Break-It-Fix-It: Unsupervised Learning for Program Repair

Michihiro Yasunaga [1] **Percy Liang** [1]

Abstract

We consider repair tasks: given a *critic* (*e.g.*, compiler) that assesses the quality of an input, the goal is to train a fixer that converts a bad example (*e.g.*, code with syntax errors) into a good one (*e.g.*, code with no syntax errors). Existing works create training data consisting of (bad, good) pairs by corrupting good examples using heuristics (*e.g.*, dropping tokens). However, fixers trained on this synthetically-generated data do not extrapolate well to the real distribution of bad inputs. To bridge this gap, we propose a new training approach, *Break-It-Fix-It* (*BIFI*), which has two key ideas: (i) we use the critic to check a fixer's output on real bad inputs and add good (fixed) outputs to the training data, and (ii) we train a *breaker* to generate realistic bad code from good code. Based on these ideas, we iteratively update the breaker and the fixer while using them in conjunction to generate more paired data. We evaluate BIFI on two code repair datasets: GitHub-Python, a new dataset we introduce where the goal is to repair Python code with AST parse errors; and DeepFix, where the goal is to repair C code with compiler errors. BIFI outperforms state-of-the-art methods, obtaining 90.5% repair accuracy on GitHub-Python (+28.5%) and 71.7% on DeepFix (+5.6%). Notably, BIFI does not require any labeled data; we hope it will be a strong starting point for unsupervised learning of various repair tasks.

1. Introduction

In many domains, one has access to a *critic* that assesses the quality of an input, but what is desired is a more constructive *fixer* that actually improves bad inputs. For instance, in programming, while a code analyzer and compiler can tell if a given code has errors, programmers need to repair the errors in the bad code. Development of an automatic code fixer is thus a key to enhancing programming productivity (Seo et al., 2014) and is an active area of research (Mesbah et al., 2019; Ding et al., 2020; Dinella et al., 2020). Other instances of this general setting include molecular design (Jin et al., 2019) which aims to improve the chemical properties (*e.g.*, drug-likeness) of molecules given a property evaluator, and essay editing (Taghipour & Ng, 2016) which aims to improve a writing given a grade. How to automatically learn a fixer given a critic (we term *critic2fixer* remains an important research problem in machine learning.

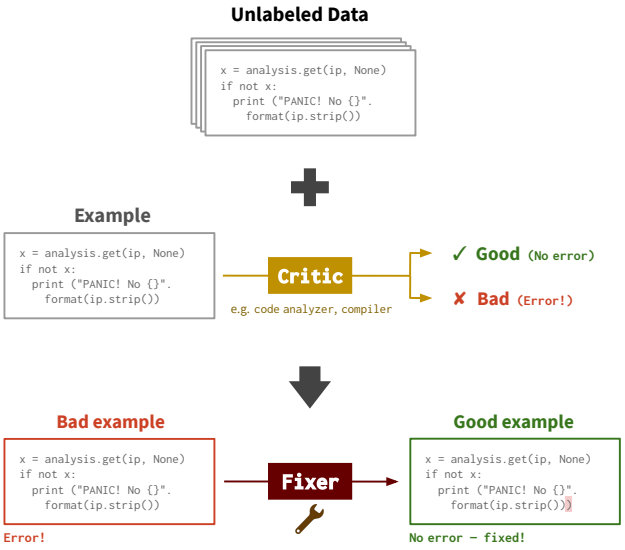

Figure 1. **Problem setup**. We are given an **unlabeled data** (code snippets) and a **critic** (*e.g.*, code analyzer, compiler) that assesses the quality of an input (*e.g.*, bad if the code has errors; good if no error). Our task is to learn a **fixer** that can actually repair a bad example into a good one (*e.g.*, fixing errors in the bad code).

In this work, we focus on the domain of code repair. Learning a fixer is challenging because manual labeling of paired data, *e.g.*, ⟨broken code, fixed code⟩, is costly. To this end, we consider learning from unlabeled data. Specifically, as illustrated in Figure 1, we are given (a) a critic (code analyzer or compiler) that assesses the quality of an input (*bad* if the code has errors; *good* if there are no errors), and (b) unlabeled data (unpaired good code and bad code, *e.g.*, collected on GitHub); our goal is to learn a fixer that repairs bad code into good code. Previous works in code repair apply random or heuristic perturbations to good code (*e.g.*, dropping tokens) and prepare synthetic paired data ⟨perturbed code, good code⟩ to train a fixer (Pu et al., 2016; Gupta et al., 2017; Ahmed et al., 2018; Hajipour et al., 2019; Yasunaga & Liang, 2020). However, such synthetically generated bad examples do not match the

[1]Stanford University, Stanford, CA. Correspondence to: Michihiro Yasunaga <myasu@cs.stanford.edu>.

Proceedings of the 38^{th} International Conference on Machine Learning, PMLR 139, 2021.

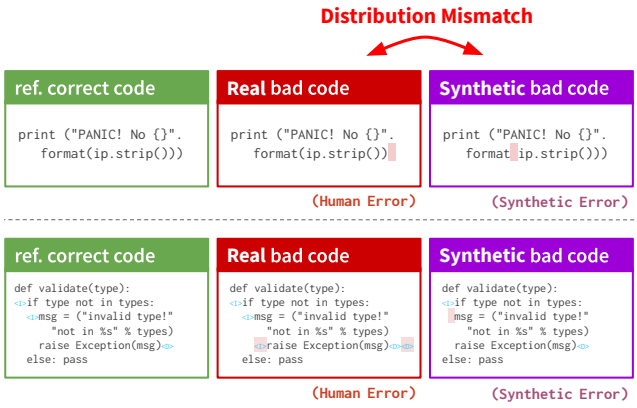

Figure 2. **Challenge of learning a fixer from unlabeled data**. Existing works randomly perturb good examples into bad examples and learn to recover. However, such synthetically generated bad examples do not match the distribution of *real* bad examples. For instance, synthetic perturbations may drop parentheses arbitrarily from code (top right), but real human-written bad code misses parentheses more often in a nested context (top center). In a more extreme case at bottom, to generate the human error (center) from the corrected code (left), a *pair* of indent and dedent tokens need to be inserted accordingly, which random perturbations generate with very small probability. Note that indentation is meaningful in Python: in the tokenized Python code, each ⟨I⟩ token means indenting the line by one unit, each ⟨D⟩ means dedenting the next line by one unit.

distribution of *real* bad examples. For instance, as shown in Figure 2, synthetic perturbations may drop parentheses arbitrarily from code, generating errors that rarely happen in real programming (Figure 2 top right; synthetic errors); in contrast, real human-written code misses parentheses more often in a nested context (Figure 2 top center; human errors). As we will show in §4, this distribution mismatch between synthetic data and real data results in low performance.

To bridge this gap, we propose *Break-It-Fix-It* (*BIFI*), a new method to learn a fixer from unlabeled data and a critic (Figure 3). BIFI is based on two core insights: (i) we can use the critic to check a fixer's output on real bad examples and add good outputs to the training data, and (ii) we train a *breaker* to generate realistic bad examples from good examples. Specifically, given an initial fixer trained on synthetic paired data ⟨synthetic bad, good⟩, BIFI improves the fixer and breaker simultaneously through rounds of data generation and training: (1) apply the fixer to real bad examples and keep fixed outputs to obtain real paired data, (2) use the resulting data to train the breaker, (3) use the learned breaker to generate code errors and obtain more paired data, and (4) train the fixer on the newly-generated paired data in (1) and (3). Intuitively, this cycle trains the fixer on increasingly more real or realistically generated bad code, adapting the fixer from the initial synthetic distributions towards real distributions of code errors.

