2022 IEEE/ACM 第 44 屆軟件工程國際會議 (ICSE)

親愛的：一種新的基於深度學習的自動程序修復方法

易立新
澤西研究所美國新澤西州技術學院
yl622@njit.edu

Shaohua Wang*新
澤西研究所 。美國新澤西州技術學院
davidsw@njit.edu

Tien N. Nguyen德克
薩斯大學達拉斯分校 美國德克薩斯州
tien.n.nguyen@utdallas.edu

抽象的

現有的基於深度學習 (DL) 的自動程序修復(APR) 模型在修復一般軟件缺陷方面存在局限性 。我們介紹了 DEAR ，這是一種基於 DL 的方法 ，它支持修復一般錯誤 ，這些錯誤需要一次對一個或多個代碼塊中的一個或多個連續語句進行相關更改 。我們首先設計了一種新的故障定位 (FL) 技術 ，用於結合傳統的基於頻譜 (SB) 的多塊 、多語句修復

具有深度學習和數據流分析的 FL 。它獲取 SBFL 模型返回的有缺陷的語句 ，檢測有缺陷的 hunks並立即修復 ，並在 hunk 中擴展有缺陷的語句以包括周圍的其他可疑語句 。我們設計了一個兩層的 、基於樹的 LSTM 模型 ，該模型結合了循環訓練並使用分而治之的策略來學習適當的代碼轉換 ，以便在由周圍子樹組成的合適的固定上下文中固定多個語句 。我們進行了幾個實驗來評估三個數據集上的 DEAR ：Defects4J（395 個錯誤） 、BigFix（+26k 個錯誤)和 CPatMiner（+44k 個錯誤） 。在 Defects4J 數據集上 ，DEAR在僅使用 top-1 補丁的自動修復錯誤數量方面優於基線 42%–683% 。在 BigFix 數據集上 ，它修復的錯誤比現有的具有top-1 補丁的基於 DL 的 APR 模型多 31-145 個錯誤 。在 CPatMiner 數據集上 ，在 667 個已修復的錯誤中 ，有 169 個 (25.3%) multi-hunk/multi-statement 錯誤。 DEAR 比最先進的基於 DL 的 APR 模型多修復了 71 和 164 個錯誤 ，包括 52 和 61 個多塊/多語句錯誤。

CCS 概念

·軟件及其工程→軟件維護工具。

關鍵詞

自動程序修復 ；深度學習 ；故障定位 ；

ACM 參考格式： Yi Li 、

Shaohua Wang 和 Tien N. Nguyen。 2022. DEAR ：一種基於深度學習的新型自動程序修復方法 。第 44 屆國際軟件工程會議 (ICSE 22) ，2022 年 5 月 21-29 日 ，美國賓夕法尼亞州匹茲堡。 ACM ，美國紐約州紐約市 ，13 頁。 https://doi.org/10。1145/3510003.3510177

通訊作者

ICSE 22 ，2022 年 5 月 21-29 日 ，美國賓夕法尼亞州匹茲堡 © 2022 計算機協會。
ACM ISBN 978–1–4503–9221–1/22/05。 …… …… 15.00
美元 https://doi.org/10.1145/3510003.3510177

1 簡介

研究人員提出了幾種方法來幫助開發人員自動識別和修復軟件中的缺陷 。這種方法稱為自動程序修復 (APR)。 APR方法一直在利用基於搜索的軟件工程 、軟件挖掘 、機器學習 (ML) 和深度學習 (DL) 領域的各種技術。

對於基於搜索的方法[9 、10 、24 、30] ，搜索策略是在通過運算符對錯誤代碼進行變異而產生的潛在解決方案空間中執行的 。其他方法使用軟件挖掘來從先前的錯誤修復[15 、17 、19 、20 、27]或類似代碼[28 、32] 中挖掘和學習修復模式 。修復模式位於源代碼級別[19 、20]或更改級別[13 、16 、40] 。機器學習已被用於挖掘修復模式 ，候選修復根據它們的可能性進行排名[21 、22 、33] 。雖然一些基於 DL 的 APR方法學習了類似的修復[11 、41 、42] ，但其他方法使用機器翻譯或具有各種代碼抽象的神經網絡模型來生成補丁 [5 、6 、12 、18 、35 、38 、39] .

儘管他們取得了成功 ，但最先進的基於 DL 的 APR方法在修復一般缺陷方面仍然受到限制 ，這些缺陷涉及修復對文件相同或不同部分或不同文件中的多個語句的更改（它們是被稱為帥哥） 。現有的基於 DL 的方法都無法通過一次對多個塊中的多個語句進行依賴更改來自動修復錯誤 。他們只支持修復單個語句 。如果我們在當前語句上使用這樣的工具 ，該工具會將該語句視為不正確 ，而將其他語句視為正確 。這不成立 ，因為要修復當前語句 ，不能將剩餘的未修復語句視為正確代碼 。因此 ，當使用現有的基於 DL 的 APR 工具修復多塊/多語句錯誤的單個語句時 ，它可能是不准確的 。雖然 DL為修復學習提供了好處 ，但這種限制使得基於 DL 的APR 方法的能力不如支持多語句修復的其他方向（基於搜索和基於模式的 APR） 。

在本文中 ，我們的目標是通過引入 DEAR 來推進基於深度學習的 APR ，DEAR 是一種基於 DL 的模型 ，支持修復一般錯誤 ，同時對屬於一個或多個錯誤代碼塊的一個或多個錯誤語句進行相關更改.為此 ，我們做出了以下關鍵技術貢獻。

首先 ，我們開發了一種針對多塊 、多語句錯誤的故障定位 (FL) 技術 ，該技術將傳統的基於頻譜的 FL (SBFL) 與 DL 和數據流分析相結合。 DEAR 使用 SBFL 方法來識別可疑錯誤語句的排名列表 。然後 ，它使用該錯誤語句列表來推導需要通過微調預訓練的 BERT 模型[8]來修復在一起的錯誤塊 ，以了解語句之間的修復關係 。我們還設計了一個擴展算法 ，它接受一個錯誤的語句

in a hunk 作為種子 ,並擴展以包含周圍其他可疑的連續語句 。為此 ,我們使用 RNN模型將語句分類為錯誤或非錯誤 ,並使用數據流分析進行調整 ,然後形成錯誤塊。

其次 ,在擴展步驟之後 ,我們已經識別出所有帶有 buggy 語句的 buggy hunk(s) 。我們開發了一種組合方法來學習 ,然後生成多塊 、多語句修復 。在我們的方法中 ,從錯誤的語句中 ,我們使用分而治之的策略來學習抽象語法樹 (AST) 中的每個子樹轉換 。具體來說 ,我們使用基於 AST 的差分技術來推導訓練數據中基於 AST 的細粒度更改以及錯誤代碼和固定代碼之間的映射 。這些細粒度的子樹映射幫助我們的模型避免錯誤和固定代碼的不正確對齊 ,因此 ,在學習修復的多個 AST 子樹轉換時更加準確。

第三 ,我們增強並編排了一個基於樹的兩層長短期記憶 (LSTM) 模型[18] ,該模型具有一個注意層和一個循環訓練 ,以幫助 DEAR 在合適的上下文中學習正確的代碼修復更改周邊代碼。

對於我們的故障定位識別出的每個有缺陷的 AST 子樹 ,我們將其編碼為向量表示 ,並應用該 LSTM 模型來導出固定代碼 。在第一層 ,它學習修復上下文 ,即圍繞有問題的 AST 子樹的代碼結構 。在第二層 ,它學習代碼轉換以使用上下文作為附加權重來修復有問題的子樹。

最後 ,可能存在多個有問題的子樹 。為了在訓練中為每個有問題的子樹構建周圍的上下文 ,我們在其他有問題的子樹 (而不是那些有問題的子樹本身)的修復之後包含 AST 子樹 。基本原理是修復後的子樹實際上代表了 的正確周圍代碼。 (注意 :在訓練中 ,固定子樹是已知的) 。

我們在三個大型數據集上進行了實驗來評估 DEAR :Defects4J [1] (395 個錯誤) 、BigFix [18] (+26k 個錯誤)和 CPat Miner 數據集[26] (+44k 個錯誤) 。基於 DL 的基線方法包括 DLFix [18]、 CoCoNuT [23]、 SequenceR [6] 、Tufano19 [38]、 CODIT [5] 和 CURE [14]。 DEAR僅使用 Top-1 在所有三個數據集上修復的錯誤分別比性能最佳的基線 CURE 多 31% (即 +11) 、5.6% (即 +41)和9.3% (即 +31)補丁 ,平均訓練參數減少七倍 。在Defects4J 上 ,它在修復錯誤的數量方面優於那些基線 42%–683% 。在 BigFix 上 ,它修復的錯誤比那些帶有 top-1 補丁的基線多 31-145 個錯誤 。在 CPatMiner 上 ,在DEAR 修復的 667 個錯誤中 ,有 169 個 (25.3%) multi-hunk/multi- statement 錯誤 。與現有的基於 DL 的 APR 工具 CoCoNuT 、DLFix 和 CURE相比 ,DEAR 修復了 71 、164 和 41 個錯誤 ,包括 52 、61 和 40 個多塊/多語句錯誤 。我們還將DEAR 與 8 種最先進的基於模式的 APR 工具進行了比較 。我們的結果表明 ,DEAR 生成的結果與基於模式的頂級 APR 工具具有可比性和互補性 。在 Defects4J 上 ,DEAR 修復了 12 個錯誤 (共 47 個) ,其中包括頂級基於模式的 APR 工具無法修復的7 個多塊/多語句錯誤。

簡而言之 ,本文的主要貢獻包括A. Advancing based APR
for general bugs with multi hunk/multi-statement fixes : DEAR advances based APR for general bug 。我們表明 ,基於深度學習的 APR 可以實現與其他 APR 方向相當和互補的結果。

B. 先進的基於 DL 的 APR 技術 :

```
1 public boolean verifyUserInfo(String UID, String password, String SSN) { 2 String retrieved_password =
    ; 3 字符串 retrieved_SSN =     ; 4 if (UID !=
null) { 5 - retrieved_password =
getPassword(UID); 6 +
retrieved_password = getPassword(toUpperCase(UID)); 7 + } 其
他 {

 8 +          返回假 ;
 9 + } 10
-布爾password_check= compare(password,retrieved_password); 11 + boolean password_check=
compare(passwordHash(password),retrieved_password);
12          if (password_check)
13            { retrieved_SSN = getSSN(UID);布爾
14            SSN_check = 比較 (SSN ,retrieved_SSN) ;如果 (SSN_check){返回
15            真 ;
16
17            }
18          }
19          返回假 ;
20 }
```

圖 1 :具有多個相關更改的一般修復

1) 一種用於多塊 、多語句修復的新型 FL 技術 ,將基於頻譜的 FL 與 DL 和數據流分析相結合 ;
2) 採用分而治之策略的組合方法
學習並生成多塊 、多語句修復 ;和
3) 雙層 LSTM 模型的設計和編排 ,通過注意力層和循環訓練進行增強。

