Adversarial Robustness for Code

Conference Paper

Author(s):
Bielik, Pavol; Vechev, Martin

Publication date:
2020

Permanent link:
https://doi.org/10.3929/ethz-b-000466229

Originally published in:
Proceedings of Machine Learning Research 119

Funding acknowledgement:
680358 - Learning from Big Code: Probabilistic Models, Analysis and Synthesis (EC)

Adversarial Robustness for Code

Pavol Bielik [1] **Martin Vechev** [1]

Abstract

Machine learning and deep learning in particular has been recently used to successfully address many tasks in the domain of code such as finding and fixing bugs, code completion, decompilation, type inference and many others. However, the issue of adversarial robustness of models for code has gone largely unnoticed. In this work, we explore this issue by: *(i)* instantiating adversarial attacks for code (a domain with discrete and highly structured inputs), *(ii)* showing that, similar to other domains, neural models for code are vulnerable to adversarial attacks, and *(iii)* combining existing and novel techniques to improve robustness while preserving high accuracy.

1. Introduction

Recent years have seen an increased interest in using deep learning to train models of code for a wide range of tasks including code completion (Brockschmidt et al., 2019; Li et al., 2018), code captioning (Alon et al., 2019; Allamanis et al., 2016; Fernandes et al., 2019), code classification (Mou et al., 2016; Zhang et al., 2019) and bug detection (Allamanis et al., 2018; Pradel & Sen, 2018; Li et al., 2019). Despite substantial progress on training accurate models of code, the issue of robustness has been overlooked. Yet, this is a very important problem shown to affect neural models in different domains (Goodfellow et al., 2015; Szegedy et al., 2014; Papernot et al., 2016).

Challenges in modeling code In our work, we focus on tasks that compute program properties (e.g., type inference), usually addressed via handcrafted static analysis, but for which a number of recent neural models with high accuracy have been introduced (Hellendoorn et al., 2018; Schrouff et al., 2019; Malik et al., 2019). Unsurprisingly, as these works do not consider adversarial robustness, their adversarial accuracy can drop significantly.

[1]Department of Computer Science, ETH Zürich, Switzerland. Correspondence to: Pavol Bielik <pavol.bielik@inf.ethz.ch>.

Proceedings of the 37th International Conference on Machine Learning, Online, PMLR 119, 2020.

Figure 1. Illustration of the three key components used in our work. Each point represents a sample, □ is a region where model abstains from making predictions, ■ and ■ are regions of model prediction, ◯ is the space of valid modifications for a given sample, and ▷ is the learned (reduced) space of valid modifications.

However, training both *robust and accurate* models of code in this setting is non-trivial and requires one to address several key challenges: *(i)* programs are highly structured and long, containing hundreds of lines of code, *(ii)* a single discrete program change can affect the prediction of a large number of properties and is much more disruptive than a slight continuous perturbation of a pixel value, and *(iii)* the property prediction problem is usually undecidable (hence, static analyzers approximate the ideal solution).

Accurate and robust models of code As a first step to address these challenges, we propose a novel method that combines three key components, illustrated in Figure 1 – as we show, all of these contribute to achieving accurate and robust models of code. First, we train a model that *abstains* (Liu et al., 2019) from making a prediction when uncertain, effectively partitioning the dataset into two parts: one part where the model makes predictions (■, ■) that should be *accurate* and *robust*, and one (□) where the model abstains and it is enough to be *robust*. Second, we instantiate *adversarial training* (Goodfellow et al., 2015) to the domain of code. Third, we develop a new method to *refine the representation* used as input to the model by learning the parts of the program relevant for the prediction. This reduces the number of places that affect the prediction and helps to make adversarial training for code effective. Finally, we create a new algorithm that trains multiple models, each learning a specialized representation that makes robust predictions on a different subset of the dataset.

We instantiate our approach to the type prediction task and show its effectiveness – we train a model that improves robustness by 15% while preserving high accuracy.

2. Accurate and Robust Models of Code

In this section, we present an overview of our approach. Without loss of generality, we define an input program p to be a sequence of words $p = w_{1:n}$. The words can correspond to a tokenized version of the program, nodes in an abstract syntax tree corresponding to p or other suitable program representations. Further, let $l \in \mathbb{N}$ be a position in the program p that corresponds to a word $w_l \in \mathbb{W}$. A training dataset $\mathcal{D} = \{(x_j, y_j)\}_{j=1}^{N}$ contains a set of samples, where $x \in \mathbb{X}$ is an input tuple $x = \langle p, l \rangle$ consisting of a program p and a position in the program l, while $y \in \mathbb{Y}$ contains the ground-truth label. As an example, the code snippet in Figure 2*a* contains 12 different samples (x, y), one for each position where a prediction should be made (annotated with their ground-truth types y).

Our goal is to learn a function $f\colon \mathbb{X} \to \mathbb{R}^{|\mathbb{Y}|}$, represented as a neural network, which for a given input program and a position in the program, computes the probability distribution over the labels. The model's prediction then corresponds to the label with the highest probability according to f.

Step 1: Augment the model with an (un)certainty score We start by augmenting the standard neural model f with an option to abstain from making a prediction. To achieve this, we adopt the recently proposed approach by (Liu et al., 2019) and introduce a selection function $g_h\colon \mathbb{X} \to \mathbb{R}$, which measures model certainty. Then, the model is defined to make a prediction only if g_h is confident enough ($g_h(x) \geq h$) and abstain from making a prediction otherwise. Here, $h \in \mathbb{R}$ is an associated threshold that controls the desired level of confidence. For example, using a high threshold $h = 0.9$, the model learns to make only five predictions for the program in Figure 2*b* and will abstain from uncertain predictions such as predicting parameter types.

The first insight from our work is that allowing the model to abstain is beneficial for achieving robustness. This step leads to simpler models, since learning to abstain is easier than learning to predict the correct label. This is in contrast with forcing the model to learn the correct label for *all* samples, which is infeasible for most practical tasks.

Step 2: Adversarial training Next, we instantiate adversarial training to the domain of code. Concretely, let $\Delta(x)$ be a set of valid modifications of sample x and let $x + \boldsymbol{\delta}$ denote a new input obtained by applying the modifications in $\boldsymbol{\delta} \subseteq \Delta(x)$ to x. As a concrete example, Figure 2*c* shows a refactoring of the program from Figure 2*b* by renaming `hex` to `color`. Even though this change does not affect the types in the program, the model suddenly predicts incorrect types for both the `color` parameter and the `substring` function. Further, even though the types of `parseInt` and `v` are still correct, the model became much more uncertain.

Intuitively, our goal is to address this issue and to ensure that the model is robust for all valid modifications $\boldsymbol{\delta} \subseteq \Delta(x)$ – when evaluated on $x + \boldsymbol{\delta}$, the model either abstains or predicts the correct label. Concretely, we use adversarial training (Goodfellow et al., 2015), which instead of minimizing the expected loss on the original distribution $\mathbb{E}_{(x,y)\sim D}[\ell((f, g_h)(x), y)]$ as usually done in standard training, minimizes the expected *adversarial loss*:

$$\mathbb{E}_{(x,y)\sim D}[\max_{\boldsymbol{\delta} \subseteq \Delta(x)} \ell((f, g_h)(x + \boldsymbol{\delta}), y)] \tag{1}$$

That is, we minimize the worst case loss obtained by applying a valid modification to the original sample x. Similar to other domains, the main challenge in this setting is solving the inner $\max_{\boldsymbol{\delta} \subseteq \Delta(x)}$ efficiently for the domain of code.

Standard adversarial training is insufficient Although adversarial training has been successfully applied in many domains (Madry et al., 2018; Wong & Kolter, 2018; Sinha et al., 2018; Raghunathan et al., 2018), in our work we show that for code, adversarial training alone is insufficient to achieve model robustness. The key reason is that, existing neural models of code typically process *the entire program* which can contain hundreds of lines of code. This is problematic as it means that any program change will affect *all* predictions and there can be infinitely many program changes in $\Delta(x)$. Further, a single discrete program change is much more disruptive in affecting the model than a slight continuous perturbation of a pixel value. At the same time, while not sufficient, in our evaluation we show that adversarial training can be used to improve robustness by 0 to 7%, depending on the model architecture.

Step 3: Representation refinement To address the issue that adversarial training alone does not work well, we develop a novel technique that: (i) learns which parts of the input program are relevant for the given prediction, and (ii) refines the model representation such that only relevant program parts are used as input to the neural network. Essentially, the technique automatically learns an abstraction α which given a program, produces a relevant representation of that program. Figure 2*d* shows an example of a possible abstraction α that takes as input the entire program but keeps only parts relevant for predicting the type of `parseInt` – it is a method call with name `parseInt` which has two arguments. To learn the abstraction α, we first represent programs as graphs and then phrase the refinement task as an optimization problem that minimizes the number of graph edges, while ensuring that the accuracy of the model before and after applying α stays roughly the same.

Finally, we apply adversarial training, but this time on the abstraction α obtained via representation refinement, resulting in new functions f and g_h. Overall, this results in an adversarially robust model $m_i = \langle f, g_h, \alpha \rangle$.

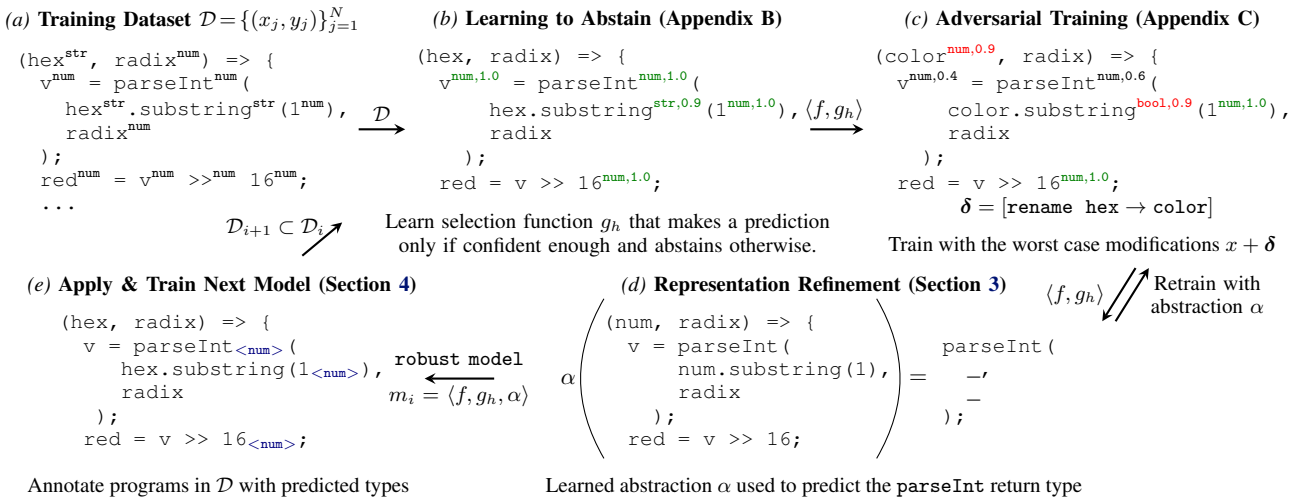


Figure 2. Overview of the main steps of our approach for learning accurate and adversarially robust models of code.

