Taxonomy of Real Faults in Deep Reinforcement Learning

***Abstract*—A growing demand is witnessed in both industry and academia for employing Deep Learning (DL) to solve real-world problems. Analysis of faults occurring in DL-based software systems is necessary, especially in safety-critical domains. Testing software systems that used Deep Reinforcement Learning (DRL), i.e., application of DL in Reinforcement Learning, has not yet received much attention from the community. In this paper, we present the first attempt to categorize faults in DRL projects. We manually analyzed 761 artifacts developed using well-known DRL frameworks (OpenAI Gym, Dopamine, Keras-rl, Tensorforce) to find bugs/issues. We labeled and taxonomized the identified faults during several rounds of discussions. We have merged our initial categories of DRL-specific faults to the most recent taxonomy of DL faults, to cover all types of faults occurring in DRL projects. Finally, we have validated our taxonomy through an online survey with 19 developers/researchers. A significant portion of the identified fault categories (10 out of 12) has been confirmed by at least half of survey participants.**

***Index Terms*—Deep Reinforcement Learning, Reinforcement Learning, Deep Learning, Faults, Software Testing, Taxonomy**

I. Introduction

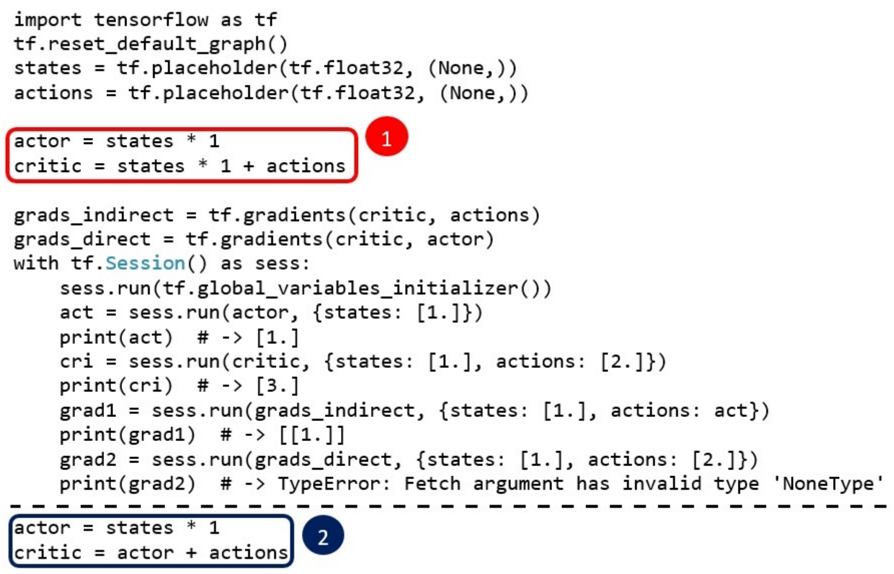
Applications of Deep Learning (DL) are growing in a variety of domains in both academia and industry. We are now seeing DL-based technologies being implemented in critical systems such as autonomous driving cars and medical diagnosis systems. Deep Reinforcement Learning (DRL) which is the application of DL in Reinforcement Learning (RL) is an active field of Machine Learning(ML), which exploits the capabilities of DL architectures to address previously unsolvable problems in RL. While DRL algorithms have been applied to problems in robotics, video games, finance and healthcare efficiently [1], their reliability is still a major source of concerns. Compare to traditional software systems, the notion of faults in DL-based software systems is more complex, because of their behavioral dependency on training data and hyperparameter settings.
In this paper, we aim to investigate the types of real faults in DRL projects to construct a taxonomy of these faults. Although other researchers have worked on categories of faults in DL programs [2], [3], to the best of our knowledge, this paper is the first research work to study the types of faults and their categories in DRL projects. Such taxonomy would help developers and researchers improve their understanding of the root cause and symptoms of faults. This would allow them to prevent common bugs during the development of DRL systems, prepare test cases, and improve DRL frameworks for facilitating development and testing.

Our methodology consists in a manual analysis of faulty software artifacts. First, we mined software repositories and Q&A forums, i.e., GitHub[1] and Stack Overflow[2](SO), to find relevant artifacts. In these artifacts, developers/researchers discussed and–or fixed issues that occurred while they were using popular DRL frameworks. We manually analyzed the artifacts and identified 761 issues. Next, we categorized the relevant issues through a multi-round labelling process. At the end, we obtained 12 distinctive types of faults that contain 28 faulty artifacts. We have merged this taxonomy to the most recent taxonomy of DL faults to cover not only DRL-specific faults, but all faults that could happen in DRL projects. We validated the obtained taxonomy through a survey with 19 participants who have various backgrounds and levels of expertise. A validated taxonomy of real faults occurring in DRL projects is the main contribution of this paper.

