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 including – finding and fixing bugs, code completion, de-compilation, malware detection, 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) developing a set of novel techniques that enable training robust and accurate models of code.

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 to half of their accuracy. 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 from making a prediction when uncertain, partitioning the dataset: one part where the model makes predictions (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 \( p \in \mathbb{P} \) 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 \( l \in \mathbb{L} \) be a position in the program \( p \) that corresponds to a word \( w_l \in \mathbb{W} \). A training dataset \( D = \{(x_j,y_j)\}_{j=1}^N \) contains a set of samples, where \( x \in X \) is an input tuple \( x = \langle p, l \rangle \) consisting of a program \( p \in \mathbb{P} \) and a position in the program \( l \in \mathbb{L} \), while \( y \in \mathbb{Y} \) contains the ground-truth label. As an example, the code snippet in Figure 2 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 \( f: X \rightarrow \mathbb{R}^{|Y|} \), 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 \( g_h: X \rightarrow \mathbb{R} \), 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 2b 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 \( x + \delta \) denote a new input obtained by applying the modifications in \( \delta \subseteq \Delta(x) \) to \( x \). As a concrete example, Figure 2c shows a refactoring of the program from Figure 2b by renaming \( \text{hex} \) to \( \text{color} \). Even though this change does not affect the types in the program, the model suddenly predicts incorrect types for both the \( \text{color} \) parameter and the \( \text{substring} \) function. Further, even though the types of \( \text{parseInt} \) 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 \( \delta \subseteq \Delta(x) \) — when evaluated on \( x + \delta \), 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 \( \mathbb{E}_{(x,y) \sim D}[\ell(\langle f, g_h \rangle(x), y)] \) as usually done in standard training, minimizes the expected adversarial loss:

\[
\mathbb{E}_{(x,y) \sim D}[\max_{\delta \subseteq \Delta(x)} \ell(\langle f, g_h \rangle(x + \delta), y)]
\]

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 \( \max_{\delta \subseteq \Delta(x)} \) 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. In fact, if used alone, adversarial training improves the robustness only slightly or even not at all. 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. Further, a single discrete program change is much more disruptive in affecting the model than a slight continuous perturbation of a pixel value. Combined with the fact that there can be infinitely many changes in \( \Delta(x) \), adversarial training alone fails to produce robust models of code.

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 \( \alpha \) which given a program, produces a relevant representation of that program. Figure 2d shows an example of a possible abstraction \( \alpha \) that takes as input the entire program but keeps only parts relevant for predicting the type of \( \text{parseInt} \) — it is a method call with name \( \text{parseInt} \) which has two arguments. To learn the abstraction \( \alpha \), 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 \( \alpha \) stays roughly the same.

Finally, we apply adversarial training, but this time on the abstraction \( \alpha \) obtained via representation refinement, resulting in new functions \( f \) and \( g_h \). Overall, this results in an adversarially robust model \( m_\alpha = \langle f, g_h, \alpha \rangle \).
Adversarial Robustness for Code

Given a training dataset \( \mathcal{D} = \{(x_i, y_i)\}_{j=1}^{N} \), we learn the prediction function \( f : \mathcal{D} \rightarrow \mathbb{R}^n \). The type of the predictions it makes. For example, while predicting the learned model \( f \) allows only if confident enough and abstains otherwise. The model \( m_i \) is robust, it provides predictions only for a sufficient part of the samples for which it has enough confidence (i.e., \( g_h(x) \geq \hat{h} \)). 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 \( m_{i+1} \) learns that the left hand side of the assignment \( v = \text{parseInt} \) has the same type as the right hand side, since the type of \( \text{parseInt} \) 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 \( \alpha \) that is specialized for the predictions it makes. For example, while predicting the type of \( \text{parseInt} \) is independent of the argument values (\( \text{parseInt}(\_ , \) radix)), predicting the second argument type is not (\( \text{parseInt}(\_ , \text{radix}) \)). Using a single abstraction to predict both would lead to either reduced robustness or accuracy, depending on which abstraction is used.