Step 4: Learning accurate models Although the model m_i is robust, it provides predictions only for a subset of the samples for which it has enough confidence (i.e., $g_h(x) \geq h$). To increase the ratio of samples for which our approach makes a prediction (i.e., does not abstain), we perform two steps: *(i)* generate a new dataset $\mathcal{D}_{i+1}$ by annotating the program with the predictions made by the learned model m_i, and removing successfully predicted samples, and *(ii)* learn another model m_{i+1} on the new dataset $\mathcal{D}_{i+1}$. We repeat this process for as long as the new learned model predicts some of the samples in $\mathcal{D}_{i+1}$.

Training multiple models is beneficial because: *(i)* the models are easier to train as well as easier to make robust as they do not try to learn all predictions, *(ii)* it allows conditioning on the predictions learned by earlier models which helps both interpretability and robustness. For example, the model m_{i+1} can learn that the left hand side of the assignment `v=parseInt` has the same type as the right hand side, since the type of `parseInt` was already predicted by m_i. Interestingly, if we think of each model as a learned set of rules, we can essentially apply the models to a given program in a fixed point style (similar to how a traditional sound static analysis works), and *(iii)* each model learns a different representation α that is specialized for the predictions it makes. For example, while predicting the type of `parseInt` is independent of the argument values (`parseInt(_, _)`), predicting the second argument type is not (`parseInt(_, radix)`). Using a single abstraction to predict both would lead to either reduced robustness or accuracy, depending on which abstraction is used.

Summary Given a training dataset $\mathcal{D}$, our approach learns a set of robust models, each of which makes robust predictions for a different subset of $\mathcal{D}$. To achieve this, we extend existing neural models of code with three key components – the ability to abstain (with associated uncertainty score), adversarial training, and learning to refine the representation. Given the limited space, we provide formal description of the the first two components that learn to abstain and apply adversarial training for code in Appendix B and Appendix C, respectively. Next, we formally describe the novel components – learning to refine the representation (Section 3) and present our training algorithm that combines all of them together (Section 4).

3. Learning to Refine Representations

As motivated in Section 2, a key issue with many existing neural models for code is that the model prediction $f(x)$ depends on the *full* program p, even though only small parts of p are typically relevant. We address this issue by learning an abstraction α that takes as input p and produces only the parts relevant for the prediction. That is, α refines the representation given as input to the neural model.

Overview Our method works as follows: *(i)* we convert the program into a graph representation, *(ii)* then define the model to be a graph neural network (e.g., (Veličković et al., 2018; Kipf & Welling, 2017; Wu et al., 2019; Li et al., 2016)), which at a high level works by propagating and aggregating messages along graph edges, *(iii)* because dependencies in graph neural networks are defined by the structure of the graph (i.e., the edges it contains), we phrase the problem of refining the representation as an optimization problem which removes the maximum number of graph edges (i.e., removes the maximum number of dependencies) without degrading model accuracy, and *(iv)* we show how to solve the optimization problem efficiently by transforming it to an integer linear program (ILP).

From programs to graphs Following prior works, we represent programs using their corresponding abstract syntax trees (AST). These are further transformed into graphs, as done in (Allamanis et al., 2018; Brockschmidt et al., 2019), by including additional edges.

Definition 3.1. (Directed Graph) A directed graph is a tuple $G = \langle V, E, \xi_V, \xi_E \rangle$ where V denotes a set of nodes, $E \subseteq V^2$ denotes a set of directed edges, $\xi_V \colon V \to \mathbb{N}^k$ is a mapping from nodes to their associated attributes and $\xi_E \colon E \to \mathbb{N}^m$ is a mapping from edges to their attributes.

We associate two attributes with each node – *type* which corresponds to the type of the AST node (e.g., `Block`, `Identifier`, `BinaryExpression`, etc.) and *value* associated with the AST node (e.g., $+, -, 0, 1, $"`GET`", `x`, `data`, etc.). For edges we use a single attribute the edge *type*, which can be: *(i) ast*, for the edges that correspond to those included in the AST, *(ii) last usage*, for edges introduced between any two usages (either read or write) of the same variable, and *(iii) returns-to*, for edges introduced between a return statement and the function declaration. All edges are initially added in both directions, but can be later removed during the training. Depending on the task, more edge types can be easily added.

Representation refinement Our goal is to learn an abstraction function $\alpha \colon \langle V, E, \xi_V, \xi_E \rangle \to \langle V, E' \subseteq E, \xi_V, \xi_E \rangle$ that removes a subset of the edges from the graph. To quantify the size of the abstraction, we use $|\alpha(x)| := |E'|$ to denote the number of edges after applying α on x.

Defining valid graph refinements Because the goal of representation refinement is to reduce the number of nodes on which a prediction depends, we need to ensure that α itself does not depend on all the graph nodes. This is necessary as otherwise we only shift the dependency on the *entire* program from the model f to the representation refinement α. To achieve this, the decision to include or remove a given edge is done *locally*, based only on the edge attributes and attributes of the nodes it connects.

Concretely, for a given edge $\langle s, t \rangle \in E$, we define an edge feature $\phi(\langle s, t \rangle) := \langle \xi_E(\langle s, t \rangle), \xi_V(s), \xi_V(t) \rangle$ to be a tuple of the edge attributes and attributes of the nodes it connects. As a form of regularization, we condition only on the *type* attribute of each node. We denote the set of all possible edge features Φ to be the range of the function ϕ evaluated over all edges in $\mathcal{D}$. Further, we define the refinement α as a subset of edge features $\alpha \subseteq \Phi$. Finally, the semantics of executing α over edges E is that only edges whose features are in α are kept, i.e., $\{e \mid e \in E \wedge \phi(e) \in \alpha\}$.

Problem statement Minimize the expected size of the refinement $\alpha \subseteq \Phi$ subject to the constraint that the expected loss of the model f stays approximately the same:

$$\underset{\alpha \subseteq \Phi}{\arg\min} \sum_{(x,y) \in \mathcal{D}} |\alpha(x)| \tag{2}$$

subject to

$$\textstyle\sum_{(x,y)\in\mathcal{D}} \ell(f(x), y) \approx \sum_{(x,y)\in\mathcal{D}} \ell(f(\alpha(x)), y)$$

Our problem statement is quite general and can be instantiated by: *(i)* using $\ell_{\texttt{AbstainCrossEntropy}}$ as the loss (Appendix B), and *(ii)* using *adversarial risk* (Appendix C).

Allowing the model to abstain from making predictions is especially important in order to obtain small α (i.e., sparse graphs). This is because the restriction that the model accuracy is roughly the same is otherwise too strict and would require that most edges are kept. Further, note that the problem formulation is defined over *all* samples in $\mathcal{D}$, not only those for which the model f predicts the correct label. This is necessary since the model needs to make a prediction for all samples, even if that prediction is to abstain.

Optimization via integer linear programming (ILP) To solve Equation 8, the key idea is that for each sample $(x, y) \in \mathcal{D}$ we first capture the relevance of each node to the prediction made by the model f by computing:

$$\boldsymbol{a}(f, x, y) = \left[\|\mathbf{G}_{i,:}\|_1, \ldots, \left\|\mathbf{G}_{|p|,:}\right\|_1 \right],$$

where $\mathbf{G} = \nabla_x \, \ell(f(x), y) \in \mathbb{R}^{|p| \times emb}$ denotes the gradient with respect to the input $x = \langle p, l \rangle$ and a given prediction y. As positions in p correspond to discrete words, the gradient is computed with respect to their embedding $emb \in \mathbb{R}$. The score for each position in p is computed by applying the L^1-norm over the embedding gradients, producing a vector of unnormalized scores $\boldsymbol{a} \in \mathbb{R}^{|p|}$. To obtain a probability distribution $\hat{\boldsymbol{a}}(f, x, y)$ over all positions in p, we normalize the entries in $\boldsymbol{a}$ accordingly.

Then, we phrase the solution of Equation 8 as an optimization problem of including the minimum number of edges necessary for a path to exist between every relevant node (according to $\hat{\boldsymbol{a}}$) and the node where the prediction is made. Preserving all paths between the prediction and relevant nodes encodes the constraint that the expected loss stays approximately the same, since it allows propagating information throughout the graph neural network. This optimization can be naturally encoded as minimum-cost maximum-flow problem and solved efficiently with off-the-shelf ILP solvers. We provide formal definition of the ILP encoding as well as concrete examples in Appendix D.

Even though our ILP formulation is very fast (in all our experiments the ILP solver takes less than a second), it does result in a more complex approach compared to an end-to-end trainable solution. We note however that an end-to-end trainable solution is also possible. For example, one

Algorithm 1 Training procedure used to learn a single adversarially robust model $\langle f, g_h, \alpha \rangle$.

1: **function** $\texttt{RobustTrain}(\mathcal{D}, t_{\texttt{acc}})$:
2: $\quad \alpha_{\texttt{last}} \leftarrow \Phi$
3: $\quad f, g_h \leftarrow \texttt{Train}\ (\mathcal{D}, t_{acc} - \epsilon)$
4: $\quad$ **while** $true$ **do**
5: $\quad\quad \alpha \leftarrow \texttt{RefineRepresentation}\ (\mathcal{D}, f, g_h)$
6: $\quad\quad$ **if** $|\alpha| \geq |\alpha_{\texttt{last}}|$ **then break**
7: $\quad\quad \alpha_{\texttt{last}} \leftarrow \alpha$
8: $\quad\quad \mathcal{D} \leftarrow \{(\alpha(x), y) \mid (x, y) \in \mathcal{D}\}$
9: $\quad\quad f, g_h \leftarrow \texttt{AdversarialTrain}\ (\mathcal{D}, f, g_h, t_{\texttt{acc}} - \epsilon)$
10: $\quad$ set threshold h in g_h such that the accuracy is $t_{\texttt{acc}}$
11: $\quad$ **return** $\langle f, g_h, \alpha \rangle$

could make α continuous by defining a learnable weight for each edge feature ϕ, encode the sparsity on α as part of the loss, and extend the graph neural network such that each message propagated along an edge e is scaled using the corresponding value of the edge feature $\phi(e)$. We have explored this option in the work of (Abstreiter et al., 2020).

4. Training Algorithm

We now describe our algorithm that combines learning to abstain, adversarial training and representation refinement.

