An Empirical Study on Numerical Bugs in Deep Learning Programs

Gan Wang
College of Intelligence and Computing, Tianjin University
Tianjin, China
wg_98@tju.edu.cn

Zan Wang
College of Intelligence and Computing, Tianjin University
Tianjin, China
wangzan@tju.edu.cn

Junjie Chen*
College of Intelligence and Computing, Tianjin University
Tianjin, China
junjiechen@tju.edu.cn

Xiang Chen
School of Information Science and Technology, Nantong University
Nantong, Jiangsu, China
xchencs@ntu.edu.cn

Ming Yan
College of Intelligence and Computing, Tianjin University
Tianjin, China
yanming@tju.edu.cn

ABSTRACT

The task of a deep learning (DL) program is to train a model with high precision and apply it to different scenarios. A DL program often involves massive numerical calculations. Therefore, the robustness and stability of the numerical calculations are dominant in the quality of DL programs. Indeed, numerical bugs are common in DL programs, producing NaN (Not-a-Number) and INF (Infinite). A numerical bug may render the DL models inaccurate, causing the DL applications unusable. In this work, we conduct the first empirical study on numerical bugs in DL programs by analyzing the programs implemented on the top of two popular DL libraries (i.e., TensorFlow and PyTorch). Specifically, We collect a dataset of 400 numerical bugs in DL programs. Then, we classify these numerical bugs into nine categories based on their root causes and summarize two findings. Finally, we provide the implications of our study on detecting numerical bugs in DL programs.

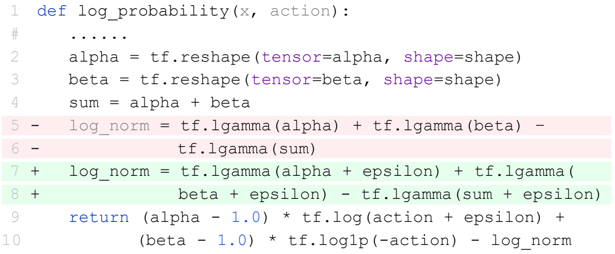
CCS CONCEPTS

• **Software and its engineering** → **Software testing and debugging**; *Empirical software validation*; • **Computing methodologies** → **Artificial intelligence**.

KEYWORDS

Deep Learning, Numerical Bug, Empirical Study

ACM Reference Format:
Gan Wang, Zan Wang, Junjie Chen, Xiang Chen, and Ming Yan. 2022. An Empirical Study on Numerical Bugs in Deep Learning Programs. In *37th IEEE/ACM International Conference on Automated Software Engineering (ASE '22), October 10–14, 2022, Rochester, MI, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3551349.3559561



```
def log_probability(x, action):
    ......
    alpha = tf.reshape(tensor=alpha, shape=shape)
    beta = tf.reshape(tensor=beta, shape=shape)
    sum = alpha + beta
-   log_norm = tf.lgamma(alpha) + tf.lgamma(beta) -
-              tf.lgamma(sum)
+   log_norm = tf.lgamma(alpha + epsilon) + tf.lgamma(
+              beta + epsilon) - tf.lgamma(sum + epsilon)
    return (alpha - 1.0) * tf.log(action + epsilon) +
           (beta - 1.0) * tf.log1p(-action) - log_norm
```

Figure 1: A numerical bug caused by `tf.lgamma()`

1 INTRODUCTION

In recent years, with the increasing development of deep learning (DL) in various areas (e.g., autonomous driving [7]), we are entering an era of Artificial Intelligence. The life cycle of a DL application can be divided into three phases: programming phase, training phase, and deployment phase. Specifically, developers need to design and implement the neural network into DL programs, and the DL model can be obtained by conducting the training procedure on the training set. Thus, the bugs that appear in the programming phase can seriously threaten the quality and safety of DL applications. To date, substantial recent studies [11, 19–21] have illustrated that testing DL programs are critical.

Among the common defects in DL programs, numerical bugs are one of the most prominent categories, as DL programs have a heavy presence of numerical computation [18, 19]. Numerical bugs come out in the form of NaN or INF, where NaN denotes that the value is Not a Number and INF denotes an infinite number. For example, NaN or INF can occur when divided-by-zero happens during computation. Once a NaN or INF appears during computation, the wrong values will affect the final output through value propagation. Due to the randomness and complexity of DL, numerical bugs in DL programs may not appear in the early stage of the training procedure but are exposed after lengthy training. Thus, developers can spend a lot of time detecting and debugging numerical bugs in DL programs, which obviously slows down the development cycle of DL applications.