The BIFI algorithm is related to backtranslation in unsupervised machine translation (Lample et al., 2018a), which uses a target-to-source model to generate noisy sources and trains a source-to-target model to reconstruct the targets (*e.g.*, the bad-side and good-side in our repair task can be viewed as the source and target). BIFI differs from backtranslation in two ways: it uses the critic to verify if the generated examples are actually fixed or broken (step 1 and 3), and it trains the fixer on *real* bad examples in addition to examples generated by the breaker (step 4), which improves the correctness and distributional match of generated paired data.

We evaluate our proposed approach on two datasets:

- **GitHub-Python**: We collected a new dataset of 3M Python code snippets from `github.com`. The task is to repair errors caught by the Python AST parser. We set the initial fixer to be an encoder-decoder Transformer (Vaswani et al., 2017) trained on random perturbations.
- **DeepFix** (Gupta et al., 2017): The task is to repair compiler errors in C code submitted by students in an introductory programming course. We set the initial fixer to be the existing best system, DrRepair (Yasunaga & Liang, 2020), which was trained on manually-designed heuristic perturbations.

Our approach (BIFI) outperforms the initial fixers, obtaining 90.5% repair accuracy on GitHub-Python (+28.5% absolute) and 71.7% repair accuracy on DeepFix (+5.6% absolute), attaining a new state-of-the-art. BIFI also improves on backtranslation by 10%. Further, we qualitatively show how the fixer and breaker adapt towards more realistic distributions of code through the BIFI algorithm (§4.3).

2. Problem statement

Figure 1 illustrates our problem setup, critic2fixer. The system is given unlabeled data $\mathcal{D}$ (code snippets) and a *critic* c (code analyzer or compiler) that returns whether an input is good or bad: *e.g.*, for a code snippet $x \in \mathcal{D}$,

$$c(x) = \begin{cases} 0 & \text{if } x \text{ has errors,} \\ 1 & \text{if } x \text{ has no error.} \end{cases} \quad (1)$$

Using the critic c, examples in $\mathcal{D}$ can be classified into bad ones $\mathcal{D}_{\text{bad}} = \{x \mid x \in \mathcal{D}, c(x) = 0\}$ and good ones $\mathcal{D}_{\text{good}} = \{y \mid y \in \mathcal{D}, c(y) = 1\}$. Our task is to learn a *fixer* f that maps a bad example $x \in \mathcal{D}_{\text{bad}}$ into a good example $f(x)$ such that it is close[1] to x and $c(f(x)) = 1$. The evaluation metric is the fixer f's repair accuracy on a held-out set of bad examples, $\mathcal{D}_{\text{bad}}^{(\text{test})}$,

$$\text{RepairAcc} = \frac{|\{x \mid x \in \mathcal{D}_{\text{bad}}^{(\text{test})}, c(f(x)) = 1\}|}{|\mathcal{D}_{\text{bad}}^{(\text{test})}|}. \quad (2)$$

3. Approach

The major challenge of learning a fixer is that we need to learn from unpaired data, *i.e.*, $\mathcal{D}_{\text{bad}}$ and $\mathcal{D}_{\text{good}}$ do not form ⟨broken, fixed⟩ pairs. Prior works in code repair apply random or

[1] We constrain the edit distance as described in §4.1. We acknowledge that while we want $f(x)$ to be semantics-preserving, it is non-trivial to ensure this automatically, so we rely on the edit distance.

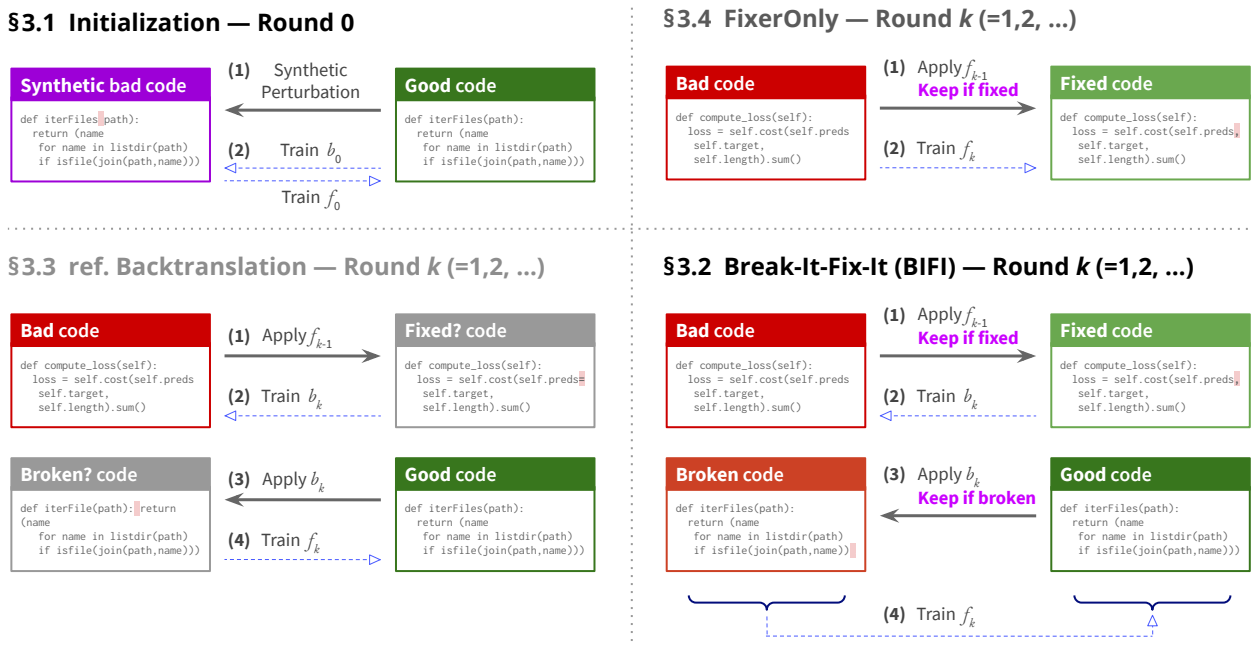

Figure 3. **Overview of our approach**. We train the initial fixer on synthetically prepared data as in prior works (**top left**). In our approach, *BIFI* (**bottom right**), we apply the initial fixer to the real bad examples and add fixed outputs (verified by the critic) to our training data (Step 1), train a breaker on the resulting paired data (Step 2), use the breaker to generate (intuitively more realistic) code errors (Step 3), and train the fixer again on the newly-generated paired data (Step 4). We iterate this cycle, improving the fixer and the breaker simultaneously. **Top right**: a version of BIFI without the breaker (*FixerOnly*). **Bottom left**: comparison of BIFI to backtranslation. The main difference is that BIFI uses the critic to verify that the fixer produces good code and the breaker produces bad code (**annotated with magenta font**).

heuristic perturbations to good examples (*e.g.*, dropping tokens) and prepare a synthetic paired data ⟨perturbed code, good code⟩ to train a fixer (Gupta et al., 2017; Ahmed et al., 2018; Yasunaga & Liang, 2020). However, such synthetically generated bad examples do not match the distribution of real bad examples. For instance, as Figure 2 (top) shows, synthetic perturbations may drop parentheses arbitrarily from code, generating errors that are rare in real programs; in contrast, real human-written code misses parentheses often in a nested context (*e.g.*, 10x more than non-nested in our collected dataset GitHub-Python). In a more extreme case (Figure 2 bottom), to make the real human error (center) from the corrected code (left), multiple tokens (in this case, a pair of indent and dedent) need to be inserted or dropped accordingly, which random perturbations would generate with extremely low probability. This distribution mismatch between synthetic data and real data results in low performance (§4).

To address this challenge, we propose *Break-It-Fix-It* (*BIFI*), an approach that adapts the fixer automatically towards real distributions of bad examples. Concretely, we first start from the synthetic paired data ⟨synthetic bad, good⟩ and train an initial fixer as in prior works (see Figure 3 top left; initialization). We then perform the following cycle (see Figure 3 bottom right): (1) we apply the initial fixer to the real bad examples and use the critic to assess if the fixer's output is good—if good, we keep the pair; (2) we train a *breaker* on the resulting paired data—as this data consists of real code errors, intuitively, the breaker learns to generate realistic code errors; (3) we apply the breaker to the good examples; (4) we finally train the fixer on the newly-generated paired data in (1) and (3). We iterate this cycle to improve the fixer and the breaker simultaneously in the process. The intuition is that a better fixer and breaker will be able to generate more realistic paired data, which in turn helps to train a better fixer and breaker.

