CIRCLE: Continual Repair across Programming Languages

Wei Yuan[*]
w.yuan@uq.edu.au
School of Information Technology
and Electrical Engineering
The University of Queensland
Australia

Quanjun Zhang[*]
quanjun.zhang@smail.nju.edu.cn
State Key Laboratory for Novel
Software Technology
Nanjing University, China

Tieke He[†]
hetieke@gmail.com
State Key Laboratory for Novel
Software Technology
Nanjing University, China

Chunrong Fang[†]
fangchunrong@nju.edu.cn
State Key Laboratory for Novel
Software Technology
Nanjing University, China

Nguyen Quoc Viet Hung
henry.nguyen@griffith.edu.au
Institute for Integrated and Intelligent
Systems
Griffith University, Australia

Xiaodong Hao
mf21320054@smail.nju.edu.cn
State Key Laboratory for Novel
Software Technology
Nanjing University, China

Hongzhi Yin
h.yin1@uq.edu.au
School of Information Technology
and Electrical Engineering
The University of Queensland
Australia

ABSTRACT

Automatic Program Repair (APR) aims at fixing buggy source code with less manual debugging efforts, which plays a vital role in improving software reliability and development productivity. Recent APR works have achieved remarkable progress via applying deep learning (DL), particularly neural machine translation (NMT) techniques. However, we observe that existing DL-based APR models suffer from at least two severe drawbacks: (1) Most of them can only generate patches for a single programming language, as a result, to repair multiple languages, we have to build and train many repairing models. (2) Most of them are developed offline. Therefore, they won't function when there are new-coming requirements.

To address the above problems, a T5-based APR framework equipped with continual learning ability across multiple programming languages is proposed, namely Cont*I*nual *R*epair a*C*ross Programming Languag*E*s (*CIRCLE*). Specifically, (1) CIRCLE utilizes a prompting function to narrow the gap between natural language processing (NLP) pre-trained tasks and APR. (2) CIRCLE adopts a difficulty-based rehearsal strategy to achieve lifelong learning for APR without access to the full historical data. (3) An elastic regularization method is employed to strengthen CIRCLE's continual learning ability further, preventing it from catastrophic forgetting. (4) CIRCLE applies a simple but effective re-repairing method to revise generated errors caused by crossing multiple programming languages.

We train CIRCLE for four languages (i.e., C, JAVA, JavaScript, and Python) and evaluate it on five commonly used benchmarks. The experimental results demonstrate that CIRCLE not only effectively and efficiently repairs multiple programming languages in continual learning settings, but also achieves state-of-the-art performance (e.g., fixes 64 Defects4J bugs) with a single repair model.

CCS CONCEPTS

• **Computing methodologies** → *Lifelong machine learning*; *Artificial intelligence*; • **Software and its engineering** → **Software testing and debugging**; **Software defect analysis**; **Empirical software validation**.

KEYWORDS

Automatic Program Repair, Neural Machine Translation, Lifelong Learning, AI and Software Engineering

ACM Reference Format:
Wei Yuan, Quanjun Zhang, Tieke He, Chunrong Fang, Nguyen Quoc Viet Hung, Xiaodong Hao, and Hongzhi Yin. 2022. CIRCLE: Continual Repair across Programming Languages. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '22), July 18–22, 2022, Virtual, South Korea.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3533767.3534219

1 INTRODUCTION

Automatic Program Repair (APR) is critical for software developers since manually detecting and fixing bugs is a labor-intensive and time-consuming task [81]. With the recent advances in deep learning (DL), a lot of APR approaches have been proposed to use neural network techniques to learn the bug-fixing patterns from accessible

[*] Both authors contributed equally to this research.
[†] Corresponding authors.

ISSTA '22, July 18–22, 2022, Virtual, South Korea
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9379-9/22/07…$15.00
https://doi.org/10.1145/3533767.3534219

code repositories [5, 17, 87]. Assisted by deep learning's powerful ability to learn hidden and intricate relationships from massive data, DL-based APRs achieve remarkable performance [25, 93].

Generally, the DL-based APRs is composed by two parts [12]: an encoder that extracts the meaning of buggy code with necessary context and converts it into fix-length vectors; and a decoder that generates the correct statements from the encoder's output [24, 34, 42]. These encoder-decoder models usually treat the program repair task as a translation from buggy code to fixed code. For example, Tufano et al. [77] adopt a classical neural machine translation (NMT) model to generate the fix patches. CoCoNut [45] utilizes two separate encoders to encode the buggy lines and surrounding context. Further, CURE [25] employs GPT to provide contextual embeddings for CoCoNut. Zhu et al. [93] design a syntax-guided edit decoder to generate patches in a refined way.

Despite the recent achievements, existing DL-based APR techniques have at least two limitations. First, *most of them can only fix bugs for a single language.* If we want to repair codes for many languages, we have to train and store a corresponding number of models. In addition, these models are trained independently, which limits them to transfer the latent knowledge learned from other languages. We argue that such "single-language learning setting" is artificial because the underlying code understanding and bug identifying abilities may share a large common across languages. For example, a developer who is proficient in certain language, when (s)he reads code with unfamiliar language, (s)he can still understand the general ideas and intuitively point out the potential problem. Recent research also indicates that multilingual training data could improve performance on several code-related tasks (e.g., code summarization and method name prediction) [2, 53]. Therefore, learning multiple languages fixing may be more efficient and effective than separately learn different languages. Besides, the scalability of cross-lingual repairing model will also be better than these traditional APR models, since the prior one can handle various languages with just one model. And the scalability problem will probably be severer with the growing of neural network models' size.

Another shortcoming for current APRs is that *they are developed in an offline manner*, so they cannot continually improve their bug-fixing ability. This shortcoming decreases the value of DL-based APRs in real-world scenarios where new task requirements increase constantly. The "task requirements increase constantly" refers to that only a part of tasks are targeted to be solved at the first time and new task requirements are proposed afterwards. This case happens when tackling all tasks at once is very complex and time-consuming or when the task requirements cannot be fully obtained initially. For example, when companies decide to provide APR service, they tend to provide service for the most in-demand programming language in the first place, since creating a high-quality and enterprise-level APR model is costly. Later, as the business grew, they would like to enrich their APR service for other languages. In addition, collecting bug-fixing corpus is also an adaptive and continual process, even though all task requirements are covered initially, the model still needs to expand their knowledge on new corpus. Conventional APR models tend to overwrite the knowledge learned from previous tasks when learning new tasks. As a result, every time the task requirement increases, APR models have to be retrained on all corpus, which is time-consuming.

To mitigate the above issues, a new DL-based APR model that can process multi-type programming languages and continually learn defects fixing is desirable. However, there are two main challenges in implementing such APR models. First, repairing defects cross languages is more difficult than for a single language [79]. Therefore, it is essential to efficiently and effectively exploit the power of deep learning [40], especially the large pre-trained neural network models, which are commonly used in NLP [59, 63, 84]. Second, models are prone to forget the knowledge obtained from previous tasks when they are learning on new corpus or new tasks, i.e., they are struggling with catastrophic forgetting[1] [16, 51]. How to prevent this forgetting is non-trivial.

In this paper, we propose *CIRCLE* (short for ContInual Repair aCross Programming LanguagEs), which is able to continually learn bug fixing across multiple programming languages. To be specific, CIRCLE incorporates a prompt template, a T5-based APR model, and a re-repairing mechanism to address the first limitation and challenge mentioned above (i.e. repairing across languages and effectively utilizing pre-trained models). Concretely, T5 [64] is a widely used NLP pre-trained model that exhibits a formidable capacity for handling multiple tasks [1, 50]. The prompt template converts bug-fixing inputs into fill-in-the-blank form, closing the gap between T5's pre-trained task and program repair. This prompt template helps model better exploit the knowledge learned from pre-trained tasks [33, 41, 69, 71]. The re-repairing mechanism is designed to eliminate incorrectly generated patches caused by crossing languages.

To tackle the second issue and challenge (i.e., continually learning bug-fixing without catastrophic forgetting), CIRCLE applies a rehearsal method and an elastic regularization. The rehearsal method stores a small set of data from past datasets to simulate the historical data distribution and replays them in the later learning period. The main challenge is how to select the set of "representative data". CIRCLE proposes a novel data selection scheme that collects representative examples from historical data for bug repairing based on the difficulty. However, since the size of selected samples is desired to be small enough to reduce the resource costs, solely using the difficulty-based example replay method cannot adequately avoid the forgetting problem. Therefore, CIRCLE employs a parameter updating regularization approach based on Elastic Weight Consolidation (EWC) [29]. Moreover, CIRCLE calculates the Fisher Matrix [74] on the chosen examples rather than on the whole historical data to approximate EWC values so that it does not need to keep the whole historical data. The EWC imposes restrictions on parameters that play a vital role in previous task learning, forcing model to learn current and future tasks via adapting other parameters.

Extensive experiments are conducted across 4 popular programming languages (C, JAVA, JavaScript, and Python) on 5 benchmarks to test the effectiveness of our CIRCLE. The experimental results demonstrate that a *single* CIRCLE model can continually learn

[1] Catastrophic forgetting means that neural network model tends to completely forget learned knowledge when learning new information.

program repair crossing multiple languages settings without severely forgetting previous knowledge. Furthermore, CIRCLE outperforms the conventional state-of-the-art DL-based APRs which are dedicated trained for certain language. At last, each component's importance is studied in the ablation study in detail. The code, experimental results and processed data is available[2].