Training a single adversarially robust model The training procedure used to learn a single adversarially robust model is shown in Algorithm 1. The input is a training dataset $\mathcal{D}$ and the desired accuracy $t_{\texttt{acc}}$ that the learned model should have. Here, setting $t_{\texttt{acc}} = 1.0$ corresponds to a model that makes no mis-prediction (i.e., 100% accuracy) while $t_{\texttt{acc}} = 0$ corresponds to training a model that never abstains.

We start by training a model f and a selection function g_h as described in Appendix B (line 3). At this point we do not use adversarial training and train with a weaker threshold $t_{\texttt{acc}} - \epsilon$, as our goal is only to obtain a fast approximation of the samples that can be predicted with high certainty. We use f and g_h to obtain an initial representation refinement α (line 5) which is applied to the dataset $\mathcal{D}$ to remove edges that are not relevant according to f and g_h (line 8). After that, we perform adversarial training (line 9) as described in Appendix C. However, instead of training from scratch, we reuse model f and g_h learned so far, which speeds-up training. Next, we refine the representation again (line 5) and if the new representation is smaller (line 6), we repeat the whole process. Note that the adversarial training also uses threshold $t_{\texttt{acc}} - \epsilon$ to account for the fact that the suitable representation is not known in advance. After the training loop finishes, we set the threshold h used by g_h to match the desired accuracy $t_{\texttt{acc}}$ (more details on this step are provided in Appendix B). The final result is a model consisting of the function f trained to make adversarially robust predictions, the selection function g_h and the abstraction α.

Algorithm 2 Training multiple adversarially robust models, each of which learns to make predictions for a different subset of the dataset $\mathcal{D}$.

function $\texttt{AccurateAndRobustTrain}(\mathcal{D}, t_{\texttt{acc}} = 1.0)$
$\quad M \leftarrow []$
$\quad$ **while** $true$ **do**
$\quad\quad \langle f, g_h, \alpha \rangle \leftarrow \texttt{RobustTrain}(\mathcal{D}, t_{\texttt{acc}})$
$\quad\quad \mathcal{D}_{\texttt{abstain}} \leftarrow \texttt{Apply}(\mathcal{D}, f, g_h, \alpha)$
$\quad\quad$ **if** $|\mathcal{D}_{\texttt{abstain}}| = |\mathcal{D}|$ **then break**
$\quad\quad \mathcal{D} \leftarrow \mathcal{D}_{\texttt{abstain}}$
$\quad\quad M \leftarrow M \cdot \langle f, g_h, \alpha \rangle$
$\quad$ **return** M

Incorporating robust predictions Once a single model is learned, it makes robust predictions on a subset of the dataset $\mathcal{D}_{\texttt{predict}} = \{(x, y) \mid (x, y) \in \mathcal{D} \wedge g_h(\alpha(x)) \geq h\}$ and abstains from making a prediction on the remainder of the samples $\mathcal{D}_{\texttt{abstain}} = \mathcal{D} \setminus \mathcal{D}_{\texttt{predict}}$. Next, for all samples in $\mathcal{D}_{\texttt{predict}}$, we use the learned model to annotate the position l in the program p (recall that each $x = \langle p, l \rangle$ consists of a program p and a position l) with the ground-truth label y (denoted as `Apply` in Algorithm 2). Annotating a program position corresponds to either defining a new attribute (as illustrated in Figure 2*e*) or replacing an existing attribute (e.g., the *value* attribute) of a given node. Note that annotating programs is useful only in cases where the same program p is shared by multiple samples $(x, y) \in \mathcal{D}$ (i.e., multiple predictions are computed for different positions in the same program).

Main training algorithm Our main training algorithm is shown in Algorithm 2. It takes as input the training dataset $\mathcal{D}$ and learns multiple models M, each of which makes robust predictions on a different subset of $\mathcal{D}$ (as motivated in Section 2). The number of models and the subsets for which they make predictions is not fixed a priori and is learned as part of our training. Model training (line 4) and model application (line 5) are performed as long as a non-empty robust model exists (i.e., it makes at least one prediction). If the goal is to make predictions for all the samples in $\mathcal{D}$, the Algorithm 2 is run iteratively, with decreasing values of $t_{\texttt{acc}}$ until the full dataset is covered.

Verifying model correctness A natural extension of our approach is to formally verify that the learned models are correct. Even though formally verifying the correctness of all samples is typically infeasible, it is possible to verify a subset of them. This can be achieved since using representation refinement significantly simplifies the problem of proving correctness of *all* positions (nodes) in the program

to a much smaller set of relevant positions. In fact, for some cases the refined representation is so small that it is possible to simply enumerate all valid modifications (e.g., a finite set of valid variable renamings) and check that the model is correct for all of them. Additionally, it would be possible to adapt the recently proposed techniques (Huang et al., 2019; Jia et al., 2019), based on Interval Bound Propagation, that verify robustness to any valid word renaming and word substitution modifications. However, applying these techniques to realistic networks in a scalable and precise ways is an open problem beyond the scope of our work.

5. Evaluation

We instantiated our approach to a well studied task – predicting types for dynamically typed languages `JavaScript` and `TypeScript` (Hellendoorn et al., 2018; Schrouff et al., 2019; Malik et al., 2019; Raychev et al., 2015). In this task, the need for model robustness is natural since the model is queried each time a program is modified by the user. Our key results are:

- *Our approach learns accurate and adversarially robust models* for the task of type inference, achieving 87.7% accuracy while improving robustness from 52.1% to 67.0%.
- We train highly accurate and robust models *for a subset of the dataset*, with 99.9% accuracy and 99.9% robustness for 29% of the samples.

Our implementation uses PyTorch (Paszke et al., 2019) and DGL library (Wang et al., 2019). We used a single Nvidia TITAN RTX for all the experiments. For our dataset, we collect the same top starred projects on Github and perform similar preprocessing steps as Hellendoorn et al. We provide detailed description in the supplementary material. The code and datasets are available at:

`https://github.com/eth-sri/robust-code`

Evaluation metrics We use two main evaluation metrics:

Accuracy is computed over the unmodified dataset $\mathcal{D}$ and corresponds to the accuracy used in prior works. Concretely, the accuracy is defined as the ratio of samples (x, y) for which the most likely label according to the model f, denoted $f(x)_{\texttt{best}}$, is the same as the ground truth label y:

$$\frac{1}{|\mathcal{D}|}\sum_{(x,y)\in\mathcal{D}}\begin{cases}1 & \text{if } f(x)_{\texttt{best}} = y\\ 0 & \text{otherwise}\end{cases}$$

Robustness is the ratio of samples $(x, y) \in \mathcal{D}$ for which the model f evaluated on all valid modifications $\boldsymbol{\delta} \subseteq \Delta(x)$ either abstains or makes a correct prediction:

$$\frac{1}{|\mathcal{D}|}\sum_{(x,y)\in\mathcal{D}}\begin{cases}0 & \text{if } \exists_{\boldsymbol{\delta}\subseteq\Delta(x)} f(x+\boldsymbol{\delta})_{\texttt{best}} \notin \{y, \texttt{abstain}\}\\ 1 & \text{otherwise}\end{cases}$$

Models We evaluate five neural model architectures:

`LSTM` is a bidirectional LSTM with attention which takes as input a sequence of AST nodes, including both types and values, obtained using pre-order traversal.

`DeepTyper` is a model proposed by Hellendoorn et al. and consists of a bidirectional LSTM layer, followed by a single layer graph neural network that connects all variables with the same name (referred as consistency layer), followed by another bidirectional LSTM layer. Our only modification is that the input to our model is a sequence of AST types and values, instead of using syntactic program tokens.

`GCN`, `GGNN` and `GNT` are three graph neural networks that use as input the graph program representation described in Section 3. Here, `GCN` is a Graph Convolutional Network (Kipf & Welling, 2017), `GGNN` is Gated Graph Neural Network (Li et al., 2016) and `GNT` is a graph implementation of a recently proposed transformer neural network architecture (Vaswani et al., 2017; Dehghani et al., 2019).

All models were trained with an embedding and hidden size of 128, batch size of 32, dropout 0.1 (Srivastava et al., 2014), initial learning rate of 0.001, using Adam optimizer (Kingma & Ba, 2014) and between 10 to 20 epochs.

Reducing dependencies via dynamic halting We further strengthen our `GNT` model by implementing the Adaptive Computation Time (ACT) (Graves, 2016) which dynamically learns how many computational steps each node requires in order to make a prediction. This is in contrast to using a fixed amount of steps as done in (Allamanis et al., 2018; Brockschmidt et al., 2019). In our experiments, ACT significantly reduces the number of steps each node performs (half of the nodes perform ≤ 3 steps).

Program modifications We instantiate the adversarial training with both semantic preserving and label-preserving modifications shown in Table 1. Here, `expr` is either an existing expression or a new expression consisting of a random binary expression over constants up to depth 3, `const` is a randomly selected constant that results in a valid expression and `x`, `y`, `z` are variables in the program scope. Our modifications extend those used by Bielik et al. (2017) but the list is not exhaustive and can be extended further.

To measure the model robustness, we run the adversarial attack for 1000 iterations for renaming modifications and additional 300 iterations for structural modifications. These thresholds are rather high and were selected with the goal

Table 1. Illustration of semantic and label preserving program modifications used in our work.

Substitutions and Renaming	Examples
Semantic Preserving	
variable renaming	$\texttt{x} \rightarrow \texttt{y}$
object field renaming	$\texttt{obj.x} \rightarrow \texttt{obj.y}$
property assignment renaming	$\{\texttt{x} : \texttt{obj}\} \rightarrow \{\texttt{y} : \texttt{obj}\}$
Label Preserving	
number substitution	$\texttt{2} \rightarrow \texttt{7}$
string substitution	$\texttt{"get"} \rightarrow \texttt{"load"}$
boolean substitution	$\texttt{true} \rightarrow \texttt{false}$

Structural Modifications	Examples
Label Preserving	
new function parameters	$\texttt{def inc(x)} \rightarrow \texttt{def inc(x, y)}$
new method arguments	$\texttt{inc(x)} \rightarrow \texttt{inc(x, expr)}$
Semantic Preserving	
ternary expressions	$\texttt{expr}_1 \rightarrow (\texttt{expr})_2 : \texttt{expr}_1 \ ? \ \texttt{expr}_1$
array access	$\texttt{expr} \rightarrow [\texttt{expr}, \texttt{expr}][\texttt{const}]$
Dead Code	
side-effect free expressions	$\emptyset \rightarrow \texttt{expr}$
adding object expressions	$\emptyset \rightarrow \{\texttt{x} : \texttt{y}, \texttt{z} : \texttt{expr}\}$

Table 2. Comparison of accuracy and robustness across various models and training techniques considered in our work for the task of type inference. Adversarial training and the ability to abstain is applicable to all the models. The representation refinement is designed specifically to models defined over graphs, including GCN, GGNN and GNT.