Below, we describe the initialization step in §3.1, our main algorithm BIFI in §3.2, and discuss two baselines of BIFI: backtranslation (§3.3) and *FixerOnly* (version of BIFI without the breaker; §3.4).

3.1. Initialization

Given unlabeled data $\mathcal{D} = (\mathcal{D}_{\text{bad}}, \mathcal{D}_{\text{good}})$, we first prepare a synthetic paired data $\mathcal{P}_{\text{synthetic}}$ by perturbing good examples:

$$\mathcal{P}_{\text{synthetic}} = \{(b_{\text{synthetic}}(y), y) \,|\, y \in \mathcal{D}_{\text{good}}\}, \qquad (3)$$

where $b_{\text{synthetic}}$ denotes a pre-defined procedure that corrupts code. For instance, we will experiment with two choices of $b_{\text{synthetic}}$: (i) **random noising**, which randomly drops/inserts /replaces tokens in code, and (ii) **heuristic noising** designed in Yasunaga & Liang (2020), which aims to generate common programming errors such as typo, punctuation and type errors. More details are described in §4.1.

We then train the initial fixer f_0 and breaker b_0 on the

synthetic paired data:

$$b_0 = \textsc{Train}^{\text{good}\to\text{bad}}(\mathcal{P}_{\text{synthetic}}) \quad (4)$$

$$f_0 = \textsc{Train}^{\text{bad}\to\text{good}}(\mathcal{P}_{\text{synthetic}}) \quad (5)$$

where $\textsc{Train}^{\text{good}\to\text{bad}}(\mathcal{P})$ denotes training an encoder-decoder model that maps good-side examples to bad-side examples in a paired data $\mathcal{P}$, and $\textsc{Train}^{\text{bad}\to\text{good}}(\mathcal{P})$ does the reverse. Note that f_0 here corresponds to the fixer learned in prior works (*e.g.*, Gupta et al. (2017); Yasunaga & Liang (2020)). We call this initialization step our *round* 0.

3.2. Break-It-Fix-It (BIFI)

BIFI aims to improve the fixer and breaker simultaneously through rounds of data generation and training: (1) use a fixer to create data for a breaker, (2) train a breaker, (3) use a breaker to create data for a fixer, and (4) train a fixer. Concretely, after the initialization step, BIFI performs the following in each round $k = 1,2,...,K$:

$$\mathcal{P}_k^{(f)} = \{(x, f_{k-1}(x)) \,|\, x \in \mathcal{D}_{\text{bad}}, c(f_{k-1}(x)) = 1\} \quad (6)$$

$$b_k = \textsc{Train}^{\text{good}\to\text{bad}}(\mathcal{P}_k^{(f)}) \quad (7)$$

$$\mathcal{P}_k^{(b)} = \{(b_k(y), y) \,|\, y \in \mathcal{D}_{\text{good}}, c(b_k(y)) = 0\} \quad (8)$$

$$f_k = \textsc{Train}^{\text{bad}\to\text{good}}(\mathcal{P}_k^{(f)} \cup \mathcal{P}_k^{(b)}). \quad (9)$$

For convenience, we call the original examples in $\mathcal{D}_{\text{bad}}$ *real* bad examples. Here Eq 6 applies the current fixer f_{k-1} to the real bad examples in $\mathcal{D}_{\text{bad}}$, and keeps outputs that are actually fixed (verified by the critic c; **red part**). This way, we can obtain new paired data $\mathcal{P}_k^{(f)}$ that is based on *real* bad examples. Eq 7 then trains the breaker b_k (fine-tunes from the previous breaker b_{k-1}) on this new paired data $\mathcal{P}_k^{(f)}$ so that intuitively it can learn to generate realistic bad examples. Next, Eq 8 applies the breaker b_k to the good examples in $\mathcal{D}_{\text{good}}$, and keeps outputs that are actually broken (verified by the critic c; **red part**). This provides an extra paired data $\mathcal{P}_k^{(b)}$ that is based on bad examples generated by the learned breaker. Finally, Eq 9 trains the fixer f_k (fine-tunes from the previous fixer f_{k-1}) on both $\mathcal{P}_k^{(f)}$ and $\mathcal{P}_k^{(b)}$, so that the fixer sees real *and* breaker-generated bad examples. Over time, this cycle adapts the fixer and breaker towards the distribution of real examples. Figure 3 (bottom right) provides an illustration of BIFI.

3.3. Comparison with Backtranslation

The BIFI algorithm is related to backtranslation (Lample et al., 2018b) in unsupervised machine translation. One may view the bad-side and good-side in our setup as two source/target languages in machine translation. Backtranslation uses a target-to-source model to generate noisy sources and trains a source-to-target model to reconstruct the targets. Specifically, in each round k, backtranslation performs the following:

$$\mathcal{P}_k^{(f)} = \{(x, f_{k-1}(x)) \,|\, x \in \mathcal{D}_{\text{bad}}\} \quad (10)$$

$$b_k = \textsc{Train}^{\text{good}\to\text{bad}}(\mathcal{P}_k^{(f)}) \quad (11)$$

$$\mathcal{P}_k^{(b)} = \{(b_k(y), y) \,|\, y \in \mathcal{D}_{\text{good}}\} \quad (12)$$

$$f_k = \textsc{Train}^{\text{bad}\to\text{good}}(\mathcal{P}_k^{(b)}). \quad (13)$$

BIFI differs in two aspects. First, as our task has a critic, BIFI uses the critic to verify the outputs of the fixer and breaker, and only keep examples whose outputs are actually fixed (Eq 6 **red part**) and whose outputs are broken (Eq 8 **red part**) to generate paired data. In contrast, backtranslation does not verify the generated paired data in Eq 10, 12. This may blindly include non-fixed code as good-side examples (consequently, the breaker might learn to output erroneous code) and non-broken code as bad-side examples, leading to noisy training data (*e.g.*, Figure 3 bottom left). Secondly, while backtranslation trains the fixer only on examples predicted by the breaker (Eq 13), BIFI trains the fixer on the *real* bad examples ($\mathcal{P}_k^{(f)}$ in Eq 9) in addition to examples generated by the breaker, which improves the correctness and distributional match of training data. We will show in our ablation study (§4.3.3) that both of these two components improve the learning of a fixer and breaker. Essentially, BIFI is an augmentation of backtranslation with a critic.

3.4. Version of BIFI without breaker: *FixerOnly*

The benefit of BIFI is to enable training the fixer on real bad examples *and* bad examples generated by a learned breaker. We consider a version (*FixerOnly*) that trains the fixer simply on the real bad examples (prepared in Eq 6) but not the bad examples generated by the breaker (Eq 8). Specifically, FixerOnly does the following in each round k:

$$\mathcal{P}_k^{(f)} = \{(x, f_{k-1}(x)) \,|\, x \in \mathcal{D}_{\text{bad}}, c(f_{k-1}(x)) = 1\} \quad (14)$$

$$f_k = \textsc{Train}^{\text{bad}\to\text{good}}(\mathcal{P}_k^{(f)}). \quad (15)$$

FixerOnly can also be viewed as self-training (Lee, 2013) with the difference that we only add fixer outputs verified by the critic to the training data. We will show in §4.3.1 that FixerOnly is especially useful when the amount of available bad examples $|\mathcal{D}_{\text{bad}}|$ is big, but the gain is smaller compared to BIFI when $|\mathcal{D}_{\text{bad}}|$ is small, because BIFI can use the breaker to generate additional paired data for training the fixer (Eq 8).

