An Empirical Study of Common Challenges in Developing Deep Learning Applications

Tianyi Zhang[†*] Cuiyun Gao[§*] Lei Ma[‡] Michael R. Lyu[§] Miryung Kim[†]
[†]University of California, Los Angeles [§]The Chinese University of Hong Kong [‡]Kyushu University
{tianyi.zhang, miryung}@cs.ucla.edu {cygao, lyu}@cse.cuhk.edu.hk malei@ait.kyushu-u.ac.jp

***Abstract*—Recent advances in deep learning promote the innovation of many intelligent systems and applications such as autonomous driving and image recognition. Despite enormous efforts and investments in this field, a fundamental question remains under-investigated—*what challenges do developers commonly face when building deep learning applications*? To seek an answer, this paper presents a large-scale empirical study of deep learning questions in a popular Q&A website, Stack Overflow. We manually inspect a sample of 715 questions and identify seven kinds of frequently asked questions. We further build a classification model to quantify the distribution of different kinds of deep learning questions in the entire set of 39,628 deep learning questions. We find that program crashes, model migration, and implementation questions are the top three most frequently asked questions. After carefully examining accepted answers of these questions, we summarize five main root causes that may deserve attention from the research community, including API misuse, incorrect hyperparameter selection, GPU computation, static graph computation, and limited debugging and profiling support. Our results highlight the need for new techniques such as cross-framework differential testing to improve software development productivity and software reliability in deep learning.**

***Index Terms*—deep learning, Stack Overflow, programming issues, software reliability**

I. Introduction

Deep learning has been successfully applied to many domains and has gained a lot of attention from both industry and academia. In the software engineering community, there is an increasing interest in applying deep learning to important software engineering problems, e.g., code completion [1], [2], code search [3], clone detection [4], [5], type inference [6], and bug prediction [7], [8]. Despite recent advances in testing deep learning applications [9]–[14], it is still unclear what kinds of programming obstacles and challenges developers face when building deep learning applications. It is also unclear how software engineering researchers should provide better tool support to address those programming pain points and improve the productivity of data scientists and machine learning engineers who build and integrate deep learning models.

Deep learning engineering significantly differs from traditional software engineering in terms of its programming paradigm and practice. Deep learning applications are *data-driven*, where developers define a desired neural network and let it automatically learn model parameters from a huge amount of training data. However, traditional software systems are *logic-driven*, where developers directly specify program logic in source code. Since model training requires heavy computation, deep learning developers often exploit data parallelism and accelerate model training using GPUs. As a result, software correctness and robustness in deep learning are more subject to training data quality, network architectures, hyperparameter selections, and configurations of computation units.

Shifting from traditional software development to deep learning engineering poses unique challenges [15]. Nowadays, developers often resort to online Q&A forums such as Stack Overflow to find solutions to programming issues they have encountered during software development. As of July 2018, Stack Overflow has accumulated 16 million programming questions and 26 million answers. Prior work has leveraged Stack Overflow to study trending topics in mobile app development [16], web development [17], and security [18]. In this paper, we analyze and mine deep learning questions asked in Stack Overflow to discover and understand common challenges in developing deep learning applications. We focus on three popular deep learning frameworks, TensorFlow, PyTorch, and Deeplearning4j, and extract 39,628 relevant deep learning questions in Stack Overflow.

Due to the large number of deep learning questions in Stack Overflow, it is challenging to manually analyze all of them. Therefore, we first manually inspect a sample of 715 deep learning questions and classify them based on Q&A contents and underlying programming issues. We identify seven kinds of programming questions—*program crash*, *model migration and deployment*, *implementation*, *training anomaly*, *comprehension*, *installation*, and *performance*. Then we build a keyword-based classification model to quantify the distribution of different kinds of deep learning questions in Stack Overflow. Among all types, program crash and model migration are the top two most frequently asked questions. In addition, performance questions are the most difficult to answer—only 25% performance questions have accepted answers compared with 34% in all other categories. Performance questions also take a longer time (4.3 hrs vs. 2.5 hrs in other categories) to receive a correct answer. We further examine the accepted or endorsed answer posts under each question to understand the root cause of underlying programming issues. We highlight five root causes that may deserve attention from the research community—API misuse, incorrect hyperparameter selection, GPU computation, static graph computation, and limited debugging and profiling support.

This study shows several development pain points in the

[*] The first two authors contributed equally.

construction, training, migration, and deployment of deep neural networks. Our findings motivate further investigation on debugging and profiling machine learning based systems as well as extending and inventing new differential testing for cross-framework model migration. The heavy utilization of GPUs also calls for new techniques that expose implicit programming constraints enforced by GPU computation.

The rest of the paper is organized as follows. Section II describes the data collection, the manual inspection procedure, and the automated classification technique. Section III describes our major findings. Section IV offers a discussion about the opportunities and challenges of extending software engineering techniques for deep learning. Section V discusses threats to validity and Section VI discusses related work. Section VII concludes this work and discusses future work.

II. Study Methodology

This section presents research questions of this study, followed by a description of data collection and analysis methods.

A. Research Questions

This study investigates the following research questions.

- *RQ1: What kinds of questions are frequently asked in deep learning?* This question aims to discover common programming issues and obstacles in the development of deep learning applications.
- *RQ2: Which kinds of deep learning questions are hard to resolve?* This question aims to examine the difficulty of resolving different kinds of programming issues by measuring the number of correct answers and the time it takes to receive those answers.
- *RQ3: What are the main root causes?* This question aims to understand the reasons for those programming issues, in order to inform software engineering researchers to design better approaches and tool support for deep learning engineering.

B. Data Collection

To identify common programming issues in deep learning, we collect Stack Overflow (SO) questions related to three representative and popular deep learning frameworks, TensorFlow [19], PyTorch [20], and Deeplearning4j [21]. These frameworks are widely adopted in practice but differ in their computation paradigm and architecture design. TensorFlow and Deeplearning4j adopt *static computation graphs*, where a neural network must be defined first before training (i.e., *define-and-run*). However, PyTorch adopts *dynamic computation graphs* and defines a neural network on-the-fly (i.e., *define-by-run*). Both TensorFlow and PyTorch provide Python APIs, while Deeplearning4j provides Java and Scala APIs. Compared with TensorFlow and PyTorch, Deeplearning4j is tightly integrated with distributed computing platforms such as Apache Spark and therefore supports distributed training by nature. From the SO data dump taken in December 2018 [22], we extract 39,628 questions that are tagged with `tensorflow`, `pytorch`, or `deeplearning4j`. Table I shows the number of questions related to each framework and their view counts.

TABLE I: The number of SO questions and their view count

Framework	#Questions	#View Count		
		Max	3rd Quartile	Median
TensorFlow	37,565	205,910	5,725	158
PyTorch	1,828	22,264	385	139
Deeplearning4j	235	5,685	362	131

C. Manual Inspection

We follow an open coding method [23] to inspect and classify deep learning questions collected from Stack Overflow. The first two authors first jointly inspect 50 deep learning questions from each framework and distill an initial set of categories based on underlying programming issues and symptoms. A question is assigned to all related categories if it is related to multiple programming issues. The two authors then independently classify more questions based on the initial categories. If one author finds a question that does not belong to an existing category, the author discusses with the other author and adds a new category as needed. It requires to sample 380 posts to achieve 95% confidence level and 5% confidence interval in the population of 39,628 SO posts related to deep learning. Yet we continue to inspect more posts till we do not find new frequent categories. This is a standard procedure in qualitative analysis to stop collecting new data when insights are converged. Finally, the authors compare their labeling results, discuss any disagreement, and refine the categories. At the end, the two authors inspect 715 SO questions and identify seven categories (discussed in Section III-A).