To sum up, the main contributions of this paper are three-fold:

- We propose CIRCLE, a novel program repair framework that can continually learn bug-fixing across multiple languages. To the best of our knowledge, we are the first to explore multi-language program repair in continual learning scenarios.
- A prompt-based template is developed to convert program repair into "fill-in-the-blank" task, allowing the pre-trained model, T5, to perform the repair task effectively. A simple but effective re-repairing mechanism is designed to revise generation errors about crossing languages. In addition, a novel difficulty-based example replay and an EWC-based regularization method are proposed to mitigate catastrophic forgetting in continual learning settings.
- Extensive experiments are conducted with 4 commonly used programming languages and evaluated on 5 bug benchmarks to demonstrate that CIRCLE can continually learn bug repair crossing languages and outperform previous neural APR models. Ablation studies and further analyses are presented to discuss CIRCLE's performance.

The remainder of this paper is organized as follows. Section 2 provides the basic background related to our work. Section 3 introduces the details of our CIRCLE. Section 4 describes the datasets and metrics adopted in this paper, followed by experimental results and discussions. Section 5 presents the related work on program repair and lifelong learning. Section 6 concludes our work.

2 BACKGROUND

2.1 DL-based APR

DL-based APRs have achieved state-of-the-art performance on program repair task [9, 11, 45, 76, 88]. Most of them treat repairing as a neural machine translation task and optimize an encoder-decoder model on a set of bug-fix pairs to learn latent patterns based on supervised learning. The inputs and neural model architectures of DL-based APRs are various. For example, CoCoNut [45] separately encodes context and buggy codes by CNN networks. SequenceR [11] abstracts the buggy context and takes it as input together with buggy lines. Recently, pre-trained models are also used in DL-based APRs. CURE [25] employs GPT as token embedding layer. Mashhadi et al. [49] utilize CodeBERT to fix Java simple bugs.

However, to the best of our knowledge, repairing multiple programming languages' defects via a single model is still underexplored. In light of this, we propose to employ the recent pre-trained model T5 as the skeleton and build a repair model that can fix bugs across languages.

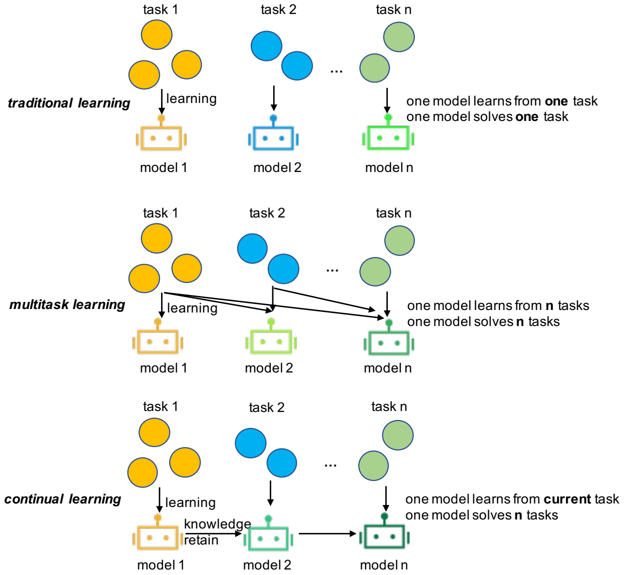


Figure 1: The difference among traditional learning, multitask learning, and continual learning paradigm.

2.2 Continual Learning

Continual Learning (also referred to as Lifelong Learning) is a type of machine learning paradigm, which aims to continually learn new tasks while not severely forget previously gained knowledge [60]. Formally, let f_t denote the model trained in task t, the new dataset D_{t+1} for task $t+1$ is used to update the model f_t. Continual Learning attempts to ensure that after updating, model f_{t+1} can have good performance in all seen tasks $1 : t + 1$.

Figure 1 illustrates the difference among traditional learning, multitask learning, and continual learning. With traditional learning settings, a model is trained on a certain dataset and can only complete a single corresponding task. Multitask Learning and Continual learning are similar in the sense that they both attempt to find a good solution across multiple tasks [54]. The main difference is that Multitask Learning has to keep access to all previous data. Once a new task and dataset are available, the model must be retrained on all historical datasets. Therefore, the cost of Multitask Learning is much expensive than Continual Learning. Formally, assuming that for task t, the dataset size is d_t and the training cost (e.g. training time, computational resources) is c_t. Then, in the task requirements growth progressively scenario, until task t arrives, the total training cost for Multitask Leaning is $\sum_{i=1}^{t} (t - i + 1)c_i$, and Multitask Learning needs to store all historical data $\sum_{i=1}^{t} d_i$. Whereas for Continual Learning, the training cost is $\sum_{i=1}^{t} c_i$ and it only needs to maintain current task's dataset. Briefly, Multitask Learning is a good choice for the issues that tasks and data are both changeless, i.e., task requirements are clear, and all data are available at the beginning stage, rather than for our paper's focused problems.

[2] https://github.com/2022CIRCLE/CIRCLE.

The major challenge of Continual Learning is catastrophic forgetting (or catastrophic interference), which indicates that artificial neural networks tend to abruptly "forget" the knowledge of previously learned tasks [29]. There are three main families of methods to mitigate this problem: Rehearsal, Regularization, and Architectural methods. Rehearsal methods rely on collecting a part of typical historical data, so that they can replay them in the future training [65–67]. Regularization methods apply some constraints to model's parameter updating, in order to strike a balance between stability and plasticity [29, 73]. Architectural methods attempt to dynamically change model's modular, however, model parameters will dramatically increase when the number of tasks grows [46, 83]. In this work, we focus on the hybrid of rehearsal and regularization methods.

2.3 Prompt for Pre-trained Model

A prompt is a piece of tokens inserted in the input, so that the original task can be formulated as a language modeling task. Prompt is used to fill the gap between pre-trained tasks and the downstream task, facilitating finetuning process. Following Raffel et al. and Khashabi et al. [28, 64], we manually design a set of prefixes as prompt to concatenate each input component.

3 APPROACH

To address the limitations and challenges mentioned in Section 1, we propose CIRCLE, a neural APR model that can continually learn defects fixing. In this section, we introduce the design and implementation of CIRCLE. First, we present the overview of CIRCLE in Section 3.1. CIRCLE aims to achieve both continual learning and multiple language repairing, which is more complex and also more practical than previous DL-based APRs. It is composed of five parts. First, CIRCLE leverages a large pre-trained model as its model skeleton to gain the strong learning ability (Section 3.3), meanwhile, it employs a prompt function to effectively finetune the pre-trained model (Section 3.2). Then, CIRCLE uses a novel difficulty-based rehearsal method (Section 3.4) and a parameter importance-based regularization (Section 3.5) to cope with forgetting problem. Finally, a simple but effective re-repairing approach is utilized to erase generation errors caused by crossing languages (Section 3.6).

3.1 Overview

Figure 2 presents the overview of our approach. As shown in the upper right part, CIRCLE can deal with task requirements increasing due to its continual program repair learning ability. Without loss of generality, we assume new language program repair tasks arrive over time with their corresponding datasets. CIRCLE automatically learns these tasks one by one based on task arriving order and does not need to retrain on or store the whole previous tasks' data.

For each task, CIRCLE consists of two stages: training stage and testing stage. During training stage, a manually designed prompt function at first converts the repairing input into a fill-in-the-blank form. Our training set is composed of two subsets: the current task corpus and a few examples selected from previous tasks. Then, these processed data are fed into a T5-based APR model. T5 utilizes a subword tokenization method to address out-of-vocabulary (OOV) problem. Unlike previous work [25], we keep the original tokenization vocabulary instead of building a new vocabulary using byte pair encoding (BPE) [70] algorithm. Because (1) we want APR model to inherit the natural language understanding ability and start learning repairing from a good initial point; (2) BPE needs to count the subwords frequency, however, the frequency is dynamically changed crossing languages. The T5-based APR generates candidate patches according to the prompted input and a loss function is applied to evaluate this generation. To alleviate catastrophic forgetting, CIRCLE should carefully update its parameters. Therefore, an EWC regularization is employed to compute the "importance" of each parameter for previous tasks, avoiding too much change of these more "important" parameters. Finally, APR model is updated based on the loss and EWC regularization. When the training is converged, we use this well-trained APR model to select a small set of examples from current task corpus via a novel data selection scheme. These selected examples are stored to be replayed in the forthcoming task training.

During each task's inference stage, the APR model receives all seen languages' buggy codes and repairs them through generating a group of candidate patches. In multiple programming language repairing scenarios, APR model is easy to incorrectly generate some keywords, since they have very similar semantic meanings. For example, Java and JavaScript's "null" is similar to Python's "None" in both program and natural language aspects. In some cases, our APR model will make mistakes about these keywords. However, the number of such keywords is not too much. We simply build a simple map to convert them after model's generation.

3.2 Prompt based Data Representation

The input of CIRCLE is composed of two parts: the buggy code and the surrounding context code. Traditional works attempt to seperately encode these two parts and then merge the encoding vectors [25, 45]. However, how to effectively fuse these seperated encoding vectors and eliminate the semantic gaps between two encoders is still worth discussing. Recently, Raffel et al. [64] propose a text-in-text-out input format, which concatenates different input components with some prefixed prompt. This mechanism is proved to be useful for finetuning pre-trained model in downstream tasks [27, 40]. In light of this, CIRCLE employs a manually designed prompt template to convert buggy code and corresponding context into a unified fill-in-the-blank format. As illustrated in Figure 3, we utilize "Buggy line:", "Context:" denote the buggy line and context codes, and then we use "The fixed code is:" to guide pre-trained model generate fixed program according to the previous input. Since T5 is pre-trained in fill-in-the-blank tasks with natural language, it is more natural for it to finetune on the prompted data.