4. Experiments

We evaluate our approach on two code repair datasets: a common benchmark **DeepFix**[2] (Gupta et al., 2017), and **GitHub-Python**, a bigger dataset we collect in this paper.

4.1. Dataset and setup

We first describe the detail and experimental setup for GitHub-Python and DeepFix.

[2] https://bitbucket.org/iiscseal/deepfix

Method		Test accuracy			
		Total	Unbalanced Parentheses	Indentation Error	Invalid Syntax
Initial	Round-0	62.0%	87.7%	39.4%	70.5%
FixerOnly	Round-1	86.8%	93.3%	79.5%	90.9%
	Round-2	88.6%	92.4%	83.7%	92.0%
BIFI	Round-1	88.0%	94.1%	81.3%	91.6%
	Round-2	**90.5%**	**94.2%**	**85.9%**	**93.5%**

Table 1. **Repair accuracy on the GitHub-Python test set**. The initial fixer is trained on synthetic bad code. Our proposed method (BIFI) enables the fixer to be fine-tuned on *real* bad code and bad code generated by the learned breaker. The result shows that BIFI outperforms the initial fixer by a large margin.

Method		Test accuracy
DeepFix	(Gupta et al., 2017)	33.4%
RLAssist	(Gupta et al., 2019)	26.6%
SampleFix	(Hajipour et al., 2019)	45.3%
DrRepair	(Yasunaga & Liang, 2020)	66.1%
Our Initial	Round-0 (= DrRepair)	66.1%
Our FixerOnly	Round-1	68.6%
	Round-2	70.5%
Our BIFI	Round-1	70.8%
	Round-2	**71.7%**

Table 2. **Repair accuracy on the DeepFix test set**. We define our initial fixer (Round 0) to be the existing best system DrRepair. Note that DrRepair's training procedure coincides with the initialization step of our BIFI algorithm, with heuristic perturbations used in Eq 3. We then apply BIFI on top of it for Round 1, 2. BIFI outperforms DrRepair, achieving a new state-of-the-art.

4.1.1. Github-Python

Dataset. To obtain an unlabeled dataset of code, we collected Python3 files from GitHub public repositories.[3] We then tokenize each code file using the builtin Python tokenizer, and keep code snippets of length 10–128 tokens, resulting in 3M code snippets. As the critic c, we use the Python AST parser,[4] which catches unbalanced parentheses, indentation errors, and other syntax errors. Concretely, we define $c(x) = 1$ (good) if the AST parser returns no errors for input code x, and $c(x) = 0$ (bad) otherwise. Using this critic, we obtain 38K snippets of bad code and 3M snippets of good code. From the 38K bad examples, we holdout 15K as the final test set, and make the remaining 23K bad examples available for BIFI. Our goal is to learn a fixer that repairs AST parse errors. We define that the fixer's repair is successful if the output code has no AST parse errors and has Levenshtein edit-distance (Levenshtein, 1966) less than 5 tokens from the input code. The evaluation metric is the fixer's repair accuracy on the test set (the heldout 15K examples of real bad code).

BIFI implementation details. For the architecture of the fixer and breaker, we use the encoder-decoder Transformer (Vaswani et al., 2017) with 4 layers, 8 attention heads, and hidden states of size 256. The model parameters are optimized by Adam (Kingma & Ba, 2015), with batch size of 20,000 tokens, learning rate 0.0001, and gradient clipping 1.0 (Pascanu et al., 2013), on one GPU (GTX Titan X). For generation, we use beam search with beam size 10, and keep predictions with Levenshtein edit-distance less than 5 tokens from the input.

To train the initial fixer f_0, we use random perturbations for the corruption procedure $b_{\text{synthetic}}$ (Eq 3), which drops, inserts, or replaces 1–3 tokens in code with uniform distribution. We apply $b_{\text{synthetic}}$ 8 times to each of the 2.6M good code snippets to prepare the initial training data $\mathcal{P}_{\text{synthetic}}$. We holdout 1% of $\mathcal{P}_{\text{synthetic}}$ as our dev set, which we use to perform early stopping. We then run the BIFI algorithm for $K = 2$ rounds.

4.1.2. DeepFix

Dataset. DeepFix (Gupta et al., 2017) contains C code submitted by students in an introductory programming course, of which 37K snippets are good (no compiler error) and 7K are bad (have compiler errors). Each code snippet has 25 lines on average. Within the 7K bad examples, we take 20% as a held-out test set. We make the remaining 80% available for BIFI. The goal is to learn a fixer that repairs compiler errors. Repair is successful if the output code has no compiler errors. The evaluation metric is the fixer's repair accuracy on the test set.

BIFI implementation details. We define the initial fixer f_0 as DrRepair (Yasunaga & Liang, 2020) (the existing best system on DeepFix), which is an encoder-decoder model trained in a procedure that corresponds exactly to the initialization step of BIFI (§3.1). Specifically, to train DrRepair, Yasunaga & Liang (2020) design heuristic perturbations for the corruption procedure $b_{\text{synthetic}}$, which mimics common code errors beginner and experienced programmers make (*e.g.*, typos, punctuation, keyword and type errors). We use the same training / dev data prepared and released by the authors to train the initial fixer. We then run the BIFI algorithm for $K = 2$ rounds. Our fixer and breaker have the same model architecture as DrRepair. At test time, following the original DrRepair, we repeatedly apply the fixer while the code still has errors, up to a maximum of 5 times.

4.2. Main results

We study the fixer's repair accuracy on GitHub-Python and DeepFix. Here "round k accuracy" means the repair accuracy of the fixer learned in round k, *i.e.*, f_k.

GitHub-Python. Table 1 shows the test results on GitHub-Python. We show the overall repair accuracy ("Total") as well as the breakdown over the error categories in the Python AST parser (table right). The initial fixer f_0 ("Initial") is trained on randomly perturbed, synthetic bad code. Our proposed method (BIFI) enables the initial fixer to be further trained on real bad code and bad code generated by the learned breaker, which outperforms the initial fixer significantly (+28.5% in overall repair accuracy, and consistently

[3] https:github.com

[4] https://docs.python.org/3/library/ast.html

Method		Test accuracy			
		Bad 100%	Bad 50%	Bad 10%	ref. Synthetic bad only
Initial	Round-0	62.0%	62.0%	62.0%	62.0%
FixerOnly	Round-2	88.6%	84.7%	78.5%	62.7%
BIFI	Round-2	90.5%	89.0%	86.7%	63.3%

Table 3. **Analysis varying the amount of (real) bad code on GitHub-Python**. "Bad 100%" is our original setting (with 23K real bad examples available for BIFI) and "Bad 50%" means only 50% of them (11.5K) are made available. "Synthetic bad only" means keeping training the fixer on synthetic bad examples. The result shows that (i) even when the amount of real bad examples is small (*e.g.*, "Bad 10%"), they are still useful, allowing FixerOnly /BIFI to perform better than using synthetic data alone; (ii) when the amount of real bad examples is small, BIFI exceeds FixerOnly by a larger margin, highlighting the usefulness of the bad examples generated by the learned breaker.

Method		Test accuracy
Initial	Round-0	62.0%
BIFI (**ours**)	Round-2	**90.5%**
– real bad	Round-2	84.6%
– critic	Round-2	84.0%
– both (backtranslation)	Round-2	80.1%

Table 4. **Performance comparison with backtranslation** on GitHub-Python. Backtranslation is equivalent to removing two components from BIFI: (i) using the critic to verify fix / break attempts ("critic") and (ii) training the fixer on real bad examples in addition to examples generated by the breaker ("real bad"). The result suggests that both of these components improve the learning of a fixer, and consequently BIFI outperforms backtranslation.

across all error categories). This result suggests that even if we start from a very simple initial fixer trained with random perturbations, BIFI can automatically turn it into a usable fixer with high repair accuracy—90.5% accuracy on real bad code. We also experimented with continuing training the initial fixer f_0 with synthetic perturbations only, for the same rounds of BIFI (hence, controlling the amount of training data seen by the fixer); however, this did not provide an improvement, suggesting that there is a performance ceiling if we only train on synthetic data.