[*] Junjie Chen is the corresponding author.

ASE '22, October 10–14, 2022, Rochester, MI, USA
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9475-8/22/10…$15.00
https://doi.org/10.1145/3551349.3559561

Figure 1 presents a simplified case of a numerical bug in a TensorFlow program from our study, in which a NAN value appears in the return value of a function `log_probability()` at line 9. In this code snippet, the NAN value first appears after the operation `tf.lgamma()` in line 5. Then, it is continuously propagated during the training process, leading to an invalid calculation result. The above bug can be fixed by adding a small epsilon value to the operands of `tf.lgamma()`.

In this work, we conduct the first empirical study to facilitate the understanding of numerical bugs in DL programs. In particular, we analyzed DL programs based on two mainstream DL libraries (i.e., TensorFlow and PyTorch). Specifically, we collected issues or posts related to numerical bugs from *GitHub*, *Stack Overflow*, and *PyTorch Forums* and built a dataset consisting of 400 issues/posts through our manual inspection. We divided the root causes of numerical bugs into *code relevant* (bugs caused by developers' incorrect implementation in programming) and *code irrelevant* (bugs caused by incorrect configurations or other external factors), and further categorized them into nine categories. We then analyzed their distributions and summarized the corresponding fixed patterns. Our analysis presents two findings to provide a better understanding of the numerical bugs in DL programs, e.g., the most dominant kind of numerical bug is caused by *Invalid Range (IR)* in mathematical operations(e.g., an `exp()` API call with a parameter greater than 88), while fewer bugs are caused by *Code Logic Error*. To motivate future research on numerical bugs in DL programs, we then provided a series of implications in Section 3.

In summary, the contributions of this work are as follows:

- We conduct the first empirical study on numerical bugs based on 400 bugs in real-world DL applications on TensorFlow and PyTorch.
- We provide a classification of root causes of numerical bugs in DL programs and some fix patterns.
- We release the collected dataset containing 400 real-world numerical bugs [2].

2 EMPIRICAL STUDY

We conduct an empirical study to comprehensively understand numerical bugs in DL programs. In the study, we aim to categorize the root causes of these bugs, analyze their distribution, and summarize our findings. Moreover, we discuss the challenges of exposing these bugs.

2.1 Data Collection

Our study considered numerical bugs in DL programs based on the two most popular DL libraries, i.e., TensorFlow [6] and PyTorch [14], which are widely used in the development of DL programs while differing in computation manners. TensorFlow adopts static computation graphs, in which the neuron network is built before the training process, while PyTorch computes the neuron network dynamically in training time. We collected potential bugs from two data sources, i.e., GitHub issues and forum posts.

For GitHub issues, we first collected all the repositories containing the label *TensorFlow* or *PyTorch*. Then, we collected all the closed issues with at least one reply and contain the keywords *NaN* or *INF* from these repositories. In this way, we collected 580 TensorFlow issues and 483 PyTorch issues as potential numerical bugs.

Regarding forum posts, TensorFlow and PyTorch have their own commonly-used forums respectively. The forum of TensorFlow is *Stack Overflow* [4], which is officially recommended by TensorFlow developers [5], while that of PyTorch is *PyTorch Forums* [3], which is also the official forum of PyTorch. From *Stack Overflow* and *PyTorch Forums*, we collected all the posts that contain the keywords *NaN* or *INF* and have an accepted answer. From *Stack Overflow*, the posts should be tagged with *TensorFlow*. In this way, we collected 323 TensorFlow posts and 400 PyTorch posts as potential numerical bugs.

Since not all of these collected issues and posts are really numerical bugs, we further conducted manual analysis by two authors. One author is responsible to analyze TensorFlow issues and posts, while another author is responsible to analyze PyTorch issues and posts. Before independently labeling the data, the two authors jointly investigated a subset of issues/posts from the entire raw dataset and determined the initial set of categories. They randomly selected 100 issues/posts and conducted manual analysis together. If the issue or post is not a numerical bug, it is discarded; otherwise, the authors further analyze the root cause and fix pattern of the numerical bug.