	Standard Training $\ell(f(x), y)$		**Adversarial Training** $\max_{\boldsymbol{\delta} \subseteq \Delta(x)} \ell(f(x+\boldsymbol{\delta}), y)$	
Model	Accuracy	Robustness	Accuracy	Robustness
LSTM	88.2 ± 0.2	44.9 ± 1.3	87.5 ± 0.4	51.9 ± 1.3
DeepTyper	88.4 ± 0.2	52.4 ± 1.2	87.1 ± 0.3	55.1 ± 2.6
GCN	82.6 ± 0.6	49.1 ± 1.1	81.9 ± 0.5	49.3 ± 3.1
GNT	89.3 ± 0.9	47.4 ± 1.0	88.3 ± 0.4	50.0 ± 0.5
GGNN	86.7 ± 0.4	52.1 ± 0.4	86.1 ± 0.2	57.9 ± 1.5

Abstain + Adversarial + Refinement $\max_{\boldsymbol{\delta} \subseteq \Delta(x)} \ell_{\texttt{AbstainCE}}((f, g_h)(\alpha(x+\boldsymbol{\delta})), y)$		
Model	Accuracy	Robustness
$t_{\texttt{acc}} = 1.00$ (Abstain $\approx 70\%$)		
GNT	99.93%	99.98%
GGNN	99.80%	99.01%
$t_{\texttt{acc}} = 0.00$		
GNT	86.6%	62.3%
GGNN	87.7%	67.0%

of closely estimating the true number of adversarial samples. Further, note that since $\boldsymbol{\delta} \subseteq \Delta(x)$ is a set, each iteration explores a set of concrete program modifications.

5.1. Accurate and Adversarially Robust Models

We summarize the main results in Table 2. The first column (left) shows the median test *accuracy* and standard deviation of various models (across three trials trained with different random seeds). The GCN achieves the worst accuracy of 82.6% and the accuracy of the remaining models is similar with GNT model performing the best with 89.3%.

Existing models are not robust While highly accurate, all models are also non-robust for up to half of the samples in the dataset. In other words, for every second sample x in our dataset, there exists a modification $\boldsymbol{\delta} \subseteq \Delta(x)$ for which $f(x)$ predicts the type correctly while $f(x+\boldsymbol{\delta})$ mispredicts it. However, since these models were not trained with the goal of adversarial robustness, it is expected for them to be (atleast partially) non-robust.

Adversarial training alone is insufficient To improve the robustness, we next train the models using adversarial training as described in Appendix C. Unfortunately, while the adversarial training increase the robustness, it does so only slightly. The best improvement was achieved for LSTM and GGNN models (7% and 5.8%, respectively). For DeepTyper and GNT the robustness increased by $\approx 2.5\%$ while for GCN it is only 0.2%. This illustrates that while useful, if used alone, *adversarial training* is not enough.

Our work: training accurate models with abstain The models trained using our approach are shown in Table 2 (right). First, we trained our models to be both *accurate* and *robust* on a subset of the dataset. This can be achieved by setting the desired accuracy thresholds, in our case $t_{acc} = 1.00$, which corresponds to training the model to make only correct predictions. For $t_{acc} = 1.00$, our approach learns an almost perfect model that is both accurate and robust for $\approx 30.0\%$ of samples. Here, GNT learned 7 models and achieved 99.98% robustness while GGNN learned 8 models with robustness of 99.01%. Learning multiple models is crucial for achieving higher coverage as a single model would not abstain for only $17 - 20\%$ of the samples, compared to 30% using multiple models.

The model did not achieve 100% accuracy and robustness for $t_{acc} = 1.00$ due to several samples included in the test set. These samples were mis-predicted because they con-

Table 3. Robustness breakdown for the GNT and GGNN models trained using our approach from Table 2 (right).

Dataset	Size	Robustness		
		∀ Correct	∃ Incorrect	Abstain
GNT	$t_{\texttt{acc}} = 1.00$			
$\mathcal{D}_{\texttt{correct}}$	29.3%	90.0%	0.00%	10.00%
$\mathcal{D}_{\texttt{abstain}}$	70.6%	–	0.01%	99.99%
GGNN	$t_{\texttt{acc}} = 1.00$			
$\mathcal{D}_{\texttt{correct}}$	30.6%	75.5%	0.06%	23.94%
$\mathcal{D}_{\texttt{abstain}}$	69.3%	–	1.46%	98.54%

$\forall$ Correct $:= \forall_{\boldsymbol{\delta} \subseteq \Delta(x)} (f, g_h)(\alpha(x + \boldsymbol{\delta}))_{\texttt{best}} = y$
$\exists$ Incorrect $:= \exists_{\boldsymbol{\delta} \subseteq \Delta(x)} (f, g_h)(\alpha(x + \boldsymbol{\delta}))_{\texttt{best}} \notin \{y, \texttt{abstain}\}$

tained code structure not seen during training and not covered by modifications $\boldsymbol{\delta} \subseteq \Delta(x)$. This illustrates that it is important that the samples in $\mathcal{D}$ are diverse and contain all the language features and corner cases of the programs, or that the modifications $\Delta(x)$ are expressive enough such that these can be discovered automatically during training.

Our work: improving robustness Next, we train models that take advantage of the highly accurate and robust models trained using $t_{acc} = 1.00$, but make predictions for all the samples (i.e., do not abstain). This can be achieved by continuing the training while reducing t_{acc} to zero and conditioning on all the models trained with higher t_{acc}. In our experiments, we train a single additional model by directly setting $t_{acc} = 0$ after training with $t_{acc} = 1.00$. The results are shown in Table 2 (right) and lead to additional robustness increase of 9.2% and 12.3% compared to using adversarial training only for GGNN and GNT, respectively. For GNT, the accuracy slightly decreases by 1.7% which is expected as increasing model robustness typically comes at the cost of reduced accuracy (Tsipras et al., 2019). Interestingly, for GGNN our robust training increases the accuracy over both the adversarial training as well as standard training by 1.9% and 1.0%, respectively.

Adversarial robustness breakdown Table 3 provides a detailed breakdown of the *robustness* metric for the GNT and GGNN models trained with $t_{acc} = 1.00$ from Table 2 (right). Here, $\mathcal{D}_{\texttt{abstain}}$ contains samples for which the model abstains from making a prediction and $\mathcal{D}_{\texttt{correct}}$ contains samples for which the model evaluated on a non-adversarial input (i.e., x without any modification) makes a correct prediction. We use $\forall$ correct to denote that a sample (x, y) is correct for *all* possible modifications $\boldsymbol{\delta} \subseteq \Delta(x)$, the $\exists$ incorrect has the same definition as robustness (i.e., there exists a modification that leads to an incorrect prediction), and abstain denotes the remaining samples.

The GNT is precise and keeps predicting the correct label in 90% of cases and abstain in the rest. This is even though the requirements for $\forall$ correct are very strict and require that all samples are correct. When considering $\mathcal{D}_{\texttt{abstain}}$, the GNT model is also precise and produces incorrect prediction for only a single sample (0.01%). For GGNN the results are similar but the model is both less precise (keeps the correct prediction in 75.5% of cases) and less robust (1.46% of samples in $\mathcal{D}_{\texttt{abstain}}$ can be modified to cause a mis-prediction). This shows that the majority of robustness errors from Table 2 are due to mis-predicted samples for which the model originally abstained.

6. Related Work

Our work is related to a number of different areas from adversarial machine learning and learning over code.

Model certainty Several approaches have been recently proposed to extend neural models with certainty measure (Gal & Ghahramani, 2016; Liu et al., 2019; Gal, 2016; Geifman & El-Yaniv, 2017; 2019). In our work, we use the method proposed by Liu et al. (2019) but in a novel way – applied to the adversarial setting with the goal of training robust models.

Learning static analyzers from data A closely related work addresses the task of learning static analyzers (Bielik et al., 2017): it defines a domain specific language (DSL) to represent static analyzers, uses decision tree learning to obtain an interpretable model, and defines a procedure that finds counter-examples the model mis-classifies (used to re-train the model). At a high-level, some of the steps are similar but the actual technical solution is very different as we address a general class of neural models and do not assume any prior knowledge (i.e., a DSL).

Adversarial training Even though the problem of adversarial robustness of code has been overlooked, the adversarial training has already been applied to related domains – natural language processing (Miyato et al., 2017; Papernot et al., 2016; Gao et al., 2018; Liang et al., 2018; Belinkov & Bisk, 2017; Ebrahimi et al., 2018) and graphs (Dai et al., 2018; Zügner et al., 2018; Zügner & Günnemann, 2019).

In the domain of *graphs*, existing works focus on attacking the graph structure (Dai et al., 2018; Zügner et al., 2018; Zügner & Günnemann, 2019) by considering that the nodes are fixed and edges can be added or removed. While this setting is natural for modelling many types of graphs, such approaches do not apply for the domain of code where graph edges can not be added and removed arbitrarily.

In *natural language processing*, existing approaches generally: *(i)* measure the contribution of individual words or

characters to the prediction (e.g., using gradients (Liang et al., 2018), forward derivatives (Papernot et al., 2016) or head/tail scores (Gao et al., 2018)), and *(ii)* replace or remove those whose contribution is high (e.g., using dictionaries (Jia et al., 2019), character level typos (Gao et al., 2018; Belinkov & Bisk, 2017; Ebrahimi et al., 2018), or handcrafted strategies (Liang et al., 2018)). The adversarial training used in our work operates similarly except our modifications are designed over programs.

Program representations A core challenge of using machine learning for code is designing a suitable program representation used as model input. Due to its simplicity, the most commonly used program representation is a sequence of words, obtained either by tokenizing the program (Hellendoorn et al., 2018) or by linearizing the abstract syntax tree (Li et al., 2018). This however ignores the fact that programs do have a rich structure – an issue addressed by representing programs as graphs (Allamanis et al., 2018; Brockschmidt et al., 2019) or as a combination of abstract syntax tree paths (Alon et al., 2019). In our work, we follow the approach proposed in recent works and represent programs as graphs. More importantly, we develop a novel technique that learns to refine the representation based on model predictions instead of fixing it a priori. As shown in our evaluation, this is crucial for learning robust models.

Adversarial attacks for code Concurrent to our work, Yefet et al. (2019) explored the task of generating adversarial attacks for code via gradient based optimization. In contrast, we introduce an approach to reduce the search space an adversarial attack needs to consider by learning to refine the representation. Such reduced search space is useful for both for renaming and structural modifications, whereas gradient based optimization has been explored only for renaming. However, both of these approaches are orthogonal and can be combined into one that learns both to reduce the search space as well as to efficiently find adversarial examples in this reduced search space.