C. 廣泛的實證評估 : 1)DEAR 優於現有的基於 DL 的 APR 工具 ; 2) DEAR 是第一個基於 DL 的 APR模型 ,在修復錯誤的數量方面與最先進的基於模式的工具處於同一水平 ,並生成互補的結果 ; 3) 我們的數據和工具是公開的 [2]。

2 動機

2.1 激勵示例讓我們展示一個錯誤修復

示例和我們對激勵的觀察 。圖 1 顯示了verifyUserInfo中的錯誤示例 ,它根據數據庫中的用戶記錄驗證給定的用戶 ID 、密碼和社會安全號碼 。這個錯誤體現在三個方面 。首先 ,開發者忘記處理UID為null的情況。

因此 ,為了修復 ,他在第 7-9 行添加了一個else分支 。二是開發者忘記對UID進行大寫轉換 ,導致錯誤 ,因為數據庫中用戶ID的記錄都是大寫字母 。相應的錯誤修復更改是在第 6 行添加了對 UID 上的 toUpperCase() 的調用 。 第三 ,由於存儲在數據庫中的密碼是通過散列編碼的 ,因此用戶輸入的密碼在比較之前需要進行散列反對數據庫中的那個 。因此 ,開發人員在第 11 行調用方法 compare()之前添加了對passwordHash()的調用 。從這個例子中 ,我們有以下觀察結果 :

觀察 1 [A Fix with Dependent Changes to Multiple Statements] :此錯誤需要在同一個修復中同時對多個語句進行依賴修復更改 :1) 添加帶有 return語句的else分支 (第 7-9 行) ,2) 添加toUpperCase在第 6 行 ,以及 3) 在第 11 行添加passwordHash 。一次更改單個語句不會修復錯誤 ,因為給定的參數UID和密碼都需要正確處理。

UID需要空檢查和大寫 ,密碼需要散列 。對多個語句的那些相關更改必須在同一個修復程序中同時發生 ,才能使程序通過測試用例。

最先進的基於 DL 的 APR 方法[6、18]一次修復一個單獨的語句。在圖1中，故障定位工具返回了兩條錯誤行：第5行和第10行。假設使用這樣一個基於DL的APR工具來修復第5行的語句。它將對第5行的語句進行修復更改（例如，修改第 5 行並添加第 7-9 行），但是，假設第 10 行和其他行的陳述是正確的。有了這個不正確的假設，這樣的修復不會使代碼通過測試用例，因為必須進行兩個更改。因此，單個語句、基於 DL 的 APR 工具無法通過一次修復一個錯誤語句來修復此錯誤。通常，錯誤可能需要對同一個修復中的多個語句（可能是多個塊）進行相關更改。

此外，基於模式的 APR 工具可能無法修復此缺陷，因為此示例中的代碼是特定於項目的，可能與任何錯誤修復模式都不匹配。

觀察 2 [多對多 AST 子樹轉換]：修復可能涉及對多個子樹的更改。例如， if語句有一個新的else分支。調用getPassword()的參數被修改為調用toUpperCase()。

此修復還涉及多對多子樹轉換。在此示例中，修復將兩個有問題的語句（第 5 行和第 10 行）轉換為四個語句（具有新else分支的if語句、第 8 行的return語句、第 6 行帶有toUpperCase的修改語句以及修改後的在第 11 行帶有passwordHash 的語句）。因此，一個修復可以分解為多個子樹轉換，如果使用具有分而治之策略的組合方法，我們可以學習各個轉換。

觀察 3 [正確修復上下文]：錯誤修復通常取決於周圍代碼的上下文。例如，要從給定的UID 獲取密碼，需要將 ID 大寫，因此在正確的代碼中，在調用getPassword方法時，很可能會調用toUpperCase方法。因此，建立正確的固定上下文很重要。在圖 1 中，模型需要學習修復（第 5 行→第 6 行)wrt 周圍的代碼，這需要包括第 11 行的修復代碼（而不是第 10 行，因為第 10 行有錯誤）。要修復第 5 行，正確的上下文必須包括第 11 行的passwordHash。因此，修復錯誤語句的正確上下文必須包括另一個錯誤語句的固定代碼，而不是本身。

2.2 關鍵思想從觀察

中，我們得出以下關鍵思想：關鍵思想1. A Fault Localization Method for Multi-hunk, Multi- statement Patches：根據觀察 1，我們設計了一種新的 FL方法，它結合了傳統的基於頻譜的 FL（SBFL）) 與 DL 和數據流分析。我們使用 SBFL 來獲得 can didate 語句的排名列表，以用它們的可疑分數來修復。我們在兩個任務中擴展了 SBFL 的結果。首先，我們設計了一個 hunk檢測算法，使用 DL 檢測需要在同一個補丁中一起依賴更改的hunk，因為 SBFL 工具返回故障的可疑候選，但不一定要一起修復。其次，我們設計了一種擴展算法，該算法採用每個檢測到的固定在一起的大塊，並將其擴展為在大塊中包含連續的可疑語句。

在圖 1 中，SBFL 工具將第 5 行返回為可疑。在 hunk detection 之後，DEAR 通過變量retrieve_password使用數據依賴性來包含第 10 行的語句以進行修復。

關鍵思想2. 學習和生成多塊、多語句修復的組合方法：學習多塊/多語句修復中的分而治之策略。要使用多個語句自動修復錯誤，工具需要對語句進行更改，即語句通常可能在修復後變成語句。一種天真的方法會讓模型學習代碼結構的變化，並在修復前後的代碼之間進行對齊。由於修復涉及多個子樹轉換（觀察 2），在訓練期間，模型可能會錯誤地對齊修復前後的代碼，從而導致錯誤地學習修復。例如，如果沒有這一步，模型可能會將第 10 行的retrieved_password映射到第 6 行的相同變量（正確的映射是第 11 行）。因此，為了便於學習錯誤修復代碼轉換，在訓練期間，我們使用分而治之的策略。我們將一個基於 AST 的細粒度更改檢測模型集成到 DEAR 中，以映射修復前後的 AST。這樣的映射使 DEAR 能夠了解對子樹的更多本地修復更改。例如，細粒度的AST變化檢測可以推導出第4-5、7行的語句變成第4、6-9行的語句；第 10 行的語句變成第 11 行的語句。我們可以將它們分成兩組，並對齊各自的 AST 子樹供 DEAR 學習。

固定多個子樹的組合方法。我們通過增強基於樹的 LSTM模型[18]的設計和編排以添加註意力層和循環訓練（第 3.3 節)來支持在一個或多個 hunk 中具有多個語句的修復。

雖然該模型一次修復一個子樹，但我們需要對其進行增強以一次修復多個 AST 子樹。

具體來說，我們修改了它在兩層中的操作，以同時考慮多個有問題的子樹。例如，在訓練期間，我們標記第 5 行語句的每個 AST 子樹和

修復前的第 10 行是錯誤的。在第一層，對於錯誤語句的每個子樹，我們將其替換為偽節點，並將新的 AST 及其（偽)節點視為錯誤語句的修復上下文。偽節點是通過嵌入技術計算的，以捕獲錯誤語句的結構（第3.2 節）。在第二層，DEAR 學習從第 5 行語句的子樹到固定子樹的轉換

第 6-9 行的語句。從第一層學習的固定上下文向量用作第二層代碼轉換學習中的權重。我們對每個錯誤語句重複相同的過程。為了修復，我們一次對所有錯誤語句執行修復轉換的組合。

關鍵思想3. 使用正確的周圍固定上下文進行轉換學習：要學習正確的上下文來修復語句，我們需要使用其他錯誤語句的固定版本來訓練模型（觀察 3）。例如，對於訓練，為了學習對第 5 行語句的修復，模型需要集成其他錯誤行的修復版本，即第 11 行的代碼，以passwordHash作為修復上下文（而不是越野車線 10)。如果使用修復前的周圍代碼（即第 10 行），模型將學習錯誤的上下文來修復第 5 行。

2.3 方法概述2.3.1訓練過程。訓練輸入

包括修復前後的源代碼（圖 2），它被解析為 AST。

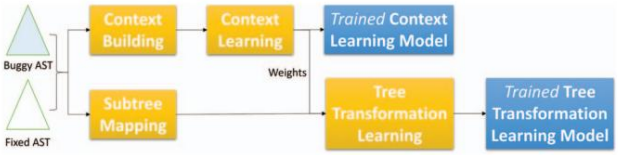


圖 2 :訓練過程概覽

輸出包括用於上下文學習和樹轉換學習（修復)的兩個訓練模型 。上下文學習模型（CTL)旨在學習權重（代表上下文的影響)以對樹變換學習結果進行調整 。樹轉換學習模型 (TTL) 旨在學習代碼轉換以修復有問題的 AST 子樹。

情境學習（第 3.2–3.3 節） 。第一步是構建訓練前/後修復上下文 。通過分而治之的策略 ,我們使用 CPatMiner [26]來導出更改 、插入和刪除的子樹（關鍵思想 2） 。結果 ,錯誤語句的 AST 子樹被映射到相應的固定子樹 。對於每個錯誤的子樹和各自的固定子樹 ,我們構建了整個方法的兩個 AST 作為上下文 ,一個在修復之前 ,一個在修復之後 ,並將它們都用於基於樹的 LSTM 上下文的輸入層和輸出層的訓練學習模型（第 3.3 節） 。

為了為每個有問題的子樹構建正確的上下文 ,我們利用了關鍵思想 3 :我們使用其他有問題的子樹的固定版本來訓練我們的模型 。最後 ,從該學習中計算出的向量用作樹變換學習中的權重。

樹轉換學習（第 3.4 節） 。我們首先使用 CPat Miner [26]來導出子樹映射 。為了學習錯誤修復樹轉換 ,每個錯誤子樹本身及其修復後的固定子樹在第二個基於樹的 LSTM 的輸入層和輸出層用於訓練 。此外 ,表示上下文的權重在上下文學習模型中計算為向量 ,在此步驟中用作附加輸入。

2.3.2固定過程 。圖 3 說明了固定過程 。輸入包括有缺陷的源代碼和測試用例集。

故障定位和 Buggy-Hunk 檢測（第 4.1 節） 。
從關鍵思想 1 開始 ,我們首先使用 SBFL 工具來定位具有可疑分數的錯誤語句。 Hunk 檢測算法使用這些語句來導出需要一起修復的錯誤 hunk。

多語句擴展（第 4.2 節） 。因為 SBFL 可能會為一個 hunk 返回一個語句 ,我們的目標是擴展到可能包括更多連續的錯誤語句 。為此 ,我們結合了 RNN [7] 和數據流分析來檢測更多錯誤語句。

基於樹的代碼修復（第 4.3 節） 。對於從多語句擴展中檢測到的錯誤語句 ,我們使用關鍵思想 2 來同時對多個錯誤子樹進行修復 。對於有問題的子樹 ,我們將方法的 AST 構建為上下文 ,並將其用作經過訓練的上下文學習 ,模型 (CTL) 的輸入 ,以產生表示上下文影響的權重。 buggy 子樹用作訓練樹轉換模型 (TTL) 的輸入 ,以生成上下文無關的固定子樹 。最後 ,該權重用於調整到候選補丁的固定子樹。

