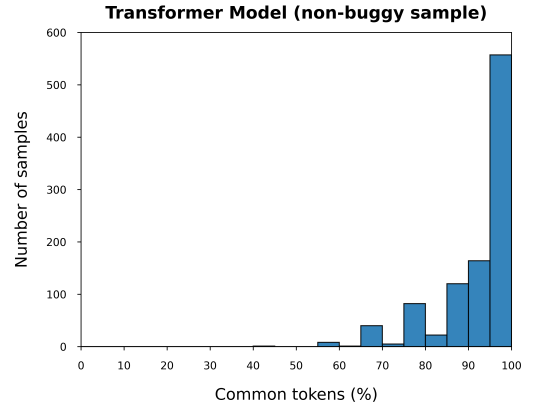

Figure 9: Percentage of common tokens between attention and reduced input programs in TRANSFORMER for non-buggy inputs.

similarity between the tokens in the reduced input program and the attention mechanism by computing $\frac{|T_{attn} \cap T_{dd}|}{|T_{dd}|}$.

Figures 8 and 9 show the similarity between the tokens in the attention and reduction in TRANSFORMER model. The results suggest that in the majority (in more than 550 cases) of the non-buggy input programs T_{attn} and T_{dd} fully match, and in buggy input programs there is a large overlap, majority over 80% match. According to the column 'Common(%)' in Table 3, we can see that the CODE2VEC and CODE2SEQ also have an average overlap over 60% and 70%, respectively. Note that RNN in this experiment does not use attention.

> **Observation 2:** On average, a large portion of tokens (between 60% and 80%) are shared between attention tokens and tokens retained by the reduction.

5.4 RQ4: Computational Cost

Column "Time" in Table 3 shows the average time spent on reduction of the input programs. The time for reduction of the program inputs in VARMISUSE task was lower than four seconds, while the average time needed for reducing an input program in the

Table 3: Summary of reduction results in correctly predicted samples.

Task	Model	# Sample	# Tokens			Reduction (%)			Common (%)			# DD Pass (Average)			Time (second)		
			Min	Avg	Max	Min	Avg	Max	Min	Avg	Max	Total	Valid	Correct	Min	Avg	Max
METHODNAME	CODE2VEC	1000	19	76.08	300	15.38	62.39	98.28	21.43	63.97	93.33	351.22	37.01	33.75	29.44	106.32	714.56
	CODE2SEQ		19	76.08	300	11.11	61.22	96.88	36.84	72.57	95.00	355.01	37.60	33.60	13.99	59.48	216.14
VARMISUSE	RNN	2000	13	82.41	501	29.41	93.61	100	-	-	-	237.32	43.53	30.57	0.13	3.68	77.57
	TRANSFORMER		13	82.41	501	20.00	89.43	100	41.67	85.85	100	241.48	47.01	29.57	0.11	2.99	108.67

Original program:

```python
def modified_time(self, name):
    try:
        file = open(self._path(mtime))
        mtime = float(file.readlines()[2])
        file.close()
    except:
        mtime = None
    return mtime
```

RNN minimized example:

```python
self, name)
        file(self._path(mtime)
        mtime(file()
        file
        mtime
    return mtime
```

Transformer minimized example:

```python
def modified_time(self, name)
        file(self(mtime
        mtime = file
        file
        mtime
    mtime
```

Figure 10: An example of two models learning different kinds of shortcuts. The misused variable is 'mtime' (in red), incorrectly passed in place of 'name' on line 3 of the original snippet.

METHODNAME task was around 60 and 107 seconds in CODE2SEQ and CODE2VEC models, respectively.

The columns under "#DD Pass" show that the search for minimal inputs in METHODNAME included creating and trying slightly more than 350 intermediate results in the delta debugging, wherein, on average, around 37 of them were syntactically correct and 33 produced the same prediction as to the original input program. Similarly, in models related to VARMISUSE, the reduction takes between 237 and 241 on average from which around 30 intermediate reduced programs produce predictions similar to the original input program.

> **Observation 3:** Input program reduction based on tokens is efficient and can reach the minified input with the same prediction in a relatively small number of steps.