As a motivating example, Fig. 1 illustrates a faulty DRL program extracted from Stack Overflow post #44444923. The developer attempted to implement the DDPG algorithm and encountered an error message of "NoneType" in the *grads_direct* variable. The root cause of the bug is missing *actor* in calculating *critic* (i.e., 1), leading to "NoneType" in *grads_direct* variable. We have labelled this sample as "Wrong calculation of gradient" in our taxonomy. Since the magnitude and value of gradient is crucial for updating neural networks, *actor* and *critic* should be updated carefully. The correct statement from the accepted answer to this post is highlighted as 2. Moreover, Fig. 2 shows another faulty sample extracted from Stack Overflow post #47750291. The symptom is expressed as bad performance in terms of low reward by the developer. However, lack of enough exploration was determined to be the root cause in the accepted answer. Environment exploration is necessary for the success of RL agents, so this sample is identified as "Missing exploration" in our study. The missing part and recommended modification is indicated in the code by 1 and 2 respectively, according to the accepted answer.

The rest of the paper is organized as follows. Section II briefly reviews the related works. In Section III, we review Deep Reinforcement Learning and its approaches. We describe the methodology followed to construct and validate our proposed taxonomy in Section IV. A fully described taxonomy is reported in Section V, followed by a discussion in Section VI. We discuss threats to validity in Section VII and conclude the paper in Section VIII.

[1]https://www.github.com
[2]https://stackoverflow.com

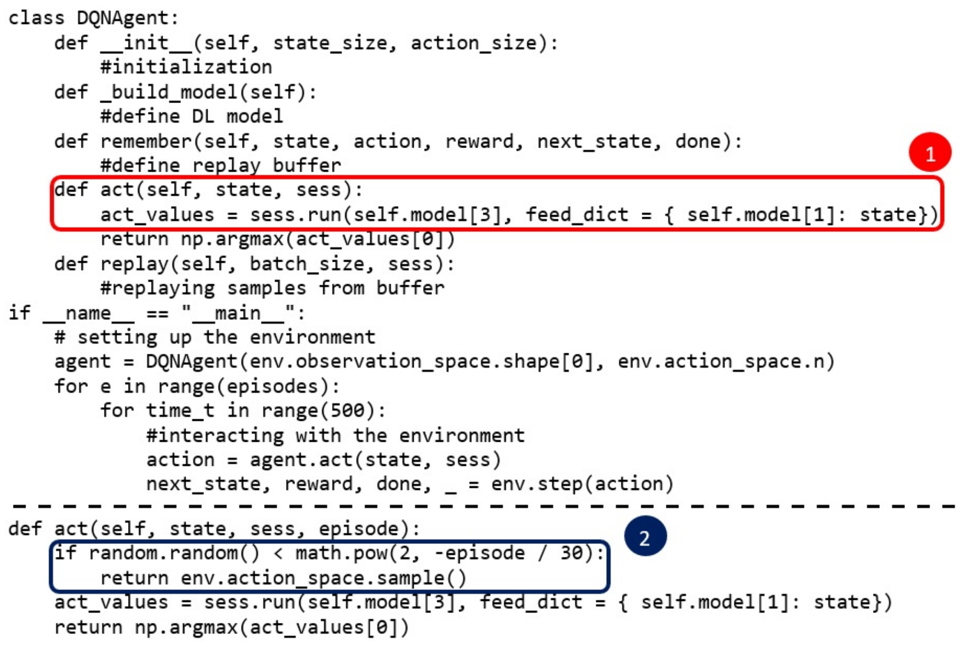


```
import tensorflow as tf
tf.reset_default_graph()
states = tf.placeholder(tf.float32, (None,))
actions = tf.placeholder(tf.float32, (None,))

actor = states * 1
critic = states * 1 + actions

grads_indirect = tf.gradients(critic, actions)
grads_direct = tf.gradients(critic, actor)
with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    act = sess.run(actor, {states: [1.]})
    print(act)  # -> [1.]
    cri = sess.run(critic, {states: [1.], actions: [2.]})
    print(cri)  # -> [3.]
    grad1 = sess.run(grads_indirect, {states: [1.], actions: act})
    print(grad1)  # -> [[1.]]
    grad2 = sess.run(grads_direct, {states: [1.], actions: [2.]})
    print(grad2)  # -> TypeError: Fetch argument has invalid type 'NoneType'
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
actor = states * 1
critic = actor + actions
```

Fig. 1. Example of *"Wrong gradient calculation"* fault from SO_44444923



```
class DQNAgent:
    def __init__(self, state_size, action_size):
        #initialization
    def _build_model(self):
        #define DL model
    def remember(self, state, action, reward, next_state, done):
        #define replay buffer
    def act(self, state, sess):
        act_values = sess.run(self.model[3], feed_dict = { self.model[1]: state})
        return np.argmax(act_values[0])
    def replay(self, batch_size, sess):
        #replaying samples from buffer
if __name__ == "__main__":
    # setting up the environment
    agent = DQNAgent(env.observation_space.shape[0], env.action_space.n)
    for e in range(episodes):
        for time_t in range(500):
            #interacting with the environment
            action = agent.act(state, sess)
            next_state, reward, done, _ = env.step(action)
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
def act(self, state, sess, episode):
    if random.random() < math.pow(2, -episode / 30):
        return env.action_space.sample()
    act_values = sess.run(self.model[3], feed_dict = { self.model[1]: state})
    return np.argmax(act_values[0])
```

Fig. 2. Example of *"Missing exploration"* fault from SO_47750291

II. Related Work

Although the number of studies on RL, DRL and DRL-based software systems has increased dramatically over the past decade, far less research has focused on testing RL systems. To the best of our knowledge, Trujillo et al. [4] is the very first work in this field. They have used neuron coverage as a well-known whitebox testing technique to investigate the evolutionary coverage of deep networks for the specific case of DRL [4]. Two different models of Deep Q-Network (DQN) have been tested for the Mountain Car Problem (MPC). Results revealed that observing good neuron coverage does not necessarily mean success in a RL task in terms of reward. A negative correlation is observed in their results and this is in contrast to typical DNNs confirming that neuron coverage is not capable of properly assessing the design or structure of DRL networks. The best possible coverage is achieved by extensive exploration which is not efficient in DRLs and does not help to maximize the reward.