我們對當前候選代碼應用語法規則和程序分析來生成固定代碼 。我們以與 DLFix [18] 中相同的方式使用測試用例重新排名和驗證固定代碼。

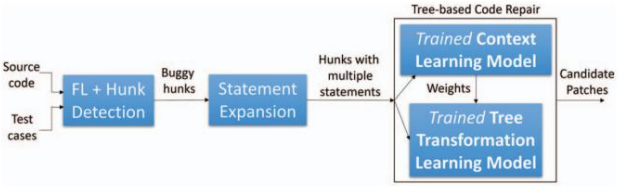


圖 3 :修復過程概覽

3 訓練過程

3.1 配對錯誤和固定子樹訓練數據包含修復前後方法源代碼對 。請注意 ,修復可能涉及多種方法 。我們沒有將整個錯誤方法與固定方法配對 ,而是使用分而治之的策略來幫助模型更好地學習適當上下文中的固定轉換 。首先 ,我們使用 CPatMiner 工具 [26] 來導出修復更改 。

如果一個子樹對應一個語句 ,我們稱它為語句子樹 。根據 CPatMiner 的結果 ,我們使用以下規則將錯誤子樹與相應的固定子樹配對： 1. 錯誤子樹 (-subtree) 是具有更新或刪除的子樹。

2. 如果一個子樹被刪除 ,我們將它與一棵空樹配對。
3. 如果一個有缺陷的子樹被標記為更新（即 ,它被更新或者它的子節點可以被插入 、刪除或更新） ,我們將這個有缺陷的子樹與它對應的固定子樹配對。
4. 如果插入一個 -subtree並且它的父節點是另一個 -subtree ,我們將它與那個父 -subtree 配對 。如果父節點不是 -子樹 ,我們將空樹與相應插入的 -子樹配對。

3.2 上下文構建圖 4 說明了我們的
上下文構建過程 。對於每對錯誤的 AST 1和固定的 AST 1 （第 3.1 節） ,我們對變量執行 alpha 重命名 。在第 1 步中 ,我們使用詞嵌入模型 GloVe [29]（捕獲良好的代碼結構)對每個 AST 節點進行編碼 ,將語句節點視為一個句子 ,將每個代碼標記視為一個詞 。我們使用這些向量來

標記1和1中的 AST 節點。 此步驟之後的 AST 是修復前後的向量化 AST 2和2。

在步驟 2 中 ,我們處理2中的每一對 buggy-subtree和 2 中對應的 fixed-subtree 。首先 ,我們使用 TreeCaps [4] 對 和 執行節點匯總 ,分別捕獲和 的樹結構到 和 中 。其次 ,對於每個其他有問題的子樹 ,例如 ,以及它們對應的固定子樹 ,例如 ,我們處理如下 。因為是我們在修復之前在生成的上下文3的構建中替換為的固定版本（關鍵思想 3） 。也就是說 ,我們用它的固定版本替換每個其他有問題的子樹 。然而 ,為了在固定後構建結果上下文3 ,我們保留它 ,因為它是固定的子樹 ,因此提供了正確的上下文。

生成的 AST 3 用作錯誤子樹的修復前上下文 ,並在上下文學習模型 (CTL) 的編碼器輸入層使用 。生成的 AST 3 用作CTL 中解碼器輸出層的修復後上下文並使用（圖 4） 。最後 ,向量和將用作後面樹變換學習的加權輸入。

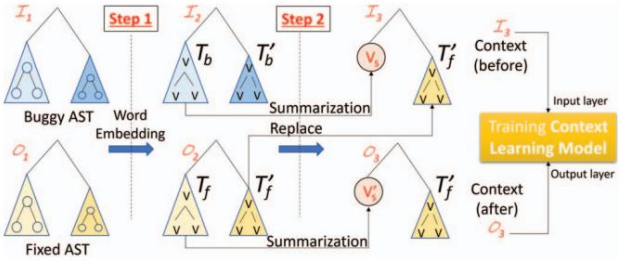


圖 4：構建上下文以訓練上下文學習模型

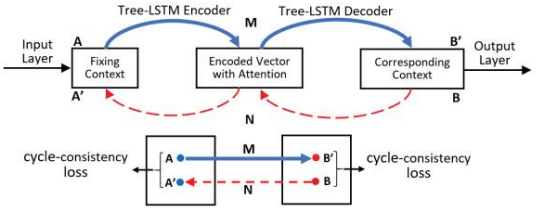


圖 5：基於注意力的 Tree-based LSTM 中的循環訓練

3.3通過帶有註意力層和循環訓練的基於樹的 LSTM 進行上下文學習

對於上下文學習和樹轉換學習，我們通過注意層和循環訓練增強了 DLFix [18]中的兩層、基於樹的 LSTM 模型。我們在該模型中添加了一個注意力層，該模型現在有 3 層：編碼器層、解碼器層和注意力層（圖 5）。對於編碼器和解碼器，為了學習用AST 表示的固定上下文，我們使用基於 Child-Sum Tree 的LSTM [36]。與針對每個時間步循環的常規 LSTM 不同，此模型針對每個子樹進行循環以捕獲結構。

我們還使用循環訓練[45]來進一步改進。循環訓練旨在通過持續訓練和重新訓練以強調輸入和輸出之間的映射來幫助模型更好地學習輸入和輸出之間的映射。這在可以通過多種方式將錯誤代碼修復為不同的固定代碼，或者可以將多個錯誤代碼修復為一個固定代碼的情況下很有用。這使得常規的基於樹的 LSTM 不太準確。通過循環訓練，強調輸入和最可能的輸出對，以減少這種一對多或多對一關係的噪聲。

循環訓練發生在編碼器和解碼器之間。我們使用
正向映射：→表示過程，反向映射：→表示過程（圖 5）。 →
→ ，
→ →

我們對兩者應用對抗性損失並得到兩個損失函數( )和, ,)。(( )和(( ))和( )之間的差異用於生成循環一致性損失，以確保學習的映射函數是循環一致的。(,)和,)。使用激勵循環一致性損失(,)，整體損失函數計算如下：

，

在數學上，我們有兩個損失函數 ， ，
(, ，

(,) = ~ ( ) [|| (( )) − ||1] (( ))
+ ~ ( ) [|| − ||1] (1)

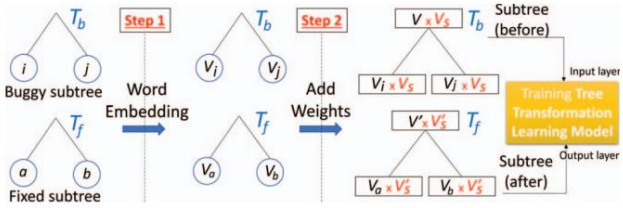


圖 6：樹轉換學習（， 在圖 4 中）

(, , ,) = ( ) , , ,) + (, , ,)
+ (, (2)

其中( )是整個循環訓練的損失函數；和 是映射到 和 的映射函數，旨在區分預測結果到；(( ))和真實結果；旨在區分預測結果(( ))和真實結果；是運行函數的循環性的 一致性損失函數是激勵循環一致性損失；是控制兩個目標相對重要參數。

， ；

3.4 樹變換學習圖6說明了樹變換學習過程。我們使用與
第 3.3 節相同的具有註意層和循環訓練的基於樹的 LSTM 模型來學習每個錯誤子樹的代碼轉換。

在第 1 步中，我們為第 3.2 節中的所有代碼標記構建詞嵌入。越野車子樹 ( ) 和固定子樹 ( ) 中的每個 AST 節點都標有其向量表示（圖 6）。接下來，我們使用從圖 4 中的上下文學習中計算得出的匯總向量作為權重，並分別對 buggy-subtree 中節點的每個向量和 fixed-subtree 中的每個向量執行叉積。

叉積後得到的兩個子樹用於基於樹的 LSTM 模型的輸入和輸出層，用於樹變換學習。我們使用叉積是因為我們的目標是將向量作為節點的標籤，並將其用作表示上下文的權重，以學習用於錯誤修復的代碼轉換。

4 固定過程

4.1 Fixing-together Hunk Detection Algorithm修復多塊、多語句錯誤的

第一步是我們的FL 方法檢測在同一補丁中固定在一起的錯誤塊。為此，我們微調了谷歌的預訓練 BERT模型[8]，以使用 BERT 的句子對分類任務學習語句之間的固定關係。然後，我們在算法中使用微調的 BERT 模型來檢測固定在一起的大塊頭。讓我們詳細解釋一下我們的 hunk 檢測算法。

4.1.1微調 BERT 以學習語句之間的固定關係。

我們首先微調 BERT 以了解是否需要將兩個語句固定在一起。讓我們成為一組為錯誤而固定在一起的大塊頭。訓練過程的輸入是訓練集中所有錯誤的所有集合。

步驟1. 對於一對帥哥和 in 語句和 ，我們從每個大塊頭中取出 ，每一對，並構建向量

與伯特 。我們考慮在同一個補丁中固定在一起的語句對 ,) 作為
（以微調 BERT。

步驟2. 對所有對和中的語句 (,) s 的所有對重複步驟 1 。我們還對所有 s 重複步驟 1 。我們使用它們來微調 BERT 模型 ,以學習所有大塊對中任意兩個語句之間的固定關係。

4.1.2使用 Fine-tuned BERT 進行 Hunk Detection 。在獲得微調的 BERT 後 ,我們用它來確定是否需要將代碼塊固定在一起 。此過程的輸入是微調的 BERT 模型 、錯誤代碼和測試用例 。輸出是需要固定在一起的大塊組 。該過程按以下步驟進行。 ,

步驟1. 我們使用基於頻譜的 FL 工具（在我們的實驗中 ,我們使用 Ochiai [3])在給定的源代碼和測試用例上運行 。它返回錯誤語句和可疑分數的列表。

第2 步 。將 FL 工具返回的方法中的連續語句組合在一起 ,形成塊1 、2 、... 、
.