To understand the root cause of a deep learning question, we carefully examine the accepted answer (if any) under the question post and summarize its explanation and solution as the root cause. Though not suggested by Stack Overflow, some SO users may also endorse a correct answer by commenting under a post and expressing gratitude. Therefore, if there is no explicitly accepted answer, we go through the comments under an answer post to identify such endorsed answers. The entire manual inspection and root cause analysis process takes about 400 man-hours for the sample of 715 deep learning questions.

D. Automated Classification

To quantify the entire set of deep learning questions, we build a classification model that automatically classifies a deep learning question to one of the seven categories identified in the previous manual inspection step. This model uses a combination of word frequency and a set of manually selected keywords in each category for classification. The first two authors manually inspected top 200 frequent words appeared in each category and selected representative keywords together, following the same method as in prior work [24], [25]. To measure the contribution of a word to each category, we assign each word w_i^c a weight θ_i^c and $\theta_i^c = tf(w_i^c) / \sum_{j=1}^{N} tf(w_j^c)$, where $tf(w)$ means the term frequency of the term w in the posts. For manually selected keywords, we assign extra weight α ($\alpha \in [0, 1]$) to emphasize their contribution to a category.

Given a deep learning question, we compute its score with respect to each category,

$$s^k = \sum_{i=1}^{N} (\theta_i^c + Manual * \alpha) * tf(w_i^c). \quad (1)$$

where k refers to the k-th category and $Manual$ is a binary variable that indicates whether a keyword is a manually selected representative keyword. We split the manually classified 715 question posts into 429 training instances (60%) and 286 test instances (40%). We fine-tune the parameter α with 0.01 granularity. When setting α to 0.18, the classification model achieves the best test accuracy with 79.6% precision and 80.3% recall. Table II shows the precision and recall for individual categories. Though a deep learning question may be related to multiple categories, it is often hard to decide a proper threshold for relevant categories. Thus, we consider the most relevant categories (i.e., the ones with the highest score) in automated classification.

TABLE II: Classification Accuracy of Different Categories

Category	#Training data	#Test data	Precision	Recall
Implementation	166	108	0.875	0.824
Program Crash	134	78	0.853	0.829
Performance	49	27	0.733	0.786
Installation	55	38	0.903	0.903
Comprehension	59	28	0.750	0.706
Training Anomaly	45	45	0.773	0.810
Model Migration	34	28	0.684	0.765
Overall	542	352	0.796	0.803

Using this automated technique, we are able to quantify the overall distribution of programming issues in the entire set of deep learning questions in Stack Overflow. We experiment with other document classification models such as using `tf-idf` in conjunction with SVM but find that those models do not behave well, since many SO posts are short documents and there is also a lack of training data. Nevertheless, we do not argue that this classification algorithm is the best algorithm nor claim it as a novel contribution. We adopt this algorithm because it is simple to build and also achieves reasonable accuracy (79.6% precision and 80.3% recall), which could provide a good estimation about the prevalence of different kinds of deep learning questions asked in Stack Overflow.

Replication package. The entire dataset of 39,628 deep learning questions, the manual inspection spreadsheet, and the automated classification tool are publicly available.[1] Software engineering researchers who wish to conduct similar analysis on deep learning could use this replication package and the dataset.

III. Analysis of the Results

A. RQ1: What deep learning questions are frequently asked?

Figure 1 shows the distribution of different kinds of deep learning questions in the entire set of 39,628 SO posts.

[1]https://github.com/tianyi-zhang/deep-learning-stack-overflow

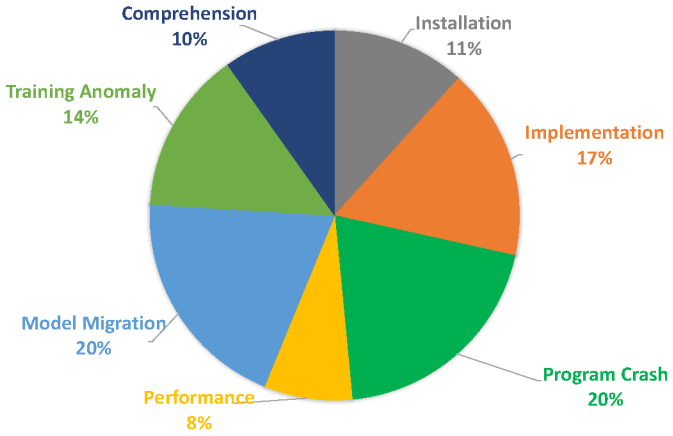

Fig. 1: Distribution of frequently asked deep learning questions in different categories

Overall, program crash, model migration, and implementation are the top three most frequently asked DL questions, while comprehension and performance are less frequently asked.

Categories such as *training anomaly* and *model migration* represent emerging programming issues that are unique to deep learning applications. Though other categories such as *program crashes* and *implementation* are well-known software engineering problems, their questions still exhibit new aspects of programming challenges due to the data-driven computation paradigm and GPU utilization in deep learning systems.

Implementation. Questions in this implementation category concern how to implement desired functionality or how to use an API of interest. Developers also wonder how to adapt the implementation of a desired neural network for a different task or a different dataset. Since deep learning heavily use GPUs for model training, many implementation questions involve how to use GPUs effectively, e.g., how to allocate a specific amount of GPU memory, how to separate the computation between CPU and GPU, etc. For instance, a developer asked how to properly assign data preprocessing operations to CPU so that GPU can focus on training only (Post 46001982).

One interesting sub-category of implementation questions is about distributed training. Training a large neural network with a large volume of data can take an impractically long time on a single GPU. Therefore, developers often resort to distributed training with multiple GPUs. A typical question in distributed training is about optimizing its parameter sharing strategy when a neural network is replicated on each GPU (i.e., *data parallelism*). Post 39595747 asks about how to first aggregate gradients among GPUs in the same machine before synchronizing with the parameter server to avoid bandwidth saturation. Compared with data parallelism, model parallelism requires developers to manually partition a neural network to different pieces and assign them to different devices (CPU and GPU), which is more tricky due to data dependencies between neurons (e.g., Post 42069147). The more dependencies a partitioning breaks, the more communication may be introduced to transfer data across devices. Overall, given a network

```
1  ...
2  def conv2d(x, W):
3    return tf.nn.conv2d(x, W, strides=[1, 1, 1, 1],
         padding='SAME')
4
5  def max_pool_2x2(x):
6    return tf.nn.max_pool(x, ksize=[1, 2, 2, 1], strides =
         [1, 2, 2, 1], padding='SAME')
7
8  mnist=input_data.read_data_sets('MNIST_data',one_hot=
       True)
9  x=tf.placeholder("float",shape=[None,784])
10 y=tf.placeholder("float",shape=[None,10])
11 x_image=tf.reshape(x,[-1,28,28,1])
12
13 #Layer 1: convolutional + max pooling
14 W_conv2=weight_variable([5,5,1,64])
15 b_conv2=bias_variable([64])
16 h_conv2=tf.nn.relu(conv2d(x_image,W_conv2)+b_conv2)
17 h_pool2=max_pool_2x2(h_conv2)
18
19 #Layer 2: ReLU+Dropout
20 W_fc1=weight_variable([7*7*64,1024])
21 b_fc1=bias_variable([1024])
22 h_pool2_flat=tf.reshape(h_pool2,[-1,7*7*64])
23 h_fc1=tf.nn.relu(tf.matmul(h_pool2_flat,W_fc1)+b_fc1)
24 ...
```

Fig. 2: A code example with shape inconsistency between the convolutional layer and the ReLU layer (Post 35138131)

architecture and a distributed environment, it is challenging to figure out how to partition data and operations across multiple devices to achieve optimal and reliable training performance.