There are some other research works that considered fault specifically in DL-based systems. A number of DL applications developed using TensorFlow have been studied by

Zhang et al. [5]. In their empirical study, they explored SO and Github to select faulty applications. From the selected faulty applications they obtained 175 TensorFlow related bugs. They manually examined the bugs to understand the challenges and strategies followed by developers to detect and localize the faults in the TensorFlow-based applications. They have presented some insights for two aspects of these faults: the root causes of bugs and the bugs impact on the application behaviour. Finally, authors classified the root causes and symptoms into seven and four different types, respectively. In this study, DRL applications that use some popular DRL frameworks, including OpenAI Gym, TensorForce, Dopamine and Keras-rl, have been studied. The popularity of these frameworks makes them representative of the state-of-the-practice in this field. There is another methodological difference in the approach followed to mine SO posts: while Zhang et al. considered SO posts with at least one answer, we consider posts with an accepted answer. This additional condition is important to ensure that the studied faults were successfully solved and their solution was accepted by a developer. Moreover, their 175 collected bugs include generic faults, while our taxonomy covers only DRL specific faults. Finally, we did not restrict this study to the authors' analysis and have validated the presented taxonomy by conducting a survey with participation of DRL developers/researchers.
Another characterisation of DL bug was reported by Islam et al. [3]. Their study aimed to discover the most frequent types of bugs in DL programs as well as their causes and symptoms. A common pattern among bugs and its dynamics over the time has been studied as well. Similar to us, they have investigated a number of SO and GitHub bugs related to five DL frameworks: Theano, Cafe, Keras, TensorFlow, and PyTorch. They leveraged the list of root causes of bugs reported by Zhang et al. [5] to categorize the causes of bugs in their study. In contrast to our study, they analyzed various fault patterns and the correlation/distribution of bugs in different frameworks, and did not construct a taxonomy of faults.
The most recent and related research to this study is a validated taxonomy of real faults in DL systems presented in [2]. Humbatova et al. [2] have constructed their taxonomy based on two sources of information: 1) manual analysis of Github artifacts and SO posts; and 2) interviews with developers/researchers. Their introduced taxonomy consists of 5 main categories containing 375 instances of 92 unique types of faults. The taxonomy is validated by conducting a survey with a distinct group of developers/researchers who verified the completeness and usefulness of the identified categories. We will discuss the relation between our taxonomy and Humbatova et. al's taxonomy in Section V.

III. Deep Reinforcement Learning

Machine Learning (ML) is classified into three main branches: Supervised Learning, Unsupervised Learning and Reinforcement Learning [6]. Supervised learning is mainly about inferring a classification/regression from labeled training data. The main goal of unsupervised learning is to draw inference from unlabeled input data. In RL, an agent interacts with an environment and the task consists in learning how to perform sequences of actions in the environment to maximize cumulative returning rewards. The agent aims to learn good behavior; meaning that it modifies or attempts new behaviours incrementally. Moreover agent uses trial-and-error experience, i.e., frequent interactions with the environment and information collection [1]. In other words, RL basically aims to handle the automatic learning of optimal decisions over time [6], [7].

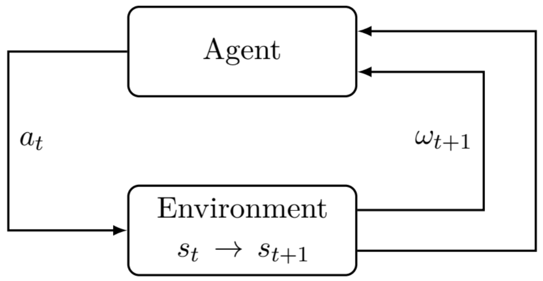


Fig. 3. Agent interacting with its environment [1].

Formally, the RL problem is formulated as a discrete-time stochastic control process in the following way: At each time step t, the agent has to select and perform an action $a_t \in A$. Upon taking the action, (1) the agent is rewarded by $r_t \in R$, (2) the state of environment is changed to $s_{t+1} \in S$, and (3) the agent perceives the next observation of $\omega_{t+1} \in \Omega$. Fig. 3 illustrates such agent-environment interaction. An RL agent is defined to find a policy $\pi \in \Pi$ that maximizes the expected cumulative reward (a discount factor $\gamma \in [0, 1]$ applies to the future rewards) [1]. The policy function could be either deterministic or stochastic. Deterministic policy function indicates the agent's action given a state: $\pi(s) : S \rightarrow A$. In the case of stochastic policy, $\pi(s, a)$ indicates the probability of choosing action a in state s by the agent.
Recently, researchers have successfully integrated DL methods in RL to solve some challenging sequential decision-making problems [8]. This combination of RL and DL is known as deepRL or DRL. DRL benefits from the advantages of DL in learning multiple levels of representation among data to address large state-action spaces with low prior knowledge, e.g., a DRL agent has successfully learned from raw visual perceptual inputs including thousands of pixels [9]. As a consequence, imitating some human-level problem solving capabilities becomes possible [10], [11].