第3 步 。為了決定是否需要將一對 hunks ( ) 固定在一起 ,我們使用經過微調的 BERT 模型 。具體來說 ,對於每對語句 ( ) ,我們使用微調的 BERT 來衡量 (,) 的固定關係得分 。和之間的固定在一起分數是對中所有語句對分數的平均值 。如果所有語句的平均分數)根據需要配對高於閾值 ,我們考慮（被固定在一起 。從檢測到的帥哥對中 ,我們構建了固定在一起帥哥的組 。帥哥組有落合可疑度得分最高的任何語句將排在第一位並固定 。理由是這樣的組包含最可疑的語句 ,因此應該首先固定。

,

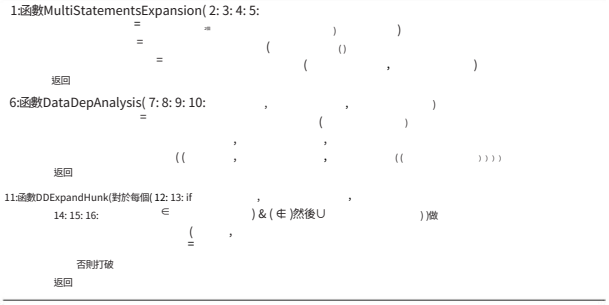
4.2 多語句擴展算法從 4.1 節的算法中檢測到的 buggy hunk 可能只包含一個語句 ,因為這些可疑語句中的每一個最初都是由 SBFL 工具派生的 ,它並不專注於檢測 hunk 中連續的 buggy 語句 。因此 ,在這一步中 ,我們從 hunk 檢測算法中獲取結果 ,並將其擴展以在 hunk 中包含更多潛在的語句。

關鍵思想 。我們的想法是將深度學習與數據流分析相結合 。我們首先使用 GRU 單元[7] （將在第 4.2.2 節中解釋)訓練一個 RNN 模型來學習判斷一個語句是否有錯誤 。我們從真實的錯誤陳述中收集該模型的訓練數據 。然後我們使用數據流分析來調整結果 。具體來說 ,如果一條語句被 RNN 模型標記為 buggy ,則不需要進行任何調整 。然而 ,即使 RNN模型將給定語句確定為無錯誤 ,並且如果數據依賴於錯誤語句 ,我們仍將其標記為錯誤。

4.2.1擴展算法。 Multi-Statement Expansion算法的輸入是buggy語句 buggyS ,即hunk的種子語句 。輸出是一大堆有問題的連續語句。

首先 ,它通過包含buggyS之前和之後的語句（第 2 行的 Expand2N CandidatesList)來生成 buggy 語句的候選列表 。在當前的實現中 ›=5 。然後，

算法 1多語句擴展算法



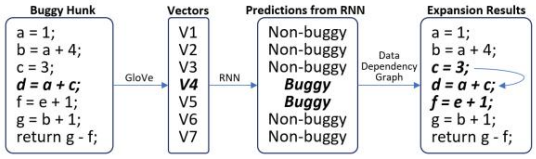


圖 7 :多語句擴展示例

它使用 RNN 模型作為分類器來預測候選列表中的每個語句（ buggyS 除外）是否存在錯誤（第 3 行的 RN NClassifier） 。為了訓練該 RNN 模型 ,我們在訓練數據的 buggy hunks 中使用了 buggy語句（參見第 4.2.2 節） 。

TreeCaps [4] 用於對語句進行編碼。

在 DataDepAnalysis（第 4 行)中 ,為了調整 RNN 模型的結果 ,我們獲得了圍繞錯誤語句buggyS 的 buggyHunk ,其中包含buggyS之前和之後的語句 ,這些語句被 RNN 模型預測為錯誤（第 7 行） 。然後 ,我們從候選列表中的中央錯誤語句（第 8 行 ,通過 TopHalf)向上和向下（第 9 行 ,通過 BotHalf)逐條檢查語句 。在 DDExpandHunk 中 ,我們繼續（向上或向下)擴展當前的 buggy hunk bug gyHunk 以包含被 RNN 模型視為錯誤或與中心 buggy 語句buggyS 具有數據依賴性的語句（第 13-14 行） 。如果我們遇到一個沒有 buggyS 的數據依賴性的非錯誤語句 ,或者我們用盡了列表（第 15 行） ,我們將停止該過程（向上或向下） 。最後 ,返回包含連續 buggy 語句的buggy hunk 。

在圖 7 中 ,SBFL 工具在第 4 行返回錯誤語句。
所有語句都通過 GloVe [29]編碼到向量集中 ,並由 RNN 模型分類 。我們從第 4 行的語句向上擴展以包括第 3 行（即使 RNN 模型預測它沒有錯誤） ,因為第 3 行通過變量c 與第 4 行的錯誤語句具有數據依賴性 。我們包括第 5 行 ,因為 RNN 模型預測第 5 行有錯誤 。此時 ,我們停止向上和向下方向 ,因為我們在第 2 行和第 6 行遇到與第 4 行沒有數據依賴性的非錯誤語句 。即排除第 1–2 行和第 6–7 行 。最終結果包括第 3-5 行的語句作為 buggy hunk。

4.2.2使用 RNN 進行錯誤語句預測 。我們介紹瞭如何使用基於 GRU 的 RNN 模型[7]來預測錯誤語句。

訓練 。為了訓練 RNN 模型 ,我們在訓練數據集中的所有塊中使用有缺陷/無缺陷的語句 。我們使用 GloVe [29]

對語句中的每個標記進行編碼，以便語句由一系列標記向量表示。我們使用基於 GRU 的 RNN 模型[7]的神經架構來使用與錯誤/非錯誤標籤關聯的語句的 GloVe 向量。

RNN 模型以時間步長運行。在時間步長(>=1)，在輸入層，GRU 消耗的 GloVe 向量被標記為 1，如果它是錯誤的語句，否則為 0。除了時間步長 + 1 的輸入，我們將時間步長的輸出饋送到 GRU。在輸出層，陳述。

預言。經過訓練的基於 GRU 的 RNN 模型在第 3 行的擴展算法中用於預測 hunk 中的語句是否有錯誤。該模型採用 GloVe 標記向量形式的語句。它採用 hunk 中所有語句的向量，並以多時間步長的方式將它們標記為錯誤或非錯誤。

4.3 基於樹的代碼修復圖 8 說明了這個過程。在導出方法中的錯誤塊後，DEAR使用經過訓練的 LSTM 模型一次對所有塊中的所有錯誤語句執行代碼修復。

基於樹的代碼修復按以下步驟進行：

第 1 步。識別有缺陷的 S-子樹。對於每個塊，我們將代碼解析為 AST，並識別與派生的錯誤語句對應的錯誤子樹。在圖 8 中，子樹1和2被標識為有問題。如果一個 buggy-subtree 是另一個更大的 buggy-subtree 的一部分，我們只需要對更大的 buggy-subtree 進行修復

-subtree 因為該修復還修復了較小的 -subtree。

第 2 步。嵌入和總結。我們使用 GloVe [29]執行詞嵌入，並使用 TreeCaps [4]在所有有問題的子樹上執行樹摘要以獲得上下文。例如，在圖 8 中， 1和2被匯總為兩個向量1和2。

第 3 步。預測上下文。我們使用經過訓練的上下文學習模型(CTL) 在具有 AST 節點的上下文上運行，其中還包括用於預測上下文的匯總節點。在生成的 AST 中，結構與輸入上下文的 AST 相同，只是匯總節點變為新節點。例如，上下文中的1和2在步驟 3 之後變為 and。

第 4 步。添加權重。來自步驟2的權重1和2用於與越野車子樹1和2 中的向量的乘積。1和2中的每個節點由節點的原始向量與權重向量之間的乘法向量表示

或者 2.

第 5 步。預測轉換。我們使用經過訓練的樹轉換學習模型來預測子樹和修復。

第 6 步。去除重量。我們從第 4 步中刪除權重以獲得有缺陷子樹的候選固定子樹。例如，我們
消除 和 獲得候選固定子樹 和 2.
然而，因為我們知道叉積和一個向量，我們可以獲得無限數量的解決方案。因此，為了產生一個單一的解決方案，我們假設
2、 對於每個節點 和 2、
和 與節點向量1垂直 1 和二合一 2.
然後，我們可以得到未加權的節點向量 合1 如下：_

$$= \frac{\times 1}{1\,1} \quad (3)$$

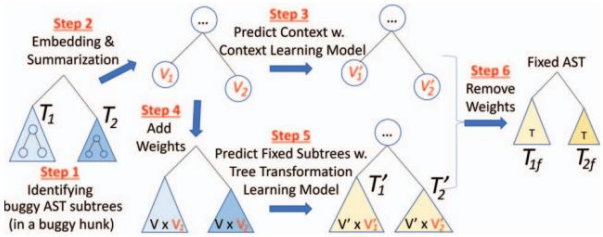


圖 8：基於樹的代碼修復

在獲得固定 S 子樹中每個節點的向量後，我們基於詞嵌入生 1，成了候選補丁。對於每個節點，我們計算了它與向量列表中所有標記的每個向量之間的餘弦相似度得分。要1生成候選補丁，我們選向量 擇列表中的令牌。固定子樹中一個節點的令牌。

有它的相似性分數通過將所有標記的所有相加，我們得到了候選人的總分。我們為每個，人選出前 5 名候選人節點生成候選人並根據 .

4.4 後處理天真的方法在形成候選修

復時會面臨組合爆炸，因為對於子樹中的每個節點，我們維護前 5 個候選。但是，當我們將固定子樹中所有節點的候選者組合在一起時，許多候選者對項目中的當前方法無效。因此，當我們通過組合節點的所有候選者形成候選者時，我們應用一組過濾器以與 DLFix [18] 中相同的方式驗證程序語義。這使我們能夠立即消除無效的候選日期。具體來說，我們使用 alpha 重命名過濾器使用包含範圍內所有有效名稱的字典將名稱更改回正常的 Java 代碼，使用語法檢查過濾器來刪除有語法錯誤的候選者，以及名稱驗證過濾器來檢查變量、方法和類的有效性。此外，為了進一步改進，我們使用集束搜索來僅維護排名靠前的候選修復。因此，我們在形成陳述時並未窮盡所有成分。這有助於維持可管理的候選人數量。

應用所有過濾器後，我們還在候選補丁上使用了 DLFix [18] 的重新排序方案。然後我們使用測試用例對這些候選者進行補丁驗證。我們從上到下驗證每個補丁，直到識別出正確的補丁並且補丁驗證結束。如果固定位置的所有候選人都不能通過所有測試用例，我們選擇下一個位置重複該過程。

5 實證評估

5.1 研究問題為了評估 DEAR，我們尋求回

答以下問題： RQ1。在 Defects4J 基准上與基於深度學習的 APR 模型進行比較研究。與 Defects4J 上現有的基於 DL 的 APR 模型相比， DEAR 的表現如何？

RQ2。與基於深度學習的 APR模型在大型錯誤數據集上的比較研究。與基於深度學習的 APR 模型相比，DEAR 在大規模錯誤數據集上的表現如何？

RQ3 。基於模式的 APR 方法對 Defects4J 的比較研究 。與最先進的基於模式的 APR 方法相比 ,DEAR 的表現如何？

RQ4。 DEAR 的敏感性分析 。各種因素如何影響 DEAR 在 APR 中的整體表現？
RQ5 。時間複雜度和模型的訓練參數。
什麼是時間複雜度和訓練參數的數量？

5.2 數據收集

我們對三個數據集進行了實證評估 :1) Defects4J v1.2.0 [1] 有 395 個錯誤和測試用例； 2) BigFix [18] 在 +180 萬個錯誤方法中有 +26k 個錯誤； 3) CPatMiner [26] 有來自 5,832 個 Java 項目的 +44k 個錯誤。

所有實驗均在配備 8 核Intel CPU 和單個 GTX Titan GPU 的工作站上進行。

5.3 實驗方法5.3.1 RQ1 。與 Defects4J 上基於深度學習的 APR 的比較。

比較基線 。我們將 DEAR 與五個最先進的基於 DL 的 APR 模型進行了比較： DLFix [18]、 CoCoNuT [23]、 SequenceR [6]、 Tufano19 [38]、 CODIT [5] 和CURE [14]。