Program Crash. Questions in this program crash category concern about runtime exceptions that crash a program. We mainly discuss three types of crash errors—*shape inconsistency*, *numerical error*, *CPU/GPU incompatibility*, which are frequently asked in deep learning applications but not so often in traditional software systems.

Shape Inconsistency. Shape inconsistency refers to runtime errors caused by unmatched multi-dimensional arrays between operations and layers. Figure 2 shows an example of shape inconsistency in a convolutional neural network. Given an input tensor in the shape of `[?,28,28,1]`, the first convolutional layer (lines 13-17) produces a tensor in the shape of `[?,14,14,64]` after striding and max pooling. However, the second ReLu layer expects an input tensor in the shape of `[?,7,7,64]`, leading to the shape inconsistency. To reduce the output tensor to the right shape, the stride of the first convolutional layer (line 3) should be set to `[1,2,2,1]` instead of `[1,1,1,1]`. When shape inconsistency occurs, developers often express the desire to view the input and output tensor shapes among network layers (Post 51460835). However, deciding tensor shapes is not straightforward, since the high-level network construction APIs in modern DL frameworks hide everything behind the scene. Furthermore, since tensors can have dynamic shapes, shape inconsistency cannot be simply detected via static type checking. Instead, it requires customized dataflow analysis accounting for both layer connectivity and operations that transform tensor shapes.

Numerical Errors. Since deep neural networks extensively use floating point computation [26], they can be highly vulnerable to numerical errors in both training and evaluation. Numerical errors are notoriously hard to debug, partly because they may only be triggered by a small set of rarely encountered inputs. One typical numerical error is not-a-number (NaN) values. For example, in Figure 3, the expression `ys_reshape*tf.log(prediction)` (line 3) may yield $0 * log(0)$ and further produces NaN values, since the previous ReLU layer can output zero. Adding a small positive value (line 4) can prevent this error. In TensorFlow, it is generally recommended to use another function, `tf.nn.softmax_cross_entropy_with_logits`, to compute cross entropy, since this function properly handles corner cases that may cause NaN errors.

```
1    ys_reshape = tf.reshape(ys,[-1,1])
2    prediction = tf.reshape(relu4,[-1,1])
3  - cross_entropy =
       tf.reduce_mean(-(ys_reshape*tf.log(prediction)))
4  + cross_entropy = tf.reduce_mean(-(ys_reshape*tf.log(
       prediction+1e-5)))
5    train_step = tf.train.AdamOptimizer(0.01).minimize(
         cross_entropy)
```

Fig. 3: An NaN error occurs at line 4 when computing cross entropy with logits (Post 40192728).

CPU/GPU Incompatibility. GPU/CPU incompatibility often occurs when switching model training from CPU to GPU. As a common practice, developers often run and test their deep neural networks on CPU first and then port them to GPU to speed up training. Compared with CPU, GPU supports different data types and operations that are customized and optimized for parallel computation across GPU kernels. To perform training on GPU, developers must make sure all data and operations in their neural networks are compatible with GPU. Modern DL frameworks often hide such implicit design decisions nicely and expose global flags and APIs to automatically transfer data between CPU and GPU. Therefore, developers only need to make small changes to their existing code to port it to GPU. However, making such subtle changes is error-prone. In Figure 4, even though the programmer converts the input data to its GPU-compatible version by calling `cuda()` at line 3, the code snippet will throw `TypeError` at line 12 when running on GPU, since the programmer forgets to convert other data (e.g., `decoder_hidden`, `decoder_context`) to GPU. Such mistakes can be hard to catch in a large and complex neural network, leading to a program crash.

Training Anomaly. This category is related to unreliable training behaviors. Neural network training is essentially a process of continuously adjusting model parameters based on previous predication errors measured by a loss function. To achieve high prediction accuracy, the training process often involves a huge amount of training data and a collection of optimization tricks such as mini-batching and feature scaling. Any training data errors or mis-conducted optimization tricks could cause training anomalies. Various training anomalies are reported on Stack Overflow, including extremely low or high

```
1  encoder_hidden = encoder_test.init_hidden()
2  if USE_CUDA:
3      word_inputs = Variable(torch.LongTensor([1, 2, 3]).
           cuda())
4  else:
5      word_inputs = Variable(torch.LongTensor([1, 2, 3]))
6
7  encoder_outputs, encoder_hidden = encoder_test(
       word_inputs, encoder_hidden)
8  decoder_attns = torch.zeros(1, 3, 3)
9  decoder_hidden = encoder_hidden
10 decoder_context = Variable(torch.zeros(1,
       decoder_test.hidden_size))
11
12 decoder_output, decoder_context, decoder_hidden,
       decoder_attn = decoder_test(word_inputs[0],
       decoder_context, decoder_hidden, encoder_outputs)
```

Fig. 4: `TypeError` occurs at line 12 when running the PyTorch model on GPU due to incompatible float tensor types between CPU and GPU (Post 46704352)

accuracy, loss values never dropping, overfitting, discontinuous accuracy values between iterations, unstable loss values, etc. Here, we discuss one representative example that is frequently asked in Stack Overflow. In Post 34743847, the training loss was initially low but quickly increased to a large value and kept bouncing back and forth without converging. The reason is that the learning rate is 0.1, which is too high for this specific task. A high learning rate can cause drastic updates on network parameters, leading to a vicious cycle of ever-increasing gradient values (i.e., *gradient exploding*). Besides, exploding gradients can also be caused by improper weight initialization. For instance, in Post 36565430, for a deep neural network with more than 10 layers, the extra layers could make the gradients too unstable, making the loss function quickly devolve to NaN. The best way to fix this is to use Xavier initialization [27]. Otherwise, the variance of initial values tends to be too high, causing instability.

Since training anomalies do not crash a program, there are no error messages or stack traces that a developer can start investigation from. To debug such anomalous behavior, experienced developers may print parameters and gradients to console and observe how their values change over training epochs. Then they may decide which hyperparameter to tune based on their own heuristics. However, such trail-and-error method is tedious and cumbersome when debugging a large neural network with thousands of neurons and millions of parameters. Existing visualization tools such as TensorBoard [28] and Visdom [29] can provide a bird's-eye view of the entire training process. However, these tools only plot high-level quantitative metrics over training epochs but lack the traceability to pinpoint which statement, operation, or hyperparameter could cause such an anomaly.