A. Value-based approaches

The value-based algorithms in RL aim to build a value function, which subsequently makes it possible to define a policy. The value function for a state is defined as the total amount of discounted reward that an agent expect to accumulate over the future, starting from that state. Q-learning algorithm [12] is the simplest and most popular value-based algorithm. In the basic version of Q-learning, a lookup table of Q-values, $Q(s, a)$, with one entry for every state-action pair is used to approximate the value function. To learn the optimal Q-value function, the Q-learning algorithm uses an incremental approach by updating Q-values with the following update rule:

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha[r_{t+1} + \gamma \max_{a'} Q(s_{t+1}, a') - Q(s_t, a_t)]$$

where α is a scalar step size called the learning rate. In this technique, the gathered Q-values regarding various actions and state are updated after taking an action. Hence, the optimal policy is defined as:

$$\pi(s) = \arg\max_{a \in A} Q(s, a)$$

The idea of *value-based deep reinforcement learning* is approximating the value function by a deep neural network. Mnih et al. [9] introduced the deep Q-network (DQN) algorithm that obtained human-level performance in an online playing of ATARI games. They have defined the state as the stack of four consecutive frames and actions as various joystick positions. The deep network consists of multiple convolutional and fully-connected layers. DQN uses two heuristics to address its instabilities: 1) using target network: instantiating fitted Q-network for some iterations and applying network's parameters update only periodically, and 2) replay memory (buffer): keeping all information of several previous steps and replaying them (as mini-batch) to reduce variance.

B. *Policy gradient approaches*

Policy gradient methods maximize a performance objective (typically the expected cumulative reward) by discovering a good policy. Basically, the policy function is directly approximated by a deep neural network meaning that the network output would be (probability of) actions instead of action values (say estimated rewards). It is acknowledged that policy-based approaches converge and train much faster specially for problems with high-dimensional or continuous action spaces [13]. The direct representation of a policy to extend DQN algorithms for addressing the restriction of discrete actions was introduced by Deep Deterministic Policy Gradient (DDPG) [14]. This algorithm updates the policy in the direction of the gradient of Q which is a computationally efficient idea.
Another approach is using an actor-critic architecture which benefits from two neural network function approximators: an actor and a critic. The actor denote the policy and the critic is estimating a value function (e.g., the Q-value function). Asynchronous advantage actor-critic (A3C) algorithm [15] can employ both feed-forward and recurrent neural approximators to learn tasks in continuous action spaces, working both on 2D and 3D games.

C. *Review of frameworks for developing DRL programs*

To select open source repositories for this study, we identify GitHub repositories that focus on RL and sort them by their number of stars. The number of stars of a repository is considered as one of the most important metrics for identifying repository popularity [16]. The number of stars identifies the number of users who used and find the repository interesting. Next, we filter out repositories that are not active today or do not have a software artifact. We choose the repositories with software artifacts, because we need real code snippets, users and bugs information, to be able to create a taxonomy. Besides, we consider a repository as an active one where it is supported to resolve the issues which are submitted to that repository. Table I presents detailed information about the studied repositories. *OpenAI Gym*, the most popular repository in our list, is a toolkit that provides a set of standardized environments for developing RL algorithms [17]. it is supported by *OpenAI* and was first released in 2016. Dopamine is another popular repository. It is a framework for prototyping RL algorithms which is developed by *Google* and released since 2018 [18]. Keras-rl is also a framework providing RL algorithms for *keras*. It was launched in 2016 [19]. Last but not least, Tensorforce is a *tensorflow* library for applied RL algorithms [20].

TABLE I
DETAILED INFORMATION OF SELECTED FRAMEWORKS

Project Name	***stars***	***commits***	***issues***	***contributors***
OpenAI Gym	$22k$	1,217	1,179	248
Dopamine	$9.1k$	197	118	8
keras-rl	$4.8k$	308	214	39
Tensorforce	$2.7k$	1,979	512	60

IV. METHODOLOGY

In this section, we describe the adopted methodology in this paper to construct and validate the proposed taxonomy of faults in DRL projects. Fig. 4 presents the main steps of our methodology.

A. *Manual Analysis of DRL programs*

We considered four popular DRL frameworks: OpenAI Gym, Tensorforce, Dopamine, keras-rl and two main sources of information: GitHub and StackOverFlow. According to GitHub information released in 2019, GitHub has more than 50 million users and about 100 million software repositories [21]. Thus it can be considered as the most important source of open source software artifacts. On the other hand, Stack Overflow is the largest Q&A website for developers in the stackExchange network. To construct our initial taxonomy, we manually analyzed issues from GitHub and SO discussions related to the four selected frameworks.