After obtaining the initial set of categories, they start to classify the bugs independently. Please note that if one issue or post is found to not belong to any initial category, the two authors will discuss and determine whether to add a new category. Once obtaining 100 valid numerical bugs for each library under each data source, this step is terminated. Then, they exchanged and checked their analysis results with each other for the obtained valid numerical bugs in order to guarantee the accuracy of manual analysis. Please note that if there is a disagreement between them for a valid numerical bug, the third author is invited for discussion. If they still cannot reach an agreement after discussion, this numerical bug is discarded. In this step, seven issues and two posts are required for discussion, and no one is discarded after tripartite discussion. Finally, we had 400 valid numerical bugs in our study, including 100 TensorFlow issues, 100 TensorFlow posts, 100 PyTorch issues, and 100 PyTorch posts.

2.2 Root Causes of Numerical Bugs

To better understand the types of numerical bugs in DL programs, we classified their root causes into nine categories of two broad categories and summarized the corresponding fix patterns in each category. Table 1 presents the nine categories of root causes and their distribution in detail. In this table, TI, TP, PI, PP are short for TensorFlow Issues, TensorFlow Posts, PyTorch Issues, and PyTorch Posts respectively. The last column calculates the proportion of each category of bugs to the corresponding broad category. According to whether the numerical bugs are caused by developers' implementation in programming or not, we first categorized their root causes into two broad categories, i.e., *code relevant* bugs and *code irrelevant* bugs. Note that there are four bugs belonging to *Other* in Table 1 since they were fixed by directly filtering NaN without pointing out real root causes in the issue reports.

2.2.1 Code Relevant Bugs. Code relevance bugs are caused by developers' incorrect implementation in programming, which is more

Table 1: The distribution of root causes in numerical bugs

Type	Root Cause	# TI	# TP	# PI	# PP	# Total	Proportion(%)
Code Relevant	IR	21	29	22	21	93	50.82%
	APIM	8	8	7	12	35	19.13%
	IDP	7	4	7	9	27	14.75%
	IFPT	6	4	4	3	17	9.29%
	CLE	1	2	3	5	11	6.01%
Code Irrelevant	GE	20	27	28	15	90	42.25%
	RE	15	3	10	20	48	22.54%
	BI	12	9	8	10	39	18.31%
	IMC	7	14	10	5	36	16.90%
Other		3	0	1	0	4	-

Table 2: Invalid Range numerical bugs

Math Function	Invalid Range
$\mathrm{div}(y, x)$	$x = 0$
$\exp(x)$	$x > 88$
$\mathrm{expm1}(x)$	$x > 88$
$\mathrm{log1p}(x)$	$x + 1 <= 0$
$\log(x)$	$x <= 0$
$\mathrm{sqrt}(x)$	$x <= 0$
$\mathrm{acos}(x)$	$x = 1 \vee x = -1$
$\mathrm{lgamma}(x)$	$x \in \{0, -1, -2, ..., -inf\}$

```
  # ......
  # define x,y as input and output placeholder
  # define w,b as model weights
  out = x * w + b
- y_pred = tf.reduce_sum(out, axis=1, name='out')
+ y_pred = tf.reduce_sum(out, axis=1, name='out',keep_dims=True)
  sub_res = y_pred - y
  square_res = tf.square(sub_res)
  # Loss as sum(error^2)
- loss = tf.reduce_sum(square_res, name='loss')
+ loss = tf.reduce_sum(square_res, name='loss',keep_dims=True)

  # ......
```

Figure 2: APIM bug reported by a TensorFlow post

complicated and requires modifying the source code to fix. In our study, 45.75% of numerical bugs belong to this broad category. The root causes of the bugs in this broad category are further divided into five categories, including API Misuse, Code Logic Error, Incorrect Floating Point Type, Incorrect Data Preprocessing, and Invalid Range.

Invalid Range (IR). This category of bugs arises when invalid values are used in mathematical functions. For example, zero is invalid for division and non-positive values are invalid for log function. Based on our study, we summarized the mathematical functions that often trigger IR bugs, and the corresponding invalid ranges in Table 2. In particular, some valid values for mathematical functions may still lead to numerical bugs. For example, zero is a valid value for the square root function but if we try to calculate its derivative at zero, NaN is induced. The fix pattern of IR bugs is to *add an extremely small value* to the variables used in mathematical functions or *clip the values into valid ranges*.