In addition, CIRCLE utilizes subword tokenization method to address OOV problem. But we do not newly build a token vocabulary because we want to fully exploit the pre-trained knowledge from T5 and the frequency of tokens from the whole datasets is not available as we mentioned in Section 3.1. In other words, finetuning T5 in APR task can be viewed as "domain-adaptive" task to some extent in this paper, i.e. the gap between down-stream task and pre-trained task is close.

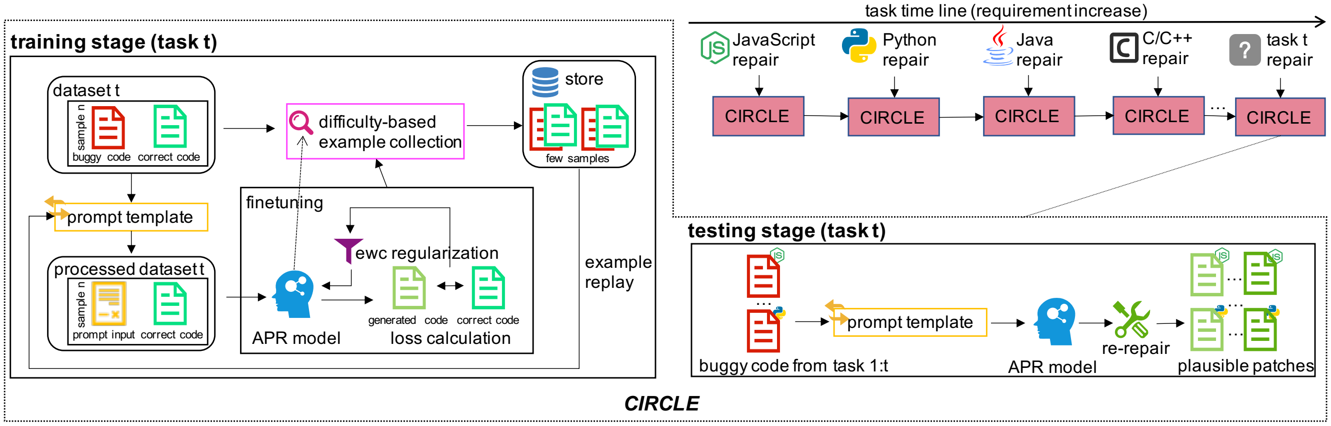
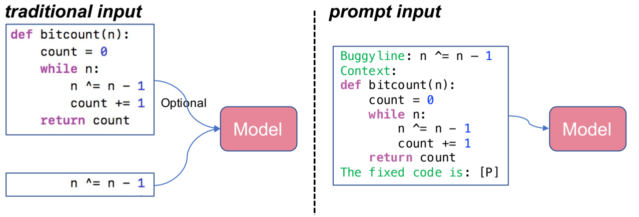


Figure 2: CIRCLE's overview.



```
def bitcount(n):
    count = 0
    while n:
        n ^= n - 1
        count += 1
    return count
```

```
        n ^= n - 1
```

```
Buggyline: n ^= n – 1
Context:
def bitcount(n):
    count = 0
    while n:
        n ^= n – 1
        count += 1
    return count
The fixed code is: [P]
```

Figure 3: The comparison of traditional input and our prompt-based input. The green text represents prompt, which indicates the semantic meaning of each input component. The prompt input formulates bug-fixing task as fill-in-the-blank task, which is similar to T5's pre-trained task.

3.3 T5 as APR Model Skeleton

T5 [64] is a kind of encoder-decoder transformer [80] model pre-trained in many tasks on over 750 GB datasets, achieving state-of-the-art performance on a variety of NLP tasks. Mastropaolo et al. [50] show that T5 can also perform well in many code-related tasks. Although they also employ T5 to solve an APR problem, we are the first to use T5 for APR in the continual learning scenarios and to solve the APR task with multiple languages simultaneously.

The encoder of T5 is a stack of transformer blocks, each of which contains two subcomponents: a multi-head self-attention layer followed by a position-wise feed-forward network. Layer normalization [4], residual connectors [20], and dropout operation [72] are also applied between each subcomponent to stabilize the training process.

The decoder of T5 is much similar to the encoder but it has attention mechanism after each self-attention so that it can attend to the output of the encoder. In addition, the attention is causality-enabled to avoid information leaking during decoding.

T5 comes in different sizes. In this paper, we do not modify the vocabulary size and use the pre-trained "t5-base" as the training starting point.

3.4 Difficulty-based Example Replay

One of the significant contributions of CIRCLE is that it enables APR models to continually learn bug fixing. To achieve this, CIRCLE incorporates a difficulty-based example replay and an EWC-based regularization to avoid forgetting. In this part, we introduce the novel difficulty-based example replay.

The core idea of example replay is to retain a small set of samples from previous datasets, and replay these samples in later training. As a result, model can avoid severely forgetting meanwhile does not need to retrain on whole historical data. Straightforwardly, the effectiveness of such rehearsal method largely relies on the selected examples. In this work, we propose to select the representative and diverse data based on difficulty. The intuition is that difficult data might be more informative and useful for improving current model. Since model has poor performance during previous training with these difficult data, it might incline to first forget the pattern learned from those data.

The difficulty selection criterion is as follows:

$$d_t(x_i^t, y_i^t) = \frac{L(x_i^t, y_i^t|\theta_t)}{|y_i^t|} \tag{1}$$

where x_i^t and y_i^t are the i-th data pair in task t. θ_t is the model that already well-trained in task t. $L(\cdot|\theta_t)$ is the loss function (e.g. cross-entropy) conditioned on model parameter θ_t. Since long sentences tend to accumulate higher loss values, we use the inverse of y's length as the normalization factor. Basically, Eq. 1 reflects the confidence of model θ_t when repairing the data x_i^t, where a higher value indicates x_i^t is more challenging to learn. We collect N samples that can maximize Eq. 1 from task t's datasets as the difficult example set E_t. N is much smaller than the whole dataset D_t. Consequently, for each task i, we maintain a corresponding example set E_i.

During continual training of APR, the previously selected difficult example set $E_{1:t}$ are integrated to the coming task $t + 1$'s dataset. Therefore, the objective of training is to find θ_{t+1} that can minimize the loss on the combined dataset:

$$\theta_{t+1} = \underset{\theta_t}{argmin} \sum_{(x,y)\in D_{t+1}\cup E_{1:t}} L(x, y|\theta_t) \tag{2}$$

3.5 Sampling-based EWC Regularization

Since the size of selected example set $|E_{1:t}|$ should be as small as possible to reduce computation and storage costs, the effects of remembering old patterns are not strong enough. Therefore, we further apply a constraint to avoid model losing previous knowledge. This constraint is based on Elastic Weight Consolidation (EWC) [29], which is widely adopted to overcome catastrophic forgetting in continual learning. EWC imposes restrictions on parameters according to the importance of the parameters for previous tasks. The more importance the parameters in previous tasks, the more tightly EWC restricts their updating. EWC uses Fisher Matrix as a measure of weight importance, the calculation is as follows:

$$F_i = \nabla^2 L(x, y|\theta_{t-1,i}) \tag{3}$$

F_i measures the importance of ith parameter in θ_{t-1}. F_i is used to regularize model's updating:

$$EWC(\theta_t) = \sum_i \lambda F_i (\theta_{t,i} - \theta_{t-1,i})^2 \tag{4}$$

where λ is hyper-parameter controls the contributions of EWC regularization.

However, directly calculate Eq. 3 is unscalable, since it needs to access to all historical data. To efficiently approaximate EWC, we only calculate Fisher Matrix on the data collected by our difficulty-based selection scheme, i.e. $(x, y) \in E_{1:t}$. Furthermore, we only sample M items from $E_{1:t}$ to further reduce the computation costs.

Finally, the objective function of training is changed to:

$$\theta_{t+1} = \underset{\theta_t}{argmin}(EWC(\theta_t) + \sum_{(x,y) \in D_{t+1} \cup E_{1:t}} L(x, y|\theta_t)) \tag{5}$$

3.6 Cross Language Re-repairing

After continual training, CIRCLE learns the latent repairing patterns of multiple language and can generate correct patches. However, unlike natural languages, programming language has strict formal grammar. Therefore, CIRCLE still faces three problems due to the complication of crossing language repairing. The first one is that CIRCLE has possibility to generate keywords that have the same semantic meaning but belongs to other languages. We name this problem as "keywords mismatch". For example, "None" in Python has the same semantic meaning to "null" in Java. In some cases, CIRCLE incorrectly generates "None" when fixing Java bugs. To address this mismatch problem, we build a simple mapping table to convert these mismatch words to corresponding one. Table 1 presents some examples from our keywords mapping table.

Table 1: Examples from keywords mapping table.

	C	Java	JavaScript	Python
1	NULL	null	null	None
2	max	Math.max	Math.max	max
3	min	Math.min	Math.min	min
4*	->	.	.	.
...	...	...	...	...

The other problem is "format mismatch", which refers to generating correct tokens but with incorrect form. The typical example is that CIRCLE tends to generate "==" as "= =" which will lead syntax error. We simply build a regular expression to remove unnecessary blank space.