6 DISCUSSION

The central motivation for using linguistics-inspired models on software has long been that source code is in a sense "natural"; that is, it contains repetitive, predictable patterns much like natural text [16]. Models that leverage this intuition have evolved from simple n-gram based models to complex neural networks, such as those used in this paper in recent years, and become applicable to (and successful at) a wide range of coding tasks. Yet, they are still built and motivated with the same core premise of learning from programs in their entirety (punctuation and all) and in lexical order. Recent results have already begun to suggest that, in practice, these models may not be using much of their input programs for at least one task [36]; however, this still focuses on otherwise natural, only simplified programs.

Our method makes no attempt at preserving any meaning or validity of the original program; we purely focus on the smallest amount of data with which our models could suffice. This allows us to show a new, stronger result: these four models across two tasks appear almost entirely indifferent to the naturalness of the provided code snippets – remove as much as 90% of tokens and both their predictions, and their confidence therein remain almost unchanged (Figure 6). We discuss the reasons and implications of this observation using several examples in the remainder of this section.

6.1 Explanations vs. Shortcuts

Figure 10 shows an example of a code snippet and its minified by an RNN and a TRANSFORMER model respectively. In this case, the (synthetically induced) error is fairly obvious; mtime is passed to the _path call where name should be. If given the task to find such a bug, a programmer might notice this particular use-before-definition bug, which could then be *explained* with a relatively small portion of the function, perhaps involving the unused parameter name and mtime's use before its assignment. Looking at the two minified programs, we see traces of these same explanations: the RNN model, which prioritizes lexically local patterns [15], primarily preserves the part of the program where mtime is passed to _path – an unlikely combination given that name is in the immediate context. The Transformer, meanwhile, prefers to focus on the subsequent assignment to mtime, which is out of place given its preceding use.

However, the existence of a short explanation does not naturally imply that much of the program is unnecessary, as it apparently does to our models. The two minified programs are arguably much harder to read than the original – especially as our model would see them, without the original for reference, or the indentation that we added here – yet our models seemed to have no more trouble predicting from them than the original. It is evident that our models have learned to take *shortcuts*: they predominantly

leverage simple syntactic patterns, quite literally to the exclusion of almost all else. Our broader results show that this behavior is ubiquitous and extends across models and tasks (section 5).

These shortcuts still capture meaningful, natural programming patterns. Neither failing to use a parameter, nor the assignment after use,[5] are necessarily syntactically invalid, but our models have clearly learned that they are sufficiently irregular to contemplate a bug. What is remarkable about our findings is not the absence of "naturalness" as a whole, but the absence of *macro-level naturalness*. Our models appear to have little to no care for the overall structure and content of the function, just for the presence or absence of specific patterns therein.

6.2 Ramifications for Deep Learning

Do our results then imply that deep learners are, in a sense, frauds, at least in software engineering applications? Not quite: it is well known that machine learners, neural networks especially, prefer to find simple explanations of their training data [18], which often hinders their generalizability. The models in our case are doing nothing less: they are presented with a single, often obvious task and learn to solve it (arguably) as efficiently as possible. That does not invalidate the quality of their learning: in practice, there are myriad patterns to heed when predicting a vocabulary of hundreds of thousands of method names, or finding arbitrary variable misuse bugs in millions of functions. One interpretation of our results is that our models arrive at a set of simple "explanations" that encompasses nearly all these cases, such as how a variable should not be used before its assignment. This vocabulary is still large and diverse, so it remains a significant challenge for models to discover – there are good reasons why model performance can differ vastly even on simple tasks [15].

We saw these differences in action, too: Transformer models are substantially better at leveraging "global" information from throughout the function than RNNs, which are largely (lexically) "local". Correspondingly, we saw that the latter permitted substantially more reduction of input programs, anecdotally because it mainly preserved features in the immediate context of the bug's location. Note that this is not a strength of the RNN: we are reducing programs to find out how much evidence these models used in their predictions. The ideal is neither the complete program nor virtually none of it.