API Misuse (APIM). This category of bugs arises as developers do not fully understand the used APIs, and hence mess up the order of parameters or miss parameters, etc. As shown in Fig. 2, due to missing the parameter `keep_dims=True` in the API `tf.reduce_sum`, the produced shape of `y_pred` is different from that of `y`, causing the broadcast mechanism in NumPy is triggered and then the loss is increased sharply. With the abnormal loss accumulating during the process of back-propagation, the program eventually outputs NaN. Among the 35 APIM bugs in our study, 16 bugs were fixed by *replacing the wrong APIs with the correct ones* and 19 bugs were fixed by *changing the parameters in APIs*.

```
  # ......
  all_labels  = genfromtxt('oh_labels.csv', delimiter=',')
  num_examples = all_labels.shape[0]
  dataset, all_labels = shuffle_examples(dataset, all_labels)

  # Split dataset into training (66%) and test (33%) set
  training_set_size = 2000
  mini_batch_size = 100
  training_set = dataset[0:training_set_size]
  training_labels = all_labels[0:training_set_size]
- total_batch = int(num_examples/mini_batch_size)
+ total_batch = int(training_set_size / mini_batch_size)
  # ......

  for i in range(total_batch):
      # example loading
      minibatch_x = training_set[i*mini_batch_size:
          (i+1)*mini_batch_size]
      minibatch_y = training_labels[i*mini_batch_size:
          (i+1)*mini_batch_size]
      # ......
```

Figure 3: CLE bug reported by a TensorFlow post

Incorrect Data Preprocessing (IDP). This category of bugs arises due to incorrect data preprocessing. In particular, missing or repeated data normalization is the most common reason for leading to NaN in this category. For example, when the data is not normalized, the values in the data can be too large for the network to normally train the model, leading to NaN eventually. Among the 27 IDP bugs in our study, 13 bugs were fixed by *correctly conducting data normalization*, while others were fixed by removing abnormal data or changing the data processing method.

Incorrect Floating Point Type (IFPT). This category of bugs arises when an incorrect floating-point type is used. Taking an example shown in Fig. 4, the hard coding of the maximum and minimum values could be out of range in `float16`, leading to NaN. Another example is that training with some optimizers (such as Adam [13]) in `float16` may result in numerical instability and thus get NaN results. To fix IFPT bugs, developers need carefully *check if there are incorrect floating-point types* that should be replaced with the correct ones or distinguished for special consideration.

Code Logic Error (CLE). This category of bugs occurs when developers mistakenly implement the functionality. For example, as shown in Fig. 3, `total_batch` (the number of batches in the training set) should be calculated by dividing `training_set_size` by `mini_batch_size`. However, developers mistakenly divided `num_examples` (the total number of inputs in both the training set and the test set) by `mini_batch_size`. Then, an empty batch is returned when the index for `training_set` exceeds its boundary due to this bug, which eventually leads to NaN of loss. For CLE bugs, it is hard to summarize their fix patterns as code logic could be wrong in various forms.

2.2.2 *Code Irrelevant Bugs.* Bugs of this type are caused by incorrect configurations or other external factors rather than developers' implementation in programming. In our study, 53.25% of numerical bugs belong to this broad category. The root causes of the bugs in this broad category are further divided into four categories, i.e., *Bad Inputs*, *Improper Model Configuration*, *Runtime Environment*, and *Gradient Explosion*.

```
-  attn_score = attn_score.float().masked_fill(
-    attn_mask[None,:,:,None], -1e30).type_as(attn_score)
+  if next(self.parameters()).dtype == torch.float16:
+    attn_score = attn_score.float().masked_fill(
+     attn_mask[None,:,:,None], -65000).type_as(attn_score)
+  else:
+    attn_score = attn_score.float().masked_fill(
+       attn_mask[None,:,:,None], -1e30).type_as(attn_score)
```

Figure 4: IFPT bug reported by a PyTorch GitHub issue

Gradient Explosion (GE). This is a special category since gradient explosion is actually an immediate flashpoint of this category of bugs rather than the root cause. Gradient explosion refers to that, an error gradient is used to update the network weights and continues accumulating errors during the training process, and finally the gradient becomes extremely large, leading to the numerical overflow of weights. However, in the issues/posts of these bugs, they directly pointed out that these numerical bugs are the problem of gradient explosion, and then instantly fixed them by either *clipping the gradients of intermediate variables* or *tuning the learning rate*. As discussed by developers, such fixing methods may not fix these numerical bugs fundamentally but effectively avoid them within limited iterations.