程序和設置 。我們複製了所有基於 DL 的 APR ,但CURE 除外 ,它不可用 。我們按照他們論文中的細節重新實現了 CURE 。我們針對 CPatMiner 數據集中的錯誤和修復訓練了所有 DL 方法 ,並針對Defects4J中的所有 395 個錯誤（兩個數據集之間沒有重疊)對其進行了測試 。所有 DL 方法都應用相同的故障定位工具 Ochiai [3] ,並使用 Defects4J 中的測試用例進行補丁驗證 。根據之前的實驗[13 、18] ,我們為補丁生成和驗證工具設置了 5 小時的運行時間限制。

我們使用beam-search 使用以下關鍵超參數調整 DEAR :(1) 用於 hunk 檢測的 BERT :epoch 大小 (e-size) (2, 3, 4, 5) ,批量大小 (b-size) (8 , 16, 32, 64),和學習率 (l-rate)
(3 -4, 1 -4, 5 -5, 3 -5, 1 -5)； (2) LSTM for Multi-Statement Expansion and code repair: e-size (100, 150, 200, 250), b-size (32, 64, 128, 256), l-rate (0.0001, 0.0005, 0.001) , 0.003, 0.005); (3) GloVe for representation vectors: vector size (v-size) (100, 150, 200, 250), l-rate (0.001, 0.003, 0.005, 0.01), b-size (32, 64, 128, 256)),和電子尺寸(100, 150, 200, 250) 。使用其他默認參數。

DEAR 的最佳設置是 (1) e-size=4, b-size=32, l-rate=1 for BERT； (2) e-size=200, l-rate=0.003, b-size=128 for LSTM; (3)對於 GloVe ,v 大小=200 ,l-rate=0.001 ,b-size=64 ,e-size=200 。對於其他模型 ,我們調整了他們論文中的參數 ,例如 word2vec 的向量長度 、學習率和 epoch 大小 ,以找到每個數據集的最佳參數。我們在同一 CPatMiner 數據集上使用上述參數調整了所有方法以獲得最佳性能 。一旦我們獲得了每個模型的最佳參數 ,我們就將它們用於以後的實驗。

10^{-4}

定量分析 。我們報告的錯誤數量
模型可以自動修復以下錯誤位置類型：
類型 1。 One-Hunk, One-Statement :修復了一個錯誤 ,只涉及一個語句。
類型 2。 One-Hunk ,Multi-Statements :修復的錯誤僅涉及一個帶有多個語句的 hunk。
類型 3。 Multi-Hunks, One-Statement :修復的錯誤涉及多個帥哥 ;每個大塊頭都有一個固定的陳述。
類型 4。 Multi-Hunks ,Multi-Statements :修復的錯誤涉及多個帥哥 ;每個大塊頭都有多個語句。
類型 5。 Multi-Hunks ,Mix-Statements :修復涉及多個 hunks 的錯誤 ,一些 hunks 有一個語句 ,而其他 hunks 有多個語句。

評估指標 。我們使用排名靠前的候選補丁報告可以正確修復的錯誤數量和合理補丁的數量（即通過所有測試用例 ,但未通過實際修復）。

5.3.2 RQ2 。與大型數據集上基於 DL 的 APR 的比較。
比較基線 。我們在兩個大型數據集 :BigFix 和 CPatMiner 上將 DEAR 與 RQ1 中的相同基線進行比較。

程序和設置 。首先 ,我們在 BigFix 和 CPatMiner 上評估了所有基於 DL 的 APR 模型 。在 DLFix 和 Sequencer 之後 ,我們將數據隨機分成 80%/10%/10% 用於訓練、調優和測試 。其次 ,我們有跨數據集評估 :在 CPatMiner 上訓練基於 DL 的方法並在 BigFix 上測試 ,反之亦然 。與 Defects4J 不同 ,BigFix 和 CPatMiner 數據集沒有測試用例。

沒有測試用例 ,我們就無法對所有 DL 方法使用故障定位和補丁驗證 。因此 ,我們將實際的 bug 位置輸入到 DL 模型中 ,包括 buggy hunk 和語句上的位置 。基於 DL 的基線不區分塊 ,而是一次處理每個錯誤語句 。我們使用開發人員的實際修復作為基本事實來評估基於 DL 的方法。

評估指標 。我們使用 top 度量 ,定義為正確補丁在排名靠前的候選列表中的次數與錯誤總數之比。

5.3.3 RQ3 。與 Defects4J 上基於模式的 APR 的比較。
比較基線 。我們將 DEAR 與 Defects4J 上最先進的基於模式的 APR 工具進行比較： Elixir [33]、 ssFix [43]、 CapGen [40]、 FixMiner [16]、 Avatar [19]、 Hercules [34]、 Sim Fix [13]和Tbar [20] 。我們能夠在相同的計算環境下複製以下基於模式的基線： Elixir 、ssFix 、FixMiner 、SimFix 、TBar 。我們將工具的時間限制設置為 5 小時 。對於其他基線 ,由於代碼不可用 ,我們使用他們論文中報告的結果 ,因為它們在同一數據集上運行 。我們使用相同的設置和評估指標。

5.3.4 RQ4 。敏感性分析 。我們評估了不同因素對 DEAR 績效的影響 。我們考慮以下幾點： (1)大塊頭檢測（Hunk）； (2) 多語句擴展(Expansion)； (3) 多語句樹模型和循環訓練； (4)數據拆分方案 。我們對每個因素都使用留一法策略。

我們在 Defects4J 上評估前三個因素 ,在 CPatMiner 上評估最後一個因素 ,因為我們需要更大的數據集來進行各種拆分。

5.3.5模型訓練的時間複雜度和參數個數 。我們測量模型的訓練和修復時間及其在數據集上進行模型訓練的參數數量。

6 實證結果

6.1 問題 1 。與基於深度學習的 APR 模型在 Defects4J 上的比較結果

6.1.1故障定位 。我們首先評估 APR 模型

當與故障定位工具 Ochiai [3] 一起使用時 。表 1 和表 2 顯示了 DEAR 和基線模型之間的比較結果。

表 1：RQ1。與基於深度學習的 APR 模型在具有故障定位的 Defects4J上的比較

項目	圖表關閉Lang Math Mockito時間總計6/9 15/19 3/5 6/12 6/8 14/18 30/55					
音序器	3/3	4/5	2/2	0/0	0/0	
它被編碼	1/2	2/5	0/0	0/0	0/0	
龍捲風19	3/4	3/5	1/1	0/0	0/0	
DL修復	5/12 6/10 5/12 12/18			1/1	1/2	
椰子 6/11		6/9	5/13 13/21 6/13	2/2	1/1	33/57
治愈	6/10 5/14 16/23			2/2	1/2	36/71
X/Y：分別	8/16	7/11	8/15 20/33	1/2	3/6 47/91	親愛的

是正確和似是而非的補丁的數量。

表 2：RQ1。在具有故障定位的 Defects4J 上與基於深度學習的 APR 模型進行詳細比較

Bug 類型類型	DLFix CoCoNUT CURE 親愛的			
1. One-Hunk One-Stmt Type 2. One-	30	33	36	29
Hunk Multi-Stmts Type 3. Multi-Hunks	0	0	0	[illegible]
One-Stmt Type 4. Multi-Hunks Multi-	0	0	0	11
Stmts Type 5. Multi-Hunks Mix-Stmts Total	0	0	0	1
	0	0	0	[illegible]
	30	33	36	47

如表 1 所示，DEAR 可以自動修復最多數量的錯誤(47) 並生成最多數量的合理補丁 (91)，這些補丁通過了 Defects4J 上的所有測試用例。特別是，DEAR 可以自動修復 32、41、33、17、14和 11 個錯誤，分別比 Sequencer、CODIT、Tufano19、DLFix、CoCoNuT 和 CURE 多（即 213%、683%、236%、57%）、42% 和 31% 的相對改進）。與 Defects4J 上的那些工具相比，DEAR 可以自動修復這些工具分別遺漏的35、34、41、18、31 和 18 個錯誤。通過DEAR 的結果與基線組合的結果之間的重疊分析， DEAR 可以修復他們遺漏的 18 個獨特的錯誤。

表 2 顯示了 DEAR 和基於深度學習的頂級基線（DLFix、CoCoNuT、CURE)與不同錯誤類型之間的比較。

對於單塊錯誤（類型 1-2），DEAR 修復了 33 個錯誤，包括其他工具遺漏的 4 個獨特的單一大塊錯誤。

對於多塊錯誤（類型 3-5），DEAR 可以修復DLFix、CoCoNuT 和 CURE 無法修復的14 個錯誤。現有的基於 DL 的APR 模型無法修復這些錯誤，因為一次修復一個語句的機制不適用於需要同時修復多個語句的相關更改的錯誤。因此，他們不會為這些情況生成正確的補丁。

對於multi-hunk 或 multi-statement錯誤（類型 2-5），親愛的修復其中 18 個（在 47 個修復錯誤中，即修復錯誤總數的 38.3%）。

6.1.2無故障定位。我們還在不受第三方 FL 工具影響的情況下，將 DEAR 與其他工具的修復能力進行了比較。所比較的所有工具（表 3)都指向正確的修復位置並執行修復。如圖所示，如果已知修復位置，DEAR 的修復能力也高於那些基線（53 個錯誤對 44、40 和 48）。

重要的是，它可以修復 20 個 multi-hunk/multi-statement 錯誤（佔總共 53 個已修復錯誤的37.7%），而 CoCoNuT、DLFix 和 CURE 只能修復 7、5 和 10 個此類錯誤。

DEAR 比現有的基於 DL 的模型更通用，因為它可以支持具有 multi-hunk 或 multi-statements 的依賴修復。

重要的是，它顯著改進了這些基於 DL 的模型和

表 3：RQ1。與沒有故障定位（即正確定位)的 Defects4J上基於深度學習的 APR 模型的比較

Bug 類型類型	DLFix CoCoNUT CURE 親愛的			
1. One-Hunk One-Stmt Type 2. One-	35	37	38	33
Hunk Multi-Stmts Type 3. Multi-Hunks	[illegible]	[illegible]	[illegible]	[illegible]
One-Stmt Type 4. Multi-Hunks Multi-	[illegible]	3	6	13
Stmts Type 5. Multi-Hunks Mix-Stmts Total	0	0	0	1
	0	[illegible]	[illegible]	[illegible]
	40	44	48號	53

表 4：RQ2。與大型數據集上的 DL APR 比較

	CPatMiner (4,415 個測試錯誤)			BigFix (2,594 個測試錯誤)		
工具/數據集 Top-1	Top-3	Top-5	Top-1	Top-3	Top-5	Sequencer 7.8% 10.8% CODIT 4.5%
9.1% Tufano19		8.9%	10.3%	8.5%	9.1%	
		7.4%	9.2%	3.9%	6.3%	
	8.6%	9.3%	11.2%	7.7%	8.8%	9.6%
DL修復	11.4%	12.3%	13.1%	11.2%	11.9%	12.5%
椰子	13.5%	14.7%	15.3%	12.2%	13.6%	14.3%
治愈	14.2%	15.1%	15.5%	12.9%	14.2%	14.1%
親愛的	15.1%	15.6%	16.8%	14.1%	15.4%	16.3%