Overall, the apparent indifference to macro-level naturalness (that functions as a whole are complete and well-formatted) is troubling. Much like prior findings, our demonstration that models rely on just a few features of their input naturally implies that they are highly vulnerable to perturbations in those inputs [7, 36]. A natural question may be: what might prompt them to read code more holistically? One answer may come in the form of multi-task learning, in which a single model is trained on a wide variety of tasks [20]. In our analyses, we believe the models learned those salient patterns that helped achieve their singular objective; a mixture of diverse objectives might prevent such shortcuts altogether. Whether and how this works in practice is an open question. One risk may be that similar types of shortcuts are useful for many tasks, especially those based on synthetically generated flaws – a common practice

---

[5]The variable might be a field.

in our field. If so, our approach should be able to elucidate this, and may well be able to serve more generally as an indicator of the complexity of a task and/or the degree of information used by a model, by using visualizations such as Figure 4 that show the amount of input data required to accomplish a task on average.

6.3 Accessible Prediction Attribution

Prediction attribution in deep learners is a fast growing field, that attempts to relate the prediction of a neural network to a part of its input [33]. Most current attribution approaches either require access to the trained model in full ("white-box" methods), including its architecture and (hyper-)parameters, or try to approximate them, e.g., [31, 32]. The methodology based on reduction that we used in this paper does not require any knowledge of either architecture or hyper-parameters of the models; it can be applied in a complete black-box fashion. This both makes it applicable to cases where internal information of models is unavailable, e.g. proprietary models, but also substantially eases its use. Deploying our model to a new pre-trained model only required knowing minimal constraints on the input (which tokens may not be reduced; whether the program must continue to be parseable), which are usually readily accessible.

Moreover, current techniques in attribution and transparency usually require a certain level of knowledge about the characteristics of learning algorithms and reasoning about the model behaviors, which the average practitioner may not have sufficient time to acquire. We envision that the application of Sivand, if deployed directly to developers, would thus be more accessible to software engineers, especially those who have prior familiarity with the concept of test reduction for failing test cases in debugging, and this knowledge can be easily transferred to reducing input programs in Sivand. Our results did show differences, sometimes significant ones, in reduction-behavior between various architectures (*e.g.* fig. 10), which may well be useful for experts to interpret. But, using Sivand does not require such knowledge; the reduced input programs speak for themselves. We plan to perform a user-study to evaluate the usability of our approach to the average practitioner.

6.4 Limits of Extractive Interpretation

The Sivand methodology proposed in this paper is best described as an *extractive* interpretation approach. Extractive interpretation methods extract and present the important regions of the inputs, leaving it to the user to characterize the behavior of the model. These approaches usually work well in cases where features are sparse and not correlated, so that highlighting one or more parts in the input provides enough insights about the model. The high rate of reduction in most cases in our work may suggest that this approach is indeed applicable here and can provide sufficient insights about the behavior of the models – although a user study is needed to validate that further. In turn, the power of this approach would be limited on models that use complex or non-syntactic features such as the number of lines of code in programs, as the basis for their prediction. This would prevent the input programs from being reduced significantly, or it might be difficult to make sense of the output and pinpoint the underlying important feature in the reduced programs.

```
public static void f(String[] args) {
    System.setProperty(Constants.DUBBO_PROPERTIES_KEY,
        "conf/dubbo.properties");
    Main.main(args);
}
```

Figure 11: Code2Vec correctly predicts the name of this method as 'main'.

6.5 Impact of Smallest Atomic Unit Choice

Choosing different smallest atomic units in the delta debugging algorithm can provide different, and potentially complementary insights about the model. For example, Code2Vec predicts the name of the code snippet in Figure 11 as 'main'. If we use hierarchical delta debugging wherein the smallest atomic unit of reduction is an AST node, the result would be "`void f (String args) {}`" suggesting that the method signature and the argument name have had a large impact on the prediction. However, if we choose characters as the smallest atomic unit and employ delta debugging, the result is "`d f(Sg[]r){em.s(C.D,"");Main(r);}`" which provides a complementary view for the prediction that shows the importance of `Main` identifier used in the method body. Future work may extend our approach to new metrics and reduction strategies, which may well provide novel insights, especially in future, more complex models that are more resistant to such simple reduction.