1) Mining GitHub and Stack Overflow: SO posts are our main source of information for bugs/issues related to DRL. We searched SO questions with five tags: four tags for our targeted frameworks (one tag for each one) and one general tag of `[reinforcement-learning]`. This search returned a total of 2072 posts. Next, we excluded questions without at least one ”accepted” answer. This filtering step is important to ensure that only questions with a verified answer are analyzed in our study. This process left us with 329 SO

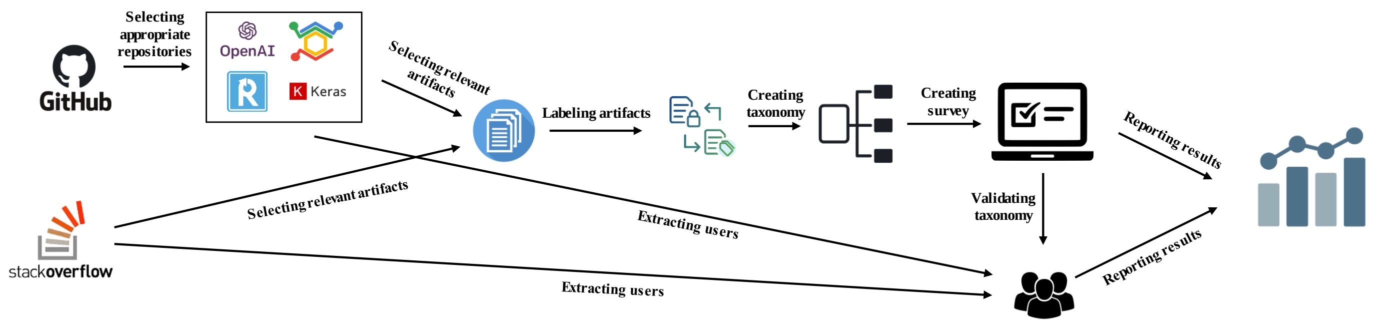


Fig. 4. The main steps of our methodology

posts with at least one accepted answer.
We have used GitHub issue tracker to investigate issues of the four DRL frameworks targeted in this study. Usually, bugs and faults related to the development of a framework are mentioned over the repository's issue tracker. But in some cases, developers post bugs found while using a framework over the GitHub issue tracker of that framework. This is why we also referred to GitHub repositories. For each framework, issues likely related to fixing problems were collected. We extracted all issues and only the ones labeled as "closed" in our study. This decision aimed to ensure that we analyze only issues that were solved. Finally, we identified 110 issues for Dopamine, 151 for OpenAI Gym, 200 for Keras-RL, and 300 for Tensorforce.

2) Manual labeling: We manually analyzed all collected data from SO and GitHub. Similar to Humbatova et. al [2] we have used an open coding procedure [22]. A shared document including the link to all artifacts have been used to make it possible for all authors to work together during the labeling process. Each artifact was inspected by reading specific parts of its document (code snippet, comment, description) and all related discussion provided by the owner or other users. We removed the underlying artifact from our analysis if: 1) it was not related to a bug-fixing activity, 2) it was related to an issue in the used framework itself, not the DRL program, 3) it was related to a common programming error not a DRL-specific bug and 4) the root cause of the fault was not clear for the authors, based on the associated information of artifact or authors' analysis.
We performed the labeling process in three rounds. For the first round, one author labelled all the documents by his own defined descriptive label. In this round, we considered only leaf groups by defining a descriptive label without proposing a hierarchical taxonomy. Then, other authors commented on his labelling by proposing a new label where applicable. In this second round, a hierarchical taxonomy was developed by grouping similar leaves. We kept all the labels on our shared document for further discussion and reuse. For the final round, all the labelled documents and the taxonomy were investigated in a team meeting. Conflict resolution was performed in this round. Overall, we found about 47 bugs/issues reported by users when working with DRL, about 86% of them were found from SO posts. We ended up with 25 real labelled faults in DRL projects in 10 distinctive leaf categories.

B. Building and validating the taxonomy

Similar to [2], we have used a bottom-up approach [23] to categorize the faults and construct the taxonomy. First, similar labels are grouped into categories. For each category, we did a double-check by investigating all artefacts in the category to make sure that they are classified correctly similar to labelling rounds. Afterward, parent categories are built based on the "is-a" relationship between each category and its subcategories. During the process, we have discussed each modified version of the taxonomy in virtual meetings with all authors. At last, the taxonomy was finalized in a virtual meeting by exploring all categories, subcategories and leaf nodes.
A validation process is required to ensure that the taxonomy is well-organized and covers real faults in the DRL program. We conducted a survey to validate the taxonomy involving DRL practitioners/researchers. We targeted practitioners or researchers with a good experience in RL and DRL. Similar to our method for building the taxonomy, we used GitHub and Stack OverFlow to extract information about suitable survey participants. To find participants with a good understanding of RL, we first identified the most popular repositories that focused on RL. Next, we filtered out the ones that do not have software artifacts. After this filtering, we extracted and sorted repositories' contributors based on their activity in the selected repositories. Next, we selected developers who have been active in GitHub during the last years. To recruit participants from SO, we used the posts extracted during the mining phase, i.e., the posts that have `reinforcement-learning` tag and at least one accepted answer. Next, we identify the users who posted accepted answers on the selected posts and sort them based on their up-vote and down-vote. Since we could not access the email address of SO users, we carried out a search for each identified user in the web to find their profile and emails from other sources such as GitHub and Google Scholar. In the end, we obtained a list of 210 users; 170 from GitHub and 40 from SO. We sent out the survey

by email to all of them and 19 persons participated in our survey which corresponds to a participation rate of 9%. We received responses from 8 researchers and 11 developers. The minimum coding experience for ML/DL and DRL were "1-3 years" and "less than 1 year", respectively. The most experienced participant had more than 5 years of experience in both ML/DL and DRL fields while the median for ML/DL and DRL was "3-5 years" and "1-3 years", respectively.