表 5：RQ2。與跨數據集上的 DL APR 比較

工具/數據集	CPatMiner(Train)/BigFix		BigFix(Train)/CPatMiner		Top-1 Top-3 Top-5	
	Top-1	Top-3	Top-5	5.4% 5.8% 6.1% 7.2%		
音序器			6.2%	5.3%		
它被編碼	2.5%	4.0%	4.4%	3.2%	5.2%	6.4%
龍捲風19	4.5%	5.4%	5.7%	5.9%	6.3%	7.6%
DL修復	6.3%	6.9%	7.3%	8.2%	8.7%	9.2%
椰子	6.7%	7.4%	8.1%	8.3%	9.6%	10.7%
治愈	7.1%	7.7%	8.2%	8.7%	9.9%	10.9%
親愛的	7.5%	8.1%	8.6%	9.6%	10.2%	11.3%

將 DL 方向提升到與其他 APR 方向（基於搜索和基於模式)相同的級別，可以處理多語句錯誤。此外，DEAR 是完全數據驅動的，不需要像基於模式的 APR 模型那樣定義固定模式。

6.2 RQ2。在大型數據集上與基於 DL 的 APR 模型的比較結果表 4 顯示，DEAR 可以修復比兩個大型數據集上任何基於 DL 的 APR 基線更多的錯誤。使用 top-1 補丁，DEAR可以修復 CPatMiner 中 4,415 個錯誤中的 15.1%。它修復了比帶有 top-1 補丁的基線多40-322個錯誤。在 BigFix 上，它可以使用 top-1 補丁修復 2,594 個錯誤中的 14.1%。它可以比使用 top-1 補丁的基線多修復31-145 個錯誤。

表 5 顯示 DEAR 在交叉數據集設置中也優於基線，在該設置中我們在 CPatMiner 上訓練模型並在 BigFix 上測試它們，反之亦然。

表 6 顯示了 CPatMiner 與不同錯誤類型的詳細比較結果。如圖所示，DEAR 可以在兩個大型數據集上的每種類型的錯誤位置上自動修復更多錯誤。在 667 個修復的錯誤中，DEAR 修復了 169 個類型 2-5 的 multi-hunk 或 multi-stmt 錯誤（即，佔修復錯誤總數的 25.33%）。 DEAR 修復了比基線 CoCoNuT、DLFix 和 CURE更多的錯誤（71、164 和 41 個），並且修復了每種錯誤類型中更多的錯誤。

表 6 :RQ2 。詳細分析 。在 CPatMiner 數據集上與基於 DL 的 APR 模型進行 Top-1 結果比較

類型（#bugs）	椰子	治愈	DL修復	親愛的
	#固定的	#固定的	#固定的	#固定的
類型 1 (1,668)	28.7% (479)	29.8%(497)	23.7% (395)	29.9% (499)
2 型 (530)	1.3% (7)	2.1% (11)	0.6% (3)	4.2% (22)
3 型 (879)	12.4% (109)	13.2% (116)	11.8% (104)	13.7% (120)
類型 4 (1,089)	0% (0)	0% (0)	0% (0)	2.0% (22)
5 型 (249)	0.4% (1)	0.8% (2)	0.4% (1)	2.0% (5)
總計 (4,415)	13.5% (596)	14.2% (626)	11.4% (503)	15.1% (667)

表 7 :RQ3 。與基於模式的 APR 模型的比較

項目 ssFix	圖表關閉Lang Math Mockito時間總計 20/60 21/25 25/31					
3/7		2/11	5/12 10/26		0/0	0/4
CapGen 4/4		0/0	5/5	12/16	0/0	0/0
修復礦工 5/8		5/5	2/3	12/14	0/0	1/1
長生不老藥 4/7 8/12 12/19		0/0			0/0	2/3
阿凡達 5/12 5/11 6/13		8/12			2/2	1/3
模擬修復 9/13 14/36		6/8			0/0	1/1
Tbar 5/13 19/36 9/14		8/12			1/2	1/3
赫拉克勒斯 8/10 8/13 10/15 20/29 0/0 49/72						3/5
親愛的8/16 7/11 8/15 20/41 1/2 47/91						3/6

26/41
27/53
34/56
43/81

X/Y :是正確且合理的補丁的數量 ;數據集 :Defects4J

DEAR 比 CoCoNuT 、DLFix 和 CURE 多修復了 52 、61 和 40 個 multi-hunk/multi-stmt 錯誤 ,以及 20 、104 和 2 個 one-hunk/one-stmt 錯誤。對於其他工具修復的多語句錯誤（類型 2 和 5） ,修復的語句是獨立的 。此結果表明一次修復每個單獨的語句是行不通的。

6.3 問題 3 。與基於模式的 APR 模型的比較結果如表 7 所示，
DEAR 在錯誤數量方面與頂級基於模式的工具
Hercules 和 Tbar處於同一水平。

表 8 顯示了不同錯誤類型的比較細節 。如圖所示 ,DEAR 修復了 Hercules 遺漏的 7 個多/混合語句錯誤（類型2 、4-5） 。進一步調查 ,我們發現 Hercules 旨在修復複製的錯誤 ,即 hunks 必須有類似的語句 。這 7 個 bug 是不可複制的 ,即buggy hunks 有不同的 buggy statements 或者 buggy hunk 有多個不相似的 buggy statements 。對於類型 1 和類型 3 ,DEAR修復的單語句錯誤比 Hercules 少 9 個 ,因為它的修復不正確。 DEAR 總共修復了 Hercules 遺漏的 12 個錯誤 :圖表7 、16 、20 、24 ;時間-7 ;關閉 - 6,10,40; 朗10 ;數學 41,50,91。

與 Tbar 相比 ,DEAR 修復了 15 個多的 multi-hunk/multi-stmt錯誤。Tbar 的設計目的不是像 DEAR 那樣一次修復多語句。
相反 ,它一次修復一個語句 ,因此 ,當這 15 個錯誤需要對多個語句進行依賴修復時 ,它就無法正常工作。
Tbar 可以修復的 3 個類型 2 錯誤是對單個語句的修復是獨立的 。同樣的原因也適用於 SimFix。 Tbar 修復了 11 個更正確的 one-hunk/one-statement 錯誤。
簡而言之 ,我們將基於 DL 的模型 DEAR 提高到可比的
以及與那些基於模式的 APR 模型的互補水平。

6.4 RQ4 。敏感性分析6.4.1 Fixing-together
Hunk 檢測的影響 。如表 9 所示 ,在沒有 hunk 檢測的情況下 ,DEAR 可以自動修復 35 個錯誤 。通過 hunk 檢測 ,DEAR 可以修復 14 個以上的 multi-hunk 錯誤（類型 3-5） 。它

表 8 :RQ3 。與基於模式的 APR 的詳細比較

Bug 類型類	SimFix Tbar Hercules 親愛的			
型 1. One-Hunk One-Stmt	30	40	34	29
Type 2. One-Hunk Multi-Stmts	1	3	0	4
Type 3. Multi-Hunks One-Stmt		0	15	11
Type 4. Multi-Hunks Multi-Stmts	0	0	0	
Type 5. Multi-Hunks Mix-Stmts	0	0	0	1 2
Total	34	43	49	47

表 9 :RQ4 。缺陷敏感性分析4J

變體沒有	大塊頭	沒有 擴展 30	沒有 注意週期	親愛的
1型26	31			29
2 型 40 2				4
Type-3	0	13	9	11
類型 4 00 1				
5型00 2				1 2
總計 40	35	43		47

由於不正確的 hunk 檢測 ,修復了兩個較少的 Type-1 錯誤 。簡而言之 ,塊檢測很有用 ,因為多塊/多語句錯誤需要同時對多個塊進行依賴修復。

6.4.2 多語句擴展的影響 。如表 9 所示 ,在沒有擴展的情況下 ,DEAR 修復了 Defects4J 中的 43 個錯誤 。通過擴展 ,它修復了類型 2 、4 、5 中的 7 個多 stmt 錯誤 ,同時修復了 2 個 Type-3 錯誤和 1 個 Type-1 錯誤 。在這兩種類型中修復較少錯誤的原因是多語句擴展可能通過將單語句錯誤視為多語句錯誤來錯誤地擴展錯誤塊 。即便如此 ,DEAR 仍然可以修復更多的錯誤 ,可見多語句展開的用處。

為了比較 Hunk Detection 和 Multi-Statement Expansion 的影響，讓我們注意到沒有 Hunk Detection的 DEAR 變體錯過了所有 14 個 multi-hunk 錯誤（類型 3 、4 、5） 。沒有擴展的變體錯過了所有 7 個多語句錯誤（類型 2 、4 、5） 。
但是 ,讓我們考慮一下修復它們的難度有多大 。在使用 Hunk-Detection 修復的 14 個多塊錯誤中 ,有 11 個錯誤屬於 Type-3（多塊/單語句） ,其中一些方法（例如， Hercules)可以通過一次修復一個語句來處理 。只有 3 個錯誤屬於類型 4-5 。相比之下 ,Expansion 修復的所有 7 個錯誤都是多語句錯誤（類型 2 、4 、5） ,現有的基於 DL 的 APR 方法無法修復這些錯誤 。因此，Expansion 有助於處理比 Hunk-Detection 更具挑戰性的錯誤。

6.4.3 具有註意力和循環訓練的基於樹的 LSTM 模型的影響。 （注意力週期）
為了衡量注意力和周期訓練的影響 ,我們從 DEAR 中刪除了這兩種機制以生成基線 。我們的結果表明 ,在 Defects4J 中 ,DEAR在所有錯誤類型上比基線多修復了 7 個錯誤（增加 17.5%）。

該結果表明這兩種機制的有用性。

6.4.4 訓練數據大小的影響 。表 10 顯示訓練數據的大小對 DEAR 的性能有影響 。如表 10 所示 ,訓練數據越多 ,DEAR 的準確性越高。

這是預期的 ,因為 DEAR 是一種數據驅動的方法 。但即使訓練數據較少（70%/30%） ,DEAR 也達到了 11.7% 的 top-1 結果 ,仍然高於 DLFix（11.4% in top-1)和 Sequencer（7.7% in top-1） ;兩者都有更多的訓練數據（90%/10% 分裂）。

表 10：訓練數據大小的影響

CPatMiner 數據集的拆分方案	90%/10%	80%/20%	70%/30%	13.8%
Top-1 的錯誤總數百分比	15.1%			11.7%

```
1 public void excludeRoot( String path) { 2 -
String url = toUrl(path) ; 3 -
findOrCreateContentRoot(url) 。 addExcludeFolder( 網址 ) ; 4 +
網址 url = toUrl(路徑); 5 +
findOrCreateContentRoot(url).addExcludeFolder(url.getUrl()); 6 } 7

public void useModuleOutput(String production, String test )
-       { modifiableRootModel.inheritCompilerOutputPath( false ) ;
9 - modifiableRootModel.setCompilerOutputPath(toUrl(production)); 10 -
modifiableRootModel.setCompilerOutputPathForTests( toUrl( test ) ) ; 11 +
modifiableRootModel.setCompilerOutputPath(toUrl(production).getUrl()); 12 +
modifiableRootModel.setCompilerOutputPathForTests(toUrl(test).getUrl()); 13 }
```

