ML4Debugging: Are we there yet?

Qianfu Tang (Edward)
University of Illinois, Urbana-Champaign

ABSTRACT

...

ACM Reference Format:
Qianfu Tang (Edward). 2021. ML4Debugging: Are we there yet?. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Program debugging involves two specific tasks: i) bug localization, which identifies and pinpoints the bug in the source code, and ii) program repair. The traditional bug localization techniques, which involve logging, breakpoints, profiling and tracing, could be very time-consuming and tedious especially when it comes to integration test. The techniques has several limitations that hinder the efficiency of debugging: 1) Re-executing the test in order to pinpoint the location of the bug requires extra budget and time; 2) these debugging tools are insensitive to the contextual setting after isolating the source code; 3) their effectiveness is heavily dependent on the quality of test suites. This research aims to conduct an empirical study and quantify the limitation of state-of-the-art M4Debugging techniques and tools, with a focus on the application of Deep Learning and Reinforcement Learning. We will propose a new technique for debugging that overcomes the challenges. The main difference between Deep Learning and Reinforcement Learning is that deep learning is learning from a training set and then applying that learning to a new data set, while reinforcement learning is dynamically learning by adjusting actions based in continuous feedback to maximize a reward. There are connections between these two methods so it is important to conduct studies on tools that use both learning methods.

2 MOTIVATING EXAMPLE

We have collected several tools that focuses on the application of Deep Learning on debugging. We focus upon [17], in which the artifact called DeepFL is made publicly available in [2]. Our goal is to analyze how the tool performs compared against other tools that are introduced in the citation of [17] or those that cited the paper. The purpose of this empirical study is to discover the weakness of the current Deep Learning methods in order to develop a new approach that can eliminate these weakness. We have collected the following tools that are publicly available for testing.

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

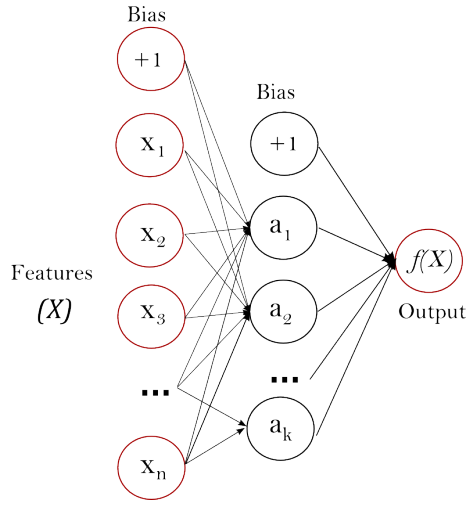

Figure 1: One-hidden-layer MLP with m inputs and 1 output

One of the papers [13] that is cited in [17] proposed a tool called TraceNN [9] that uses word embedding, RNN layers to conduct unsupervised learning in order to represent the artifact semantics.

The tool FLUCCS mentioned in [18] extends SBFL with code and change metrics that have been studied in the context of defect prediction, and will also be tested in this research.

3 RELATED WORK

The Defects4j dataset is publically available on GitHub as mentioned in [16], as posed by R.Just et al in the paper , the techniques of bug reproduction and isolation are explained.
In [14], a multi-model approach of bug localization has been introduced. A model called Network-clustered Multi-modal Bug Localization (NetML) is being used.

The main idea from the DeepFL [17] tool is to use deep learning to predict potential fault via incorporating various dimensions of fault diagnosis information. The main architecture is called Multi-Layer Perceptron(MLP). It is a supervised learning algorithm that learns a function on a trained dataset. Figure 1 is an example of one-hidden-layer MLP from [7].

Recurrent Neural Network(RNN) is very widely used in Natural Language Processing. The idea is that the output of the RNN is dependent on the prior element of the sequence. Figure 2 shows the difference between RNN and Feedforward Neural Network[10].

The idea behind the word embedding technique is to learn the representation of each natural language word into a high-dimensional space. Thus the resulting vectors can be used to encode the semantic and syntactic relationship between words. The tool used in [9] is called GloVe and is publicly available for download at [6].

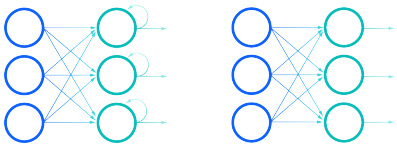

Figure 2: Recurrent Neural Network vs. Feedforward Neural Network

The architecture behind the RNN-based embedding in DeepCS [12] is a model that takes a word vector $s = w_1, w_2, ..., w_t$ and define each hidden state $\boldsymbol{h_t}$ as

$$\boldsymbol{h_t} = \tanh(\boldsymbol{W}\,[\boldsymbol{h_{t-1}}, \boldsymbol{w_t}])\forall t = 1, 2, 3, ..., T,$$

Where $\boldsymbol{W}$ is a matrix that contains trainable parameters from RNN.

