Supplementary Material: Adversarial Robustness for Code

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

We provide the following four appendices:

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

A. Evaluation

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

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

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

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

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

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

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

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

We compute the ground-truth types using the TypeScript compiler version 3.4.5 based on manual type annotations, library specifications and analyzing all project files. While we reuse the same GitHub projects and part of DeepTyper's infrastructure[2] to obtain the dataset, the datasets are not directly comparable for a number of reasons. First, we fixed a bug due to which DeepTyper incorrectly included some type annotations as part of the

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

Proceedings of the 37th International Conference on Machine Learning, Online, PMLR 119, 2020. Copyright 2020 by the author(s).

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

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

input. Second, the projects we used are subset of those used in `DeepTyper` since some are no longer available and were removed from GitHub. Third, we additionally predict the types corresponding to all intermediate expressions and constants (e.g., the expression $x + y$ contains three predictions for x, y and $x+y$). This improves model performance as it is explicitly trained also on the intermediate steps required to infer the types. Finally, we train all the models to predict four primitive types (`string`, `number`, `boolean`, `void`), four function types ($() \Rightarrow$ `string`, $() \Rightarrow$ `number`, $() \Rightarrow$ `boolean`, $() \Rightarrow$ `void`) and a special `unk` label denoting all the other types. While this is similar to types predicted by some other works such as `JSNice` (Raychev et al., 2015), it is only subset of types considered in `DeepTyper`.

All results presented in Tables 1 and 2 are obtained by training our models using a dataset that contains 3000 programs split equally between training, validation and test datasets. Because each program contain multiple type predictions, the number of training samples is significantly higher than the number of programs. Concretely, there are $139,915$, $223,912$ and $121,153$ samples in training, validation and test datasets. We note that this is only a subset of the full dataset that can be obtained by processing all the files included in the projects used by Hellendoorn et al. We make the dataset available online at https://github.com/eth-sri/robust-code.

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

B. Training Neural Models to Abstain

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

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

$$(f, g_h)(x) := \begin{cases} f(x) & \text{if } g_h(x) \geq h \\ \texttt{abstain} & \text{otherwise} \end{cases} \tag{1}$$

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

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

$$\ell_{\texttt{CrossEntropy}}(\boldsymbol{p}, \boldsymbol{y}) := -\sum_{i=1}^{|\mathbb{Y}|} y_i \log(p_i) \tag{2}$$

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

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

$$\ell_{\texttt{AbstainCrossEntropy}}(\boldsymbol{p}, \boldsymbol{y}) := -\sum_{i=1}^{|\mathbb{Y}|} y_i \log(p_i o_i + p_{\texttt{abstain}}) \tag{3}$$

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

Obtaining a model which never mis-predicts on $\mathcal{D}$ For the $\ell_{\texttt{AbstainCrossEntropy}}$ loss, it is possible to always obtain a model f' that never mis-predicts on samples in $\mathcal{D}$. Such a model f' corresponds to minimizing the loss incurred by Equation 3 which corresponds to maximizing $p_i o_i + p_{\texttt{abstain}}$ (assuming i is the correct label). This can be simplified and bounded from above to $p_i + p_{\texttt{abstain}} \leq 1$, by setting $o_i = 1$ and for any valid distribution it holds that $1 = \sum_{p_i \in \boldsymbol{p}} p_i$. Thus, $p_i o_i + p_{\texttt{abstain}}$ has a global optimum trivially obtained if $p_{\texttt{abstain}} = 1$ for all samples in $\mathcal{D}$. That is, the correctness (no mis-predictions) can be achieved by rejecting all samples in $\mathcal{D}$. However, this leads to zero recall and is not practically useful.

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

Summary We described an existing technique (Liu et al., 2019) for training a model that learns to abstain from making predictions, allowing us to trade-off correctness (precision) and recall. A key advantage of this technique is its generality – it works with any existing neural model with two simple changes: *(i)* adding an abstain label, and *(ii)* using the loss function in Equation 3. To remove clutter and keep discussion general, the rest of our work interchangeably uses $f(x)$ and $(f, g_h)(x)$.

C. Adversarial Training for Code

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

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

$$\mathbb{E}_{(x,y)\sim D}[\max_{\boldsymbol{\delta} \subseteq \Delta(x)} \ell(f(x+\boldsymbol{\delta}), y)] \tag{4}$$

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

$$\frac{1}{|\mathcal{D}|} \sum_{(x,y)\in\mathcal{D}}^{|\mathcal{D}|} \max_{\boldsymbol{\delta} \subseteq \Delta(x)} \ell(f(x+\boldsymbol{\delta}), y) \tag{5}$$

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

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

C.1. Label Preserving Program Modifications

We define three types of label preserving program modifications – word substitutions, word renaming, and sequence substitutions. Note that label preserving modifications are a strict superset of semantic preserving modifications. This

is because while label preserving modifications only require that the correct label does not change, the semantic preserving modifications require that both the label does not change as well as that the overall program semantics do not change. Preserving programs semantics is for many properties unnecessarily strict and therefore we focus on the more general label preserving modifications.