Finally, since T5 model's tokenizer is designed for natural language, although it applies wordpiece algorithms, it still slightly suffers out-of-vocabulary problem for a few rare symbols. In our experiments, we find three symbols which are OOV tokens, however, they are frequently used in programming languages [3]. When generating these symbols, T5 will simply generate unknown tokens, which is quite inappropriate. The re-repairing mechanism replaces these unknown tokens with those special symbols.

4 EXPERIMENTAL SETUP

In this section, we introduce the experimental design, including the research questions we studied, the training datasets, evaluation benchmarks, and implementation details in the experiments.

4.1 Research Questions

CIRCLE is designed to fix multiple language bugs and to achieve continually learning bug fixes. To this end, we explore the following research questions (RQ):

- RQ1. Can CIRCLE effectively learn bug fixing in "task requirements increase constantly" scenario?
- RQ2. What is the performance of a single CIRCLE model compared to state-of-the-art APR methods?
- RQ3. What are the contributions of the different components of CIRCLE?

4.2 Datasets

Following previous works [25, 45], we directly use the CoCoNut's training data released on Github[4] as our method's training corpus. Same as these works, to make the evaluation realistic, we remove Java data committed after 2006. The size of original datasets is very large: 3 241 966, 480 777, 2 735 506, and 3 217 093 bug-fix pairs for Java, Python, C, and JavaScript, respectively. Limitted of computation power, we randomly select a part of remaining data (0.4 million) for training. The following experimental results indicate that training on such part of data still get great performance. We utilize the prompt template to concatenate the context data and buggy data, meanwhile, we truncate the inputs whose length are longer than 512 after subword tokenization.

For RQ1, to better align with continual learning scenario in real life, we assume that CIRCLE learns different languages repairing in the order of the language's popularity[5]: JavaScript → Python → Java → C. This setting is reasonable since in practice, companies often attempt to build tools for the most popular part and then refine them for other parts. The final trained CIRCLE model is used to compare with all state-of-the-art APR models to answer RQ2.

[3] We test the following symbols: "+ - * / % ** // == != <> > < >= <= = += -= *= /= %= //= **= & | ˆ « » { } \ \\ # $ ( )" only find three symbols meet this requirement: "<", "^", and "{".

[4] https://github.com/lin-tan/CoCoNut-Artifact/releases

[5] the popularity is according to language's change rate on Github. https://madnight.github.io/githut/#/pull_requests/2021/3

4.3 Implementation Details

All of our approaches are built based on PyTorch. We use the HuggingFace [85] implementation version of T5 and utilize "t5-base" as the initial point, considering previous work recommendation [15, 64] and our devices' limits. "t5-base" model contains 12 layers of transformer blocks and 12 attention heads. The optimizer is AdamW [44] with $3e-4$ learning rate. For each task (or splitted data in RQ2.), we train at most 20 epochs, and if the validation loss does not decrease after 3 epochs, the training process will be early stopped. The batch size is 64, the max length of input is set to 512, and the λ of EWC is 110000. The size of example set in difficulty-based example replay is restricted to 20000, which is much smaller than the total size of training data.

In inference stage, we use beam search with 250 beam size. Meanwhile, we apply top-k and top-p sampling during each step's token selection. Then, we re-repair the generated patches using the mapping table mentioned in Section 3.6. As a result, at most 1000 candidate patches are created by CIRCLE. For evaluation purpose only, following previous works [25, 45], three authors manually verify plausible patches (i.e. patches that successfully pass the test) based on ground truth patches (i.e., developer patches). And the plausible patches is considered to be correct only if all three authors agree it is equivalent to ground truth data semantically. All the training and evaluation of our methods are conducted on one CentOS 7.7 server with eight Tesla V100-SXM2 GPUs.

4.4 Benchmarks and Baselines

We use five benchmarks with four popular programming languages: Defects4J [26] and QuixBugs [35] for Java, BugIDs [19] for JavaScript, ManyBugs [31] for C, and QuixBugs [35] for Python, all of which have been adopted in previous APR work [18, 25, 45].

In order to verify the continual learning ability of CIRCLE, i.e. RQ1, we train a T5 model in the traditional finetuning way as our baseline. Finetuning way means that with the progress of tasks, model is initialized with checkpoint obtained from the last task and then, it is finetuned with current task's data. We name this model as Finetuned-APR. The training and inference parameters of Finetuned-APR is the same as CIRCLE. The comparison between CIRCLE and Finetuned-APR indicates the superiority of our approaches in task streaming settings.

For RQ2, we employ five benchmarks commonly used for APR that contain realistic bugs. To enable sufficient evaluations, we compare CIRCLE against 30 APR techniques covering different programming languages and technique categories. Specifically, all APR tools in the previous evaluation [45] and three recent state-of-the-art NMT-based tools [25, 93] are considered.

5 EVALUATION AND RESULTS

In this section, we evaluate CIRCLE and answer to four research questions based on experimental results.

5.1 RQ1: Can CIRCLE effectively learn bug fixing in "task requirements increase constantly" scenario?

We compare CIRCLE with the Finetuned-APR which we mentioned in Section 4.4. Note that the only difference between CIRCLE and Finetuned-APR is that the latter does not incorporate continual learning modules and is directly trained in a finetuning way. Figure 4 reports the performance trend of CIRCLE and Finetuned-APR with regarding to the task progress. To be specific, during the learning progress, we account how many bugs the model can fix for both current task and previous learned tasks. For example, after model learned Task 3 (i.e. Java) APR, we calculate the number of bugs it can fix on JavaScript (BugAID), Python (QuixBugs), and Java (Defects4J and QuixBugs) benchmarks. In Figure 4, CIRCLE and Finetuned-APR have the same performance for the first task. Then, Finetuned-APR is gradually falling behind our CIRCLE. Moreover, we can observe that: (1) For historical tasks, CIRCLE significantly outperform Finetuned-APR. For instance, after Task 3, Finetuned-APR can only repair 18 bugs on QuixBugs-Py, in contrast, CIRCLE fixes 26 bugs. This observation supports the effectiveness of our proposed continual learning modules. (2) Even for current tasks, CIRCLE still have better performance than Finetuned-APR. Take Task 2 as examples, Finetuned-APR correctly generates patches for 23 bugs on QuixBugs-Py, however, CIRCLE generates correct codes for 28 bugs. This observation supports our argument that the underlying code understanding and patches construction abilities are largely common among programming languages. As a result, if model does not severely forget previous obtained knowledge, this knowledge will help them in the following tasks. To further demonstrate that the previous task's knowledge is helpful, we train a T5-based APR model for Python without learning any previous task. It fixes 21 bugs, which is worse than Finetune-APR's performance on Task 2. This experimental result shows that although Finetune-APR tends to forget most previous knowledge, it still remembers a part of learned transferred knowledge and therefore, achieve better performance than the independently trained model.

Besides, we also draw the estimated upperbound in Figure 4 to show the "forgetting" problem that our CIRCLE still suffered. At each task point, we choose the best performance that CIRCLE or Finetuned-APR achieved on each benchmark as the upperbound performance. This upperbound curve indicates the degree of forgetting CIRCLE still suffers. Figure 4 shows that CIRCLE slightly falls behind the estimated upperbound compared to Finetuned-APR, further indicating the effectiveness of our continual learning strategies.

5.2 RQ2: What is the performance of a single CIRCLE model compared to the dedicatedly trained state-of-the-art APR methods?

To evaluate the performance of CIRCLE, we compare it with state-of-the-art techniques, including the traditional and DL-based ones. We adopt the final trained CIRCLE in RQ1 to perform repair tasks for five bug benchmarks across four programming languages. Table 2 shows the repair performance of a single CIRCLE model and the all selected baselines. In Table 2, each cell is represented as x/y, where x is the number of correct patches and y is the number of produced plausible patches. The results show that a single CIRCLE model achieves state-of-the-art performance in different languages.

As shown in Table 2, CIRCLE fixes 120 bugs for all bug benchmarks in four programming languages. For Java benchmark, CIRCLE fixes 19 bugs on QuickBugs, outperforming CoCoNut and is

Table 2: Comparison with state-of-the-art techniques. Note that due to the the accessible of source code and time consuming of repair process, following the most of existing APR work [18, 93], we reuse the released results from the recent work [45]. Meanwhile, the results of most recent tools (i.e., CURE and Recoder) are extracted from the original papers. According to [45], the two duplicated bugs (i.e., Closure 63 and Closure 93) are excluded since they are same with Closure 62 and 92, respectively. (†) No plausible patch of Recoder is reported in the original work and five originally reported correct patches are false positives in the latest official repository (https://github.com/pkuzqh/Recoder). (‡) The number is calculated by the exact match of the generated patch and the developer patch, and indicates minimal number of correct patches.