4 EVALUATION

RQ1: *Test Re-execution.* Description of RQ
Does the tool require test re-execution in order to pinpoint the bug location? If so, how much does it cost? Is there any way to improve based upon that?
RQ2: *Dataset.* Description of RQ
What are the nature of defects in the dataset?
RQ3: *Test Suite.* Description of RQ
How do the tools perform with a test suite with lower quality?

4.1 Experimental Setup

We will use both LLVM and GCC to test the performance of our debugging tool. We will use Clang Libtooling Library [1] and PyTorch [8] to collect compiler coverage information.
Dataset: Defect4j Includes 835 open source projects.

There features include, as quoted in [3] , that the bugs being fixed in a single commit, irrelevant changes pruned out in the commit, being fixed by modifying the source code and a triggering test.

4.2 DeepFL

The tool is made available at [12] and the next step is to run it through Defects4j.

4.3 FLUCCS

The tool mentioned in [18] is made available at [5], this is a model that uses different code metrics on Spectrum-Based Fault Localization that also can be evaluated with Defects4j

4.4 Savant

Savant is a tool mentioned in the paper [11] that employs learning-to-rank strategy and uses likely invariant diffs and suspiciousness scores as features. The artifact is not made available but we have emailed the authors hoping for their response.

4.5 RQ1: Test Re-execution

In this case we need to determine if we need the test to be re-executed on the tools that we are working with in order to pinpoint the bugs.

Based on the artifact page of DeepFL, the deep learning architecture of fault localization implements the following steps: the spectrum-based fault localization gives suspicious values for the fault diagnosis from execution, and the code-specific gives diagnosis from code complexity, and the combination of these features gives a direct rank of suspiciousness, without test re-execution. This is benefitted from the spectrum-based suspiciousness that directly takes the test cases as input, and no more execution is needed after obtaining the spectrum data, in any of the categories (e.g. Ochiai, Jaccard, etc)

The result re-execution of all the tools the DeepFL paper compared with is attached here:

Subjects	Techniques	Top-1	Top-3	Top-5	MFR	MAR
	Ochiai	6	14	15	9	9.51
	Me-Ochiai	7	15	17	12.68	13.31
	MULTRIC	7	15	16	8.08	8.85
Chart	FLUCCS	15	19	20	3.68	4.3
	TraPT	10	15	16	5.04	5.7
	MLP_DFL(2)	12	20	20	3.52	4.11
	Ochiai	24	44	50	4.63	5.01
	Me-Ochiai	32	51	56	2.84	3.15
	MULTRIC	23	42	49	5.53	5.85
Lang	FLUCCS	40	53	55	3.4	3.63
	TraPT	42	55	58	2.89	3.18
	MLP_DFL(2)	46	54	59	2.15	2.53
	Ochiai	23	52	62	9.73	11.72
	Me-Ochiai	20	51	71	7.74	9.31
	MULTRIC	21	50	58	10.44	12.69
Math	FLUCCS	48	77	83	4.64	5.66
	TraPT	34	63	77	5.2	6.84
	MLP_DFL(2)	63	85	91	3.72	4.84
	Ochiai	6	11	13	15.96	18.87
	Me-Ochiai	7	12	15	12.35	14.82
	MULTRIC	6	13	13	24.58	27.33
Time	FLUCCS	8	15	18	9	11.9
	TraPT	7	13	16	11.85	13.19
	MLP_DFL(2)	13	17	17	11.92	12.62
	Ochiai	7	14	18	20.22	24.77
	Me-Ochiai	9	15	21	23.47	28.38
	MULTRIC	6	12	18	21.33	26.37
Mockito	FLUCCS	7	19	22	14.78	18.63
	TraPT	12	20	22	22.67	26.37
	MLP_DFL(2)	12	19	22	11.75	13.78
	Ochiai	14	30	38	90.28	102.28
	Me-Ochiai	19	47	64	23.06	27.47
	MULTRIC	17	31	41	87.34	100.85
Closure	FLUCCS	42	66	77	36.61	48.61
	TraPT	51	83	92	14.11	19.34
	MLP_DFL(2)	67	87	96	9.2	12.14
	Ochiai	80	165	196	37.74	43.09
	Me-Ochiai	94	191	244	14.28	16.93
	MULTRIC	80	163	195	37.71	43.68
Overall	FLUCCS	160	249	275	16.53	21.53
	TraPT	156	249	281	9.94	12.7
	MLP_DFL(2)	213	282	305	6.63	8.27

It is worth noting that FLUCCS, which uses spectrum-based fault localization as their feature, has very promising performance. The learning strategies for FLUCCS [18] include genetic programming and support vector machine, while DeepFL uses deep learning. The outcome of the data in the table seems to support that as the first deep learning based fault localization model, DeepFL has a dominating performance across other models, and thus our new

proposed technique can also use deep learning (MLP, RNN, etc) as a basis for improvement.