圖 9：CPatMiner 中的多塊/多語句修復

6.5 RQ5。時間複雜度和參數DEAR 在 CPatMiner 上的訓練時間為 +22 小時，在 CPatMiner 上預測每個候選補丁需要 2.4-3.1 秒。在 BigFix 上訓練DEAR 花費了 18-19 小時，在 BigFix 上預測每個候選人花費了 3.6-4.2 秒。由於數據集小得多，對候選人的Defects4J 預測只用了 2.1 秒。

每個測試用例的測試執行時間為 +1 秒。對所有錯誤修復的測試用例進行測試驗證需要 2-20 分鐘。

最佳基線 CURE [14]比 DEAR（RQ1和 RQ2)修復的錯誤更少，並且在 CPatMiner 和 BigFix 上分別需要比 DEAR 多 7 倍和 7.3 倍的訓練參數。具體來說， DEAR 和 CURE 在CPatMiner 上需要 0.39M 和 3.1M 訓練參數，在 BigFix 上需要 0.42M 和 3.5M 參數。因此，DEAR比 CURE 更簡單，同時取得了更好的結果。

有效性的威脅。我們測試了 Java 代碼。 DEAR 中的關鍵模塊是獨立於語言的，除了第三方 FL 和帶有程序分析的後處理。基於模式的 APR工具需要帶有測試用例的數據集，因此，我們僅在 Defects4J 上比較了它們。我們盡力重新實現基於模式的APR 基線和 CURE 以進行公平比較。

說明性示例。圖 9 顯示了來自 DEAR 的正確修復。它正確地檢測到兩個越野車大塊頭；每個都有多個語句。
DEAR 利用同一方法中存在的變量名稱（第 8 行的 modifiableRootModel）編寫第 11-12 行的固定代碼。基於 DL 的基線，Sequencer [6]和 CoCoNuT [23]，將代碼視為序列，並且不能很好地得出此修復程序的結構更改。
DLFix 一次修復一個語句，因此不起作用（第 2 行和第 3 行的修復相互依賴）。對於基於模式的 APR [13、34]，沒有針對此錯誤的修復模板。

限制。親愛的有以下限制。首先，與 ML方法一樣，修復罕見或詞彙外的名稱具有挑戰性。有了更多的訓練數據，DEAR 就有更高的機會遇到生成新名稱的成分。其次，我們只關注導致測試失敗的錯誤。安全性、漏洞和非失敗測試錯誤仍然是它的局限性。第三，我們不能通過添加幾個新語句或任意大的依賴固定語句來生成修復。四、擴容算法會產生不正確的塊以進行修復，從而導致在正確的語句中進行修復。最後，我們目前專注於 Java，然而， DEAR 中使用的基本表示，例如令牌、AST、依賴項，對任何程序語言都是通用的。只有第三方 FL 和帶有語義檢查器的後處理是語言相關的。

7 相關工作

基於深度學習的 APR 方法。 DeepRepair [42]學習代碼相似性以從與錯誤代碼相似的代碼片段中選擇修復成分。 DeepFix [11]學習語法規則來修復語法錯誤。棘輪[12]，圖法諾等人。 [39]和 SequenceR [6]主要使用神經網絡機器翻譯 (NMT) 和基於注意力的編碼器-解碼器和代碼抽象來生成補丁。 CODIT [5]對代碼結構進行編碼，學習代碼編輯，並採用 NMT 模型來提出修復建議。圖法諾等人。 [38]使用帶有代碼抽象和關鍵字替換的序列到序列 NMT 來學習代碼更改。 DLFix [18]有一個基於樹的翻譯模型來學習修復。 CoCoNuT [23]開發了一個上下文感知的NMT 模型。 CURE [14]提出了一種使用 GPT模型[31]的代碼感知 NMT 。現有的基於 DL 的 APR 模型一次修復單個語句並且對 multi-hunk/multi-stmt 錯誤無效。

基於模式的 APR 方法。這些方法從先前的修復[15、17、19、27]中自動或半自動地挖掘和學習修復模式[17、19、20、27]。 Prophet [22]從一組成功的人類補丁中學習代碼正確性模型。

Droix [37]使用基於搜索的修復來學習崩潰的常見根本原因。 Genesis [21]自動從用戶提交的補丁中推斷出補丁生成。 HDRepair [17]用圖形挖掘修復模式。 ELIXIR [33]使用來自 PAR 的模板和局部變量、字段或常量來構建固定表達式。 CapGen [40]、 SimFix [13]、 FixMiner [16]依賴於從現有補丁中提取的頻繁代碼更改。 Avatar [19]利用靜態分析違規的修復模式。 Tbar [20]是一個基於模板的 APR 工具，具有收集的修復模式。 Angelix [25]捕獲程序語義來修復方法。 ARJA [44]生成較低粒度的補丁表示，從而實現高效搜索。我們沒有與Angelix 進行比較，因為我們與性能優於 Angelix 的 CapGen 進行了比較。

我們無法重現 ARJA，但是，ARJA 僅修復了 18 個錯誤，而 DEAR 在 Defects4J 中的相同四個項目中修復了 42 個錯誤。

8 結論

在這項工作中，我們做出了三個關鍵貢獻：1)一種新穎的 FL 技術，用於將傳統SBFL 與深度學習和數據流分析相結合的多塊、多語句修復； 2) 使用分而治之策略生成多塊、多語句修復的組合方法； 3) 增強和編排具有註意力層和循環訓練的兩層 LSTM 模型。

在 Defects4J 上，DEAR在修復錯誤的數量方面優於基於 DL 的 APR 基線 42%–683%。在 BigFix 上，它使用 top-1 補丁修復了 31-145 個以上的錯誤。在 CPatMiner 上，它修復了比基線多 40-52 個 multi-hunk/multi-stmt 錯誤。

致謝

這項工作部分得到美國國家科學基金會(NSF) 贈款 CNS-2120386、CCF-1723215、CCF-1723432、TWC 1723198、CCF-1518897 和 CNS-1513263 的支持。

參考

[1] 2019.Defects4J 數據集。 https://github.com/rjust/defects4j [2] 2021.
親愛的 :一種基於深度學習的新型自動程序修復方法。 https://github.com/AutomatedProgramRepair-2021/dear-auto-fix [3] Rui Abreu 、Peter Zoeteweij
和 Arjan Jc Van Gemund。 2006.軟件故障定位的相似係數評估 。在第 12 屆環太平洋國際可靠計算研討會 (PRDC) 的會議記錄中。 39–46。 https://doi.org/10.1109/PRDC.2006.18 [4] Nghi DQ Bui 、Yijun Yu 和 Lingxiao Jiang。 2021. TreeCaps :用於源代碼處理的基於樹的膠囊網絡。 AAAI 人工智能會議論文集 35,
1（2021 年 5 月） ,30-38。 https://ojs.aaai.org/index.php/AAAI/article/view/16074 _

[5] Saikat Chakraborty 、Yangruibo Ding 、Miltiadis Allamanis 和 Baishakhi Ray。
2020. CODIT :使用基於樹的神經模型進行代碼編輯。 IEEE軟件工程彙刊 (2020)。 https://doi.org/10.1109/TSE.2020.3020502 [6] Zimin Chen 、Steve James
Kommrusch 、Michele Tufano 、Louis-Noël Pouchet 、 Denys Poshyvanyk 和 Martin Monperrus。 2019. SEQUENCER :用於端到端程序修復的序列到序列學習。 IEEE 軟件工程彙刊 (2019)。 https://doi.org/10.1109/TSE.2019.2940179 [7] Kyunghyun Cho 、Bart van Merriënboer 、Caglar Gulcehre 、Dzmitry Bahdanau 、
Fethi Bougares 、Holger Schwenk 和 Yoshua Bengio。 2014. 使用 RNN 編碼器-解碼器學習短語表示以進行統計機器翻譯。

在 2014 年自然語言處理經驗方法會議 (EMNLP)會議記錄中 。計算語言學協會 ,多哈 ,卡塔爾， 1724-1734 年。 https://doi.org/10.3115/v1/D14-1179 [8] Jacob Devlin 、Ming-Wei Chang 、Kenton Lee 和 Kristina Toutanova。
2019. BERT :用於語言理解的深度雙向轉換器的預訓練 。在計算語言學協會北美分會 2019 年會議記錄中 :人類語言技術 ,第 1 卷（長文和短文） 。計算語言學協會 ,明尼蘇達州明尼阿波利斯， 4171–4186。 https://doi.org/10.18653/v1/N19-1423 [9] Claire Le Goues 、Michael Dewey-Vogt 、Stephanie Forrest 和 Westley Weimer。

2012. 自動程序修復的系統研究 :以每個 8 美元的價格修復 105 個錯誤中的 55 個 。在第 34 屆國際軟件工程會議 (ICSE 12) 的會議記錄中。 3–13。 https://doi.org/10.1109/ICSE.2012.6227211 [10] Claire Le Goues 、ThanhVu Nguyen 、Stephanie
Forrest 和 Westley Weimer。 2012.
GenProg :自動軟件修復的通用方法。 IEEE 軟件工程彙刊 38, 1（2012 年 1 月） ,54–72。 https://doi.org/10.1109/TSE.2011.104 [11] Rahul Gupta 、Soham Pal 、Aditya Kanade
和 Shirish Shevade。 2017. DeepFix :通過深度學習修復常見的 C 語言錯誤 。在第 30 屆 AAAI 人工智能會議（美國加利福尼亞州舊金山)(AAAI 17)的會議記錄中。 AAAI 出版社 ,1345–1351。

[12] Hideaki Hata 、Emad Shihab 和 Graham Neubig。 2018. 學習使用神經機器翻譯生成正確的補丁。 arXiv 預印本 arXiv:1812.07170 (2018)。

[13] Jiajun Jiang, Yingfei Xiong, Hongyu Zhang, Qing Gao, and Xiangqun Chen.
2018. 使用現有補丁和類似代碼塑造程序修復空間。
在第 27 屆 ACM SIGSOFT 軟件測試和分析國際研討會（荷蘭阿姆斯特丹）(ISSTA 2018)的會議記錄中 。美國紐約州計算機協會 ,298–309。 https://doi.org/10.1145/ 3213846.3213871

[14]姜南 ,Thibaud Lutellier ,譚林， 2021. CURE :用於自動程序修復的代碼感知神經機器翻譯 。在第 43 屆國際軟件工程會議 (ICSE 21)的會議記錄中。 1161–1173。 https://doi.org/10.1109/ICSE43902.2021.00107 _

[15] Dongsun Kim 、Jaechang Nam 、Jaewoo Song 和 Sunghun Kim。 2013.從人工編寫的補丁中學習自動補丁生成 。在第 35 屆國際軟件工程會議 (ICSE 13)的會議記錄中。 802–811。 https://doi.org/10.1109/ICSE.2013.6606626 [16] Anil Koyuncu 、Kui Liu 、Tegawendé F Bissyandé 、Dongsun Kim 、Jacques
Klein 、Martin Monperrus 和 Yves Le Traon。 2020. Fixminer :挖掘相關修復模式以進行自動程序修復 。實證軟件工程 25 (2020)， 1980–2024。 https://doi.org/10.1007/s10664-019-09780-z

[17] Xuan Bach D. Le 、David Lo 和 Claire Le Goues。 2016. 歷史驅動程序修復 。在第 23 屆 IEEE 軟件分析 、進化和再工程國際會議 (SANER 16) 的論文集中 ,卷。 1. 213–224。 https://doi.org/10.1109/SANER.2016.76 [18] Yi Li 、Shaohua Wang 和 Tien N. Nguyen。 2020. DLFix :用於自動程序修復的基於上下文
的代碼轉換學習 。在ACM/IEEE 第 42 屆軟件工程國際會議（韓國首爾)(ICSE 20)的會議記錄中。 美國紐約州計算機協會， 602–614。 https://doi.org/10.1145/3377811.3380345 [19] Kui Liu 、Anil Koyuncu 、Dongsun Kim 和 Tegawendé F. Bissyandè。 2019.