7 RELATED WORK

There has been a lot of work in the area of transparent or interpretable AI, computer vision, and natural language processing that focuses on understanding neural networks. Interpretable, transparent machine learning has numerous benefits, including making predictions explainable (and thus more useful), using learned models to generate new insights, and improving the quality and robustness of the models themselves [30]. These objectives are generally studied under the umbrella of "explainable AI". While some work studies the properties of neural models in general [6], many studies are more ad-hoc, focusing on specific domains and tasks [1].

7.1 Software Engineering

There is a growing body of work in the domain of robust neural models for source code or code intelligence in general. Bui *et al.* [8] evaluated the impact of a specific code fragment by deleting it from the original program, Rabin *et al.* [25] compared the performance of neural models and SVM models with handcrafted features and found comparable performance with a relatively small number of features. Wang *et al.* [36] propose a mutate-reduce approach to find important features in the code summarization models. Several other studies [7, 24, 26, 27] have evaluated the robustness of neural models of code under semantic program transformations and found that transformations can change the results of the prediction in many cases.

7.2 Computer Vision and NLP

The need for neural model interpretability and robustness was firmly established by Goodfellow *et al.*, who showed that a convolutional neural network could be tricked into changing its image classification into virtually any label by adding an imperceptible amount of targeted noise [10]. This produced a flood of research on both improving the robustness of neural networks and attacking current architectures [9], often in tandem. Importantly, comprehensive studies of robustness and adversarial perturbations have yielded rich scientific insights into the learning behavior of these models, both for vision and beyond [18].

The computer vision community has proposed many ways to visualize what parts of the input are most significant to a neural network, both to individual neurons and its output classification. A popular example, and similar to our approach, is occlusion analysis [38], in which regions of interest are identified by evaluating the network's prediction when various parts of the input image are occluded. Locations at which prediction accuracy rapidly drops when removed are likely especially integral to the prediction. For linguistic expressions, input perturbations are usually less obvious: while certain words (such as stop words) may safely be removed without altering the meaning of a sentence, more general changes quickly risk producing very different inputs. Recent input-related methods rely on synonym datasets and swap out similar names to ensure that they generate semantically similar phrases [19, 29]. Our work shows that, at least for current models and tasks, this is significantly less of a concern in software engineering, where many tokens can be removed with little consequence.

8 THREATS TO VALIDITY

We evaluated Sivand on four neural models and two tasks, trained on millions but tested on a few thousands of random samples from their datasets. As such, our findings may naturally not extend to other inputs, models, and domains. Nevertheless, we argue both that our analysis is broad, spanning more models and domains than comparable work [36], and that the design of our approach is applicable to many other problem settings in our field, which commonly take tokens and ASTs as inputs to yield a single, or few outputs, all covered by the models in this work. To ensure software quality, we used the tools and datasets shared by the original developers of the models, each from public repositories used by dozens of developers and cited in multiple studies. For our input reduction, we adapted Zeller *et al.*'s implementation of delta debugging [39], which has been widely used in the industry and other research studies over decades.

9 CONCLUSION

We proposed Sivand, a simple, model-agnostic methodology for interpreting a wide range of code intelligence models, which works by reducing the size of input programs using the well-known delta-debugging algorithm. We apply Sivand to four popular neural code intelligence models across two datasets and two tasks, showing that our method can significantly reduce the size of input programs while preserving the prediction of the model, thereby exposing the most significant input features to the various models. Our results hint at the idea that these models often use just a few simple syntactic shortcuts in their prediction. This sets the stage for broader use of transparency-enhancing techniques to better understand and develop neural code intelligence models.

REFERENCES

[1] ALLAMANIS, M., BARR, E. T., DEVANBU, P., AND SUTTON, C. A survey of machine learning for big code and naturalness. *ACM Comput. Surv. 51*, 4 (July 2018).

[2] ALLAMANIS, M., BROCKSCHMIDT, M., AND KHADEMI, M. Learning to represent programs with graphs. In *International Conference on Learning Representations* (2018).

[3] ALLAMANIS, M., PENG, H., AND SUTTON, C. A convolutional attention network for extreme summarization of source code. In *Proceedings of the 33nd International Conference on Machine Learning, ICML* (2016).

[4] ALON, U., LEVY, O., AND YAHAV, E. code2seq: Generating sequences from structured representations of code. In *International Conference on Learning Representations* (2019).