FL	ID	Tool	Java		C	Python	JavaScript
			Defects4J	QuixBugs	ManyBugs	QuixBugs	BugAID
			393 bugs	40 bugs	69 bugs	40 bugs	12 bugs
Standard	T1	Angelix	-	-	18/39	-	-
	T2	Prophet	-	-	15/39	-	-
	T3	SPR	-	-	11/38	-	-
	T4	Astor	-	6/11	-	-	-
	T5	LSRepair	19/37	-	-	-	-
	T6	DLFix	29/65	-	-	-	-
Supplemented	T7	JAID	9/31	-	-	-	-
	T8	HD-Repair	13/23	-	-	-	-
	T9	SketchFix	19/26	-	-	-	-
	T10	ssFix	20/60	-	-	-	-
	T11	CapGen	21/25	-	-	-	-
	T12	ConFix	22/92	-	-	-	-
	T13	Elixir	26/41	-	-	-	-
	T14	Hercules	49/72	-	-	-	-
Perfect	T15	SOSRepair	-	-	16/23	-	-
	T16	Nopol	2/9	1/4	-	-	-
	T17	(j)Kali	2/8	1/2	3/27	-	-
	T18	(j)GenProg	6/16	0/2	2/18	-	-
	T19	RSRepair	10/24	2/4	2/10	-	-
	T20	ARJA	12/36	-	-	-	-
	T21	SequenceR	12/19	-	-	-	-
	T22	ACS	16/21	-	-	-	-
	T23	SimFix	27/50	-	-	-	-
	T24	kPAR	29/56	-	-	-	-
	T25	AVATAR	29/50	-	-	-	-
	T26	FixMiner	34/62	-	-	-	-
	T27	TBar	52/85	-	-	-	-
	T28	CoCoNut	44/85	13/ 20	7/-	19/21	3/-
	T29	CURE	57 /104	26/35	-	-	-
	T30	Recoder†	64/-	-	-	-	-
	T31	CIRCLE	64/182	19‡/-	9‡/-	23‡/-	5‡/-

competitive with CURE. It is worthy noting that 19 bugs is calculated by exact match, which is usually the minimal of correct patch, while CoCoNut and CURE run all generated patches against the test suite. Meanwhile, CIRCLE correctly repairs 64 bugs and outperforms all of the previous traditional APR techniques on Defects4J v1.2. In particular, CIRCLE repairs 23.1% (12 bugs) more bugs than the state-of-the-art traditional approach (e.g., TBar). Meanwhile, CIRCLE can fix more bugs than most DL-based APR techniques (e.g., CoCoNut and CURE). CIRCLE is also found to be competitive with the most recent DL-based approach, Recoder, which is reported to be the first DL-based APR approach that has outperformed the traditional APR approaches. For other three language benchmarks, Table 2 shows that CIRCLE is the best technique on two of the three benchmarks (i.e., fixs 19 bugs for QuickBugsPY, 9 bugs for ManyBugs, 5 bugs for BugAID, respectively), indicating that the repair ability across different programming languages with a single CIRCLE model. It is worthy noting that most existing DL-based approaches (e.g., CURE) consider complex code-aware characteristics (e.g., code edits and abstract syntax tree) [93], while CIRCLE treats the program repair process as a simple machine translation task on a sequence of tokens. We only use around 4.13% (400 000/9 675 342) of the data dataset compared with CoCoNut. We also set the beam size as 250 while CURE's beam is configured to 1000. According to previous work[58, 76], a larger training set and beam size may

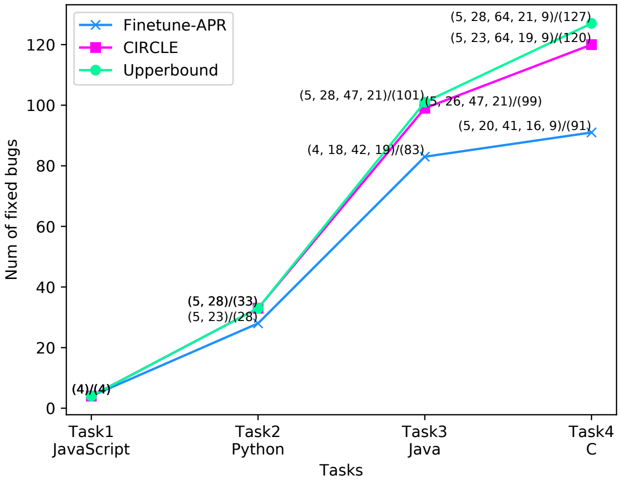
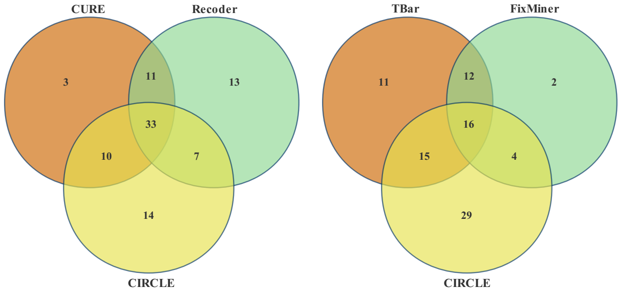


Figure 4: The performance trend comparison of traditional training approach and our CIRCLE in “task requirements increase constantly” scenario. The dot value (x)/(y) reports model’s behavior on all seen tasks. The number in “x” shows the performance on each benchmark following the order: BugAID, QuixBugs-Py, Defects4J, QuixBugs-Java, and ManyBugs. “y” is the sum of “x”.



(a) Traditional approaches (b) DL-based approaches

Figure 5: The overlaps of the bugs fixed by different approaches

lead to better repair performance. Despite these points, CIRCLE still outperforms most of state-of-the-art APR techniques. Thus, we highlight this direction of continual learning ability across multiple programming languages for automatic program repair.

To investigate what extent CIRCLE complements existing APR techniques, we further calculate the the overlaps of the bugs fixed by different techniques. We focus on the Defecst4J benchmark as it is widely evaluated by most previous APR work [34, 93] and thus has rich performance results for overlap analysis. Two best-performing traditional techniques (i.e., Tbar and FixMiner) and two best-performing DL-based techniques (i.e., Cure and Recoder) are selected. As shown in Figure 5, CIRCLE fixes 29 and 14 unique bugs when compared with traditional and DL-based approaches. Moreover, CIRCLE fixes 33, 44, 21 and 24 unique bugs compared with TBar, FixMinder, CURE and Recoder, respectively. This result shows that CIRCLE is complementary to these best-performing existing techniques.

5.3 RQ3: What are the contributions of the different components of CIRCLE?

We investigate the impact of four components of CIRCLE: (1) the prompt-based data representation; (2) the cross language re-repairing; (3) the lifelong module.

5.3.1 Impact of prompt-based data representation. According to recent research in NLP [40], adding prompt in input during finetuning can close the gap between the pre-trained task and down-stream task. In this work, we use T5 as skeleton model, which is mainly pre-trained with natural language tasks and therefore is much different from the APR task. To fill such gap, we concatenate context code and buggy code with some prompt words. Intuitively, these simple prompt words mark the input with natural language, helping model better exploit its pre-trained knowledge.

To investigate the actual impact of these prompt words, we train two CIRCLE model on the Java dataset. One is fed with prompt-based input and the other is fed with no-prompt input. Except the input form, all the parameters of these two models are the same. Figure 6 presents the loss curve of these two models. Within same number of epochs, the prompt-based inputs let model reach lower validation loss than no-prompt input. In other words, the prompt-based input representation promotes the convergence of the model.

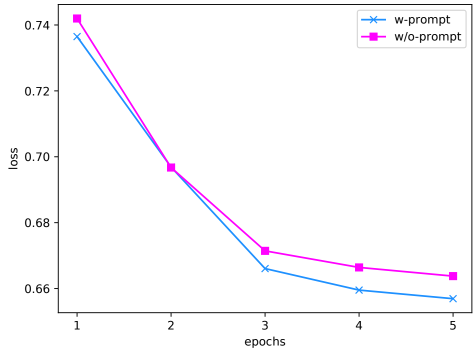


Figure 6: The validation loss of with/without prompt as input for the same model.

5.3.2 Impact of re-repairing. As mentioned in Section 3.6, we employ re-repairing mechanism to solve “keywords mismatch”, “format mismatch”, and “rare symbol” problems. In this subsection, we detailly analyze the influence of this re-repairing mechanism.

Figure 10 displays the statistics of fixed bugs on all benchmarks when using or not using re-repairing. As it shows, the simple re-repairing mechanism improves the number of fixed bugs on all benchmarks, demonstrating the efficacy of this module. Figure 7, 8, 9. contains three examples illustrating how re-repairing mechanism addresses the “keywords mismatch”, “format mismatch”, and “rare symbol” problems. For the first example, CIRCLE generates “NULL”

buggy line
```
-   while True:
```
patch generated by developer
```
+   while queue:
```
patch generated before re-repairing
```
+   while (NULL!= queue)
```
patch generated after re-repairing
```
+   while (None!= queue)
```

Figure 7: Keywords mismatch example for BREADTH_ FIRST_ SEARCH from QuixBugs-Py

buggy line
```
-   if (typeof opt.default!=’undefined’) self.default(key, opt.default);
```
patch generated by developer
```
+   if (typeof opt.default !== ’undefined’) self.default(key, opt.default);
```
patch generated before re-repairing
```
+   if (typeof opt.default!= = ’undefined’) self.default(key, opt.default);
```
patch generated after re-repairing
```
+   if (typeof opt.default!== ’undefined’) self.default(key, opt.default);
```

Figure 8: Format mismatch example for INCORRECT_ COMPARISON1_2011 from BugAID

buggy line
```
-   if (excerpt.equals(LINE) && 0 <= charno && charno < sourceExcerpt.length()) {
```
patch generated by developer
```
+   if (excerpt.equals(LINE) && 0 <= charno && charno <= sourceExcerpt.length()) {
```
patch generated before re-repairing
```
+   re-repairing: if (excerpt.equals(LINE) && 0 <unk>= charno &&
    charno <unk>= sourceExcerpt.length()) <unk>
```
patch generated after re-repairing
```
+   if (excerpt.equals(LINE) && 0<= charno && charno<= sourceExcerpt.length()){
```

Figure 9: Rare symbol example for Closure #63 from Defects4J

in python patch, because the meaning of “NULL” is much similar to “None” in both programming language and natural language. Our re-repairing module corrects this mistake, letting the generated patch be equivalent to the ground truth code. The second example shows the procession that our re-repairing removes incorrect blank. In the last sample, re-repairing fill rare symbols in the unknown token position to generate the fixed code. From these example, we can observe that T5 generally generates the correct patches, however, it could make some naive mistakes. The re-repairing mechanism is designed to fix these mistakes.