AVATAR :使用靜態分析違規修復模式修復語義錯誤。
在第 26 屆 IEEE 軟件分析 、演化和再工程國際會議論文集 (SANER 19) 中。 1-12。 https://doi.org/10.1109/SANER。 2019.8667970

[20] Kui Liu 、Anil Koyuncu 、Dongsun Kim 和 Tegawendé F. Bissyandé。 2019. TBar :重新審視基於模板的自動化程序修復 。在第 28 屆 ACM SIGSOFT 軟件測試與分析國際研討會（中國北京)(ISSTA 19) 的會議記錄中 。美國紐約州計算機協會， 31-42。 https://doi.org/10.1145/3293882.3330577 [21] Fan Long 、Peter Amidon 和 Martin Rinard。 2017.用於補丁生成的代碼轉換的自動推理 。在第 11 屆軟件工程基
礎聯席會議（德國帕德博恩)(ESEC/FSE 17) 的記錄中。

美國紐約州計算機協會 ,727–739。 https://doi.org/10.1145/3106237.3106253 [22]範龍和馬丁里納德。 2016. 通過學習正確代
碼自動生成補丁 。在第 43 屆 ACM SIGPLAN-SIGACT 編程語言原理年度研討會（美國佛羅里達州聖彼得堡)的會議記錄中

（POPL 16） 。美國紐約州計算機協會 ,298–312。 https://doi.org/10.1145/2837614.2837617
[23] Thibaud Lutellier 、Hung Viet Pham 、Lawrence Pang 、Yitong Li 、Moshi Wei 和Lin Tan。 2020. CoCoNuT :使用 Ensemble 結合上下文感知神經翻譯模型進行程序修復 。在第 29 屆 ACM SIGSOFT軟件測試與分析國際研討會論文集（虛擬活動 ,美國）

（ISSTA 20） 。美國紐約州計算機協會 ,101–114。 https://doi.org/10.1145/3395363.3397369
[24]馬蒂亞斯·馬丁內斯和馬丁·蒙佩魯斯。 2016. ASTOR :Java 程序修復庫（演示） 。在第 25 屆軟件測試與分析國際研討會（德國薩爾布呂肯)(ISSTA 16) 的會議記錄中 。美國紐約州計算機協會 ,441–444。 https://doi.org/10.1145/2931037.2948705 _

[25] Sergey Mechtaev 、Jooyong Yi 和 Abhik Roychoudhury。 2016. Angelix :通過符號分析的可擴展多行程序補丁合成 。在第 38 屆國際軟件工程會議（德克薩斯州奧斯汀）(ICSE 16)的會議記錄中。
美國紐約州紐約市計算機協會 ,691–701。 https://doi.org/10.1145/2884781.2884807 [26] Hoan Anh Nguyen 、Tien N.
Nguyen 、Danny Dig 、Son Nguyen 、Hieu Tran 和Michael Hilton。 2019. 基於圖的野外細粒度語義代碼更改模式挖掘 。在第 41 屆國際軟件工程會議(ICSE 19) 的會議記錄中。 IEEE 出版社 ,819–830。 https://doi.org/10.1109/ICSE.2019.00089 _

[27] Hoang Duong Thien Nguyen 、Dawei Qi 、Abhik Roychoudhury 和 Satish Chan dra。 2013. SemFix :通過語義分析進行程序修復 。在第 35 屆國際軟件工程會議 (ICSE 13) 的會議記錄中。 772–781。 https://doi.org/10.1109/ICSE.2013.6606623

[28] Tung Thanh Nguyen 、Hoan Anh Nguyen 、Nam H. Pham 、Jafar Al-Kofahi 和Tien N. Nguyen。 2010. 面向對象程序中的重複錯誤修復 。在第 32 屆 ACM/IEEE 軟件工程國際會議論文集中 -第 1 卷（南非開普敦)(ICSE 10) 。計算機機械協會 ,美國紐約州紐約市 ,315–324。 https://doi.org/10.1145/1806799.1806847 [29] Jeffrey Pennington 、Richard Socher 和 Christopher D. Manning。 2014. GloVe :用於詞表示的全局向量 。在自然語言處理 (EMNLP) 的經驗方法中。 1532–1543。 http://www.aclweb.org/anthology/D14-1162 _

[30] Yuhua Qi, Xiaoguang Mao, Yan Lei, Ziying Dai, and Chengsong Wang. 2014.
自動程序修復隨機搜索的強度 。在第 36 屆國際軟件工程會議論文集（印度海得拉巴）
（ICSE 14） 。美國紐約州計算機協會 ,254–265。 https://doi.org/10.1145/2568225.2568254

[31] 亞歷克·拉德福德 、卡爾西克·納拉西姆漢 、蒂姆·薩利曼斯和伊利亞·薩茨克維爾。 2018. 我通過生成預訓練證明語言理解。 (2018)。
[32] Baishakhi Ray 和 Miryung Kim。 2012. 分叉項目中跨系統移植的案例研究 。在 ACM SIGSOFT 第 20 屆軟件工程基礎國際研討會（北卡羅來納州卡里)(FSE 12) 的會議記錄中。

美國紐約州紐約市計算機協會 ,第 53 條 ,11 頁。 https://doi.org/10.1145/2393596.2393659

[33] Ripon K Saha 、Yingjun Lyu 、Hiroaki Yoshida 和 Mukul R Prasad。 2017. Elixir :有效的面向對象程序修復 。在第 32 屆 IEEE/ACM國際自動化軟件工程會議 (ASE 17) 的會議記錄中。 648–659。 https://doi.org/10.1109/ASE.2017.8115675

[34] Seemanta Saha 、Ripon K. Saha 和 Mukul R. Prasad。 2019. 利用進化進行多塊程序修復。在第 41 屆國際軟件工程會議 (ICSE 19) 的會議記錄中。 IEEE 出版社 ,13–24。 https://doi.org/10.1109/ICSE.2019.00020 _

[35] Abigail See 、Peter J Liu 和 Christopher D Manning。 2017. 進入正題 :使用指針生成器網絡進行總結。 arXiv 預印本 arXiv:1704.04368 (2017)。

[36] Kai Sheng Tai 、Richard Socher 和 Christopher D. Manning。 2015.樹結構長短期記憶網絡作品的改進語義表示 。在計算語言學協會第 53 屆年會和第 7 屆自然語言處理國際聯合會議論文集（第 1 卷 :長篇論文)中 。中國北京計算語言學協會 ,1556-1566 年。 https://doi.org/10.3115/v1/P15-1150

[37] Shin Hwei Tan、Zhen Dong、Xiang Gao 和 Abhik Roychoudhury。 2018. 修復Android 應用程序中的崩潰。在第 40 屆國際軟件工程會議（瑞典哥德堡)(ICSE 18) 的會議記錄中。計算機機械協會，美國紐約州紐約市，187–198。 https://doi.org/10.1145/3180155。 3180243

[38] Michele Tufano、Jevgenija Pantiuchina、Cody Watson、Gabriele Bavota 和 Denys Poshyvanyk。 2019. 關於通過神經機器翻譯學習有意義的代碼更改。在第 41 屆 IEEE/ ACM 國際軟件工程會議 (ICSE 19) 的會議記錄中。 25–36。 https://doi.org/10.1109/ ICSE.2019.00021 [39] Michele Tufano、Cody Watson、Gabriele Bavota、Massimiliano Di Penta、Martin White 和 Denys Poshyvanyk。 2018. 對通過神經機器翻譯在野外學習錯誤修復補丁的實證研究。在第 33 屆 ACM/IEEE 自動化軟件工程國際會議（法國蒙彼利埃） (ASE 18)的會議記錄中。美國紐約州計算機協會，832–837。 https://doi.org/ 10.1145/3238147.3240732 [40] Ming Wen, Junjie Chen, Rongxin Wu, Dan Hao, and Shing-Chi Cheung. 2018.

用於更好地自動修復程序的上下文感知補丁生成。在第 40 屆國際軟件工程會議（瑞典哥德堡)(ICSE 18) 的會議記錄中。美國紐約州計算機協會，1-11。 https://doi.org/ 10.1145/3180155.3180233

[41] Martin White、Michele Tufano、Matias Martinez、Martin Monperrus 和 Denys Poshyvanyk。 2019. 通過深度學習代碼相似性對程序修復成分進行排序和轉換。在第 26 屆 IEEE軟件分析、演化和再工程國際會議論文集 (SANER 19) 中。 479–490。 https:// doi.org/10.1109/SANER.2019.8668043

[42] Martin White、Michele Tufano、Christopher Vendome 和 Denys Poshyvanyk。 2016. 用於代碼克隆檢測的深度學習代碼片段。在第 31 屆 IEEE/ACM 國際自動化軟件工程 (ASE)會議記錄中。美國計算機學會，87–98。

[43]齊鑫和史蒂文·賴斯。 2017. 利用語法相關代碼進行自動程序修復。在第 32 屆 IEEE/ACM 國際自動化軟件工程會議（美國伊利諾伊州厄巴納-香檳)(ASE 17) 的會議記錄中。 IEEE出版社，660–670。

[44]元元和沃爾夫岡·班扎夫。 2020. ARJA：通過多目標遺傳編程自動修復 Java程序。 IEEE 軟件工程彙刊 (TSE) 46, 10 (2020), 1040–1067。 https://doi.org/10.1109/TSE.2018。 2874648

[45] J. Zhu、T. Park、P. Isola 和 AA Efros。 2017.使用循環一致的對抗網絡進行不成對的圖像到圖像的轉換。 2017年IEEE國際計算機視覺會議（ICCV）。 2242–2251。 https://doi.org/ 10.1109/ICCV。 2017.244