DeepFix. Table 2 shows the test results on DeepFix, along with prior works. Here we use the existing best system DrRepair as our initial fixer ("Initial"). BIFI outperforms the initial fixer by a substantial margin (+5.6% absolute over DrRepair), achieving a state-of-the-art accuracy of 71.7%. It is notable that DrRepair was trained with manually-designed heuristic perturbations, where the authors (Yasunaga & Liang, 2020) studied and mimicked various code errors beginner and experienced programmers make (*e.g.*, typos, punctuation and type errors); nevertheless, our result suggests that there is still room for improving the adaptation to a more realistic distribution of coding errors, and BIFI

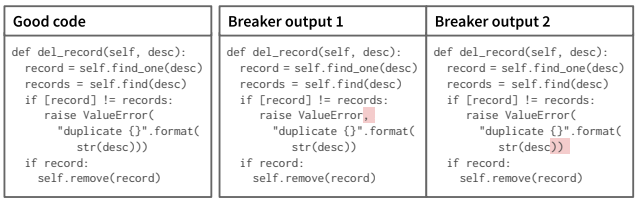

Figure 4. **Example of breaker outputs**. Given good code on the left, we sampled two outputs made by the breaker learned in BIFI round 1 (right). We observe that the breaker places large probability on errors seen in real bad code (*i.e.*, obsolete usage of `raise`, unbalanced parentheses in nested context).

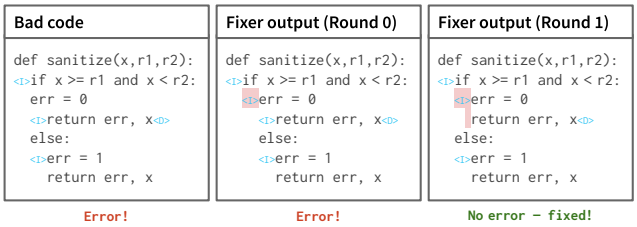

Figure 5. **Example of fixer outputs**. Given the bad code on the left (with an indentation error), the initial fixer (center) attempts to fix it by inserting an indent token ($\langle$I$\rangle$) to line 3, but fails to adjust (delete) the indent token on the next line. The initial fixer commonly makes this mistake due to the distribution mismatch between real errors and synthetic perturbations on which the initial fixer was trained (see §4.3.4). After one round of BIFI, the fixer in round 1 (right) learns to insert and delete the correct pair of indent tokens, fixing the error.

boosts repair accuracy without additional manual effort.

4.3. Analysis

We now analyze the key insights of BIFI. As the main properties of BIFI are to (i) add real bad examples to the training data if the critic accepts the fixer's output (the FixerOnly version), and to (ii) train the breaker to generate realistic bad examples (BIFI), we analyze their effects in §4.3.1 and §4.3.2. We also compare with backtranslation in §4.3.3. We then analyze how our fixer adapts towards real code distributions through quantitative and qualitative studies (§4.3.4).

4.3.1. Effect of real bad examples

FixerOnly enables training the fixer on real bad examples (but does not use the bad examples generated by the breaker). As Table 1 and 2 show, FixerOnly outperforms the initial fixer by a large margin (+27% on GitHub-Python). This result highlights the the importance of training on real bad examples.

We further analyze the effect of varying the *amount* of real bad examples, shown in Table 3. Here, "Bad 100%" is our original setting (with 23K real bad examples available for BIFI and FixerOnly) and "Bad 50%" means only 50% of them (11.5K) are made available. "Synthetic bad only" means keeping training the fixer on synthetic bad examples only. We find that while the repair accuracy drops as we decrease the amount of available real bad examples, a small amount of real bad examples (*e.g.*, "Bad 10%") is still useful,

Code error category	#Examples	Initial Round-0 Acc.	FixerOnly		BIFI	
			Round-1 Acc.	Round-2 Acc.	Round-1 Acc.	Round-2 Acc.
Total	15055	62.0%	86.8%	92.4%	88.0%	90.5%
Unbalanced Parentheses	3999	87.7%	93.3%	92.4%	94.1%	94.2%
Unclosed left parenthesis (not nested)	226	92.5%	94.6%	94.6%	94.7%	94.4%
Unclosed left parenthesis (nested)	3014	85.8%	92.8%	91.7%	93.8%	93.8%
Redundant right parenthesis	759	93.8%	95.3%	94.7%	95.4%	95.7%
Indentation Error	6307	39.4%	79.5%	83.7%	81.3%	85.9%
Expected indent	4311	46.4%	81.2%	84.1%	82.0%	85.3%
Unexpected indent	1966	24.6%	76.8%	83.8%	80.9%	88.4%
Invalid Syntax	4749	70.5%	90.9%	92.0%	91.6%	93.5%
Missing colon	663	98.3%	97.3%	97.4%	98.2%	98.0%
Missing comma (single-line list/tuple/dict)	694	95.4%	98.1%	97.4%	98.4%	98.3%
Missing comma (multi-line list/tuple/dict)	451	88.9%	92.5%	92.0%	94.5%	94.9%
Missing newline	52	84.6%	86.5%	88.5%	86.5%	88.5%
Missing parenthesis pair	634	82.5%	85.0%	86.4%	87.1%	88.3%
Redundant comma	152	73.7%	84.2%	91.4%	84.9%	92.1%
Redundant parenthesis pair	698	13.8%	80.7%	86.1%	80.1%	89.4%
Invalid use of comma (*e.g.*, "`raise OSError, "msg"`" → "`raise OSError("msg")`")	1138	61.3%	98.8%	99.1%	98.7%	99.4%
Other	267	60.7%	66.3%	64.4%	67.4%	66.7%

Table 5. **Code error categories in GitHub-Python, and repair accuracy**. Due to the mismatch between real errors and synthetic perturbations used for training, the initial fixer has lower accuracy on "nested" than "not nested" for "unbalanced parentheses" errors, but it catches up in BIFI round 1, 2. Similarly, the initial fixer's repair accuracy is very low for "redundant parenthesis pair" and "indentation error", but it improves significantly in round 1, 2. This result illustrates the effect of BIFI adapting the fixer towards real errors. See §4.3.4 for more analysis.

making FixerOnly perform better than using synthetic data alone (*i.e.*, 78.5% vs 62.7%).

4.3.2. Effect of bad examples generated by breaker

Recall that BIFI trains the fixer on both real bad examples and bad examples generated by the learned breaker. As Table 1 and 2 show, BIFI consistently provides an extra boost over FixerOnly, suggesting that the use of breaker outputs improves the fixer. Moreover, another benefit of the breaker is that one can sample many bad examples from the breaker to augment real bad examples, if their amount is limited. In Table 3 we find that BIFI is especially stronger than FixerOnly when the amount of available real bad examples is small (*e.g.*, "Bad 10%").

Figure 4 shows sample outputs made by the learned breaker b_1 given the good code on the left. We observe that the breaker generates errors seen in real bad code, *i.e.*, obsolete usage of `raise` in Python3 (center) and unbalanced parentheses in nested context (right). Compared to random perturbations that arbitrarily drop/insert tokens, the learned breaker improves the coverage and efficiency of the training data for the fixer.

4.3.3. Comparison with backtranslation

Table 4 compares our method (BIFI) with backtranslation. As discussed in §3.3, backtranslation is equivalent to removing two components from BIFI: (i) using the critic to verify fix/break attempts in data generation ("critic" in Table) and (ii) training the fixer on real bad examples besides examples generated by the breaker ("real bad"). We find that removing each component from BIFI hurts the performance (*e.g.*, 90%→84%), which suggests that both components are important to improve the quality of training data. With these two innovations, BIFI outperforms backtranslation by a large margin (+10% absolute on GitHub-Python).