Model Migration and Deployment. Questions in this category concern about porting model implementations between different frameworks or deploying saved models across different frameworks or platforms. In such a migration or deployment process, behavior inconsistency often occurs due to variations or incompatibilities between different frameworks, platforms, or library versions. For instance, in Post 49447270, the developer wanted to import the weights of a Theano-based CNN model into a predictor written in PyTorch. As Theano and PyTorch adopt different matrix formats in a convolutional layer by default, the predictor could not behave properly without matrix format conversion. As another example, a developer found that the output quality was much lower when restoring a saved model that was trained on a server in a mobile device (Post 49454430). It is critical to ensure behavior consistency of the same model in different settings for deployment. However, such behavior inconsistency is often hard to diagnose due to a lack of tool support for systematically testing and comparing model behavior between different platforms or frameworks.

To deploy a neural model on mobile devices or embedded systems, quantization techniques are commonly used to reduce model size [30]. However, quantization can cause various errors such as program crashes and performance issues. For instance, after reducing floating point precision to 8 bit, a developer found that the quantized model was about 20X slower than the original one (Post 39469708). With an increasing demand of migrating a deep learning model from cloud servers to mobile devices and IoT devices, a differential testing framework that systematically detects behavior consistency of the same model in different settings is necessary for improving the robustness of a model.

Performance. Questions in this performance category concern about training time and memory usage of deep learning models. Training neural networks is computationally demanding. Therefore, developers often concern about training performance in terms of execution time and memory usage. Many performance questions in Stack Overflow ask for advice on how to optimize model implementation and how to fix performance bugs such as memory leaks. Developers also ask about the performance difference between different platforms, frameworks, and GPUs. For instance, a developer implemented an image recognition model for ImageNet using both MXNet and TensorFlow but found that, when training using four GPUs, TensorFlow was much slower than MXNet (Post 36047937). There are similar questions about TensorFlow vs. Caffe (Post 37255626), PyTorch vs. TensorFlow (Post 50784130), etc. When such divergent behavior occurs in different settings or frameworks, developers often wonder whether it is caused by framework differences or implementation bugs in their own code. This indicates a need for diagnosing and empirically benchmarking performance differences among different platforms and frameworks.

In addition to training time, GPU utilization ratio is another important metric that developers often use to diagnose performance bugs in deep learning. GPU usage depends on many factors including the size of a neural network, batch size, and data pre-processing. For instance, using GPUs can actually slow down the overall training process for small neural networks with light computation, since the overhead of memory allocation and data transferring can overwhelm

the speed-up benefit (e.g., Post 38688777, Post 47900761). Mini-batch gradient descent is a common trick to exploit the full power of GPUs by feeding multiple training instances to train a neural network all at once. Developers often encounter a low GPU usage rate due to small batch sizes (Post 47971512, Post 52159053). Using a large batch size can make the training process converge more quickly by computing gradients from multiple training instances in parallel. On the other hand, a large batch size can also lead to out-of-memory errors, since it requires a lot of memory to hold such large batches and corresponding tensors (Post 38010666, Post 45916769). CPU computation can also become a bottleneck of GPU usage. For instance, in Post 35274405, the developer found that the GPU usage rate always bounced back and forth between 0% and 80% during training. The reason is that each training iteration starts from fetching and preprocessing a batch of training instances from a queue, which is really slow and thus blocks GPU computation.

Comprehension. Questions in this comprehension category ask for clarifications on concepts, algorithms, and framework APIs in deep learning. Oftentimes, these comprehension questions are simply answered by quoting or summarizing existing learning resources. Yet just like any other emerging techniques, a large portion of SO posts are about the clarity of documentations and the lack of code examples in deep learning. For example, in Post 34240703, the developer is confused about the usage of `logits` in the cross-entropy functions and asks for code examples to illustrate the difference between two similar functions, `tf.nn.softmax` and `tf.nn.softmax_cross_entropy_with_logits`.

Installation. Questions in this installation category are related to software reliability issues caused by incompatible library installation. Deep learning frameworks are notoriously hard to install due to complex software and hardware dependencies and version incompatibilities. For example, each TensorFlow version is only compatible with certain cuDNN library versions, which further depends on certain CUDA versions and Nvidia GPU models. Due to the rapid evolution in DL frameworks, many framework versions are not backward compatible. Package managers such as `conda` and `pip` do not always make installation easier, especially when DL framework dependencies are cluttered with other software. Upgrading a single library can become catastrophic, breaking the entire dependency chain and affecting other installed software.

We also analyze the distribution of deep learning questions in different issue categories over years, as shown in Figure 5. Starting from 2015, the number of deep learning questions increases dramatically each year, indicating the increasing popularity of deep learning. Although there are no major topic changes over years, the percentage of installation questions decreases from 20% in 2015 to 11% in 2018. Similarly, the percentage of implementation questions also decreases slightly from 19% to 16%. This can be caused by the fact that DL frameworks are getting more stable and their documentations are also getting more comprehensive. For instance, now there

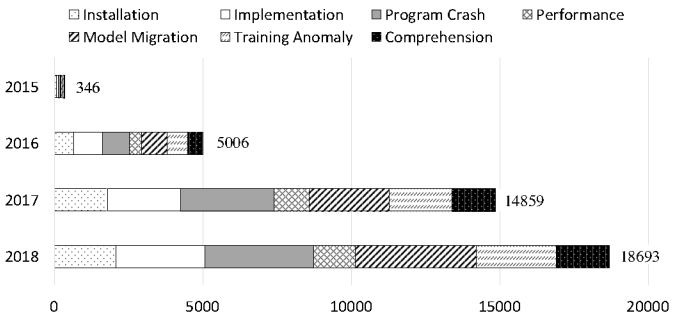

Fig. 5: Distribution of deep learning questions in different categories over time

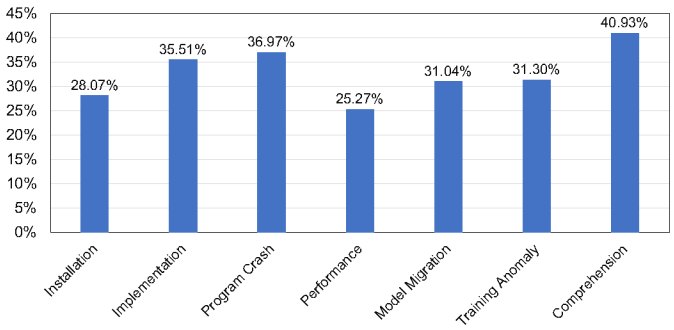

Fig. 6: Percentage of deep learning questions with accepted answers in Stack Overflow

is a TensorFlow webpage that keeps track of build and installation issues posted on Stack Overflow and GitHub [31]. By contrast, the percentage of model migration questions are constantly increasing over time, from 14 % in 2015 to 22 % in 2018. Based on manual inspection, the majority of model migration questions are about migrating from Theano and Caffe to TensorFlow and PyTorch, rather than the other way around. This can be caused by the fact that, with strong development teams devoted from Google and Facebook, TensorFlow and PyTorch now provide more functionality and tool support and thus attrack more developers than Theano and Caffe. In fact, Theano has already stopped active development and maintenance due to the competition from industry [32].

B. RQ2: Which kinds of deep learning questions are harder to resolve?

Stack Overflow encourages users to accept an answer, if the answer is correct or useful to resolve the question. Therefore, we consider that a Stack Overflow question is resolved, if the question has an accepted answer. We investigate the difficulty of resolving different kinds of deep learning questions from two perspectives. First, we analyze how many deep learning questions in a category have accepted answers. Among those questions with accepted answers, we further analyze how long it takes to receive a correct answer after a question is posted on Stack Overflow.