5.3.3 Impact of the continual learning module. The lifelong module in CIRCLE is mainly performed by the combination of the difficulty-based example replay and the sampling-based EWC regularization. As the EWC’s value is calculated on the selected sample set, we analyze them together. In deed, the “Finetune-APR” in RQ1 is the version of CIRCLE removed lifelong learning module. In RQ1, we show the impact of continual learning module through comparing the overall performance of CIRCLE and Finetune-APR. Here, we give a detailed example from our experiments to show the continual learning’s influence in a sample-level. As shown in Figure 11, when the java training set is first given, both CIRCLE and Finetune-APR can generate the correct patch, which is the same as the developer patch. However, Finetune-APR forgets to generate the condition “endIndex < 0” when the new C/C++ training set is in involved.

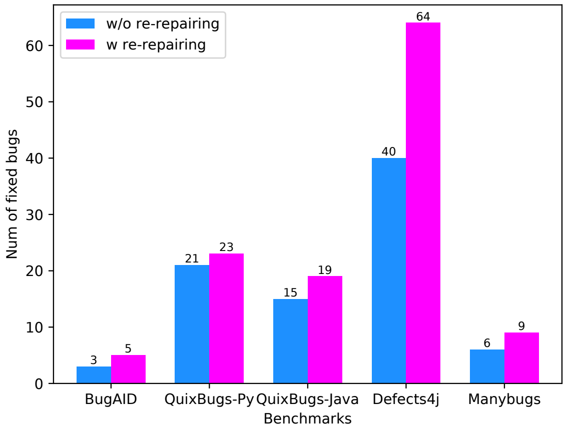


Figure 10: Number of fixed bugs on each benchmark with or without re-repairing mechanism.

buggy line
```
-   if (endIndex < 0) {
```
patch generated by developer
```
+   if ((endIndex < 0) || (endIndex < startIndex)) {
```
patch generated by CIRCLE
```
+   if (endIndex <0 || startIndex >endIndex){
```
patch generated by Finetune-APR
```
+   if (endIndex <startIndex){
```

Figure 11: Continual learning example for Chart #9 from Defects4J

5.4 Case Study

For multiple languages repairing, one common concern is that “can model distinguish different meanings of the same keywords among different languages?”. For example, “def” is a keyword in Python, but it can be a variable name in other languages. Will model be confused with such phenomenon? To investigate this question, we conduct case study on two samples selected from QuixBugs-Java and QuixBugs-Py. Specifically, we choose the Java and Python version of BITCOUNT program as an example. We replace the variable name “n” with “def” in Java version, and replace “n” with “public” in Python version. As shown in Figure 12, text with blue color represents the variable name replaced with other languages’ keyword. Our CIRCLE can correctly generate the patch without influenced by the ambiguious meaning of other language’s keywords.

context line (Java)

```
public class BITCOUNT { public static int bitcount(int def) { int
count = 0; while (def != 0) { def = (def ˆ (def - 1)); count++; } return
count; }}
```

buggy line

```
-   def = (def ˆ (def - 1));
```

patch generated by developer

```
+   def = (def & (def - 1));
```

patch generated by CIRCLE

```
+   def = (def &(def - 1));
```

context line (Python)

```
def bitcount(public): count = 0 while public: public ˆ= public - 1
count += 1 return count
```

buggy line

```
-   public ˆ= public - 1
```

patch generated by developer

```
+   public &= public - 1
```

patch generated by CIRCLE

```
+   public &= public - 1
```

Figure 12: Case study for keyword meaning's ambiguity.

6 RELATED WORK

6.1 APR

Over the past decade, researchers have proposed a variety of techniques to generate patches based on different hypotheses [17, 56]. Following recent work [6, 39, 92], we categorize them into four main categories: heuristic-based [32, 47, 90], constraint-based [14, 52, 86], template-based [30, 37, 38] and DL-based repair techniques [34, 45, 93].

Recently, DL-based repair techniques, which attempt to fix bugs enhanced by machine learning techniques, is getting growing attentionRecently, due to the large available open-source code. Tufano et al. [77] extensively evaluate the ability of adopting neural machine translation techniques to generate patches from bug-fixes commits in the wild. Li et al. [34] adopt a tree-based RNN encoder-decoder model (i.e., DLFix) to learn code contexts and transformations from previous bug fixes. Lutellier et al. [45] propose a new context-aware NMT architecture (i.e., CoCoNut) that represents the buggy source code and its surrounding context separately, to automatically fix bugs in multiple programming languages. Jiang et al. [25] propose a novel approach (i.e., CURE) combines a program language model, a code-aware search strategy, and a subword tokenization technique. The results demonstrate CURE can outperform all existing techniques on two popular benchmark (i.e., Defects4J and QuixBugs) when published. Recently, Zhu et al. [93] use a syntax-guided edit decoder (i.e., Recoder) with provider/decider architecture to ensure accurate patch generation. Compared to existing work, CIRCLE is the first work that aims to address the the generalizability issue of APR by repairing multiple languages in a lifelong learning scenario.

6.2 Continual Learning

The general concept and technical categories of Continual Learning has been introduced in Section 2.2. Recently, to better fit real life applications, where tasks and data always change, Continual Learning has been widely used in many Natural Language Processing (NLP) [7, 57, 89] and Computer Vision (CV) areas [60], such as: Named Entity Recognition [55], Neural Machine Translation [8], Dialogue Systems [36], Question Anwering [61], Text Classification [10, 23], Image Classification [13, 48, 78], and so on. However, none of existing works studied Automatic Program Repair (APR) with Continual Learning before. This paper is the first one to explore continually learning APR tasks.

The idea of continual learning starts in 1990s [75]. However, achieving continual learning is challenging because of catastrophic forgetting [16, 29]. Most of recent continual learning works focus on dealing with the forgetting problem. Generally, these methods can be classified into three categories: Rehearsal, Regularization, and Architectural methods. Rehearsal methods aim to replay some selected data in the forthcoming task. iCaRL [66] is one of the most well-known rehearsal method. It selects training data using Herding techniques [82]. Based on the replaying idea, some works build generators for previous task to create pseudo data for future learning [21, 22]. Regularization approaches replies on additional loss term to consolid learned knowledge. The classical regularization method is EWC [29]. It restricts the update of "important" parameters. Except EWC, other regularization methods such as GEM [43], MAS [3], IS [91] are also widely used to tackle catastrophic forgetting. Architectural approaches prevent forgetting by applying modular changes to neural network models [46, 62, 68, 83]. However, they will dynamically increase models' parameters when the number of tasks grows up. In this paper, we combine rehearsal method with regularization to achieve lifelong learning in APR.

7 CONCLUSION

In this paper, we propose CIRCLE, an automatic program repairing framework that can continually learn to fix bugs crossing various programming languages. Specifically, CIRCLE consists of five components: a prompt-based representation, a T5-based model, a difficulty-based example replay, an EWC-based regularization, and a re-repairing mechanism. The T5-based model is the skeleton of APR model. The prompt-based representation converts program repairing to fill-in-the-blank task, filling the gap between T5's pre-trained task and APR task. The difficulty-based replay and EWC-based regularization are two lifelong strategies, enabling CIRCLE to continually update its parameters according to the incremental task requirements. Finally, a simple yet effective re-repairing method is applied to eliminate the form error caused by multiple languages repairing. To the best of our knowledge, it is the first time to construct an APR model simultaneously addressing multiple programming languages based on continual learning approaches. We conduct extensive experiments with 4 programming languages on 5 benchmarks to demonstrate the effectiveness of our CIRCLE. Experimental results show that our CIRCLE (1) can continually learn bug fixing crossing languages; (2) achieves state-of-the-art performance on all benchmarks using a single model.

ACKNOWLEDGEMENT

This work is partially supported by the National Key Research and Development Program of China (2021YFB1715600), National Natural Science Foundation of China (No. 62141215), Australian Research Council Future Fellowship (No. FT210100624), and Australian Discovery Project (No. DP190101985).

REFERENCES

[1] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified Pre-training for Program Understanding and Generation. *arXiv preprint arXiv:2103.06333* (2021).

[2] Toufique Ahmed and Premkumar Devanbu. 2021. Multilingual training for Software Engineering. *arXiv preprint arXiv:2112.02043* (2021).

[3] Rahaf Aljundi, Francesca Babiloni, Mohamed Elhoseiny, Marcus Rohrbach, and Tinne Tuytelaars. 2018. Memory aware synapses: Learning what (not) to forget. In *Proceedings of the European Conference on Computer Vision (ECCV)*. 139–154.

[4] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. 2016. Layer normalization. *arXiv preprint arXiv:1607.06450* (2016).

[5] Johannes Bader, Andrew Scott, Michael Pradel, and Satish Chandra. 2019. Getafix: Learning to fix bugs automatically. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 1–27.

[6] Samuel Benton, Xia Li, Yiling Lou, and Lingming Zhang. 2021. Evaluating and Improving Unified Debugging. *IEEE Transactions on Software Engineering* (2021).