Type inference We evaluated our work on the task of type inference for which a number of recent works improve accuracy by proposing a new neural architectures. In contrast, the goal of our work is to study and improve robustness of these models. To achieve this, we compare to two prior works in our evaluation (Schrouff et al., 2019; Hellendoorn et al., 2018). In addition to predicting types from source code, Malik et al. (2019) showed that it is possible to predict parameter types using natural language information obtained from method docstrings. Here, existing attacks on text (LSTM) can be used to assess its robustness but evaluating text models is outside the scope of our work. Finally, two concurrent works to ours have proposed new models to improve accuracy: `Typilus` (Allamanis et al., 2020) and `LambdaNet` (Wei et al., 2020). Both of these works represent programs as graphs and use graph neural networks as the underlying model architecture, which makes our approach applicable. However, we note that for `LambdaNet` we expect the model to be quite robust as the authors manually designed a sparse graph representation (by designing a static analysis to extract the type dependence graph) over which to learn.

7. Conclusion

We presented a new technique to train *accurate* and *robust* neural models of code. Our work addresses two key challenges inherent to the domain of code: the difficulty of computing the correct label for all samples (i.e., the input is incomplete code snippet, program semantics are unknown) as well as the fact that programs are significantly larger and more structured compared to images or natural language.

To address the first challenge, we allow the model to abstain from making a prediction, rather than forcing the model to make predictions for all samples (as done in prior works). To address the second challenge, we learn which parts of the program are relevant for the prediction, and abstract the rest (instead of using the *entire* program as input).

Further, we introduce a new procedure that trains multiple models, instead of one. This has several advantages, as each model is simpler and thus easier to train robustly, the learned representation is specialized to the kind of predictions it makes, and the model directly conditions on predictions of prior models (instead of having to re-learn them). However, a disadvantage of our approach is that the models are learned sequentially which slows down the training (i.e., training 10 models will take $10\times$ more time). To speed up the training, it would be interesting to allow learning multiple models in parallel at each sequential step and then combine them as explored by Shazeer et al. (2017).

We believe than our work is only one step in addressing the task of adversarially robust models of code and that many challenges remain open. For example, it remains to be seen how effective our approach is at other tasks over code, beyond type inference. Further, we optimize for the worst case adversarial robustness, which corresponds to learning a robust model for all programs. An interesting future work is to optimize with respect to those modification that are common among developers, especially if it is not possible to be robust for all of them. While we checked robustness for a wide range of program modifications, these are still far from exhaustive and more work is needed in defining new ones. Finally, as the number of possible modification is large and growing, an interesting area is designing how they can be combined efficiently, as explored recently by Ramakrishnan et al. (2020) and Zhang et al. (2020).

Acknowledgements

We would like to thank the anonymous reviewers who gave useful comments and provided interesting suggestions on how our work can be improved and extended. Further, we would like to acknowledge the work of (Hellendoorn et al., 2018) which is publicly available and provided useful infrastructure for generating datasets used in our work. The research leading to these results was partially supported by an ERC Starting Grant 680358.

References

Abstreiter, K., Bielik, P., and Vechev, M. Improving robustness for models of code via sparse graph neural networks. Technical report, ETH Zurich, june 2020. URL https://www.research-collection.ethz.ch/handle/20.500.11850/431559.

Allamanis, M., Peng, H., and Sutton, C. A convolutional attention network for extreme summarization of source code. In *International Conference on Machine Learning (ICML)*, 2016.

Allamanis, M., Brockschmidt, M., and Khademi, M. Learning to represent programs with graphs. In *International Conference on Learning Representations*, ICLR'18, 2018.

Allamanis, M., Barr, E. T., Ducousso, S., and Gao, Z. Typilus: Neural type hints. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI'20, pp. 91–105, 2020.

Alon, U., Zilberstein, M., Levy, O., and Yahav, E. Code2Vec: Learning distributed representations of code. In *Proceedings of the 46st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, volume 3 of *POPL '19*, pp. 40:1–40:29, 2019.

Belinkov, Y. and Bisk, Y. Synthetic and natural noise both break neural machine translation. *CoRR*, abs/1711.02173, 2017.

Bielik, P., Raychev, V., and Vechev, M. Learning a static analyzer from data. In *International Conference on Computer Aided Verification*, CAV'17, pp. 233–253, 2017.

Brockschmidt, M., Allamanis, M., Gaunt, A. L., and Polozov, O. Generative code modeling with graphs. In *International Conference on Learning Representations*, ICLR'19, 2019.

Dai, H., Li, H., Tian, T., Huang, X., Wang, L., Zhu, J., and Song, L. Adversarial attack on graph structured data. In *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *ICML'18*, pp. 1115–1124. PMLR, 2018.

Dehghani, M., Gouws, S., Vinyals, O., Uszkoreit, J., and Kaiser, L. Universal transformers. In *International Conference on Learning Representations*, ICLR'19, 2019.

Ebrahimi, J., Rao, A., Lowd, D., and Dou, D. HotFlip: White-box adversarial examples for text classification. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, ACL'18, pp. 31–36, 2018.

Fernandes, P., Allamanis, M., and Brockschmidt, M. Structured neural summarization. In *International Conference on Learning Representations*, ICLR'19, 2019.

Gal, Y. *Uncertainty in Deep Learning*. PhD thesis, University of Cambridge, Department of Engineering, 9 2016.

Gal, Y. and Ghahramani, Z. Dropout as a bayesian approximation: Representing model uncertainty in deep learning. In *Proceedings of The 33rd International Conference on Machine Learning*, volume 48 of *ICML'16*, pp. 1050–1059, 2016.

Gao, J., Lanchantin, J., Soffa, M. L., and Qi, Y. Black-box generation of adversarial text sequences to evade deep learning classifiers. *CoRR*, abs/1801.04354, 2018.

Geifman, Y. and El-Yaniv, R. Selective classification for deep neural networks. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NeurIPS'17, pp. 4885–4894, 2017.

Geifman, Y. and El-Yaniv, R. SelectiveNet: A deep neural network with an integrated reject option. In *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *ICML'19*, pp. 2151–2159, 2019.

Goodfellow, I. J., Shlens, J., and Szegedy, C. Explaining and harnessing adversarial examples. In *3rd International Conference on Learning Representations*, ICLR'15, 2015.

Graves, A. Adaptive computation time for recurrent neural networks. *CoRR*, abs/1603.08983, 2016.

Gurobi Optimization, L. Gurobi optimizer reference manual, 2020. URL http://www.gurobi.com.

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*, ESEC/FSE'18, 2018.

Huang, P.-S., Stanforth, R., Welbl, J., Dyer, C., Yogatama, D., Gowal, S., Dvijotham, K., and Kohli, P. Achieving verified robustness to symbol substitutions via interval bound propagation. In *Empirical Methods in Natural Language Processing*, EMNLP'19, 2019.

Jia, R., Raghunathan, A., Göksel, K., and Liang, P. Certified robustness to adversarial word substitutions. In *Empirical Methods in Natural Language Processing*, EMNLP'19, 2019.

Kingma, D. and Ba, J. Adam: A method for stochastic optimization. In *International Conference on Learning Representations*, ICLR'14, 2014.

Kipf, T. N. and Welling, M. Semi-supervised classification with graph convolutional networks. In *International Conference on Learning Representations*, ICLR'17, 2017.

Li, J., Wang, Y., Lyu, M. R., and King, I. Code completion with neural attention and pointer networks. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence*, IJCAI'18, pp. 4159–25, 2018.

Li, Y., Zemel, R., Brockschmidt, M., and Tarlow, D. Gated graph sequence neural networks. In *International Conference on Learning Representations*, ICLR'16, 2016.

Li, Y., Wang, S., Nguyen, T. N., and Van Nguyen, S. Improving bug detection via context-based code representation learning and attention-based neural networks. *Proc. ACM Program. Lang.*, (OOPSLA):162:1–162:30, 2019.

Liang, B., Li, H., Su, M., Bian, P., Li, X., and Shi, W. Deep text classification can be fooled. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence*, IJCAI'18, pp. 4208–4215, 2018.

Liu, Z., Wang, Z., Liang, P. P., Salakhutdinov, R. R., Morency, L.-P., and Ueda, M. Deep gamblers: Learning to abstain with portfolio theory. In *Advances in Neural Information Processing Systems 32*, NeurIPS'19, pp. 10622–10632. 2019.

Madry, A., Makelov, A., Schmidt, L., Tsipras, D., and Vladu, A. Towards deep learning models resistant to adversarial attacks. In *International Conference on Learning Representations*, ICLR'18, 2018.

Malik, R. S., Patra, J., and Pradel, M. NL2Type: Inferring javascript function types from natural language information. In *Proceedings of the 41st International Conference on Software Engineering*, ICSE '19, pp. 304–315, 2019.

Miyato, T., Dai, A. M., and Goodfellow, I. Adversarial training methods for semi-supervised text classification. In *International Conference on Learning Representations*, ICML'17, 2017.

Mou, L., Li, G., Zhang, L., Wang, T., and Jin, Z. Convolutional neural networks over tree structures for programming language processing. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, AAAI'16, pp. 1287–1293, 2016.

Papernot, N., McDaniel, P. D., Swami, A., and Harang, R. E. Crafting adversarial input sequences for recurrent neural networks. *CoRR*, abs/1604.08275, 2016.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S. PyTorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, NeurIPS'19, pp. 8024–8035. 2019.

Pradel, M. and Sen, K. DeepBugs: A learning approach to name-based bug detection. *Proc. ACM Program. Lang.*, (OOPSLA):147:1–147:25, 2018.

Raghunathan, A., Steinhardt, J., and Liang, P. Certified defenses against adversarial examples. In *International Conference on Learning Representations*, ICLR'18, 2018.

Ramakrishnan, G., Henkel, J., Wang, Z., Albarghouthi, A., Jha, S., and Reps, T. Semantic robustness of models of source code, 2020.

Raychev, V., Vechev, M., and Krause, A. Predicting program properties from "Big Code". In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pp. 111–124, 2015.

Schrouff, J., Wohlfahrt, K., Marnette, B., and Atkinson, L. Inferring javascript types using graph neural networks. In *Representation Learning on Graphs and Manifolds. ICLR Workshop*, 2019.

Shazeer, N., Mirhoseini, A., Maziarz, K., Davis, A., Le, Q. V., Hinton, G. E., and Dean, J. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. *CoRR*, abs/1701.06538, 2017.

Sinha, A., Namkoong, H., and Duchi, J. Certifiable distributional robustness with principled adversarial training. In *International Conference on Learning Representations*, ICLR'18, 2018.

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. Dropout: A simple way to prevent neural networks from overfitting. *J. Mach. Learn. Res.*, 15(1):1929–1958, January 2014. ISSN 1532-4435.

Szegedy, C., Zaremba, W., Sutskever, I., Bruna, J., Erhan, D., Goodfellow, I., and Fergus, R. Intriguing properties of neural networks. In *International Conference on Learning Representations*, ICLR'14, 2014.