**Step 4: Learning accurate models** Although the model \( m_i \) is robust, it provides predictions only for a sufficient part of the samples for which it has enough confidence (i.e., \( g_h(x) \geq \hat{h} \)). 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 \( \mathcal{D}_{i+1} \). 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 \( \alpha \) that takes as input \( p \) and produces only the parts relevant for the prediction. That is, \( \alpha \) 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 $G = (V, E, \xi_V, \xi_E)$ where $V$ denotes a set of nodes, $E \subseteq V^2$ denotes a set of directed edges, $\xi_V : V \rightarrow \mathbb{N}^k$ is a mapping from nodes to their associated attributes and $\xi_E : E \rightarrow \mathbb{N}^m$ 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 $\alpha : (V, E, \xi_V, \xi_E) \rightarrow (V', E', \xi_V, \xi_E)$ 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 $\alpha$ 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 $\alpha$ 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 $\alpha$. 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 $(s, t) \in E$, we define an edge feature $\phi((s, t)) := \langle \xi_E((s, t)), \xi_V(s), \xi_V(t) \rangle$ 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 $\Phi$ to be the range of the function $\phi$ evaluated over all edges in $D$. Further, we define the refinement $\alpha$ as a subset of edge features $\alpha \subseteq \Phi$. Finally, the semantics of executing $\alpha$ over edges $E$ is that only edges whose features are in $\alpha$ are kept, i.e., $\{ e \mid e \in E \land \phi(e) \in \alpha \}$.

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:

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

subject to

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

Our problem statement is quite general and can be instantiated by: (i) using $\ell_{\text{AbstainCrossEntropy}}$ 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 $\alpha$ (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 $D$, 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 $(x, y) \in D$ we first capture the relevance of each node to the prediction made by the model $f$ by computing:

$$\alpha(f, x, y) = [\|G_i;_1, \ldots, G_{|p|};_1]$$

where $G = \nabla_x \ell(f(x), y) \in \mathbb{R}^{|p| \times \text{emb}}$ 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 $\text{emb} \in \mathbb{R}$. 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 $\tilde{a} \in \mathbb{R}^{|p|}$. To obtain a probability distribution $\hat{a}(f, x, y)$ over all positions in $p$, we normalize the entries in $\tilde{a}$ 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 $\hat{a}$) 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 \((f, g_h, \alpha)\).

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

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

1: function AccurateAndRobustTrain\((D, t_{acc} = 1.0)\) :
2: \(M \leftarrow []\)
3: while true do
4: \((f, g_h, \alpha) \leftarrow \text{RobustTrain}\((D, t_{acc})\) \)
5: \(D_{\text{abstain}} \leftarrow \text{Apply}\((D, f, g_h, \alpha)\) \)
6: if \(|D_{\text{abstain}}| = |D|\) then break
7: \(D = D_{\text{abstain}}\)
8: \(M = M : (f, g_h, \alpha)\)
9: return \(M\)

trainable solution is also possible. For example, one could define \(\alpha\) to be continuous by defining a learnable weight for each edge feature \(\phi\), encode the sparsity on \(\alpha\) as part of the loss, and extend the graph neural network such that each message propagated along an edge is scaled according to the corresponding value of the edge feature \(\phi(e)\). We leave exploring this option for future work.

4. Training Algorithm

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

Training a single adversarially robust model We start by defining a training procedure used to learn a single adversarially robust model as shown in Algorithm 1. The input is a training dataset \(D\) and the desired accuracy \(t_{acc}\) that the learned model should have. Here, setting \(t_{acc} = 1.0\) corresponds to a model that makes no mis-prediction (i.e., 100% accuracy) while \(t_{acc} = 0\) 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 \(t_{acc} - \epsilon\), 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 \(\alpha\) (line 5) which is applied to the dataset \(D\) 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 \(t_{acc} - \epsilon\) 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 \(t_{acc}\) (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 \(\alpha\).

Incorporating robust predictions Once a single model is learned, it makes robust predictions on a subset of the dataset \(D_{\text{predict}} = \{(x, y) \mid (x, y) \in D \land g_h(\alpha(x)) \geq h\}\) and abstains from making a prediction on the remainder of the samples \(D_{\text{abstain}} = D \setminus D_{\text{predict}}\). Next, for all samples in \(D_{\text{predict}}\), we use the learned model to annotate the position \(l\) in the program \(p\) (recall that each \(x = (p, l)\) consists of a program \(p\) and a position \(l\) with the ground-truth label \(y\) (denoted as \(\text{App1y}\) in Algorithm 2). Annotating a program position corresponds to either defining a new attribute (as illustrated in Figure 2e) 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 \((x, y) \in D\).