[7] Magdalena Biesialska, Katarzyna Biesialska, and Marta R Costa-jussà. 2020. Continual lifelong learning in natural language processing: A survey. *arXiv preprint arXiv:2012.09823* (2020).

[8] Yue Cao, Hao-Ran Wei, Boxing Chen, and Xiaojun Wan. 2021. Continual Learning for Neural Machine Translation. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. 3964–3974.

[9] Saikat Chakraborty, Yangruibo Ding, Miltiadis Allamanis, and Baishakhi Ray. 2020. Codit: Code editing with tree-based neural models. *IEEE Transactions on Software Engineering* (2020).

[10] Hongxu Chen, Hongzhi Yin, Tong Chen, Quoc Viet Hung Nguyen, Wen-Chih Peng, and Xue Li. 2019. Exploiting centrality information with graph convolutions for network representation learning. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, 590–601.

[11] Zimin Chen, Steve James Kommrusch, Michele Tufano, Louis-Noël Pouchet, Denys Poshyvanyk, and Martin Monperrus. 2019. Sequencer: Sequence-to-sequence learning for end-to-end program repair. *IEEE Transactions on Software Engineering* (2019).

[12] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. 2014. Learning phrase representations using RNN encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078* (2014).

[13] Matthias Delange, Rahaf Aljundi, Marc Masana, Sarah Parisot, Xu Jia, Ales Leonardis, Greg Slabaugh, and Tinne Tuytelaars. 2021. A continual learning survey: Defying forgetting in classification tasks. *IEEE Transactions on Pattern Analysis and Machine Intelligence* (2021).

[14] Thomas Durieux and Martin Monperrus. 2016. Dynamoth: dynamic code synthesis for automatic program repair. In *Proceedings of the 11th International Workshop on Automation of Software Test*. 85–91.

[15] Ahmed Elnaggar, Wei Ding, Llion Jones, Tom Gibbs, Tamas Feher, Christoph Angerer, Silvia Severini, Florian Matthes, and Burkhard Rost. 2021. CodeTrans: Towards Cracking the Language of Silicon's Code Through Self-Supervised Deep Learning and High Performance Computing. *arXiv preprint arXiv:2104.02443* (2021).

[16] Robert M French. 1999. Catastrophic forgetting in connectionist networks. *Trends in cognitive sciences* 3, 4 (1999), 128–135.

[17] Luca Gazzola, Daniela Micucci, and Leonardo Mariani. 2017. Automatic software repair: A survey. *IEEE Transactions on Software Engineering* 45, 1 (2017), 34–67.

[18] Ali Ghanbari, Samuel Benton, and Lingming Zhang. 2019. Practical program repair via bytecode mutation. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 19–30.

[19] Quinn Hanam, Fernando S de M Brito, and Ali Mesbah. 2016. Discovering bug patterns in JavaScript. In *Proceedings of the 2016 24th ACM SIGSOFT international symposium on foundations of software engineering*. 144–156.

[20] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 770–778.

[21] Wenpeng Hu, Zhou Lin, Bing Liu, Chongyang Tao, Zhengwei Tao, Jinwen Ma, Dongyan Zhao, and Rui Yan. 2018. Overcoming catastrophic forgetting for continual learning via model adaptation. In *International Conference on Learning Representations*.

[22] Xinting Hu, Kaihua Tang, Chunyan Miao, Xian-Sheng Hua, and Hanwang Zhang. 2021. Distilling Causal Effect of Data in Class-Incremental Learning. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 3957–3966.

[23] Yufan Huang, Yanzhe Zhang, Jiaao Chen, Xuezhi Wang, and Diyi Yang. 2021. Continual Learning for Text Classification with Information Disentanglement Based Regularization. *arXiv preprint arXiv:2104.05489* (2021).

[24] Jiajun Jiang, Yingfei Xiong, Hongyu Zhang, Qing Gao, and Xiangqun Chen. 2018. Shaping program repair space with existing patches and similar code. In *Proceedings of the 27th ACM SIGSOFT international symposium on software testing and analysis*. 298–309.

[25] Nan Jiang, Thibaud Lutellier, and Lin Tan. 2021. CURE: Code-Aware Neural Machine Translation for Automatic Program Repair. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 1161–1173.

[26] René Just, Darioush Jalali, and Michael D Ernst. 2014. 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*. 437–440.

[27] Nitish Shirish Keskar, Bryan McCann, Lav R Varshney, Caiming Xiong, and Richard Socher. 2019. Ctrl: A conditional transformer language model for controllable generation. *arXiv preprint arXiv:1909.05858* (2019).

[28] Daniel Khashabi, Sewon Min, Tushar Khot, Ashish Sabharwal, Oyvind Tafjord, Peter Clark, and Hannaneh Hajishirzi. 2020. Unifiedqa: Crossing format boundaries with a single qa system. *arXiv preprint arXiv:2005.00700* (2020).

[29] James Kirkpatrick, Razvan Pascanu, Neil Rabinowitz, Joel Veness, Guillaume Desjardins, Andrei A Rusu, Kieran Milan, John Quan, Tiago Ramalho, Agnieszka Grabska-Barwinska, et al. 2017. Overcoming catastrophic forgetting in neural networks. *Proceedings of the national academy of sciences* 114, 13 (2017), 3521–3526.

[30] Anil Koyuncu, Kui Liu, Tegawendé F Bissyandé, Dongsun Kim, Jacques Klein, Martin Monperrus, and Yves Le Traon. 2020. : Mining relevant fix patterns for automated program repair. *Empirical Software Engineering* 25, 3 (2020), 1980–2024.

[31] Claire Le Goues, Neal Holtschulte, Edward K Smith, Yuriy Brun, Premkumar Devanbu, Stephanie Forrest, and Westley Weimer. 2015. The ManyBugs and IntroClass benchmarks for automated repair of C programs. *IEEE Transactions on Software Engineering* 41, 12 (2015), 1236–1256.

[32] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2011. Genprog: A generic method for automatic software repair. *Ieee transactions on software engineering* 38, 1 (2011), 54–72.

[33] Brian Lester, Rami Al-Rfou, and Noah Constant. 2021. The power of scale for parameter-efficient prompt tuning. *arXiv preprint arXiv:2104.08691* (2021).

[34] Yi Li, Shaohua Wang, and Tien N Nguyen. 2020. Dlfix: Context-based code transformation learning for automated program repair. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 602–614.

[35] Derrick Lin, James Koppel, Angela Chen, and Armando Solar-Lezama. 2017. QuixBugs: A multi-lingual program repair benchmark set based on the Quixey Challenge. In *Proceedings Companion of the 2017 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity*. 55–56.

[36] Bing Liu and Sahisnu Mazumder. 2021. Lifelong and continual learning dialogue systems: learning during conversation. *Proceedings of AAAI-2021* (2021).

[37] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F Bissyandé. 2019. Avatar: Fixing semantic bugs with fix patterns of static analysis violations. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 1–12.

[38] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F Bissyandé. 2019. Tbar: revisiting template-based automated program repair. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 31–42.

[39] Kui Liu, Shangwen Wang, Anil Koyuncu, Kisub Kim, Tegawendé F Bissyandé, Dongsun Kim, Peng Wu, Jacques Klein, Xiaoguang Mao, and Yves Le Traon. 2020. On the efficiency of test suite based program repair. In *Proceedings of ICSE*.

[40] Pengfei Liu, Weizhe Yuan, Jinlan Fu, Zhengbao Jiang, Hiroaki Hayashi, and Graham Neubig. 2021. Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing. *arXiv preprint arXiv:2107.13586* (2021).

[41] Xiao Liu, Yanan Zheng, Zhengxiao Du, Ming Ding, Yujie Qian, Zhilin Yang, and Jie Tang. 2021. GPT Understands, Too. *arXiv preprint arXiv:2103.10385* (2021).

[42] Fan Long and Martin Rinard. 2016. Automatic patch generation by learning correct code. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 298–312.

[43] David Lopez-Paz and Marc'Aurelio Ranzato. 2017. Gradient episodic memory for continual learning. *Advances in neural information processing systems* 30 (2017), 6467–6476.

[44] Ilya Loshchilov and Frank Hutter. 2017. Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101* (2017).

[45] Thibaud Lutellier, Hung Viet Pham, Lawrence Pang, Yitong Li, Moshi Wei, and Lin Tan. 2020. Coconut: combining context-aware neural translation models using ensemble for program repair. In *Proceedings of the 29th ACM SIGSOFT international symposium on software testing and analysis*. 101–114.

[46] Massimiliano Mancini, Elisa Ricci, Barbara Caputo, and Samuel Rota Bulo. 2018. Adding new tasks to a single network with weight transformations using binary masks. In *Proceedings of the European Conference on Computer Vision (ECCV) Workshops*. 0–0.

[47] Matias Martinez and Martin Monperrus. 2016. Astor: A program repair library for java. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*. 441–444.

[48] Marc Masana, Xialei Liu, Bartlomiej Twardowski, Mikel Menta, Andrew D Bagdanov, and Joost van de Weijer. 2020. Class-incremental learning: survey and performance evaluation on image classification. *arXiv preprint arXiv:2010.15277*

(2020).

[49] Ehsan Mashhadi and Hadi Hemmati. 2021. Applying CodeBERT for Automated Program Repair of Java Simple Bugs. *arXiv preprint arXiv:2103.11626* (2021).