Figure 6 shows the percentage of deep learning questions that have accepted answers in each category. Around 41 %

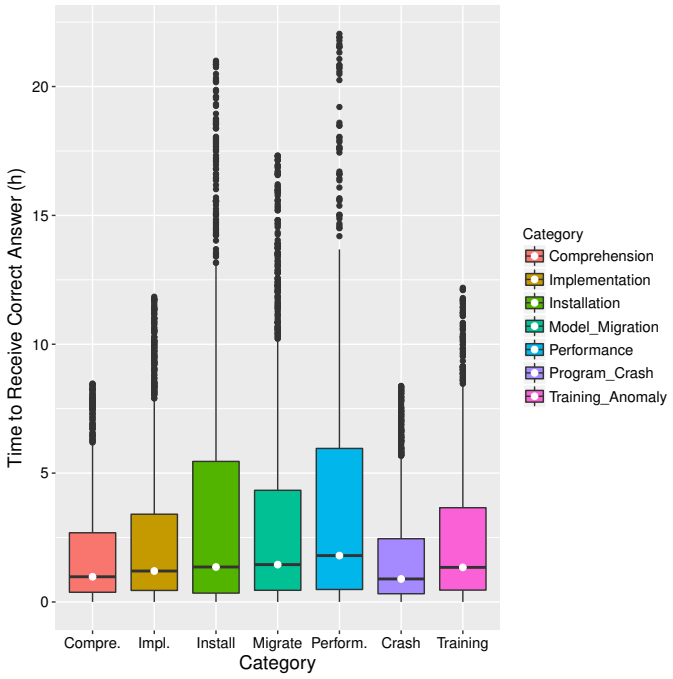

Fig. 7: Distribution of the time length to receive a correct answer for different kinds of deep learning questions

of comprehension questions have accepted answers, which is the highest rate among all categories. Based on the manual inspection, many comprehension questions are asked because developers are not familiar with machine learning concepts or because the documentation is unclear. Such comprehension questions are relatively easy to answer compared with other kinds of questions. By contrast, only 25 % of performance questions have accepted answers, which is the lowest. Furthermore, among all categories, performance questions also take the longest time to receive a correct answer. As shown in Figure 7, performance questions take 4.3 hours on average to receive a correct answer, while program crash questions take only 1.7 hours. Compared with other issues such as program crashes, performance issues do not have error messages that trace back to specific lines of code. Developers have to use profiling tools to first diagnose which operation is the bottleneck in order to further investigate the root cause. However, profiling support from deep learning frameworks is still primitive. For instance, since PyTorch does not provide any profiler, PyTorch developers have to manually instrument a neural network to gather performance metrics using Python modules such as `time` (Post 49989414). In addition, performance bugs often occur in the training stage and many of them require specific GPUs or a distributed environment to reproduce (e.g., Post 49235599, Post 37255626). On the other hand, many program crashes such as type errors and shape inconsistencies often occur in the graph construction stage, which only requires CPU and is therefore relatively easy to be reproduced by other developers.

C. RQ3: What are the main root causes?

Certainly, some SO questions are simply because developers do not have enough machine learning background or do not read documentations carefully. However, a large portion of questions are caused by common mistakes that different developers repetitively make. In addition, the lack of tool support also makes it hard to diagnose certain types of errors in deep learning and thus developers have no choice but turn to online Q&A forums, which may deserve attention from framework developers and software engineering researchers. We identify five main root causes of the common programming issues during the manual inspection.

API misuse. API usage violations are the root cause of various runtime errors and training anomalies. For instance, developers must call `zero_()` after `loss.backward()` to zero out gradients to avoid exceptions in PyTorch (Post 46513276). Some API usage violations are implicit and hard to spot. For instance, TensorFlow requires to initialize all `tf.Variable` objects by calling `initialize_all_variables`, which is easy to follow for user-defined variables. However, certain functions such `adam` and `momentum` optimizers define variables internally and thus developers still need to explicitly initialize variables before training (Posts 33788989, 36007883). In such cases, it is critical to provide more *information transparency* about critical internals of APIs and functions.

GPU computation. Utilizing GPUs raises a spectrum of programming challenges when building deep learning applications. GPUs impose critical programming constraints that must be followed to avoid runtime errors, training anomalies, and performance bugs. For example, developers cannot directly convert a PyTorch tensor `x` to an equivalent `NumPy` array by calling `x.data.numpy()` in GPU. Instead, developers have to move a tensor to CPU first and then convert it to `NumPy` by calling `x.data.cpu().numpy()` (Post 44351506). When porting a deep learning model from CPU to GPU, such unsupported function calls must be rewritten first before training. Though GPUs play such an important role in deep learning, many GPU interfaces and usage scenarios are not well documented. For example, Post 38580201 illustrates how to use a function `list_local_devices` to list available GPUs in TensorFlow, which receives 128 upvotes. However, this function is not documented at all.

Incorrect hyperparameter selection. Modeling mistakes such as improper hyperparameters are often the root cause of unexpected model behavior such as low accuracy and overfitting. For example, in Post 37914647, the developer built a simple neural network to learn arithmetic addition but found that the training accuracy was always 100 %, even with garbage data. The reason was that the developer intended to learn a linear regression model to approximate addition, but mistakenly used the cross-entropy loss function, which was designed for classification problems. The developer should use the mean square error (MSE) loss function instead. Another developer sets the embedding dimension too high for a small vocabulary, which causes overfitting (Post 48541336). Thus, it

```
1  # define a neural network
2  def train_op():
3      images, true_labels = inputs()
4      predictions = NET(images)
5      true_labels = tf.cast(true_labels, tf.float32)
6      loss = tf.nn.softmax_cross_entropy_with_logits(
           predictions, true_labels)
7      return OPTIMIZER.minimize(loss)
8
9  # train the network
10 def train():
11     ...
12     while not coord.should_stop():
13           # i-th training epoch
14           training = train_op()
15           sess.run(training)
16     ...
```

Fig. 8: A TensorFlow code snippet that defines a neural network within the training process and thus hangs forever during training, simplified from Post 35274405.

can be beneficial to mine common combinations and values of hyperparameters and show how others set hyperparameters.

Static graph computation. Unlike PyTorch, TensorFlow adopts static computation graphs, where a neural network must be defined before training. Though such *define-and-run* paradigm can improve training performance via mini-batching, it is not straightforward to program with, leading to many programming mistakes. A common mistake is to define a neural network within the training process, as shown in Figure 8. The function call, `train_op` at line 14 keeps adding new operations to the neural network in each training iteration, making the network grow bigger and bigger and thus quickly draining out the computation power.

Static graph computation also causes difficulty in debugging training behavior, accounting for many debugging questions in Stack Overflow. General-purpose debuggers such as `pdb`[2] cannot be used to inspect runtime values when training the computational graph in TensorFlow. Instead, TensorFlow uses the `tf.Session` interface to communicate between a static graph and the training process. Developers have to evaluate a tensor using `tf.Session` functions first and print its values to the console for further analysis, which is not straightforward. In Stack Overflow, many developers feel confused about how to inspect runtime values in TensorFlow (e.g., Post 47710467, Post 33679382, Post 33633370). This is very different from traditional software debugging, where developers can simply inject a breakpoint and step through the execution. Though TensorFlow now provides a debugger called `tfdbg`[3] for TensorFlow models, where you can step through the execution and check values, it is cumbersome to set up as complained by many developers in Stack Overflow (e.g., Post 44211871, Post 46025966, Post 46104266).