[5] ALON, U., ZILBERSTEIN, M., LEVY, O., AND YAHAV, E. Code2vec: Learning distributed representations of code. *Proc. ACM Program. Lang. 3*, POPL (Jan. 2019), 40:1–40:29.

[6] BALDUZZI, D., FREAN, M., LEARY, L., LEWIS, J. P., MA, K. W.-D., AND MCWILLIAMS, B. The shattered gradients problem: If resnets are the answer, then what is the question? In *Proceedings of the 34th International Conference on Machine Learning* (International Convention Centre, Sydney, Australia, 06–11 Aug 2017), D. Precup and Y. W. Teh, Eds., vol. 70 of *Proceedings of Machine Learning Research*, PMLR, pp. 342–350.

[7] BIELIK, P., AND VECHEV, M. Adversarial robustness for code. *Proceedings of the International Conference on Machine Learning (ICML)* (2020).

[8] BUI, N. D. Q., YU, Y., AND JIANG, L. Autofocus: Interpreting attention-based neural networks by code perturbation. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)* (Nov 2019), pp. 38–41.

[9] CARLINI, N., AND WAGNER, D. Towards evaluating the robustness of neural networks. In *2017 ieee symposium on security and privacy (S&P)* (2017), IEEE, pp. 39–57.

[10] GOODFELLOW, I. J., SHLENS, J., AND SZEGEDY, C. Explaining and harnessing adversarial examples. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings* (2015), Y. Bengio and Y. LeCun, Eds.

[11] GROCE, A., ALIPOUR, M. A., ZHANG, C., CHEN, Y., AND REGEHR, J. Cause reduction for quick testing. In *Seventh IEEE International Conference on Software Testing, Verification and Validation, ICST 2014, March 31 2014-April 4, 2014, Cleveland, Ohio, USA* (2014), IEEE Computer Society, pp. 243–252.

[12] GROCE, A., ALIPOUR, M. A., ZHANG, C., CHEN, Y., AND REGEHR, J. Cause reduction: delta debugging, even without bugs. *Softw. Test. Verification Reliab. 26*, 1 (2016), 40–68.

[13] GU, X., ZHANG, H., ZHANG, D., AND KIM, S. Deep api learning. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (2016), pp. 631–642.

[14] HELLENDOORN, V. J., BIRD, C., BARR, E. T., AND ALLAMANIS, M. Deep learning type inference. In *Proceedings of the 2018 26th acm joint meeting on european software engineering conference and symposium on the foundations of software engineering* (2018), pp. 152–162.

[15] HELLENDOORN, V. J., SUTTON, C., SINGH, R., MANIATIS, P., AND BIEBER, D. Global relational models of source code. In *International Conference on Learning Representations* (2020).

[16] HINDLE, A., BARR, E. T., SU, Z., GABEL, M., AND DEVANBU, P. On the naturalness of software. In *2012 34th International Conference on Software Engineering (ICSE)* (2012), IEEE, pp. 837–847.

[17] HORNIK, K., STINCHCOMBE, M., AND WHITE, H. Multilayer feedforward networks are universal approximators. *Neural networks 2*, 5 (1989), 359–366.

[18] ILYAS, A., SANTURKAR, S., TSIPRAS, D., ENGSTROM, L., TRAN, B., AND MADRY, A. Adversarial examples are not bugs, they are features. In *Advances in Neural Information Processing Systems* (2019), pp. 125–136.

[19] JIN, D., JIN, Z., ZHOU, J. T., AND SZOLOVITS, P. Is bert really robust? a strong baseline for natural language attack on text classification and entailment. *Proceedings of the AAAI Conference on Artificial Intelligence 34*, 05 (Apr. 2020), 8018–8025.

[20] KANADE, A., MANIATIS, P., BALAKRISHNAN, G., AND SHI, K. Pre-trained contextual embedding of source code. *arXiv preprint arXiv:2001.00059* (2019).

[21] KARAMPATSIS, R.-M., AND SUTTON, C. How often do single-statement bugs occur? the manysstubs4j dataset. In *Proceedings of the 17th International Conference on Mining Software Repositories* (2020), pp. 573–577.