Our survey was created using Google Forms [24], a well-known online tool for creating and sharing online surveys and quizzes. The survey started by some background questions about job title, DL/DRL-specific programming experience and familiar languages/frameworks. Then, particular questions about our taxonomy were asked. We put distinctive multiple-choice questions for each of our 12 leaf nodes including a short description of the corresponding faults. In each question, the participants are asked to answer whether they have seen such fault in their own experience or not. The positive answer option asked them to also select a severity level for the fault (minor/major) and the required amount of effort to detect/fix the fault (easy/hard). This allowed us to assess the observed occurrence and the perceived severity of faults provided in our taxonomy, by developers/researchers, at the same time. At the end, we asked if the participant has observed any problems related to DRL that have not been considered in the survey and her availability for an interview to discuss the subject in detail. By these final questions we aimed to find out missing bugs/issues in our presented taxonomy and identify possibilities of further investigation.

V. The taxonomy

We have prepared a replication package that includes the materials used in this study and anonymized data collected during our survey [25].

Prior to describing our proposed taxonomy for faults in DRL, we should discuss the relation between the faults we investigate in this paper and the ones observed in DL programs. Thanks to a recent interesting research that categorizes DL-specific faults [2], we can discuss this relation systematically. This taxonomy of DL faults is constructed based on the faults observed in three popular DL frameworks (Tensorflow, PyTorch and Keras). The authors ended up with five main categories of faults, namely Tensors and inputs, Model, GPU usage, Training and API. These faults could happen in any DL program regardless of the application domain including DRL programs. For example, selecting a wrong architecture for a network, missing a specific layer, wrong initialization of network's parameters or wrong loss definition could be observed in any DL program. Therefore, we have excluded such faults in our study as it is described in Section IV focusing on DRL-specific faults. However, we have blended our proposed taxonomy of DRL faults with the DL fault taxonomy introduced in [2] to 1) make the taxonomy self-contained in terms of faults that may happen in DRL programs and 2) help the reader to understand the significance and hierarchy of various faults. There is an exception for faults related to training data. Since in RL, an agent is interacting with an environment and the learning process performs based on information gathered through interactions, traditional "training data" which has been used in classification tasks is not relevant to DRL programs. Hence, we removed faults categories related to "training data" including preprocessing and quality issues. Fig. 5 illustrates the final taxonomy. DRL-specific sub-taxonomy includes 4 categories and 12 leaves. The numbers after each title name represent the number of posts assigned to such title during manual labelling. In the following, we first briefly describe DL faults imported from [2]. Then, our proposed categories of DRL faults will be discussed.