Runtime Environment (RE). This category of bugs are caused by third-party libraries (e.g., cuDNN, TensorFlow, and PyTorch) or hardware (e.g., CPU and GPU). *Updating the versions of libraries* or *replacing the hardware device* can fix RE bugs. Among the 48 RE bugs in our study, 34 bugs were fixed by the former and 14 bugs were fixed by the latter. We used two examples to further illustrate RE bugs. The first one is that a user came across NaN during training ResNet-18 with ImageNet. After deep analysis, she/he finally found the reason lies in the old version of cuDNN [1]. In the second one, a user spent a month discussing with others a numerical bug in her/his post, and finally solved the problem by sending the GPU card back to the manufacturer.

Bad Inputs (BI). This category is caused by ill-formed inputs. For example, when an input itself contains NaN, it must lead to NaN during program execution. Also, when the mapping between data and labels is wrong, it may cause numerical bugs. Interestingly, we also found that the format of images could cause numerical bugs. For example, a user used OpenCV to generate images in the JPEG format as the training set. However, JPEG is a lossy compression standard method for images, which could lose annotation information in images. In the meanwhile, OpenCV can reduce the quality of images in JPEG by default, and thus such loss eventually leads to a numerical bug. For BI bugs, *modifying or discarding bad inputs* (23 out of 39 bugs) and *correcting the map between data and labels* are two most common fix patterns.

Improper Model Configuration (IMC). This category is caused by improper model structure or parameters. Therefore, the fix pattern of IMC bugs is to *modify the improper components of a network*. Among the 36 IMC bugs in our study, 17 bugs were fixed by modifying the activation function, loss function, or optimizer of the network, and four bugs were fixed by modifying the network structure, and 15 bugs were fixed by modifying the network's parameters and the initialization of weights. For example, a user did not standardize data and adopted the SGD optimizer, leading to a numerical bug. This is because the SGD optimizer cannot perform well on such non-standardized data.

3 IMPLICATIONS

Detecting various types of bugs. Existing numerical bug detection techniques [18, 22] for DL programs just target IR bugs. According to our study, IR bugs only account for 23.25% of all the studied numerical bugs, which is the most common but still limited. Hence, it is necessary to propose more techniques to handle other types of numerical bugs. For example, GE bugs also occupy a significant proportion (22.5%) and it is possible to handle GE bugs by modifying hyper-parameter values and gradient clipping.

Investigating code relevant bugs. In our study, code relevant bugs account for 55.65%, most of which can be fixed by replacing API or clipping the parameter values into a valid range. There is an opportunity that mines the frequent patterns occurring with numerical bugs. For instance, an envisioned technique can focus on detecting code relevant numerical bugs based on these patterns.

Cross-hardware differential testing. According to our study, several numerical bugs (12%) are caused by incompatible hardware devices or iterations of DL libraries. Therefore, researchers can systematically reveal the inconsistencies generated by the same model in different environments by designing a differential testing framework, thereby finding more numerical bugs in DL programs.

4 RELATED WORK

Deep learning and deep neural network have been widely tested [8–10, 15–17]. Besides, there are a number of empirical studies on DL application bugs [11, 12, 20, 21]. Islam et al. [11] analyzed 2,716 Stack Overflow posts and 500 GitHub bug commits on five popular deep learning libraries to learn the root causes and impacts of DL bugs. They also analyzed the challenges of automated bug repair tools [12]. Zhang et al. [20] presented an empirical study on 715 questions collected from Stack Overflow and provided challenges in developing deep learning applications. Zhang et al. [21] studied DL applications built on TensorFlow and analyzed program bugs from Stack Overflow posts or GitHub issues. These studies analyzed general DL application bugs without deeply analyzing numerical bugs in DL applications. Unlike them, we conducted the first empirical study on DL numerical bugs by profoundly analyzing their root causes and fix patterns. Besides, there are also research projects on DL testing at other levels, such as model level [9, 17], library level [8, 16], and compiler level [15].

5 CONCLUSION

We conduct the first empirical study of numerical bugs in deep learning programs. In this study, we analyze 1,063 GitHub issues and 723 Stack Overflow posts and select 400 of them as our dataset. We classify these numerical bugs into nine categories by analyzing their root causes, and summarize some fix patterns for these categories. Finally, we provide the implications of detecting numerical bugs in deep learning programs according to our study findings.