[22] LECUN, Y., BENGIO, Y., AND HINTON, G. Deep learning. *nature 521*, 7553 (2015), 436.

[23] MURDOCH, W. J., SINGH, C., KUMBIER, K., ABBASI-ASL, R., AND YU, B. Definitions, methods, and applications in interpretable machine learning. *Proceedings of the National Academy of Sciences 116*, 44 (2019), 22071–22080.

[24] RABIN, M. R. I., BUI, N. D., WANG, K., YU, Y., JIANG, L., AND ALIPOUR, M. A. On the generalizability of neural program models with respect to semantic-preserving program transformations. *Information and Software Technology* (2021), 106552.

[25] RABIN, M. R. I., MUKHERJEE, A., GNAWALI, O., AND ALIPOUR, M. A. Towards demystifying dimensions of source code embeddings. In *Proceedings of the 1st ACM SIGSOFT International Workshop on Representation Learning for Software Engineering and Program Languages* (New York, NY, USA, 2020), RL+SE&PL 2020, Association for Computing Machinery, p. 29–38.

[26] RABIN, M. R. I., WANG, K., AND ALIPOUR, M. A. Testing neural program analyzers. *34th IEEE/ACM International Conference on Automated Software Engineering (Late Breaking Results-Track)* (2019).

[27] RAMAKRISHNAN, G., HENKEL, J., WANG, Z., ALBARGHOUTHI, A., JHA, S., AND REPS, T. Semantic robustness of models of source code. *arXiv preprint arXiv:2002.03043* (2020).

[28] RAY, B., HELLENDOORN, V., GODHANE, S., TU, Z., BACCHELLI, A., AND DEVANBU, P. On the "naturalness" of buggy code. In *Proceedings of the 38th International Conference on Software Engineering* (New York, NY, USA, 2016), ICSE '16, ACM, pp. 428–439.

[29] REN, S., DENG, Y., HE, K., AND CHE, W. Generating natural language adversarial examples through probability weighted word saliency. In *Proceedings of the 57th annual meeting of the association for computational linguistics* (2019), pp. 1085–1097.

[30] SAMEK, W., MONTAVON, G., LAPUSCHKIN, S., ANDERS, C. J., AND MÜLLER, K.-R. Toward interpretable machine learning: Transparent deep neural networks and beyond. *arXiv preprint arXiv:2003.07631* (2020).

[31] SELVARAJU, R. R., COGSWELL, M., DAS, A., VEDANTAM, R., PARIKH, D., AND BATRA, D. Grad-cam: Visual explanations from deep networks via gradient-based localization. In *Proceedings of the IEEE international conference on computer vision* (2017), pp. 618–626.

[32] SIMONYAN, K., VEDALDI, A., AND ZISSERMAN, A. Deep inside convolutional networks: Visualising image classification models and saliency maps. *arXiv preprint arXiv:1312.6034* (2013).

[33] SUNDARARAJAN, M., TALY, A., AND YAN, Q. Axiomatic attribution for deep networks. In *Proceedings of the 34th International Conference on Machine Learning* (2017), pp. 3319–3328.

[34] VASIC, M., KANADE, A., MANIATIS, P., BIEBER, D., AND SINGH, R. Neural program repair by jointly learning to localize and repair. In *International Conference on Learning Representations* (2019).

[35] VASWANI, A., SHAZEER, N., PARMAR, N., USZKOREIT, J., JONES, L., GOMEZ, A. N., KAISER, Ł., AND POLOSUKHIN, I. Attention is all you need. In *Advances in neural information processing systems* (2017), pp. 5998–6008.

[36] WANG, Y., GAO, F., AND WANG, L. Demystifying code summarization models. *arXiv preprint arXiv:2102.04625* (2021).

[37] WHITE, M., VENDOME, C., LINARES-VÁSQUEZ, M., AND POSHYVANYK, D. Toward deep learning software repositories. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories* (2015), IEEE, pp. 334–345.

[38] ZEILER, M. D., AND FERGUS, R. Visualizing and understanding convolutional networks. In *European conference on computer vision* (2014), Springer, pp. 818–833.

[39] ZELLER, A., AND HILDEBRANDT, R. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering 28*, 2 (2002), 183–200.