Tsipras, D., Santurkar, S., Engstrom, L., Turner, A., and Madry, A. Robustness may be at odds with accuracy. In *International Conference on Learning Representations*, ICLR'19, 2019.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. Attention is all you need. In *Advances in Neural Information Processing Systems 30*, NeurIPS'17, pp. 5998–6008. 2017.

Veličković, P., Cucurull, G., Casanova, A., Romero, A., Liò, P., and Bengio, Y. Graph Attention Networks. *International Conference on Learning Representations*, 2018.

Wang, M., Yu, L., Zheng, D., Gan, Q., Gai, Y., Ye, Z., Li, M., Zhou, J., Huang, Q., Ma, C., Huang, Z., Guo, Q., Zhang, H., Lin, H., Zhao, J., Li, J., Smola, A. J., and Zhang, Z. Deep Graph Library: Towards efficient and scalable deep learning on graphs. *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019.

Wei, J., Goyal, M., Durrett, G., and Dillig, I. LambdaNet: Probabilistic type inference using graph neural networks. In *International Conference on Learning Representations*, ICLR'20, 2020.

Wong, E. and Kolter, Z. Provable defenses against adversarial examples via the convex outer adversarial polytope. In *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *ICML'18*, pp. 5286–5295, 2018.

Wu, F., Souza, A., Zhang, T., Fifty, C., Yu, T., and Weinberger, K. Simplifying graph convolutional networks. In *Proceedings of the 36th International Conference on Machine Learning*, ICML'19, pp. 6861–6871. PMLR, 2019.

Yefet, N., Alon, U., and Yahav, E. Adversarial examples for models of code, 2019.

Zhang, J., Wang, X., Zhang, H., Sun, H., Wang, K., and Liu, X. A novel neural source code representation based on abstract syntax tree. In *Proceedings of the 41st International Conference on Software Engineering*, ICSE '19, pp. 783–794, 2019.

Zhang, Y., Albarghouthi, A., and D'Antoni, L. Robustness to programmable string transformations via augmented abstract training. In *Proceedings of the 37th International Conference on Machine Learning*, ICML'20, 2020.

Zügner, D. and Günnemann, S. Adversarial attacks on graph neural networks via meta learning. In *International Conference on Learning Representations*, ICLR'19, 2019.

Zügner, D., Akbarnejad, A., and Günnemann, S. Adversarial attacks on neural networks for graph data. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, KDD '18, pp. 2847–2856, 2018.

Supplementary Material

We provide the following four appendices:

- Appendix A provides details of our dataset and additional experiments that evaluates the effect of dataset size.
- Appendix B describes the method (introduced by Liu et. al. 2019) used in our work for training neural models that abstain from making predictions if uncertain.
- Appendix C describes application of the adversarial training in the domain of code via a set of program mutations.
- Appendix D provides a formal definition of the integer linear encoding used to solve the problem in Equation 2 efficiently. Additionally, we provide a concrete example illustrating the encoding.

A. Evaluation

Implementation All our models are implemented in PyTorch (Paszke et al., 2019). The graph neural networks are implemented using the DGL library v0.4.3 (Wang et al., 2019). To solve the integer linear program we use Gurobi solver v8.11 (Gurobi Optimization, 2020).

Adaptive computation time (ACT) (Graves, 2016) Our `GNT` model implements the Adaptive Computation Time (ACT) (Graves, 2016) technique which dynamically learns how many computational steps each node requires in order to make a prediction. This is instead of using a fixed amount of steps as done for example in (Allamanis et al., 2018; Brockschmidt et al., 2019). To achieve this, recall that for each node $v_i \in V$ in the graph, a graph neural network computes sequence of hidden states $\boldsymbol{s}_t^i$ where $t \in \mathbb{N}$ is the timestep[1]. Following (Graves, 2016), the number of timesteps that the model performs is controlled by introducing an extra sigmoidal halting unit $h_t^i \in \mathbb{R}^{(0,1)}$ with associated learnable weight matrix W_h and bias b_h:

$$h_t^i = \sigma(W_h s_t^i + b_h)$$

The output of the halting unit is then used to determine the halting probability p_t^i as follows:

$$p_t^i = \begin{cases} 1 - \sum_{k=0}^{t-1} h_k^i & \text{if } t = T \text{ (last timestep)} \\ 1 - \sum_{k=0}^{t-1} h_k^i & \text{if } \sum_{k=0}^{t} h_k^i \geq 1 - \epsilon \\ h_t^i & \text{otherwise} \end{cases}$$

[1] Note we assume only that $\boldsymbol{s}_t^i$ is computed for each timestep which is independent of the concrete graph neural architecture used to compute $\boldsymbol{s}_t^i$.

where $T \in \mathbb{N}$ is the maximum allowed number of timesteps and $\epsilon \in \mathbb{R}^{(0,1)}$ is a small constant introduced to allow the network to stop after a single step (we use $\epsilon = 0.01$ in our experiments). Finally, the halting probability p_t^i is used to define the final state $\boldsymbol{s}_T^i$ of a node v_i as a weighted average of its intermediate states:

$$\boldsymbol{s}_T^i = \sum_{t=0}^{T} p_t^i \cdot \boldsymbol{s}_t^i$$

Dataset To obtain the datasets used in our work, we extend the infrastructure from `DeepTyper` (Hellendoorn et al., 2018), collect the same top starred projects on GitHub, and perform similar preprocessing steps – remove `TypeScript` header files, remove files with less than 100 or more than 3,000 tokens and split the projects into train, validation and test datasets such that each project is fully contained in one of the datasets. Additionally, we remove exact file duplicates and files that are similar to each other ($\approx 10\%$ of the files). We measure file similarity by collecting all 3-grams (excluding comments and whitespace) and removing files with Jaccard similarity greater than 0.7.

We compute the ground-truth types using the `TypeScript` compiler version $3.4.5$ based on manual type annotations, library specifications and analyzing all project files. While we reuse the same GitHub projects and part of `DeepTyper`'s infrastructure[2] to obtain the dataset, the datasets are not directly comparable for a number of reasons. First, we fixed a bug due to which `DeepTyper` incorrectly included some type annotations as part of the input. Second, the projects we used are subset of those used in `DeepTyper` since some are no longer available and were removed from GitHub. Third, we additionally predict the types corresponding to all intermediate expressions and constants (e.g., the expression $x + y$ contains three predictions for x, y and $x+y$). This improves model performance as it is explicitly trained also on the intermediate steps required to infer the types. Finally, we train all the models to predict four primitive types (`string`, `number`, `boolean`, `void`), four function types ($() \Rightarrow$ `string`, $() \Rightarrow$ `number`, $() \Rightarrow$ `boolean`, $() \Rightarrow$ `void`) and a special `unk` label denoting all the other types. While this is similar to types predicted by some other works such as `JSNice` (Raychev et al., 2015), it is only subset of types considered in `DeepTyper`.

All results presented in Tables 1 and 2 are obtained by training our models using a dataset that contains 3000 programs split equally between training, validation and test datasets. Because each program contain multiple type predictions, the number of training samples is significantly higher than the number of programs. Concretely, there are $139,915$, $223,912$ and $121,153$ samples in training, validation and test datasets. We note that this is only

[2] https://github.com/DeepTyper/DeepTyper

a subset of the full dataset that can be obtained by processing all the files included in the projects used by Hellendoorn et al. We make the dataset available online at https://github.com/eth-sri/robust-code.

During adversarial training, we explore 20 different modifications $\boldsymbol{\delta} \subseteq \Delta(x)$ applied to each sample $(x, y) \in \mathcal{D}$ which effectively increases dataset size by up to two orders of magnitude since for each training epoch the modifications are different. For the purposes of evaluation, we increase the number of explored modifications to 1300 for each sample – 1000 for renaming modifications and further 300 for renaming together with structural modifications.

B. Training Neural Models to Abstain

We now present a method for training neural models of code that provide an uncertainty measure and can abstain from making predictions. This is important as essentially all practical tasks contain some examples for which it is not possible to make a correct prediction (e.g., due to the task hardness or because it contains ambiguities). In the machine learning literature this problem is known as selective classification (supervised-learning with a reject option) and is an active area with several recently proposed approaches (Gal & Ghahramani, 2016; Liu et al., 2019; Gal, 2016; Geifman & El-Yaniv, 2017; 2019). In our work, we use one of these methods (Liu et al., 2019) which is briefly summarized below. For a full description, we refer the reader to the original paper (Liu et al., 2019).

Let $\mathcal{D} = \{(x_j, y_j)\}_{j=1}^{N}$ be a training dataset and $f\colon \mathbb{X} \to \mathbb{Y}$ an existing model trained to make predictions on $\mathcal{D}$. The existing model f is augmented with an option to abstain from making a prediction by introducing a selection function $g_h\colon \mathbb{X} \to \mathbb{R}^{(0,1)}$ with an associated threshold $h \in \mathbb{R}^{(0,1)}$, which leads to the following definition:

$$(f, g_h)(x) \coloneqq \begin{cases} f(x) & \text{if } g_h(x) \geq h \\ \texttt{abstain} & \text{otherwise} \end{cases} \tag{3}$$

That is, the model makes a prediction only if the selection function g_h is confident enough (i.e., $g_h(x) \geq h$) and abstains from making a prediction otherwise. Although conceptually the model is now defined by two functions f (the original model) and g_h (the selection function), it is possible to adapt the original classification problem such that a single function f' encodes both. To achieve this, an additional `abstain` label is introduced and a function $f'\colon \mathbb{X} \to \mathbb{Y} \cup \{\texttt{abstain}\}$ is trained in the same way as f (i.e., same network architecture, hyper-parameters, etc.) with two exceptions: *(i)* f' is allowed to predict the additional `abstain` label, and *(ii)* the loss function used to train f' is changed to account for the additional label. After f' is obtained, the selection function is defined as $g_h \coloneqq 1 - f'(x)_{\texttt{abstain}}$, that is, to be the probability of selecting any label other than `abstain` according to f'. Then, f is defined to be re-normalized probability distribution obtained by taking the distribution produced by f' and assigning zero probability to `abstain` label. Essentially, as long as there is sufficient probability mass h on labels outside `abstain`, f decides to select one of these labels.

Loss function for abstaining To gain an intuition behind the loss function used for training f', recall that the standard way to train neural networks is to use cross entropy loss:

$$\ell_{\texttt{CrossEntropy}}(\boldsymbol{p}, \boldsymbol{y}) \coloneqq -\sum_{i=1}^{|\mathbb{Y}|} y_i \log(p_i) \tag{4}$$

Here, for a given sample $(x, y) \in \mathcal{D}$, $\boldsymbol{p} = f(x)$ is a vector of probabilities for each of the $|\mathbb{Y}|$ classes computed by the model and $\boldsymbol{y} \in \mathbb{R}^{|\mathbb{Y}|}$ is a vector of ground-truth probabilities. Without loss of generality, assume only a single label is correct, in which case $\boldsymbol{y}$ is a one-hot vector (i.e., $y_j = 1$ if j-th label is correct and zero elsewhere). Then, the cross entropy loss for an example where the *j*-label is correct is $-\log(p_j)$. Further, the loss is zero if the computed probability is $p_j = 1$ (i.e., $-\log(1) = 0$) and positive otherwise.