4.3.4. How does the fixer adapt?

We take a closer look at how our fixer performs and adapts towards the real distribution of bad code. Table 5 shows fine-grained categories of code errors seen in GitHub-Python, and their repair accuracy.

In this categorization, we observe two examples of distribution mismatch between the real errors and synthetic perturbations: (i) random perturbations can generate this category of errors with high probability but with a wrong "sub-distribution" within it (*e.g.*, can generate "unbalanced parentheses" or "missing comma" errors but do not match the low-level distribution of real bad code, such as errors occurring more often in nested parentheses or in a multi-line list/tuple/dict; recall the Figure 2 top example); (ii) random perturbations can only generate this category of errors with very small probability (*e.g.*, "redundant parenthesis pair" and "indentation error", for which an *exact pair* of parentheses or indents/dedents need to be dropped or inserted; recall the Figure 2 bottom example). For (i), the result shows that the initial fixer trained with random perturbations has lower accuracy on "nested" than "not nested" for "unbalanced parentheses" errors, and on "multi-line" than "single-line" for "missing comma" errors; but the performance catches up in round 1,2, suggesting the effect of BIFI for addressing the low-level distribution mismatch. For (ii), the result shows that the initial fixer's repair accuracy is very low for "redundant parenthesis pair" and "indentation error", but it achieves significantly higher performance in round 1, 2 (*e.g.*, 39%→85% for indentation errors). A possible explanation is that as BIFI iteratively adds the successfully repaired cases to training, the fixer adapts to up-weight this category of error fixing, leading to improved accuracy.

Figure 5 provides examples of fixer outputs. Given the bad code on the left (with an indentation error), the initial fixer (center) attempts to fix it by inserting an indent token (⟨I⟩) to line 3 but fails to adjust (delete) the indent token on the following line. The initial fixer commonly makes this mistake for indentation errors, due to the mismatch between real errors and synthetic perturbations discussed above. After one round of BIFI, the fixer (Figure 5 right) learns to insert and delete the correct pair of indents, fixing the error.

5. Related work and discussion

Learning to repair code. Several works learn to repair code from labeled datasets of source code edits made by programmers, *e.g.*, error resolution records (Just et al., 2014; Chen et al., 2019; Mesbah et al., 2019; Bader et al., 2019; Tarlow et al., 2020; Ding et al., 2020). As labeled data is costly to prepare, various works study learning code fixers from unlabeled or synthetic data (Pu et al., 2016; Parihar et al., 2017; Ahmed et al., 2018; Pradel & Sen, 2018; Wang et al., 2018; Vasic et al., 2019; Gupta et al., 2019; Hajipour et al., 2019; Hellendoorn et al., 2020). In particular, Gupta et al. (2017) is an early work that randomly perturbs good code to prepare a synthetic paired data and trains a seq2seq neural network model as a fixer. Yasunaga & Liang (2020) improve on it by designing heuristic perturbations that mimic common errors made by programmers. Different from the above work, our method (BIFI) adapts a naive fixer trained with synthetic errors towards real errors without manual, domain-specific effort. For a more comprehensive review of automatic code repair, we refer readers to Monperrus (2020).

Denoising autoencoding. Denoising autoencoding (Vincent et al., 2008) trains a model that recovers original data from randomly corrupted versions, and is widely used as an effective self-supervised representation learning and pre-training strategy, *e.g.*, in computer vision (Erhan et al., 2010) and natural language processing (NLP) (Lewis et al. (2020), which randomly drops/replaces tokens in a sentence and learns to recover). Our initial fixer trained with random perturbations is a denoising autoencoder. Crucially, instead of using it purely for representation learning, we show that through the BIFI algorithm, one can turn the vanilla denoising autoencoder into a usable fixer that repairs real-world bad examples with high accuracy. On a related note, Lee et al. (2019) adapt a denoising autoencoder into an autocomplete system.

Domain adaptation. Domain adaptation aims to address the mismatch in data distribution between training and test domains (Daume III & Marcu, 2006; Quionero-Candela et al., 2009; Koh et al., 2021). Such domain shifts typically occur across related but different datasets (Torralba & Efros, 2011; Fang et al., 2013; Venkateswara et al., 2017; Yu et al., 2018b; 2019; Peng et al., 2019; Kamath et al., 2020), as well as from synthetic data to real data (including sim2real) (Wang et al., 2015; Ganin & Lempitsky, 2015; Richter et al., 2016; Peng et al., 2018; Hellendoorn et al., 2019; Xu et al., 2020; Bellemare et al., 2020), as synthetic data can be easier to obtain than real data. In our repair task, unpaired real data (code snippets on GitHub) is available but *paired* real data is costly to obtain. Hence we considered adaptation from synthetic paired data to real paired data. Within domain adaptation, our task is also related to the setting where unlabeled data in the test domain is available (Sun & Saenko, 2016; Hoffman et al., 2018; Sun et al., 2020) (in our case, real bad code). The difference is that as our outputs are structured, we have a critic to check if the fixer's output on the unlabeled data is correct. Leveraging this property, BIFI takes the correct outputs to create training data in the test domain (Eq 6); and trains a breaker to generate more data that simulates the test domain (Eq 8).

Data augmentation. Data augmentation aims to generate extra training data. A common approach is to increase the source side data, for instance by adding modifications or sampling from generative models (Hannun et al., 2014; Jia & Liang, 2016; Krizhevsky et al., 2017; Antoniou et al., 2017; Yasunaga et al., 2018; Yu et al., 2018a; Lee et al., 2019; Berthelot et al., 2019; Xie et al., 2020). Several works also study target side augmentation, which keeps multiple (valid) target predictions made by a model and adds to training. This is commonly used in structured generation problems such as semantic parsing, program synthesis and molecule generation (Liang et al., 2013; Berant et al., 2013; Guu et al., 2017; Min et al., 2019; Zhong et al., 2020; Yang et al., 2020). While our method also augments training data, it differs in two aspects: 1) we use the fixer and breaker to augment both the source and target sides; 2) our goal is not only to increase the amount of training data, but also to adapt to the distribution of interest (real bad examples).

Self-training. Self-training (Lee, 2013; McClosky et al., 2006; Kumar et al., 2020; Xie et al., 2021) applies a trained model to unlabeled data, obtains predicted targets (pseudo-labels), and uses them as extra training examples. Similarly, co-training (Blum & Mitchell, 1998) and tri-training (Zhou & Li, 2005) train multiple models and add predicted targets on which these models agree. Our method also applies trained models (breaker and fixer) to unlabeled data to generate targets, with a difference that we use the critic to verify the predictions and only keep correct ones.

Unsupervised machine translation (MT). Unsupervised MT learns translators from unpaired corpora (Artetxe et al., 2018a;b; Lachaux et al., 2020). Backtranslation (Sennrich et al., 2016; Lample et al., 2018a;b) is a common approach that uses the target-to-source model to generate noisy sources and then trains the source-to-target model to reconstruct the targets (also related to cycle-consistency (Zhu et al., 2017; Hoffman et al., 2018) and style transfer (Shen et al., 2017; Yang et al., 2018; Zhang et al., 2019)). One may view the "bad-side" and "good-side" in our repair task as two source/target languages in MT and apply backtranslation. The main difference is that the repair task has a critic, which motivated our BIFI algorithm: BIFI (i) uses the critic to verify if the generated examples are actually fixed or broken (Eq 6, 8), and (ii) trains the fixer on *real* bad examples besides

examples generated by the breaker (Eq 9). We found these techniques improve the correctness of the generated training data (§4.3.3). While we focused on code repair in this work, we hope that BIFI can be applied to unsupervised MT and style transfer by introducing a human-based or learned critic.

6. Conclusion

We considered the problem of learning a fixer from unpaired data and a critic (code analyzer or compiler), and proposed a new approach, Break-It-Fix-It (BIFI). The idea of BIFI is to train a breaker and use the critic to amass more realistic and correct paired data for training the fixer. Using two code repair datasets (GitHub-Python and DeepFix), we showed how BIFI adapts baseline fixers towards realistic distributions of code errors, achieving improved repair performance. We note that BIFI is not about simply collecting more training data, but rather turning raw unlabeled data into *usable* paired data with the help of a critic. This is a potentially powerful and general framework applicable to many areas such as molecular design, text editing, and machine translation.