4.6 RQ2: Dataset

The dataset defects4j includes 835 different bugs from different identifiers, in which the DeepFL tool, along with other tools, was tested on Lang, Math, Closure, Mockito, Chart, Time as in [2].

From what I have ran the tool from version control history of the buggy files, for each bug localized by DeepFL, the runtime performance for bugs in each identifier is as follows:

	Chart	Lang	Time	Math	Mockito	Closure
mean(s)	5.55	5.18	5.12	4	5.71	(100+)
std.dv(s)	2.77	5.04	1.02	4.16	2.72	
file number	26	64	27	106	38	136

In the meanwhile, as identifying the root cause of each bug in all categories, most of bugs in the 6 identifiers (Lang, Math, Mockito, Time, Closure, Chart) are caused by the AssertionError in 1-3 test cases, while in Closure there are several buggy files that fails up to 7 generated test cases in Defects4j. From the data, it seems like we have a relatively steady runtime performance in all the identifiers, excluding Closure. It is possible that the discrepancy of Closure being signicificantly more costly than others is coming from the runtime memory.

4.7 Test Suite

Next we explore whether the performance of the debugging tool is affected by the quality of test suites. This might not be such a case when we have performed our test on Defects4j, considering Defects4j has implemented a component for test generation that is based on EvoSuite[4]. However, there are components in the original DeepFL architecture that takes into account of the quality of test suites, and we can analyse the impact of using a lower-quality test suite mathematically.

DeepFL uses a combination of spectrum-based fault localization (SBFL) and mutation-based fault localization (MBFL). I will use the Tarantula coefficient [15] as an example of SBFL. For a coverage entity e:

$$suspiciousness(e) = \frac{\frac{failed(e)}{totalfailed}}{\frac{passed(e)}{totalpassed} + \frac{failed(e)}{totalfailed}}$$

An example of how Tarantula coefficient works in a given test suite is as follows[15], note that the coverage entity in this case is the individual line of the program.

	mid() { int x,y,z,m;	Test Case 3,3,5	1,2,3	3,2,1	5,5,5	5,3,4	2,1,3	suspiciousness	rank
1:	read("Enter 3 numbers:",x,y,z);	●	●	●	●	●	●	0.5	7
2:	m = z;	●	●	●	●	●	●	0.5	7
3:	if (y<z)	●	●	●	●	●	●	0.5	7
4:	if (x<y)	●	●			●	●	0.63	3
5:	m = y;		●					0.0	13
6:	else if (x<z)	●				●	●	0.71	2
7:	m = y; // *** bug ***	●					●	0.83	1
8:	else			●	●			0.0	13
9:	if (x>y)			●	●			0.0	13
10:	m = y;			●				0.0	13
11:	else if (x>z)				●			0.0	13
12:	m = x;							0.0	13
13:	print("Middle number is:",m);	●	●	●	●	●	●	0.5	7
	} Pass/Fail Status	P	P	P	P	P	F		

Assume we have a low quality test suite in which there is no passed tests in N test cases for a given element $\hat{e}$. We get:

$$suspiciousness(\hat{e}) = \frac{N/N}{0 + N/N} = 1$$

In this case, before we apply mutation testing, the given elements' suspiciousness will be maxed out at 1, and if all elements in the test suite behave the same way, then the suspiciousness of the overall ranking will be all maxed out, thus SBFL will be vulnerable to low-quality test suites.

Next we focus on MBFL. With a mutant m, the formula is given by

$$suspiciousness(m) = \frac{|T_f^{(m)}(e)|}{|T_f^{(m)}(e)| + |T_p^{(m)}(e)|}$$

on each element e. With a mutant on the original program, it is easy for programmers to modify the code with respect to the test suites, and on an element e, $suspiciousness(m) \leq suspiciousness(e)$. As multiple SBFL formulae can be utilized, the addition of mutation-based testing could make the suspiciousness score less prone to the quality of test suite, as programmers can perform mutations on multiple lines of program in order to pinpoint the place where the mutation suspiciousness score is slightly lower than the maximum, even in the worst of test suites. Therefore the combination of both SBFL and MBFL is advantageous in mitigating the effect of test suites.