Now, to incorporate the additional `abstain` label, the abstain cross entropy loss is defined as follows:

$$\ell_{\texttt{AbstainCrossEntropy}}(\boldsymbol{p}, \boldsymbol{y}) \coloneqq -\sum_{i=1}^{|\mathbb{Y}|} y_i \log(p_i o_i + p_{\texttt{abstain}}) \tag{5}$$

Here $\boldsymbol{p} \in \mathbb{R}^{|\mathbb{Y}|+1}$ is a distribution over the classes (including `abstain`), $o_i \in \mathbb{R}$ is a constant denoting the weight of the *i*-th label and $p_{\texttt{abstain}}$ is the probability assigned to `abstain`. Intuitively, the model either: *(i)* learns to make "safe" predictions by assigning the probability mass to $p_{\texttt{abstain}}$, in which case it incurs constant loss of $p_{\texttt{abstain}}$, or *(ii)* tries to predict the correct label, in which case it potentially incurs smaller loss if $p_i o_i > p_{\texttt{abstain}}$. If the scaling constant o_i is high, the model is encouraged to make predictions even if it is uncertain and potentially makes lot of mistakes. As o_i decreases, the model is penalized more and more for making mis-predictions and learns to make "safer" decisions by allocating more probability mass to the `abstain` label.

Obtaining a model which never mis-predicts on $\mathcal{D}$ For the $\ell_{\texttt{AbstainCrossEntropy}}$ loss, it is possible to always obtain a model f' that never mis-predicts on samples in $\mathcal{D}$. Such a model f' corresponds to minimizing the loss incurred by Equation 5 which corresponds to maximizing $p_i o_i + p_{\texttt{abstain}}$ (assuming i is the correct label). This can be simplified and bounded from above to $p_i + p_{\texttt{abstain}} \leq 1$,

by setting $o_i = 1$ and for any valid distribution it holds that $1 = \sum_{p_i \in \boldsymbol{p}} p_i$. Thus, $p_i o_i + p_{\texttt{abstain}}$ has a global optimum trivially obtained if $p_{\texttt{abstain}} = 1$ for all samples in $\mathcal{D}$. That is, the correctness (no mis-predictions) can be achieved by rejecting all samples in $\mathcal{D}$. However, this leads to zero recall and is not practically useful.

Balancing correctness and recall To achieve both correctness and high recall, similar to Liu et. al., we train our models using a form of annealing. We start with a high $o_i = |\mathbb{Y}|$, biasing the model away from abstaining, and then train for a number of epochs n. We then gradually decrease o_i to 1 for a fixed number of epochs k, slowly nudging it towards abstaining. Finally, we keep training with $o_i = 1$ until convergence. We note that the threshold h is not used during the training. Instead, it is set after the model is trained and is used to fine-tune the trade-off between recall and correctness (precision). Further, note that $o_i = 1$ is used only if the desired accuracy is 100% and otherwise we use $o_i = 1 + \epsilon$. Here, ϵ is selected by decreasing the value o_i as before but stopping just before the model abstains from making all predictions.

Summary We described an existing technique (Liu et al., 2019) for training a model that learns to abstain from making predictions, allowing us to trade-off correctness (precision) and recall. A key advantage of this technique is its generality – it works with any existing neural model with two simple changes: *(i)* adding an `abstain` label, and *(ii)* using the loss function in Equation 5. To remove clutter and keep discussion general, the rest of our work interchangeably uses $f(x)$ and $(f, g_h)(x)$.

C. Adversarial Training for Code

In Section B, we described how to learn models that are correct on subset of the training dataset $\mathcal{D}$ by allowing the model to abstain from making a prediction when uncertain. We now discuss how to achieve robustness (that is, the model either abstains or makes a correct prediction) to a much larger (potentially infinite) set of samples beyond those included in $\mathcal{D}$ via so-called *adversarial training* (Goodfellow et al., 2015).

Adversarial training The goal of adversarial training (Madry et al., 2018; Wong & Kolter, 2018; Sinha et al., 2018; Raghunathan et al., 2018) is to minimize the expected adversarial loss:

$$\mathbb{E}_{(x,y)\sim D}[\max_{\boldsymbol{\delta} \subseteq \Delta(x)} \ell(f(x + \boldsymbol{\delta}), y)] \tag{6}$$

In practice, as we have no access to the underlying distribution but only to the dataset $\mathcal{D}$, the expected adversarial loss is approximated by *adversarial risk* (which training aims to minimize):

$$\frac{1}{|\mathcal{D}|} \sum_{(x,y)\in\mathcal{D}}^{|\mathcal{D}|} \max_{\boldsymbol{\delta} \subseteq \Delta(x)} \ell(f(x + \boldsymbol{\delta}), y) \tag{7}$$

Intuitively, instead of training on the original samples in $\mathcal{D}$, we train on the worst perturbation of each sample. Here, $\boldsymbol{\delta} \subseteq \Delta(x)$ denotes an ordered sequence of modifications while $x + \boldsymbol{\delta}$ denotes a new input obtained by applying each modification $\delta \in \boldsymbol{\delta}$ to x. Recall that each input $x = \langle p, l \rangle$ is a tuple of a program p and a position l in that program for which we will make a prediction. Applying a modification $\delta \colon \mathbb{X} \to \mathbb{X}$ to an input x corresponds to generating both a new program as well as updating the position l if needed (e.g., in case the modification inserted or reordered program statements). That is, δ can modify *all* positions in p, not only those for which a prediction is made. Further, note that the sequence of modifications $\boldsymbol{\delta} \subseteq \Delta(x)$ is computed for each x separately, rather than having the same set of modifications applied to all samples in $\mathcal{D}$.

Using adversarial training in the domain of code requires a set of *label preserving* modifications $\Delta(x)$ over programs which preserve the output label y (defined for a given task at hand), and a technique to solve the optimization problem $\max_{\boldsymbol{\delta} \subseteq \Delta(x)}$ efficiently. We elaborate on both of these next.

C.1. Label Preserving Program Modifications

We define three types of label preserving program modifications – word substitutions, word renaming, and sequence substitutions. Note that label preserving modifications are a strict superset of semantic preserving modifications. This is because while label preserving modifications only require that the correct label does not change, the semantic preserving modifications require that both the label does not change as well as that the overall program semantics do not change. Preserving programs semantics is for many properties unnecessarily strict and therefore we focus on the more general label preserving modifications.

- *Word substitutions* are allowed to substitute a word at a single position in the program with another word (not necessarily contained in the program). Examples of word substitutions include changing constants or values of binary/unary operators.
- *Word renaming* is a modification which includes renaming variables, parameters, fields or methods. In order to produce valid programs, this modification needs to ensure that the declaration and all usages are replaced jointly. Because of this, renaming a single variable in practice always corresponds to making multiple changes to the program (i.e., $|\delta| > 1$ unless the variable is used only once).

- *Sequence substitution* is the most general type of modification which can perform any label preserving program change such as adding dead code or reordering independent program statements.

The main property differentiating the modification types is that word renaming and substitution do not change program structure. This is used both to compute which substitution should be made as well as to provide formal correctness guarantees (discussed in Section 4). Further, is it used for efficient implementation that allows us to implement word substitutions and word renaming directly on the batched tensors, thus making them fast. In contrast, sequence substitutions require parsing batched tensors back to programs, applying modifications on the programs and the processing the resulting programs back to batched tensors. As a result, word substitutions and renaming take 0.1 second to apply once over the full training dataset while structural modifications are $\approx 70\times$ slower and take 7 seconds.

Additionally, it is also possible to define modifications that are not label preserving (i.e., change the ground-truth label), in which case the user has to additionally provide an oracle that computes the correct label y. However, such oracles are typically expensive to design and run (i.e., one would need to run a static analysis over the program or execute the program) and therefore label preserving modification are a preferred option whenever available.

C.2. Finding Adversarial Examples

Given a program x, associated ground-truth label y, and a set of valid modifications $\Delta(x)$ that can be applied over x, our goal is to select a subset of them $\boldsymbol{\delta} \subseteq \Delta(x)$ such that the inner term in the adversarial risk formula $\max_{\boldsymbol{\delta} \subseteq \Delta(x)} \ell(f(x + \boldsymbol{\delta}), y)$ is maximized. Solving for the optimal $\boldsymbol{\delta}$ is highly non-trivial since: *(i)* $\boldsymbol{\delta}$ is an ordered sequence rather than a single modification, *(ii)* the set of valid modifications $\Delta(x)$ is typically very large, and *(iii)* the modification can potentially perform arbitrary rewrites of the program (due to sequence substitutions). Thus, we focus on solving this maximization approximately, inline with how it is solved in other domains. In what follows, we discuss three approximate approaches to achieve this and discuss their advantages and limitations.

C.2.1. Greedy Search

The first approach is a greedy search that randomly samples a sequence of modifications $\boldsymbol{\delta} \subseteq \Delta(x)$. The sampling can be performed for a predefined number of steps with the goal of maximizing the adversarial risk, or until an adversarial example is found (i.e., $f(x+\boldsymbol{\delta}) \neq f(x)$). Concretely, for a given input $x = \langle p, l \rangle$, let us define the space of valid modifications $\Delta(x) \subseteq \Delta(p, l_1) \times \Delta(p, l_2) \times \cdots \times \Delta_n(p, l_n)$ as the Cartesian product of possible modification applied to each position in the program $l_{1:n}$. We select $\boldsymbol{\delta}$ using the following procedure: sample a threshold value $t \sim \mathcal{N}(0.1, 0.4)$ and apply the modification at each location with probability t. If $|\Delta_i(p, l_i)| > 1$, then the modification to apply is sampled at random from the set $\Delta_i(p, l_i)$. Sampling of the threshold value t is done per each sample x and ensures variety in the number of modifications applied.

Limitations and advantages The main advantage of this technique is that it is simple, easy to implement, and very fast. Given its simplicity, this technique is independent of the actual modification and applies equally to words substitutions, word renamings as well as sequence substitutions. However, a natural limitation of this technique is that is uses no information about which positions and which values are important to the prediction is used.

C.2.2. Gradient-based Search

Similar to prior works, gradient information can be used to guide the search for an adversarial examples. This can done in two ways – *(i)* finding a program position to change, and *(ii)* finding both a program positions as well as the new value to change. To find a program position, we can use gradients to measure the importance of each position $\boldsymbol{a}$ for a given prediction in the same way as described in Section 3. Once the *attribution* score $\boldsymbol{a}$ is computed, the adversarial attack can be generated by sampling positions to be modified proportionally to $\boldsymbol{a}$, instead of the uniform sampling used in the greedy search.

