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 37^{th} 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 $f : \mathbb{X} \to \mathbb{R}^{|\mathbb{Y}|}$, 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 $g_h : \mathbb{X} \to \mathbb{R}$, 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 $m_i = \langle f, g_h, \alpha \rangle$.

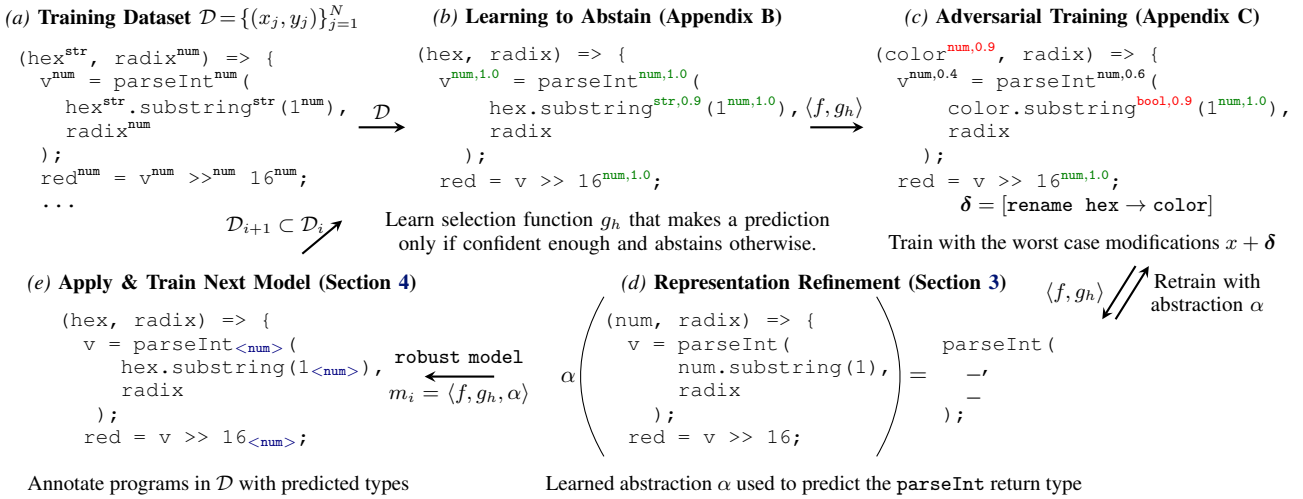

Figure 2. Overview of the main steps of our approach for learning accurate and adversarially robust models of code.

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:

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

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 2, 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:

$$\boldsymbol{a}(f, x, y) = \left[\|\mathbf{G}_{i,:}\|_1, \ldots, \ \mathbf{G}_{|p|,:\ 1} \right],$$

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 2 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

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

1: **function** `RobustTrain`$(\mathcal{D}, t_{\texttt{acc}})$:
2: $\quad \alpha_{\texttt{last}} \leftarrow \Phi$
3: $\quad f, g_h \leftarrow$ `Train` $(\mathcal{D}, t_{acc} - \epsilon)$
4: $\quad$ **while** $true$ **do**
5: $\quad\quad \alpha \leftarrow$ `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: $\quad\quad f, g_h \leftarrow$ `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** $\langle f, g_h, \alpha \rangle$

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

1: **function** `AccurateAndRobustTrain`$(\mathcal{D}, t_{\texttt{acc}} = 1.0)$
2: $\quad M \leftarrow []$
3: $\quad$ **while** $true$ **do**
4: $\quad\quad \langle f, g_h, \alpha \rangle \leftarrow$ `RobustTrain`$(\mathcal{D}, t_{\texttt{acc}})$
5: $\quad\quad \mathcal{D}_{\texttt{abstain}} \leftarrow$ `Apply`$(\mathcal{D}, f, g_h, \alpha)$
6: $\quad\quad$ **if** $|\mathcal{D}_{\texttt{abstain}}| = |\mathcal{D}|$ **then break**
7: $\quad\quad \mathcal{D} \leftarrow \mathcal{D}_{\texttt{abstain}}$
8: $\quad\quad M \leftarrow M \cdot \langle f, g_h, \alpha \rangle$
9: $\quad$ **return** M

trainable solution is also possible. For example, one could define α to be 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 according to 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.

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 repre-

sentation 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 task studied by a number of prior works – predicting types for two 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.

We implemented our code in 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	`x` → `y`
object field renaming	`obj.x` → `obj.y`
property assignment renaming	{`x : obj`} → {`y : obj`}
Label Preserving	
number substitution	`2` → `7`
string substitution	"`get`" → "`load`"
boolean substitution	`true` → `false`

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

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

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

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 ∀ correct to denote that a sample (x, y) is correct for *all* possible modifications $\boldsymbol{\delta} \subseteq \Delta(x)$, the ∃ 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 ∀ 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 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 domain specific language).

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 perform two steps: *(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 proposed new neural architectures with the goal of improving accuracy. 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.

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.