Acknowledgments

We thank Michael Xie, members of the Stanford P-Lambda, SNAP and NLP groups, as well as our anonymous reviewers for valuable feedback. This work was supported in part by Funai Foundation Fellowship and NSF CAREER Award IIS-1552635.

Reproducibility

Code and data are available at `https://github.com/michiyasunaga/bifi`. Experiments are available at `https://worksheets.codalab.org/worksheets/0xfddb2ef01a9f4dc0b5d974a5a97174be`.

References

Ahmed, U. Z., Kumar, P., Karkare, A., Kar, P., and Gulwani, S. Compilation error repair: for the student programs, from the student programs. In *International Conference on Software Engineering (ICSE)*, pp. 78–87, 2018.

Antoniou, A., Storkey, A., and Edwards, H. Data augmentation generative adversarial networks. *arXiv preprint arXiv:1711.04340*, 2017.

Artetxe, M., Labaka, G., and Agirre, E. Unsupervised statistical machine translation. *arXiv preprint arXiv:1809.01272*, 2018a.

Artetxe, M., Labaka, G., Agirre, E., and Cho, K. Unsupervised neural machine translation. In *International Conference on Learning Representations (ICLR)*, 2018b.

Bader, J., Scott, A., Pradel, M., and Chandra, S. Getafix: Learning to fix bugs automatically. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–27, 2019.

Bellemare, M. G., Candido, S., Castro, P. S., Gong, J., Machado, M. C., Moitra, S., Ponda, S. S., and Wang, Z. Autonomous navigation of stratospheric balloons using reinforcement learning. *Nature*, 588, 2020.

Berant, J., Chou, A., Frostig, R., and Liang, P. Semantic parsing on freebase from question-answer pairs. In *Empirical Methods in Natural Language Processing (EMNLP)*, 2013.

Berthelot, D., Carlini, N., Goodfellow, I., Papernot, N., Oliver, A., and Raffel, C. A. Mixmatch: A holistic approach to semi-supervised learning. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2019.

Blum, A. and Mitchell, T. Combining labeled and unlabeled data with co-training. In *Conference on computational learning theory*, 1998.

Chen, Z., Kommrusch, S. J., Tufano, M., Pouchet, L.-N., Poshyvanyk, D., and Monperrus, M. Sequencer: Sequence-to-sequence learning for end-to-end program repair. *IEEE Transactions on Software Engineering*, 2019.

Daume III, H. and Marcu, D. Domain adaptation for statistical classifiers. *Journal of artificial Intelligence research*, 2006.

Dinella, E., Dai, H., Li, Z., Naik, M., Song, L., and Wang, K. Hoppity: Learning graph transformations to detect and fix bugs in programs. In *International Conference on Learning Representations (ICLR)*, 2020.

Ding, Y., Ray, B., Devanbu, P., and Hellendoorn, V. J. Patching as translation: the data and the metaphor. In *Automated Software Engineering (ASE)*, 2020.

Erhan, D., Courville, A., Bengio, Y., and Vincent, P. Why does unsupervised pre-training help deep learning? In *Artificial Intelligence and Statistics (AISTATS)*, pp. 201–208, 2010.

Fang, C., Xu, Y., and Rockmore, D. N. Unbiased metric learning: On the utilization of multiple datasets and web images for softening bias. In *International Conference on Computer Vision (ICCV)*, 2013.

Ganin, Y. and Lempitsky, V. Unsupervised domain adaptation by backpropagation. In *International Conference on Machine Learning (ICML)*, 2015.

Gupta, R., Pal, S., Kanade, A., and Shevade, S. K. Deepfix: Fixing common C language errors by deep learning. In *Association for the Advancement of Artificial Intelligence (AAAI)*, 2017.

Gupta, R., Kanade, A., and Shevade, S. Deep reinforcement learning for programming language correction. In *Association for the Advancement of Artificial Intelligence (AAAI)*, 2019.

Guu, K., Pasupat, P., Liu, E. Z., and Liang, P. From language to programs: Bridging reinforcement learning and maximum marginal likelihood. In *Association for Computational Linguistics (ACL)*, 2017.

Hajipour, H., Bhattacharya, A., and Fritz, M. Samplefix: Learning to correct programs by sampling diverse fixes. *arXiv preprint arXiv:1906.10502*, 2019.

Hannun, A., Case, C., Casper, J., Catanzaro, B., Diamos, G., Elsen, E., Prenger, R., Satheesh, S., Sengupta, S., Coates, A., et al. Deep speech: Scaling up end-to-end speech recognition. *arXiv preprint arXiv:1412.5567*, 2014.

Hellendoorn, V. J., Proksch, S., Gall, H. C., and Bacchelli, A. When code completion fails: A case study on real-world completions. In *International Conference on Software Engineering (ICSE)*, 2019.

Hellendoorn, V. J., Sutton, C., Singh, R., Maniatis, P., and Bieber, D. Global relational models of source code. In *International Conference on Learning Representations (ICLR)*, 2020.

Hoffman, J., Tzeng, E., Park, T., Zhu, J.-Y., Isola, P., Saenko, K., Efros, A., and Darrell, T. Cycada: Cycle-consistent adversarial domain adaptation. In *International Conference on Machine Learning (ICML)*, 2018.

Jia, R. and Liang, P. Data recombination for neural semantic parsing. In *Association for Computational Linguistics (ACL)*, 2016.

Jin, W., Yang, K., Barzilay, R., and Jaakkola, T. Learning multimodal graph-to-graph translation for molecular optimization. In *International Conference on Learning Representations (ICLR)*, 2019.

Just, R., Jalali, D., and Ernst, M. D. Defects4j: A database of existing faults to enable controlled testing studies for java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pp. 437–440, 2014.

Kamath, A., Jia, R., and Liang, P. Selective question answering under domain shift. In *Association for Computational Linguistics (ACL)*, 2020.

Kingma, D. and Ba, J. Adam: A method for stochastic optimization. In *International Conference on Learning Representations (ICLR)*, 2015.

Koh, P. W., Sagawa, S., Marklund, H., Xie, S. M., Zhang, M., Balsubramani, A., Hu, W., Yasunaga, M., Phillips, R. L., Beery, S., et al. Wilds: A benchmark of in-the-wild distribution shifts. In *International Conference on Machine Learning (ICML)*, 2021.

Krizhevsky, A., Sutskever, I., and Hinton, G. E. Imagenet classification with deep convolutional neural networks. *Communications of the ACM*, 2017.

Kumar, A., Ma, T., and Liang, P. Understanding self-training for gradual domain adaptation. In *International Conference on Machine Learning (ICML)*, 2020.

Lachaux, M.-A., Roziere, B., Chanussot, L., and Lample, G. Unsupervised translation of programming languages. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2020.

Lample, G., Conneau, A., Denoyer, L., and Ranzato, M. Unsupervised machine translation using monolingual corpora only. In *International Conference on Learning Representations (ICLR)*, 2018a.

Lample, G., Ott, M., Conneau, A., Denoyer, L., and Ranzato, M. Phrase-based & neural unsupervised machine translation. In *Empirical Methods in Natural Language Processing (EMNLP)*, 2018b.

Lee, D.-H. Pseudo-label: The simple and efficient semi-supervised learning method for deep neural networks. In *Workshop on challenges in representation learning, International Conference on Machine Learning (ICML)*, 2013.

Lee, M., Hashimoto, T. B., and Liang, P. Learning autocomplete systems as a communication game. In *Advances in Neural Information Processing Systems (NeurIPS) Workshop on Emergent Communication*, 2019.

Levenshtein, V. I. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady*, 1966.