[50] Antonio Mastropaolo, Simone Scalabrino, Nathan Cooper, David Nader Palacio, Denys Poshyvanyk, Rocco Oliveto, and Gabriele Bavota. 2021. Studying the usage of text-to-text transfer transformer to support code-related tasks. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 336–347.

[51] Michael McCloskey and Neal J Cohen. 1989. Catastrophic interference in connectionist networks: The sequential learning problem. In *Psychology of learning and motivation*. Vol. 24. Elsevier, 109–165.

[52] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2016. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *Proceedings of the 38th international conference on software engineering*. 691–701.

[53] Zhu Ming, Suresh Karthik, and K. Reddy Chandan. 2022. Multilingual Code Snippets Training for Program Translation. In *Proceedings of the Thirty-Sixth AAAI Conference on Artificial Intelligence*.

[54] Seyed Iman Mirzadeh, Mehrdad Farajtabar, Dilan Gorur, Razvan Pascanu, and Hassan Ghasemzadeh. 2020. Linear mode connectivity in multitask and continual learning. *arXiv preprint arXiv:2010.04495* (2020).

[55] Natawut Monaikul, Giuseppe Castellucci, Simone Filice, and Oleg Rokhlenko. 2021. Continual Learning for Named Entity Recognition. In *Proceedings of the Thirty-Fifth AAAI Conference on Artificial Intelligence*.

[56] Martin Monperrus. 2020. The living review on automated program repair. (2020).

[57] Thanh Tam Nguyen, Chi Thang Duong, Matthias Weidlich, Hongzhi Yin, and Quoc Viet Hung Nguyen. 2017. Retaining data from streams of social platforms with minimal regret. In *Twenty-sixth International Joint Conference on Artificial Intelligence*.

[58] Changan Niu, Chuanyi Li, Vincent Ng, Jidong Ge, Liguo Huang, and Bin Luo. 2022. SPT-Code: Sequence-to-Sequence Pre-Training for Learning the Representation of Source Code. *arXiv preprint arXiv:2201.01549* (2022).

[59] Daniel W Otter, Julian R Medina, and Jugal K Kalita. 2020. A survey of the usages of deep learning for natural language processing. *IEEE Transactions on Neural Networks and Learning Systems* 32, 2 (2020), 604–624.

[60] German I Parisi, Ronald Kemker, Jose L Part, Christopher Kanan, and Stefan Wermter. 2019. Continual lifelong learning with neural networks: A review. *Neural Networks* 113 (2019), 54–71.

[61] Anselmo Peñas, Mathilde Veron, Camille Pradel, Arantxa Otegi, Guillermo Echegoyen, and Alvaro Rodrigo. 2021. Continuous learning for question answering. In *Increasing Naturalness and Flexibility in Spoken Dialogue Interaction: 10th International Workshop on Spoken Dialogue Systems*. Springer Singapore, 337–341.

[62] Jonas Pfeiffer, Andreas Rücklé, Clifton Poth, Aishwarya Kamath, Ivan Vulić, Sebastian Ruder, Kyunghyun Cho, and Iryna Gurevych. 2020. Adapterhub: A framework for adapting transformers. *arXiv preprint arXiv:2007.07779* (2020).

[63] Xipeng Qiu, Tianxiang Sun, Yige Xu, Yunfan Shao, Ning Dai, and Xuanjing Huang. 2020. Pre-trained models for natural language processing: A survey. *Science China Technological Sciences* (2020), 1–26.

[64] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. 2019. Exploring the limits of transfer learning with a unified text-to-text transformer. *arXiv preprint arXiv:1910.10683* (2019).

[65] Tiago Ramalho and Marta Garnelo. 2019. Adaptive posterior learning: few-shot learning with a surprise-based memory module. *arXiv preprint arXiv:1902.02527* (2019).

[66] Sylvestre-Alvise Rebuffi, Alexander Kolesnikov, Georg Sperl, and Christoph H Lampert. 2017. icarl: Incremental classifier and representation learning. In *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*. 2001–2010.

[67] Anthony Robins. 1995. Catastrophic forgetting, rehearsal and pseudorehearsal. *Connection Science* 7, 2 (1995), 123–146.

[68] Andrei A Rusu, Neil C Rabinowitz, Guillaume Desjardins, Hubert Soyer, James Kirkpatrick, Koray Kavukcuoglu, Razvan Pascanu, and Raia Hadsell. 2016. Progressive neural networks. *arXiv preprint arXiv:1606.04671* (2016).

[69] Timo Schick and Hinrich Schütze. 2020. Exploiting cloze questions for few shot text classification and natural language inference. *arXiv preprint arXiv:2001.07676* (2020).

[70] Rico Sennrich, Barry Haddow, and Alexandra Birch. 2015. Neural machine translation of rare words with subword units. *arXiv preprint arXiv:1508.07909* (2015).

[71] Taylor Shin, Yasaman Razeghi, Robert L Logan IV, Eric Wallace, and Sameer Singh. 2020. Autoprompt: Eliciting knowledge from language models with automatically generated prompts. *arXiv preprint arXiv:2010.15980* (2020).

[72] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2014. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research* 15, 1 (2014), 1929–1958.

[73] Peng Su, Shixiang Tang, Peng Gao, Di Qiu, Ni Zhao, and Xiaogang Wang. 2020. Gradient Regularized Contrastive Learning for Continual Domain Adaptation. *arXiv preprint arXiv:2007.12942* (2020).

[74] Brian Thompson, Jeremy Gwinnup, Huda Khayrallah, Kevin Duh, and Philipp Koehn. 2019. Overcoming catastrophic forgetting during domain adaptation of neural machine translation. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. 2062–2068.

[75] Sebastian Thrun. 1998. Lifelong learning algorithms. In *Learning to learn*. Springer, 181–209.

[76] Michele Tufano, Jevgenija Pantiuchina, Cody Watson, Gabriele Bavota, and Denys Poshyvanyk. 2019. On learning meaningful code changes via neural machine translation. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 25–36.

[77] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. 2019. An empirical study on learning bug-fixing patches in the wild via neural machine translation. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 28, 4 (2019), 1–29.

[78] Gido M Van de Ven and Andreas S Tolias. 2019. Three scenarios for continual learning. *arXiv preprint arXiv:1904.07734* (2019).

[79] Rijnard van Tonder and Claire Le Goues. 2019. Towards s/engineer/bot: Principles for program repair bots. In *2019 IEEE/ACM 1st International Workshop on Bots in Software Engineering (BotSE)*. IEEE, 43–47.

[80] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in neural information processing systems*. 5998–6008.

[81] Cathrin Weiss, Rahul Premraj, Thomas Zimmermann, and Andreas Zeller. 2007. How long will it take to fix this bug?. In *Fourth International Workshop on Mining Software Repositories (MSR'07: ICSE Workshops 2007)*. IEEE, 1–1.

[82] Max Welling. 2009. Herding dynamical weights to learn. In *Proceedings of the 26th Annual International Conference on Machine Learning*. 1121–1128.

[83] Yeming Wen, Dustin Tran, and Jimmy Ba. 2020. Batchensemble: an alternative approach to efficient ensemble and lifelong learning. *arXiv preprint arXiv:2002.06715* (2020).

[84] Thomas Wolf, Julien Chaumond, Lysandre Debut, Victor Sanh, Clement Delangue, Anthony Moi, Pierric Cistac, Morgan Funtowicz, Joe Davison, Sam Shleifer, et al. 2020. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*. 38–45.

[85] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, et al. 2019. Huggingface's transformers: State-of-the-art natural language processing. *arXiv preprint arXiv:1910.03771* (2019).

[86] Jifeng Xuan, Matias Martinez, Favio Demarco, Maxime Clement, Sebastian Lamelas Marcote, Thomas Durieux, Daniel Le Berre, and Martin Monperrus. 2016. Nopol: Automatic repair of conditional statement bugs in java programs. *IEEE Transactions on Software Engineering* 43, 1 (2016), 34–55.

[87] He Ye, Matias Martinez, Thomas Durieux, and Martin Monperrus. 2021. A comprehensive study of automatic program repair on the QuixBugs benchmark. *Journal of Systems and Software* 171 (2021), 110825.

[88] He Ye, Matias Martinez, and Martin Monperrus. 2021. Neural Program Repair with Execution-based Backpropagation. *arXiv preprint arXiv:2105.04123* (2021).

[89] Wei Yuan, Hongzhi Yin, Tieke He, Tong Chen, Qiufeng Wang, and Lizhen Cui. 2022. Unified Question Generation with Continual Lifelong Learning. *arXiv preprint arXiv:2201.09696* (2022).

[90] Yuan Yuan and Wolfgang Banzhaf. 2018. Arja: Automated repair of java programs via multi-objective genetic programming. *IEEE Transactions on Software Engineering* 46, 10 (2018), 1040–1067.

[91] Friedemann Zenke, Ben Poole, and Surya Ganguli. 2017. Continual learning through synaptic intelligence. In *International Conference on Machine Learning*. PMLR, 3987–3995.

[92] Quanjun Zhang, Yuan Zhao, Weisong Sun, Chunrong Fang, Ziyuan Wang, and Lingming Zhang. 2022. Program Repair: Automated vs. Manual. *arXiv preprint arXiv:2203.05166* (2022).

[93] Qihao Zhu, Zeyu Sun, Yuan-an Xiao, Wenjie Zhang, Kang Yuan, Yingfei Xiong, and Lu Zhang. 2021. A Syntax-Guided Edit Decoder for Neural Program Repair. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Athens, Greece) *(ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 341–353.