Main training algorithm Our main training algorithm is shown in Algorithm 2. It takes as input the training dataset \(D\) and learns multiple models \(M\), each of which makes robust predictions on a different subset of \(D\) (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 \(D\), the Algorithm 2 is run iteratively, with decreasing values of \(t_{acc}\) 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 way 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 $f(x)_{\text{best}}$, is the same as the ground truth label $y$:

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

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

  \[
  \frac{1}{|\mathcal{D}|} \sum_{(x,y) \in \mathcal{D}} \begin{cases} 0 & \text{if } \exists \delta \subseteq \Delta(x) \text{ such that } f(x + \delta)_{\text{best}} \notin \{y, \text{abstain}\} \\ 1 & \text{otherwise} \end{cases}
  \]

**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. The model is the same as described by Hellendoorn et al. except that we use as input sequence of AST types and values, instead of using the syntactic program tokens.

- **GNN**, **GGNN** and **GNT** are three graph neural networks that use as input the graph program representation described in Section 3. Here, **GNN** 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 depending on the model.

**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 $\leq 3$ steps).

**Program modifications** We instantiate the adversarial training with both semantic preserving as well as label-preserving modifications. Concretely, we use the following types of program renaming modifications – variable renaming, constant substitution, object field renaming (e.g., `obj.x → obj.y`) and property assignment renaming (e.g., `{x: obj} → {y: obj}`). Further, we use the following modifications that change the program structure – wrapping expressions, variables and constants in a ternary operator (e.g., `x → (expr : x ? x)`, adding function parameter (e.g., `def inc(x) → def inc(x, y)`), adding extra method call parameters (e.g., `inc(x) → inc(x, expr)`), wrapping
Adversarial Robustness for Code

Table 1. 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.

<table>
<thead>
<tr>
<th>Model</th>
<th>Standard Training</th>
<th>Adversarial Training</th>
<th>Abstain + Adversarial + Refinement</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$\ell(f(x), y)$</td>
<td>$\max_{\delta \subseteq \Delta(x)} \ell(f(x + \delta), y)$</td>
<td>$\max_{\delta \subseteq \Delta(x)} \ell_{\text{abstain}}(f, y_t)(\alpha(x + \delta), y)$</td>
</tr>
<tr>
<td></td>
<td>Accuracy</td>
<td>Robustness</td>
<td>Accuracy</td>
</tr>
<tr>
<td>LSTM</td>
<td>88.2%</td>
<td>44.9%</td>
<td>87.5%</td>
</tr>
<tr>
<td>DeepTyper</td>
<td>88.2%</td>
<td>52.4%</td>
<td>86.7%</td>
</tr>
<tr>
<td>GCN</td>
<td>82.6%</td>
<td>49.1%</td>
<td>81.4%</td>
</tr>
<tr>
<td>GNT</td>
<td>89.5%</td>
<td>47.4%</td>
<td>88.3%</td>
</tr>
<tr>
<td>GGNN</td>
<td>86.7%</td>
<td>52.1%</td>
<td>85.8%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

variables as arrays (e.g., $x \rightarrow [x, x]\{\text{const}\}$), adding dead code (e.g., $\emptyset \rightarrow \text{expr}$) and adding new object expressions (e.g., $\emptyset \rightarrow \{x : y, z : \text{expr}\}$). Here, expr consists of a random binary expression over constants up to depth 3, const a randomly selected constant that results in a valid expression and $x, y, z$ are variables in the program scope. Naturally, the list of modification we use 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 of closely estimating the true number of adversarial samples. Furthermore, note that since $\delta \subseteq \Delta(x)$ is a set, each iteration explores a set of concrete program modifications, rather than a single modification.

5.1. Accurate and Adversarially Robust Models

We summarize the main results in Table 1. The first column (left) show the test accuracy of various models, which corresponds to how existing models of code are currently trained and evaluated. 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 89.5%.