Lewis, M., Liu, Y., Goyal, N., Ghazvininejad, M., Mohamed, A., Levy, O., Stoyanov, V., and Zettlemoyer, L. Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. In *Association for Computational Linguistics (ACL)*, 2020.

Liang, P., Jordan, M. I., and Klein, D. Learning dependency-based compositional semantics. *Computational Linguistics*, 2013.

McClosky, D., Charniak, E., and Johnson, M. Effective self-training for parsing. In *North American Association for Computational Linguistics (NAACL)*, 2006.

Mesbah, A., Rice, A., Johnston, E., Glorioso, N., and Aftandilian, E. Deepdelta: learning to repair compilation errors. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 925–936, 2019.

Min, S., Chen, D., Hajishirzi, H., and Zettlemoyer, L. A discrete hard em approach for weakly supervised question answering. In *Empirical Methods in Natural Language Processing (EMNLP)*, 2019.

Monperrus, M. The living review on automated program repair. *Technical Report hal-01956501. HAL/archives-ouvertes.fr.*, 2020.

Parihar, S., Dadachanji, Z., Praveen Kumar Singh, R. D., Karkare, A., and Bhattacharya, A. Automatic grading and feedback using program repair for introductory programming courses. In *ACM Conference on Innovation and Technology in Computer Science Education*, 2017.

Pascanu, R., Mikolov, T., and Bengio, Y. On the difficulty of training recurrent neural networks. In *International conference on machine learning (ICML)*, pp. 1310–1318, 2013.

Peng, X., Usman, B., Kaushik, N., Wang, D., Hoffman, J., and Saenko, K. Visda: A synthetic-to-real benchmark for visual domain adaptation. In *Computer Vision and Pattern Recognition (CVPR)*, 2018.

Peng, X., Bai, Q., Xia, X., Huang, Z., Saenko, K., and Wang, B. Moment matching for multi-source domain adaptation. In *International Conference on Computer Vision (ICCV)*, 2019.

Pradel, M. and Sen, K. Deepbugs: A learning approach to name-based bug detection. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):1–25, 2018.

Pu, Y., Narasimhan, K., Solar-Lezama, A., and Barzilay, R. sk_p: a neural program corrector for moocs. In *Companion Proceedings of the 2016 ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity*, pp. 39–40, 2016.

Quionero-Candela, J., Sugiyama, M., Schwaighofer, A., and Lawrence, N. D. *Dataset shift in machine learning*. The MIT Press, 2009.

Richter, S. R., Vineet, V., Roth, S., and Koltun, V. Playing for data: Ground truth from computer games. In *European conference on computer vision*, 2016.

Sennrich, R., Haddow, B., and Birch, A. Improving neural machine translation models with monolingual data. In *Association for Computational Linguistics (ACL)*, 2016.

Seo, H., Sadowski, C., Elbaum, S., Aftandilian, E., and Bowdidge, R. Programmers' build errors: A case study at google. In *International Conference on Software Engineering (ICSE)*, 2014.

Shen, T., Lei, T., Barzilay, R., and Jaakkola, T. Style transfer from non-parallel text by cross-alignment. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2017.

Sun, B. and Saenko, K. Deep coral: Correlation alignment for deep domain adaptation. In *European Conference on Computer Vision (ECCV)*, 2016.

Sun, Y., Wang, X., Liu, Z., Miller, J., Efros, A. A., and Hardt, M. Test-time training for out-of-distribution generalization. In *International Conference on Machine Learning (ICML)*, 2020.

Taghipour, K. and Ng, H. T. A neural approach to automated essay scoring. In *Empirical Methods in Natural Language Processing (EMNLP)*, 2016.

Tarlow, D., Moitra, S., Rice, A., Chen, Z., Manzagol, P.-A., Sutton, C., and Aftandilian, E. Learning to fix build errors with graph2diff neural networks. In *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*, pp. 19–20, 2020.

Torralba, A. and Efros, A. A. Unbiased look at dataset bias. In *Computer Vision and Pattern Recognition (CVPR)*, 2011.

Vasic, M., Kanade, A., Maniatis, P., Bieber, D., and Singh, R. Neural program repair by jointly learning to localize and repair. In *International Conference on Learning Representations (ICLR)*, 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 (NeurIPS)*, 2017.

Venkateswara, H., Eusebio, J., Chakraborty, S., and Panchanathan, S. Deep hashing network for unsupervised domain adaptation. In *Computer Vision and Pattern Recognition (CVPR)*, pp. 5018–5027, 2017.

Vincent, P., Larochelle, H., Bengio, Y., , and Manzagol, P.-A. Extracting and composing robust features with denoising autoencoders. In *International Conference on Machine Learning (ICML)*, 2008.

Wang, K., Singh, R., and Su, Z. Dynamic neural program embeddings for program repair. In *International Conference on Learning Representations (ICLR)*, 2018.

Wang, Y., Berant, J., and Liang, P. Building a semantic parser overnight. In *Association for Computational Linguistics (ACL)*, 2015.

Xie, Q., Dai, Z., Hovy, E., Luong, M.-T., and Le, Q. V. Unsupervised data augmentation for consistency training. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2020.

Xie, S. M., Kumar, A., Jones, R., Khani, F., Ma, T., and Liang, P. In-n-out: Pre-training and self-training using auxiliary information for out-of-distribution robustness. In *International Conference on Learning Representations (ICLR)*, 2021.

Xu, S., Semnani, S. J., Campagna, G., and Lam, M. S. Autoqa: From databases to qa semantic parsers with only synthetic training data. In *Empirical Methods in Natural Language Processing (EMNLP)*, 2020.

Yang, K., Jin, W., Swanson, K., Barzilay, R., and Jaakkola, T. Improving molecular design by stochastic iterative target augmentation. In *International Conference on Machine Learning (ICML)*, 2020.

Yang, Z., Hu, Z., Dyer, C., Xing, E. P., and Berg-Kirkpatrick, T. Unsupervised text style transfer using language models as discriminators. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2018.

Yasunaga, M. and Liang, P. Graph-based, self-supervised program repair from diagnostic feedback. In *International Conference on Machine Learning (ICML)*, 2020.

Yasunaga, M., Kasai, J., and Radev, D. Robust multilingual part-of-speech tagging via adversarial training. In *North American Association for Computational Linguistics (NAACL)*, 2018.

Yu, T., Yasunaga, M., Yang, K., Zhang, R., Wang, D., Li, Z., and Radev, D. Syntaxsqlnet: Syntax tree networks for complex and cross-domaintext-to-sql task. In *Empirical Methods in Natural Language Processing (EMNLP)*, 2018a.

Yu, T., Zhang, R., Yang, K., Yasunaga, M., Wang, D., Li, Z., Ma, J., Li, I., Yao, Q., Roman, S., et al. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-sql task. In *Empirical Methods in Natural Language Processing (EMNLP)*, 2018b.

Yu, T., Zhang, R., Yasunaga, M., Tan, Y. C., Lin, X. V., Li, S., Er, H., Li, I., Pang, B., Chen, T., et al. Sparc: Cross-domain semantic parsing in context. In *Association for Computational Linguistics (ACL)*, 2019.

Zhang, Z., Ren, S., Liu, S., Wang, J., Chen, P., Li, M., Zhou, M., and Chen, E. Style transfer as unsupervised machine translation. In *Association for the Advancement of Artificial Intelligence (AAAI)*, 2019.

Zhong, V., Lewis, M., Wang, S. I., and Zettlemoyer, L. Grounded adaptation for zero-shot executable semantic parsing. In *Empirical Methods in Natural Language Processing (EMNLP)*, 2020.

Zhou, Z.-H. and Li, M. Tri-training: Exploiting unlabeled data using three classifiers. *IEEE Transactions on knowledge and Data Engineering*, 2005.

Zhu, J.-Y., Park, T., Isola, P., and Efros, A. A. Unpaired image-to-image translation using cycle-consistent adversarial networks. In *International Conference on Computer Vision (ICCV)*, 2017.