5 CONCLUSION

We can see that the deep learning model that was implemented in DeepFL actually saves the effort of re-executing the test in order to pinpoint the bug location. We have collected elements that could effectively ameliorate the performance of a debugging tool that is less prone to the three challenges of state-of-art techniques. While DeepFL does provide a good example of using MLP and RNN to perform deep learning, it is worth considering to implement architectures like Variational Auto-Encoders and Generative Adversarial Networks to compare their performance.

REFERENCES

[1] [n.d.]. Clang 12 documentation LIBTOOLING. https://clang.llvm.org/docs/LibTooling.html. Accessed: 2020-02-27.

[2] [n.d.]. DeepFaultLocalization GitHub. https://github.com/DeepFL/DeepFaultLocalization. Accessed: 2020-03-25.
[3] [n.d.]. Defects4j. https://defects4j.org. Accessed: 2020-03-25.
[4] [n.d.]. EvoSuite. https://www.evosuite.org.
[5] [n.d.]. FLUCCS tool. https://bitbucket.org/teamcoinse/fluccs/src/master/. Accessed: 2020-03-25.
[6] [n.d.]. GloVe: Global Vectors for Word Representation. https://nlp.stanford.edu/projects/glove/. Accessed: 2020-03-25.
[7] [n.d.]. Neural network models (supervised). https://scikit-learn.org/stable/modules/neural_networks_supervised.html#multi-layer-perceptron. Accessed: 2020-03-25.
[8] [n.d.]. PyTorch. https://pytorch.org. Accessed: 2020-02-27.
[9] [n.d.]. TraceNN GitHub. https://github.com/jin-guo/TraceNN. Accessed: 2020-03-25.
[10] [n.d.]. What are recurrent neural networks? https://www.ibm.com/cloud/learn/recurrent-neural-networks. Accessed: 2020-03-25.
[11] Tien-Duy B. Le, David Lo, Claire Le Goues, and Lars Grunske. 2016. A Learning-to-Rank Based Fault Localization Approach Using Likely Invariants. In *Proceedings of the 25th International Symposium on Software Testing and Analysis* (Saarbrücken, Germany) *(ISSTA 2016)*. Association for Computing Machinery, New York, NY, USA, 177–188. https://doi.org/10.1145/2931037.2931049
[12] X. Gu, H. Zhang, and S. Kim. 2018. Deep Code Search. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. 933–944. https://doi.org/10.1145/3180155.3180167
[13] Jin Guo, Jinghui Cheng, and Jane Cleland-Huang. 2017. Semantically Enhanced Software Traceability Using Deep Learning Techniques. In *Proceedings of the 39th International Conference on Software Engineering* (Buenos Aires, Argentina) *(ICSE '17)*. IEEE Press, 3–14. https://doi.org/10.1109/ICSE.2017.9
[14] T. Hoang, R. J. Oentaryo, T. B. Le, and D. Lo. 2019. Network-Clustered Multi-Modal Bug Localization. *IEEE Transactions on Software Engineering* 45, 10 (2019), 1002–1023. https://doi.org/10.1109/TSE.2018.2810892
[15] James A. Jones and Mary Jean Harrold. 2005. Empirical Evaluation of the Tarantula Automatic Fault-Localization Technique. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering* (Long Beach, CA, USA) *(ASE '05)*. Association for Computing Machinery, New York, NY, USA, 273–282. https://doi.org/10.1145/1101908.1101949
[16] 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* (San Jose, CA, USA) *(ISSTA 2014)*. Association for Computing Machinery, New York, NY, USA, 437–440. https://doi.org/10.1145/2610384.2628055
[17] Xia Li, Wei Li, Yuqun Zhang, and Lingming Zhang. 2019. DeepFL: Integrating Multiple Fault Diagnosis Dimensions for Deep Fault Localization. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Beijing, China) *(ISSTA 2019)*. Association for Computing Machinery, New York, NY, USA, 169–180. https://doi.org/10.1145/3293882.3330574
[18] Jeongju Sohn and Shin Yoo. 2017. FLUCCS: Using Code and Change Metrics to Improve Fault Localization. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Santa Barbara, CA, USA) *(ISSTA 2017)*. Association for Computing Machinery, New York, NY, USA, 273–283. https://doi.org/10.1145/3092703.3092717