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 $\delta \subseteq \Delta(x)$ for which $f(x)$ predicts the type correctly while $f(x + \delta)$ mis-predicts 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 5.7%, 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 1 (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 $t_{\text{acc}} = 1.00$, which corresponds to training the model to make only correct predictions. Using $t_{\text{acc}} = 1.00$, our approach learns an almost perfect model that is both accurate and robust for $\approx 30.0\%$ of samples using both GNT and GGNN. Here, GNT learned 7 models and achieved 99.98% while GGNN learned 8 models achieve slightly lower 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 $t_{\text{acc}} = 1.00$ due to several samples included in the test set. These samples were mis-predicted because they contained code structure not seen during training and not covered by modifications $\delta \subseteq \Delta(x)$. This illustrates that it is important that the samples in $D$ 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 $t_{\text{acc}} = 1.00$, but make predictions for all the samples (i.e., do not abstain). This can be achieved by continuing the training while reducing $t_{\text{acc}}$ to zero and conditioning on all the models trained with higher $t_{\text{acc}}$. In
Table 2. Robustness breakdown for the GNT and GGNN models trained using our approach from Table 1 (right).

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Size</th>
<th>∀ Correct</th>
<th>∃ Incorrect</th>
<th>Abstain</th>
</tr>
</thead>
<tbody>
<tr>
<td>GNT</td>
<td>( t_{acc} = 1.00 )</td>
<td>90.0%</td>
<td>0.00%</td>
<td>10.00%</td>
</tr>
<tr>
<td>( D_{correct} )</td>
<td>29.3%</td>
<td>90.0%</td>
<td>0.00%</td>
<td>10.00%</td>
</tr>
<tr>
<td>( D_{abstain} )</td>
<td>70.6%</td>
<td>-</td>
<td>0.01%</td>
<td>99.99%</td>
</tr>
<tr>
<td>GGNN</td>
<td>( t_{acc} = 1.00 )</td>
<td>75.5%</td>
<td>0.06%</td>
<td>23.94%</td>
</tr>
<tr>
<td>( D_{correct} )</td>
<td>30.6%</td>
<td>75.5%</td>
<td>0.06%</td>
<td>23.94%</td>
</tr>
<tr>
<td>( D_{abstain} )</td>
<td>69.3%</td>
<td>-</td>
<td>1.46%</td>
<td>98.54%</td>
</tr>
</tbody>
</table>

 ∀ Correct \( \coloneqq \forall \delta \subseteq \Delta(x)(f, g_h)(\alpha(x + \delta))_{best} = y \)

∃ Incorrect \( \coloneqq \exists \delta \subseteq \Delta(x)(f, g_h)(\alpha(x + \delta))_{best} \notin \{y, abstain\} \)

Adversarial robustness breakdown Table 2 provides a detailed breakdown of the robustness metric for the GNT and GGNN models trained with \( t_{acc} = 1.00 \) from Table 1 (right). Here, \( D_{abstain} \) contains samples for which the model abstains from making a prediction and \( D_{correct} \) 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 \( \delta \subseteq \Delta(x) \), 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 \( D_{abstain} \), 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 \( D_{abstain} \) can be modified to cause a mis-prediction). This shows that the majority of robustness errors from Table 1 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.

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 (Helendoorn 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 priory. As shown in our evaluation, this is crucial for learning 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 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 exploring the task of learning adversarially robust models of code and that many challenges remain open. While we have tried to make our approach general, it is yet to be seen how effective it is at other tasks over code. Further, we optimize for the worst case adversarial robustness which corresponds to learning a robust model for all programs, not just the common ones on GitHub. An interesting future work would be to optimize with respect to those modification that are also commonly performed by the developers, especially if it is not possible to be perfectly robust for all of them. Finally, we used wide range of different program modifications that include variable, fields and property renamings, token substitutions as well as many structural changes. These are however far from exhaustive and more work is needed in defining new ones as well as exploring ways how different modifications can be combined efficiently.

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. The research leading to these results was partially supported by an ERC Starting Grant 680358.
References


Adversarial Robustness for Code