ACKNOWLEDGMENTS

We thank all the anonymous reviewers for their valuable comments. This work has been supported by the National Natural Science Foundation of China 61782263 and 62002256.

REFERENCES

[1] Accessed: 2022. GitHub. https://github.com/tensorpack/tensorpack/issues/1188.

[2] Accessed: 2022. Homepage. https://github.com/Jacob-yen/numerical_bugs_in_dl_programs.

[3] Accessed: 2022. PyTorch Forums. https://discuss.pytorch.org/.

[4] Accessed: 2022. Stack Overflow. https://stackoverflow.com/.

[5] Accessed: 2022. TensorFlow. https://tensorflow.google.cn/community/forums.

[6] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek Gordon Murray, Benoit Steiner, Paul A. Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*. USENIX Association, 265–283.

[7] Chenyi Chen, Ari Seff, Alain Kornhauser, and Jianxiong Xiao. 2015. Deepdriving: Learning affordance for direct perception in autonomous driving. In *Proceedings of the IEEE International Conference on Computer Vision*. 2722–2730.

[8] Junjie Chen, Yihua Liang, Qingchao Shen, and Jiajun Jiang. 2022. Toward Understanding Deep Learning Framework Bugs. *arXiv preprint arXiv:2203.04026* (2022).

[9] Junjie Chen, Zhuo Wu, Zan Wang, Hanmo You, Lingming Zhang, and Ming Yan. 2020. Practical accuracy estimation for efficient deep neural network testing. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 29, 4 (2020), 1–35.

[10] Junjie Chen, Ming Yan, Zan Wang, Yuning Kang, and Zhuo Wu. 2020. Deep neural network test coverage: How far are we? *arXiv preprint arXiv:2010.04946* (2020).

[11] Md Johirul Islam, Giang Nguyen, Rangeet Pan, and Alessandra Russo Hridesh Rajan. 2019. A comprehensive study on deep learning bug characteristics. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019*. ACM, 510–520.

[12] Md Johirul Islam, Rangeet Pan, Giang Nguyen, and Hridesh Rajan. 2020. Repairing Deep Neural Networks: Fix Patterns and Challenges. *CoRR* abs/2005.00972 (2020).

[13] Diederik P. Kingma and Jimmy Ba. 2015. Adam: A Method for Stochastic Optimization. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*. http://arxiv.org/abs/1412.6980

[14] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *NeurIPS*. 8024–8035.

[15] Qingchao Shen, Haoyang Ma, Junjie Chen, Yongqiang Tian, Shing-Chi Cheung, and Xiang Chen. 2021. A comprehensive study of deep learning compiler bugs. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 968–980.

[16] Zan Wang, Ming Yan, Junjie Chen, Shuang Liu, and Dongdi Zhang. 2020. Deep learning library testing via effective model generation. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 788–799.

[17] Zan Wang, Hanmo You, Junjie Chen, Yingyi Zhang, Xuyuan Dong, and Wenbin Zhang. 2021. Prioritizing test inputs for deep neural networks via mutation analysis. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 397–409.

[18] Ming Yan, Junjie Chen, Xiangyu Zhang, Lin Tan, Gan Wang, and Zan Wang. 2021. Exposing numerical bugs in deep learning via gradient back-propagation. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 627–638.

[19] Ru Zhang, Wencong Xiao, Hongyu Zhang, Yu Liu, Haoxiang Lin, and Mao Yang. 2020. An empirical study on program failures of deep learning jobs. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 1159–1170.

[20] Tianyi Zhang, Cuiyun Gao, Lei Ma, Michael R. Lyu, and Miryung Kim. 2019. An Empirical Study of Common Challenges in Developing Deep Learning Applications. In *30th IEEE International Symposium on Software Reliability Engineering, ISSRE 2019, Berlin, Germany, October 28-31, 2019*. IEEE, 104–115.

[21] Yuhao Zhang, Yifan Chen, Shing-Chi Cheung, Yingfei Xiong, and Lu Zhang. 2018. An empirical study on TensorFlow program bugs. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 129–140.

[22] Yuhao Zhang, Luyao Ren, Liqian Chen, Yingfei Xiong, Shing-Chi Cheung, and Tao Xie. 2020. Detecting numerical bugs in neural network architectures. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 826–837.