Limited debugging and profiling support. Debugging a deep learning model is fundamentally different from debugging a regular program. Because the decision logic of deep learning models is not directly specified by developers, but learned from training data. The backbone of a deep learning model is a data-flow graph. Though a stack trace may point to a specific line of code, the real fault can reside in the training data, hyperparameter selection, hardware, and versioning. Different training data and hyperparameters can lead to unexpected behavioral divergence, which is hard to debug by just comparing high-level metrics such as training loss over time. Setting random seeds, data dropout, and some GPU operations can cause non-determinism in model training, making it harder to debug (Post 39938307). When an error occurs in GPU or in a distributed setting, developers cannot easily track which operation at which step raises the error (Post 50661415). There is also limited tool support for runtime monitoring or profiling in GPU and in a distributed environment (Post 34775522). Developers cannot do much except for waiting the training process to finish, staring at log files, and tuning the model via trial and error.

When performance issues occur, developers find it hard to identify which line of code or which operation introduces the bottleneck. The profiling support in PyTorch and Deeplearning4j is still primitive. In PyTorch, developers have to manually instrument a neural network using the built-in Python modules such as `time`[4] and `gc`[5] to collect the execution time and memory fingerprint. Similarly, developers also have to use a general-purpose JVM profiler to profile a Deeplearning4j model. TensorFlow provides its own profiling feature, *runtime statistics* feature[6], which is tightly integrated with its visualization tool, TensorBoard. However, TensorBoard is cumbersome to use, since developers have to write extra boilerplate code to set up TensorBoard and collect performance metrics. In Stack Overflow, we observe a variety of errors such as web browser compatibility errors (Post 33680397) and display errors (Post 34416702), when setting up and using TensorBoard.

Existing visualization tools such as TensorBoard [28] and Visdom [29] provide a good summary of high-level metrics such as training accuracy over iterations. However, these tools are not suitable for low-level debugging activities such as inspecting runtime values and setting a watchpoint for NaN values. A common practice is still to print runtime values to a console or log files and manually scan for potential errors such as exploding gradients. Due to the ever-increasing model complexity and data volume, now it is common to train deep neural networks with multiple GPUs or a cluster of workers. Such distributed training brings more challenges in debugging deep learning models, since runtime errors and training anomalies are compounded with communication bandwidth and latency as well as neural network partitioning and placement in a distributed setting.

IV. Research Opportunities

Certain categories such as installation and comprehension are general issues in software engineering rather than specific

[2]https://docs.python.org/3/library/pdb.html
[3]https://www.tensorflow.org/guide/debugger
[4]https://docs.python.org/3/library/time.html
[5]https://docs.python.org/2/library/gc.html
[6]https://www.tensorflow.org/guide/graph_viz#runtime_statistics

to deep learning applications. However, in deep learning applications, many issues such as program crashes and unexpected behavioral inconsistencies after model deployment are due to the incompatibility between installed libraries and hardware or a lack of understanding of library functions. Therefore, in order to build reliable deep learning systems, it is important to understand commonly asked questions related to those aspects and think about solutions for better library dependency management and documentation, especially given the rapid revolution of deep learning libraries and the heavy utilization of hardware units compared with traditional software systems. We briefly discuss three major implications of our findings on future research below.

Implication 1: Mining implicit API usage protocols and hyperparameter combinations is needed. There are many implicit but critical API usage protocols enforced by deep learning frameworks and GPU computation. Failing to follow these constraints will lead to program crashes, training anomalies, and performance bugs. Deep learning frameworks often lack documentations of these API usage protocols. Therefore, it is beneficial to automatically mine such API usage protocols from neural network implementations available on GitHub. In addition, since hyperparameters are often specified as function parameters, mining common combinations of hyperparameters and showing how other developers set them can provide guidance for model design and tuning.

Implication 2: Facilitating debugging and profiling is needed. Runtime monitoring and profiling support needs to be improved, especially when training with multiple GPUs and machines. Testing and runtime verification techniques could be useful to detect training anomalies. For instance, an envisioned technique should enable data scientists and machine learning engineers to expressively specify desired high-level properties as test oracles or assertations that can be propagated across training iterations. Then, an abnormal training execution state that violates those properties (e.g., numerical errors, concurrency issues, incorrect parameter configuration) could be captured at runtime for further analysis.

Implication 3: Cross-framework and cross-platform differential testing is needed. Behavior inconsistency often occurs when migrating a model between different frameworks or deploying a model to a different platform (e.g., Android, IOS, web browsers). However, it is hard to uncover and diagnose such behavioral inconsistency due to a lack of tool support. Existing testing techniques mainly focus on identifying test inputs that improve a given test coverage metric for test generation [9]–[12], [33]. Therefore, a differential testing framework that systematically tests and uncovers behavior inconsistency of the same model in different settings is needed, especially with an increasing demand for migrating and deploying a deep learning model from cloud servers to mobile devices and other edge computing devices.

As future work, we plan to send surveys and conduct semi-structured interviews with professional machine learning engineers and data scientists to understand common practices and principles to address these challenges in industry and solicit feedback about desired tool support.

V. Threats to Validity

Internal Validity. The combination of manual analysis and automated classification raises several threats to internal validity. First, the manual analysis of deep learning questions in Stack Overflow is subjective. To reduce the bias, two authors independently inspected a total of 715 deep learning questions and discussed classification disagreements. The taxonomy was finalized based on the consensus of two authors after resolving all disagreements.

Since it is challenging to manually inspect all 39,628 deep learning questions, we develop an automated technique to classify them to seven categories identified in the manual analysis. Both the precision and recall of our classification technique are about 80%, which is commonly acceptable for automatic natural language analysis in software engineering [34]–[37]. Nevertheless, questions can still be inevitably misclassified. The automated classification is based on the manually identified categories that may not fully cover all categories in the entire set of 39,628 questions. To mitigate this issue, the authors followed an open-coding method [23] to iteratively develop and refine the categories of frequently asked questions and continued to inspect more SO posts, until no new programming issue categories were found.

In RQ2, we consider the accepted answer of a SO question as the correct answer. Yet, some SO users may endorse correct answers by commenting below answer posts, rather than explicitly accepting them as correct answers. Therefore, we may miss correct answers that are not accepted by question asksers. As a result, the percentage of real correct answers in each category could be slightly higher in Figure 6.

External Validity. The external validity concerns about the generalizability of our results. In order to mitigate this issue, we analyze programming questions related to three popular and representative deep learning frameworks with different computation paradigms and infrastructures. But still, we may miss unique programming issues in other deep learning frameworks that are not included in our study scope, such as MXNet [38] and Caffe [39]. Since we only analyze deep learning questions asked in Stack Overflow, we may overlook valuable insights from other sources. In future work, we plan to conduct in-depth interviews with professional deep learning engineers and solicit their feedback to mitigate this issue. Furthermore, given the rapid evolution of deep learning frameworks, it is unclear how long our findings may stay valid and what new challenges may emerge in the future. Any breakthrough in industry and academia may significantly change the way developers write deep learning programs. For example, Microsoft and Facebook released the first version of Open Neural Network Exchange (ONNX), a new neural network exchange format and ecosystem for interchangeable machine learning models in 2017 [40]. Later, IBM, Huawei, Intel, AMD, ARM and Qualcomm announced support for the

initiative. Increasing adoption and support for ONNX may shift the themes of questions related to model migration.