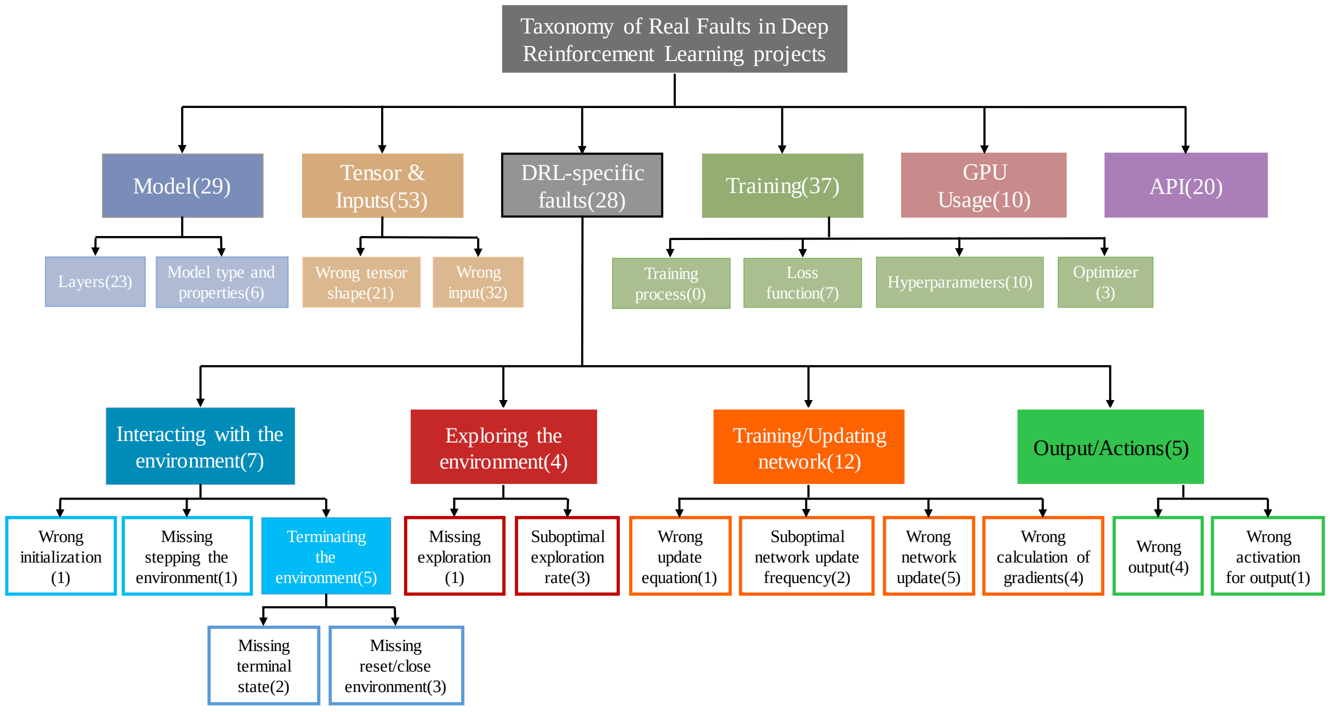
A. *DL faults*

According to [2], there are five main categories of faults in DL programs: Model, Tensors and Inputs, GPU Usage, Training and API. Any fault related to the structure and properties of the DL model is classified as **Model**. It has two subcategories of *Model type & properties* and *Layers*. Faults related to the model as a whole not its components are categorized as Model type & properties. For example, wrongly selected model for a task, too few or too many layers in a model which is identified as suboptimal network structure, or wrong initialization of weights in the network. *Layers* covers faults related to a particular layer as a component of a model, e.g., missing average pooling layer after convolution layer (s), missing activation function or mismatch of layers' dimensions.

Tensors and Inputs gathers issues related to a wrong dimension, type or format of data processed by the network and has two subcategories: *Wrong tensor shape* and *Wrong input*. The former covers bugs raised by operation on tensors with incompatible or incorrectly defined shape, e.g., using a transposed tensor instead of the normal one. Incompatible format, type or dimension of input data to a layer or methods result in faults of *Wrong input*.

All faults that are encountered during the training process of DL programs are categorized as **Training**. *Hyperparameters* covers problems due to tuning the hyperparameters of the DL model including learning rate, number of epochs and batch size. Wrong calculation, wrong selection or missing a loss function leads to faults from the subcategory of *Loss function*, which affect the effectiveness of learning algorithms. *Optimizer* includes faults like wrong selection or bad parameter setting of the model training optimizer. Faults such as attempting to fit a too big model into memory or to refer to non-existing checkpoint during model restoration fall in the subcategory *Training model*.

Usually, DL frameworks employ GPU devices to run the code, so the **GPU Usage** subcategory includes faults that occur while using such devices. For instance, wrong reference to a GPU device or faulty data transfer to a GPU device. Finally, failures related to the wrong usage of framework APIs (i.e., using an API in a way that does not comply with its definition) fall into the **API** subcategory.



Taxonomy of Real Faults in Deep Reinforcement Learning projects
Model(29)
Tensor & Inputs(53)
DRL-specific faults(28)
Training(37)
GPU Usage(10)
API(20)
Layers(23)
Model type and properties(6)
Wrong tensor shape(21)
Wrong input(32)
Training process(0)
Loss function(7)
Hyperparameters(10)
Optimizer (3)
Interacting with the environment(7)
Exploring the environment(4)
Training/Updating network(12)
Output/Actions(5)
Wrong initialization (1)
Missing stepping the environment(1)
Terminating the environment(5)
Missing exploration (1)
Suboptimal exploration rate(3)
Wrong update equation(1)
Suboptimal network update frequency(2)
Wrong network update(5)
Wrong calculation of gradients(4)
Wrong output(4)
Wrong activation for output(1)
Missing terminal state(2)
Missing reset/close environment(3)

Fig. 5. Taxonomy of real faults in DRL projects.

B. *Interacting with the environment*

In a DRL program, an agent must interact with an environment to learn. Usually in each DRL program, there are some parts for initializing, retrieving current state, submitting action and receiving next state and the reward. DRL frameworks support some well-known environments with the possibility of defining new environments. They provide developers with APIs to work with the environments. This category considers faults that occur when a DRL program tries to interact with the environment.

1) Wrong initialization: Problems related to initializing the environment in a wrong way (including parameters, initial state, actions) are classified in this category. Prior to working with the environment, it must be properly initialized. A failure to properly initialize the environment will lead to problems during further interactions with the environment. As an example, we have observed missed initialization functions during our manual analysis.

2) Missing stepping the environment: Failure to timely push the environment to a new state and get the associated reward lead to problems in learning. During each episode of interaction between an agent and its environment, the environment must be moved to a new state getting the reward associated with the transition. Otherwise, the developer would lose the track of reward or sequence of the environments' states.

Terminating the environment: This subcategory includes faults related to the ending of an episode of interaction with the environment or restarting the environment for the next episode.

3) Missing terminal state: This category contains problems related to missing or wrongly detected terminal state of the environment. Finally, each episode of agent interaction with its environment should be terminated normally by reaching the terminal state. Missing this state may lead to inefficient learning. For example, using default definitions of the framework's API for detecting the terminal state leads to such fault.

4) Missing reset/close environment: It includes problems related to missing or bad termination (and restarting) of each round of agent interaction with its environment. At the end of each episode, the environment must be properly closed or reset to its default configurations for the next episodes. Actually, execution of the next episode and correctness of state sequences depend on the successful termination of the previous episode. Wrongly positioned API call for resetting environment is an example of this type of faults.