Additionally, as shown in the concurrent work (Yefet et al., 2019), the gradients can also be used to select both the program position and the new value to be used (instead of sampling from all valid values uniformly at random).

Limitations and advantages The main advantage of gradient-based approach is that the decision of which position to changes as well as what the new value should be is guided, rather than random. Further, for renaming modifications, such approach has shown to be quite effective (Yefet et al., 2019) at finding the adversarial examples. However, the main limitation of this approach is that it works only for replacing single value (i.e., word substitutions and word renaming) and not when the value is a complex structure (i.e., sequence substitution). Sequence substitutions are important class of modifications which are however hard to optimize for as in general, they can perform arbitrary changes to the program (e.g., adding dead code, adding/removing statements, etc.).

C.2.3. Reducing the Search Space

The third technique is orthogonal to the first two and aims to reduce the search space of relevant modifications a priori, rather than searching it efficiently. Concretely, for

a position l_i in the program p at which the prediction is made, it refines the set of valid program modifications as $\Delta(x) \subseteq \prod_{l_j} \Delta(p, l_j)$ for all positions $\{l_j \mid l_j \in l_{1:n} \wedge \texttt{reachable}(l_j, l_i)\}$. Here, we use $\texttt{reachable}(l_j, l_i)$ to denote that position l_j can affect position l_i. When representing programs as graphs, this can be computed a priori by checking the reachability between the two corresponding nodes. Additionally, when used together with gradient based optimization, such check is not necessary as the gradients will naturally be zero. To obtain a program representation where dependencies between many program locations are removed, we learn to refine program representation as described in Section 3 and Appendix D.

Limitations and advantages The main advantage of this approach is that it applies to both renaming and structural modifications. The main disadvantage is that it depends on the fact the dependencies between program locations can be check efficiently and learned as part of the training. While we show how this can be done for graph neural networks, our approach currently does not support other models such as recurrent neural networks.

Summary In this section, we described how adversarial attacks can be applied to code via set of program modifications. The adversarial attacks we consider are applied on the discrete input (i.e., the attack always correspond to a concrete program) rather than considering attacks in the latent space that are not directly interpretable. We describe two existing techniques that can be used to guide the search for adversarial attacks (greedy search and gradient-based search) and one makes the attacks easier by reducing the search space. As such, these techniques are quite general and can be applied to number of tasks over code. In our experiments, we use the greedy search technique together with reducing the search space.

D. Learning to Refine Representation

In this section, we provide formal definition of the integer linear program (ILP) encoding used to solve the optimization problem presented in Section 3. Recall, that the problem statement is as follows.

Problem statement Minimize the expected size of the refinement $\alpha \subseteq \Phi$ subject to the constraint that the expected loss of the model f stays approximately the same:

$$\arg\min_{\alpha \subseteq \Phi} \sum_{(x,y)\in\mathcal{D}} |\alpha(x)| \tag{8}$$

subject to

$$\textstyle\sum_{(x,y)\in\mathcal{D}} \ell(f(x), y) \approx \sum_{(x,y)\in\mathcal{D}} \ell(f(\alpha(x)), y)$$

Our problem statement is quite general and can be directly instantiated by: *(i)* using $\ell_{\texttt{AbstainCrossEntropy}}$ as the loss (Appendix B), and *(ii)* using *adversarial risk* (Appendix C).

The motivation of solving Equation 8 by phrasing it as ILP problem is that existing off-the-shelve ILP solvers can solve it efficiently and produce the optimal solution. We discuss an alternative end-to-end solution that does not depend on an external ILP solver at the end of Section 3.

Optimization via integer linear programming To solve Equation 2 efficiently, the key idea is that for each sample $(x, y) \in \mathcal{D}$ we: *(i)* capture the relevance of each node to the prediction made by the model f by computing the *attribution* $\boldsymbol{a}(f, x, y) \in \mathbb{R}^{|V|}$ (as described in Section 3), and *(ii)* include the minimum number of edges necessary for a path to exist between every relevant node (according to the attribution $\boldsymbol{a}$) and the node where the prediction is made. Preserving all paths between the prediction and relevant nodes encodes the constraint that the expected loss stays approximately the same.

Concretely, let us define a *sink* to be the node for which the prediction is being made while *sources* are defined to be all nodes v with *attribution* $a_v > t$. Here, the threshold $t \in \mathbb{R}$ is used as a form of regularization. To encode the *sources* and the *sink* as an ILP program, we define an integer variable r_v associated with each node $v \in V$ as:

$$r_v = \begin{cases} -\sum_{v' \in V \setminus \{v\}} r_{v'} & \text{if } v \text{ is predicted node [\textit{sink}]} \\ \lfloor 100 \cdot a_v \rfloor & \text{else if } a_v > t \qquad \text{[\textit{sources}]} \\ 0 & \text{otherwise} \end{cases}$$

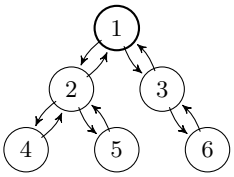
That is, r_v for a source is its attribution value converted to an integer and r_v for a sink is a negative sum of all source values. Note that in our definition it is not possible for a single node to be both *source* and a *sink*. For cases when the *sink* node has a non-zero attribution, this attribution is simply left out since every node is trivially connected to itself.

We then define our ILP formulation of the problem as shown in Figure 3. Here $cost_q$ is an integer variable associated with each edge feature and denotes the edge capacity (i.e., the maximum amount of flow allowed to go trough the edge with this feature), f_{st} is an integer variable denoting the amount of flow over the edge $\langle s, t\rangle$, the constraint $0 \leq f_{st} \leq cost_{\phi(\langle s,t\rangle)}$ encodes the edge capacity, and $r_v + \sum_{\{s|(s,v)\in E\}} f_{sv} = \sum_{\{t|(v,t)\in E\}} f_{vt}$ encodes the flow conservation constraint which requires that the flow generated by the node r_v together with the flow from all the incoming edges $\sum_{\{s|(s,v)\in E\}} f_{sv}$ has to be the same as the flow leaving the node $\sum_{\{t|(v,t)\in E\}} f_{vt}$. The solution to this ILP program is a *cost* associated with each

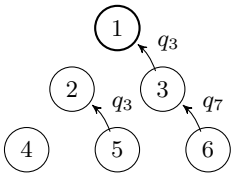
$$\underset{\forall(\langle V,E,\xi_V,\xi_E\rangle,y)\in\mathcal{D}}{\text{minimize}} \quad \sum_{q\in\Phi} cost_q \quad \text{subj. to} \quad \begin{array}{lll} 0 \le f_{st} \le cost_{\phi(\langle s,t\rangle)} & \forall\langle s,t\rangle \in E & \text{[edge capacity]} \\ r_v + \sum_{\{s|(s,v)\in E\}} f_{sv} = \sum_{\{t|(v,t)\in E\}} f_{vt} & \forall v \in V & \text{[flow conservation]} \end{array}$$

Figure 3. Formulation of the refinement problem from Equation 8 as a minimum cost maximum flow integer linear program.

(a) **Original Graph** G



(d) **Abstracted Graph** $\alpha(G)$



(b) **Graph Nodes** V

V	ξ_V	$\boldsymbol{a}$	r_v	flow conservation constraints
1	A	0.3	$r_1 = -70$	$r_1 + f_{21} + f_{31} = f_{12} + f_{13}$
2	A	0	$r_2 = 0$	$r_2 + f_{12} + f_{42} + f_{52} = f_{21} + f_{24} + f_{25}$
3	B	0	$r_3 = 0$	$r_3 + f_{13} + f_{63} = f_{31} + f_{36}$
4	C	0	$r_4 = 0$	$r_4 + f_{24} = f_{42}$
5	B	0	$r_5 = 0$	$r_5 + f_{25} = f_{52}$
6	D	0.7	$r_6 = 70$	$r_6 + f_{36} = f_{63}$

(c) **Graph Edges** E

E	ξ_E	$\phi(\langle s,t\rangle)$	edge capacity constraints
$\langle 1,2\rangle$	ast	$q_1 = \langle ast, A, A\rangle$	$0 \le f_{12} \le cost_{q1}$
$\langle 1,3\rangle$	ast	$q_2 = \langle ast, A, B\rangle$	$0 \le f_{13} \le cost_{q2}$
$\langle 3,1\rangle$	ast	$q_3 = \langle ast, B, A\rangle$	$0 \le f_{31} \le cost_{q3}$
$\langle 5,2\rangle$	ast	$q_3 = \langle ast, B, A\rangle$	$0 \le f_{52} \le cost_{q3}$
		$\cdots$	
$\langle 6,3\rangle$	ast	$q_7 = \langle ast, D, B\rangle$	$0 \le f_{63} \le cost_{q7}$

Optimization Problem

$$\text{minimize} \ \sum_{i=1}^{7} cost_{q_i} \quad \text{subj. to} \quad \begin{array}{l} \text{edge capacity constraints} \\ \text{flow conservation constraints} \end{array}$$

Solution

$$cost_{q3,q7} = 70 \wedge cost_{q1,q2,q4,q5,q6} = 0$$
$$\alpha \ = \ \{q_3, q_7\}$$

Figure 4. Illustration of ILP encoding from Figure 3 on a single graph where the prediction should be made for node 1.

edge feature $q \in \Phi$. If the cost for a given edge feature is zero, it means that this feature was not relevant and can be removed. As a result, we define the refinement $\alpha = \{q \mid q \in \Phi \wedge cost_q > 0\}$ to contain all edge features with non-zero weight.

Example As a concrete example, consider the initial graph shown in Figure 4*a* and assume that the prediction is made for node 1. For simplicity, each node has a single attribute ξ_V, as shown in Figure 4*b*, and all edges are of type ast. The edge feature for edge $\langle 1,3\rangle$ is therefore $\langle ast, A, B\rangle$, since $\xi_E(\langle 1,3\rangle) = ast$, $\xi_V(1) = A$ and $\xi_V(3) = B$, as shown in Figure 4*c*. The *attribution* $\boldsymbol{a}$ reveals two relevant nodes for this prediction – the node itself with score 0.3 and node 6 with score 0.7. We therefore define a single source $r_6 = 70$ and a sink $r_1 = -70$ and encode both the edge capacity constraints, and the flow conservation constraints as shown in Figure 4 (note that according to Figure 3, we would encode all samples in $\mathcal{D}$ jointly). The minimal cost solution assigns cost 70 to edge features q_3 and q_7 which are needed to propagate the flow from node 6 to node 1. The graph obtained by applying the abstraction $\alpha = \{q_3, q_7\}$ is shown in Figure 4*d* and makes the prediction independent of the subtree rooted at node 2. Notice however, that an additional edge is included between nodes 5 and 2. This is because α is computed using *local* edge features ϕ only, which are the same for edges $\langle 3,1\rangle$ and $\langle 5,2\rangle$.