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

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

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

C.2. Finding Adversarial Examples

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

C.2.1. Greedy Search

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

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

C.2.2. Gradient-based Search

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

Additionally, as shown in the concurrent work (Yefet et al.,

2019), the gradients can also be used to select both the program position and the new value to be used (instead of sampling from all valid values uniformly at random).

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

C.2.3. Reducing the Search Space

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

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

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

D. Learning to Refine Representation

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

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

$$\arg\min_{\alpha \subseteq \Phi} \sum_{(x,y) \in \mathcal{D}} |\alpha(x)| \tag{6}$$

subject to

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

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

The motivation of solving Equation 6 by phrasing it as ILP problem is that existing off-the-shelve ILP solvers can solve it efficiently and produce the optimal solution. We discuss an alternative end-to-end solution that does not depend on an external ILP solver at the end of Section 3.

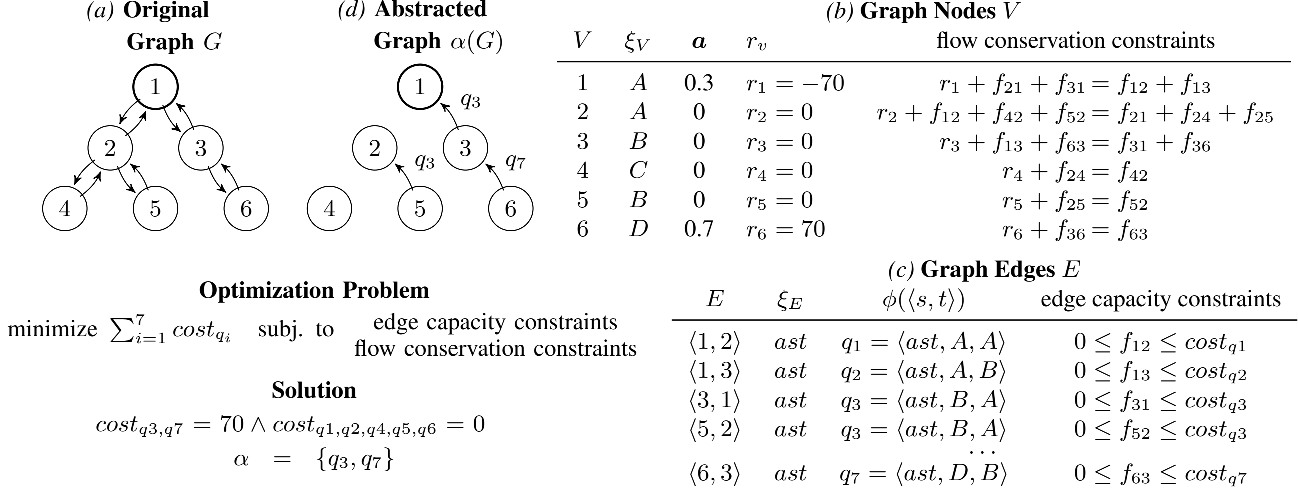
Optimization via integer linear programming To solve Equation 2 efficiently, the key idea is that for each sample $(x, y) \in \mathcal{D}$ we: *(i)* capture the relevance of each node to the prediction made by the model f by computing the *attribution* $\boldsymbol{a}(f, x, y) \in \mathbb{R}^{|V|}$ (as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

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

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

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

Figure 1. Formulation of the refinement problem from Equation 6 as a minimum cost maximum flow integer linear program.



(b) **Graph Nodes** V

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

(c) **Graph Edges** E

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

Figure 2. Illustration of ILP encoding from Figure 1 on a single graph where the prediction should be made for node 1.

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

We then define our ILP formulation of the problem as shown in Figure 1. Here $cost_q$ is an integer variable associated with each edge feature and denotes the edge capacity (i.e., the maximum amount of flow allowed to go trough the edge with this feature), f_{st} is an integer variable denoting the amount of flow over the edge $\langle s,t\rangle$, the constraint $0 \le f_{st} \le cost_{\phi(\langle s,t\rangle)}$ encodes the edge capacity, and $r_v + \sum_{\{s|(s,v)\in E\}} f_{sv} = \sum_{\{t|(v,t)\in E\}} f_{vt}$ encodes the flow conservation constraint which requires that the flow generated by the node r_v together with the flow from all the incoming edges $\sum_{\{s|(s,v)\in E\}} f_{sv}$ has to be the same as the flow leaving the node $\sum_{\{t|(v,t)\in E\}} f_{vt}$. The solution to this ILP program is a *cost* associated with each edge feature $q \in \Phi$. If the cost for a given edge feature is zero, it means that this feature was not relevant and can be removed. As a result, we define the refinement $\alpha = \{q \mid q \in \Phi \wedge cost_q > 0\}$ to contain all edge features with non-zero weight.

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

References

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

Brockschmidt, M., Allamanis, M., Gaunt, A. L., and Polozov, O. Generative code modeling with graphs. 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.

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

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

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

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

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

Hellendoorn, V. J., Bird, C., Barr, E. T., and Allamanis, M. Deep learning type inference. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE'18, 2018.

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.

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.

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

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.

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

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.

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