VI. Related Work

The most related work is a recent study on 175 software bugs in deep learning applications built by TensorFlow [41]. Our work extends this study by identifying seven categories of frequently asked questions about deep learning. We analyze 715 Stack Overflow questions not just related to TensorFlow, but also two other popular deep learning frameworks with different computation paradigm and architecture design. In addition to software defects, we also identify other development issues such as training anomaly and model migration. We also find that, in addition to enhance fault localization and repair techniques as suggested in [41], it is also important to expose implicit API usage constraints, provide more interactive debugging and profiling support, and enable differential testing for migrating and deploying models across different frameworks and platforms.

Both Thung et al. [42] and Sun et al. [43] study categories of software bugs in machine learning libraries and frameworks. Thung et al. [42] manually inspected a sample of 500 bugs in Apache Mahout, Lucene, and OpenNLP, and grouped them based on bug categories proposed by Seaman et al. [44]. They further analyzed the bug severity, as well as the average time and effort needed to fix a bug in each category. Sun et al. [43] manually inspected 328 closed bugs in three machine learning systems, Scikit-learn, Paddle, and Caffe. They proposed seven new bug categories and also identified twelve fix patterns that are commonly used to fix these bugs. Our work differs by focusing on software development issues and challenges when using deep learning frameworks, rather than building and maintaining these frameworks.

Recently, the software engineering community made many advances in testing deep learning applications [9]–[14], [45]–[49]. DeepXplore [9] proposes a test effectiveness metric called neuron coverage and develops a neuron-coverage-guided differential testing technique that uncovers behavioral inconsistencies between different deep learning models. DeepTest [10] is a test generation framework that adapts labeled driving scenes with a set of pre-defined image transformations in order to increase the neuron coverage of autonomous driving systems. DeepRoad [13] improves on DeepTest by leveraging the generative adversarial network (GAN) to automatically transform images to different driving scenes, e.g, snowy or rainy conditions. DeepConcolic [12] is a concolic testing approach that incrementally generates test inputs for deep neural networks by alternating between concrete execution and symbolic analysis. DeepGauge [11] and DeepCT [50] extend the neuron coverage metric by proposing a set of neuron value-based testing criteria and neural interaction-based criteria, and demonstrated that the new criteria were more effective in capturing software defects caused by adversarial examples. Kim et al. [14] propose a new test criterion called surprise adequacy that measures how much input data follows the statistical distribution of training data in terms of neuron activation status. Note that most of these techniques focus on improving the robustness of deep learning models by generating test inputs. In this study, we find that developers are also likely to make implementation bugs or choose sub-optimal hyperparameters, leading to various program crashes and training anomalies that are rarely seen in traditional software systems. Our findings indicate the need of testing and fault localization support that allows developers to systematically check desired behaviors and properties of deep learning applications, e.g., a gradient should not be NaN, similar to how developers write unit test cases to check conventional software systems.

Ma et al. [33] present a debugging technique called MODE for deep neural networks. MODE performs state analysis of neural network to identify neurons that are responsible for incorrectly predicted test data, and selects new input samples to retrain the faulty neurons. MODE is designed to address overfitting and underfitting problems caused by inadequate training data. Our study finds other kinds of bugs such as numerical errors and performance issues, which requires more interactive debugging and profiling techniques.

VII. Conclusion

This paper presents a large-scale study about programming issues and mistakes in deep learning. We manually inspect a sample of 715 deep learning questions in Stack Overflow and identify seven types of frequently asked questions. We find that the new data-driven programming paradigm in deep learning has introduced new software development issues such as training anomalies and model migration, which has not been observed in traditional software systems. Based on the insights from manual inspection, we build an automated classification technique that quantifies different kinds of deep learning questions in Stack Overflow. Among all categories, program crashes and model migration are the top two most frequently asked topics. We also find that performance questions are the most difficult deep learning questions to answer in terms of receiving correct answers and the wait time for correct answers.

Despite the successful adoption of deep learning to many domains, our study reveals that the development tool chain support is still at an early stage. Our study motivates the need of debugging and profiling machine learning and AI based systems as well as extending and inventing new differential testing for cross-framework model migration, which is a timely topic and urgent issue to be addressed to impact real-world systems of today and future.

Acknowledgment

Thanks to anonymous reviewers for their valuable feedback. This work is supported by NSF grants CCF-1764077, CCF-1527923, CCF-1460325, CCF-1723773, ONR grant N00014-18-1-2037, Intel CAPA grant, Hong Kong RGC GRF grant CUHK-14210717, JSPS KAKENHI Grant-19H04086, and Qdai-Jump Research Program NO.01277.

REFERENCES

[1] V. Raychev, M. Vechev, and E. Yahav, "Code completion with statistical language models," in *Acm Sigplan Notices*, vol. 49, no. 6. ACM, 2014, pp. 419–428.

[2] X. Gu, H. Zhang, D. Zhang, and S. Kim, "Deep api learning," in *Proceedings of 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2016, pp. 631–642.

[3] X. Gu, H. Zhang, and S. Kim, "Deep code search," in *Proceedings of the 40th International Conference on Software Engineering*. ACM, 2018, pp. 933–944.

[4] L. Li, H. Feng, W. Zhuang, N. Meng, and B. Ryder, "Cclearner: A deep learning-based clone detection approach," in *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Sep. 2017, pp. 249–260.

[5] G. Zhao and J. Huang, "Deepsim: Deep learning code functional similarity," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2018. New York, NY, USA: ACM, 2018, pp. 141–151. [Online]. Available: http://doi.acm.org/10.1145/3236024.3236068

[6] V. J. Hellendoorn, C. Bird, E. T. Barr, and M. Allamanis, "Deep learning type inference," in *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering*. ACM, 2018.

[7] M. Pradel and K. Sen, "Deepbugs: A learning approach to name-based bug detection," *Proc. ACM Program. Lang.*, vol. 2, no. OOPSLA, pp. 147:1–147:25, Oct. 2018. [Online]. Available: http://doi.acm.org/10.1145/3276517

[8] S. Wang, T. Liu, J. Nam, and L. Tan, "Deep semantic feature learning for software defect prediction," *IEEE Transactions on Software Engineering*, 2018.

[9] K. Pei, Y. Cao, J. Yang, and S. Jana, "Deepxplore: Automated whitebox testing of deep learning systems," in *Proceedings of the 26th Symposium on Operating Systems Principles*. ACM, 2017, pp. 1–18.

[10] Y. Tian, K. Pei, S. Jana, and B. Ray, "Deeptest: Automated testing of deep-neural-network-driven autonomous cars," in *Proceedings of the 40th International Conference on Software Engineering*. ACM, 2018, pp. 303–314.

[11] L. Ma, F. Juefei-Xu, and J. S. et al., "DeepGauge: Multi-Granularity Testing Criteria for Deep Learning Systems," in *The 33rd IEEE/ACM International Conference on Automated Software Engineering*, 2018.

[12] Y. Sun, M. Wu, W. Ruan, X. Huang, M. Kwiatkowska, and D. Kroening, "Concolic testing for deep neural networks," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE 2018, 2018, pp. 109–119.

[13] M. Zhang, Y. Zhang, L. Zhang, C. Liu, and S. Khurshid, "Deeproad: Gan-based metamorphic testing and input validation framework for autonomous driving systems," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE 2018, pp. 132–142.

[14] J. Kim, R. Feldt, and S. Yoo, "Guiding Deep Learning System Testing using Surprise Adequacy," in *2019 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, 2019.

[15] S. Amershi, A. Begel, C. Bird, R. DeLine, H. Gall, E. Kamar, N. Nagappan, B. Nushi, and T. Zimmermann, "Software engineering for machine learning: A case study," in *Proceedings of the 41th International Conference on Software Engineering, ICSE 2019, Montral, Canada, May 25 - May 31, 2019*, 2019.

[16] C. Rosen and E. Shihab, "What are mobile developers asking about? a large scale study using stack overflow," *Empirical Software Engineering*, vol. 21, no. 3, pp. 1192–1223, 2016.

[17] K. Bajaj, K. Pattabiraman, and A. Mesbah, "Mining questions asked by web developers," in *Proceedings of the 11th Working Conference on Mining Software Repositories*. ACM, 2014, pp. 112–121.

[18] N. Meng, S. Nagy, D. Yao, W. Zhuang, and G. Arango-Argoty, "Secure coding practices in java: Challenges and vulnerabilities," in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 2018, pp. 372–383.

[19] "Tensorflow," accessed: 2019-01-21.

[20] "Pytorch," https://pytorch.org/, accessed: 2019-01-21.

[21] "Deeplearning4j," https://deeplearning4j.org/, accessed: 2019-01-21.

[22] *Stack Overflow data dump*, 2018, https://archive.org/details/stackexchange, accessed on Aug 15, 2018.

[23] B. L. Berg, H. Lune, and H. Lune, *Qualitative Research Methods for the Social Sciences*. Pearson Boston, MA, 2004, vol. 5.

[24] B. Ray, D. Posnett, V. Filkov, and P. T. Devanbu, "A large scale study of programming languages and code quality in github," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*, 2014, pp. 155–165.

[25] P. S. Kochhar, D. Wijedasa, and D. Lo, "A large scale study of multiple programming languages and code quality," in *IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering, SANER 2016, Suita, Osaka, Japan, March 14-18, 2016 - Volume 1*, 2016, pp. 563–573.

[26] D. Goldberg, "What every computer scientist should know about floating-point arithmetic," *ACM Computing Surveys (CSUR)*, vol. 23, no. 1, pp. 5–48, 1991.

[27] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, 2010, pp. 249–256.

[28] "Tensorboard: Visualizing learning," https://www.tensorflow.org/guide/summaries_and_tensorboard, accessed: 2019-01-21.

[29] "Visdom: A flexible tool for creating, organizing, and sharing visualizations of live, rich data." https://github.com/facebookresearch/visdom, accessed: 2019-01-21.

[30] Q. Guo, S. Chen, X. Xie, L. Ma, Q. Hu, H. Liu, Y. Liu, J. Zhao, and X. Li, "An empirical study towards characterizing deep learning development and deployment across different frameworks and platforms," in *Proceedings of the 34rd ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE 2019.

[31] "Build and install error messages," https://www.tensorflow.org/install/errors, accessed: 2019-01-21.

[32] "Mila and the future of theano," https://groups.google.com/forum/#!topic/theano-users/7Poq8BZutbY, accessed: 2019-01-21.

[33] S. Ma, Y. Liu, W.-C. Lee, X. Zhang, and A. Grama, "Mode: Automated neural network model debugging via state differential analysis and input selection," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 2018, pp. 175–186.

[34] B. Lin, F. Zampetti, G. Bavota, M. D. Penta, M. Lanza, and R. Oliveto, "Sentiment analysis for software engineering: how far can we go?" in *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, 2018, pp. 94–104.

[35] W. Fu, T. Menzies, and X. Shen, "Tuning for software analytics: Is it really necessary?" *Information & Software Technology*, vol. 76, pp. 135–146, 2016.

[36] T. Menzies, Z. Milton, B. Turhan, B. Cukic, Y. Jiang, and A. B. Bener, "Defect prediction from static code features: current results, limitations, new approaches," *Autom. Softw. Eng.*, vol. 17, no. 4, pp. 375–407, 2010.

[37] T. B. Le, L. Bao, and D. Lo, "DSM: a specification mining tool using recurrent neural network based language model," in *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*, 2018, pp. 896–899.

[38] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang, "Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems," *arXiv preprint arXiv:1512.01274*, 2015.

[39] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," in *Proceedings of the 22nd ACM international conference on Multimedia*. ACM, 2014, pp. 675–678.

[40] "Facebook and microsoft introduce new open ecosystem for interchangeable ai frameworks," https://research.fb.com/blog/2017/09/facebook-and-microsoft-introduce-new-open-ecosystem-for-interchangeable-ai-frameworks/, accessed: 2019-01-21.

[41] Y. Zhang, Y. Chen, S.-C. Cheung, Y. Xiong, and L. Zhang, "An empirical study on tensorflow program bugs," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2018, 2018, pp. 129–140.

[42] F. Thung, S. Wang, D. Lo, and L. Jiang, "An empirical study of bugs in machine learning systems," in *2012 IEEE 23rd International Symposium on Software Reliability Engineering*, 2012, pp. 271–280.

[43] X. Sun, T. Zhou, G. Li, J. Hu, H. Yang, and B. Li, "An empirical study on real bugs for machine learning programs," in *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*, 2017, pp. 348–357.

[44] C. B. Seaman, F. Shull, M. Regardie, D. Elbert, R. L. Feldmann, Y. Guo, and S. Godfrey, "Defect categorization: making use of a decade of widely varying historical data," in *Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement*. ACM, 2008, pp. 149–157.

[45] J. M. Zhang, M. Harman, L. Ma, and Y. Liu, "Machine Learning Testing: Survey, Landscapes and Horizons," *arXiv e-prints*, Jun 2019.

[46] L. Ma, F. Zhang, J. Sun, M. Xue, B. Li, F. Juefei-Xu, C. Xie, L. Li, Y. Liu, J. Zhao *et al.*, "Deepmutation: Mutation testing of deep learning systems," *The 29th IEEE International Symposium on Software Reliability Engineering (ISSRE)*, 2018.

[47] X. Xie, L. Ma, F. Juefei-Xu, M. Xue, H. Chen, Y. Liu, J. Zhao, B. Li, J. Yin, and S. See, "Deephunter: A coverage-guided fuzz testing framework for deep neural networks," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2019, 2019, pp. 146–157.

[48] X. Du, X. Xie, Y. Li, L. Ma, Y. Liu, and J. Zhao, "Deepstellar: Model-based quantitative analysis of stateful deep learning systems," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2019, 2019, pp. 477–487.

[49] X. Xie, L. Ma, H. Wang, Y. Li, Y. Liu, and X. Li, "Diffchaser: Detecting disagreements for deep neural networks," in *Proceedings of the 28th International Joint Conference on Artificial Intelligence*, 2019.

[50] L. Ma, F. Juefei-Xu, M. Xue, B. Li, L. Li, Y. Liu, and J. Zhao, "Deepct: Tomographic combinatorial testing for deep learning systems," in *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, Feb 2019, pp. 614–618.