C. *Exploring the environment*

A RL agent must prefer high-reward actions that have been tried previously to obtain more reward. On the other hand, to detect such actions, the agent must investigate actions that have not been taken before. The agent not only has to exploit its experience of actions with higher reward, but also has to explore to improve action selection in the future. The dilemma is that neither exploration nor exploitation can be successful exclusively without failure. A variety of actions has to be attempted by the agent, gradually favoring those that appear to be best. Although there are various methods to perform exploration in DRL, enough exploration of the environment is crucial for an effective performance. Faults in this category fall into two groups: missing the exploration phase in the code and choosing suboptimal exploration rate.

1) Missing exploration: This tag is identified as the failure to explore the environment in the case that it is necessary according to the algorithm. Lack of exploration leads to poor performance of the algorithm in terms of mean reward. Sometimes developers rely on the output of the neural network (e.g., DQN) to have enough flexibility to cause sufficient exploration but using explicit methods like epsilon greedy is more effective.

2) Suboptimal exploration rate: It is widely acknowledged in the RL community that balancing exploration and exploitation is crucial to achieving good performance [26]. This tag covers problems related to suboptimal exploration parameters (e.g., epsilon in epsilon-greedy method) or suboptimal decay rate that leads to poor performance of the algorithm.

D. Training/Updating network

This is the largest category in our taxonomy. It includes faults related to training or updating deep neural networks in DRL programs. According to the adopted DRL architecture/algorithm, various updating or training procedures should take place in DRL programs, e.g., training Q-network and updating target network in DQN, policy and value networks in policy gradient algorithms, and updating replay buffer.

1) Wrong update rule: A new experience from the environment should be incorporated into the existing experiences by an update rule. This leaf covers issues related to using an incorrect update rule for value or policy function including suboptimal learning rate and wrong implementation of the update rule.

2) Suboptimal network update frequency: Problems related to suboptimal update frequency of networks' parameters (including the target network) leading to unstable learning and increasing loss value are categorized in this group. For example, in DQN, the target network must be updated frequently, so choosing a too high update rate can lead to unstable learning process.

3) Wrong network update: This tag covers faults related to the wrong update of networks or its parameters. Each network or set of parameters (like Q and target network in DQN or policy and value networks in policy gradient algorithms) must be properly updated based on new values or recent observations. Examples are update the wrong network, wrong update statement and missing the update statement(s).

4) Wrong calculation of gradients: Problems related to wrong calculation of gradients for learning, including computation of one network's gradients with respect to another network's are categorized in this tag. Since in some complicated DRL methods, different networks must be trained according to the output of other networks or returning reward from the environment, it is a quite frequent fault in such methods.

E. Output

When DL is applied to typical machine learning problems like classification, the output of the network is much familiar in comparison to RL problems. A particular action, a vector of state-action values or a probability distribution over possible actions could be the output of deep neural networks in DRL programs. Due to this diversity, faults related to the network output are observed in our study. We have categorized these faults in two leaves.

1) Wrong output: This tag includes issues related to failure to define a correct output layer for the network with respect to the environment and algorithm. This type of faults leads to issues in determination of actions and poor learning.

2) Wrong activation for output: Failure to define a correct activation function for the output layer leading to incorrect action determination is labelled in this group. As a real example, if sigmoid activation function is used when the output is expected to be a reward value, the activated output range will be different from the original reward.

F. Validation results

Table II shows the results of the validation survey for the proposed taxonomy. The ratios of answers for each category are reported. The answers reveal whether or not the participants ever faced the related faults, their perceived severity for each category of fault (as "minor/major"), and their estimation of the effort required to fix the faults (as "easy/hard to fix"). The results confirm the relevance of all categories in the taxonomy since all of them have been encountered by at least 37% of participants. *Exploring the environment* is the most popular category; it has been experienced by 89% of participants. Participants have determined *Missing exploration* as the most critical faults by 63% of answers (highest rate for "Yes, major and hard"). The least popular type is *Missing stepping the environment* experienced by 37% of all the survey participants which is a non negligible fraction. The average of answers that included "yes" is 71%. This result show that the taxonomy covers relevant types of faults that has been experienced by most of the participants as DRL developers/researchers. However, 2 types of faults received more than 50% "no" answers as the least observed type of faults in DRL. Consequently, participants validated 10 categories out of 12 on average, ratio of categories that received more than 50% of responses that included "yes". We have also received 8 positive answers for attending an interview.

In response to our question about the completeness of our taxonomy, some participants mentioned examples of faults they thought we missed in the reported taxonomy. Generic coding problems were commented by two participants (an example is: *"Wrong data type when storing the environment state somewhere"*), the effect of the fault was described in two others rather than the root cause of symptom (for example, *"diverging behaviour and instability"* and *"not converging entropy/gradients"*), and one reported faults related to DL and which are not specific to DRL (such as *"neural network initialization"*). The remaining three participants commented on the definition and formulation of a DRL problem including evaluation metrics and reward function(e.g. *"Problems related*

TABLE II
RESULTS OF VALIDATING SURVEY

Type of faults	Responses				
	No	Yes, minor and easy	Yes, minor but hard	Yes, major but easy	Yes, major and hard
Wrong initialization	11%	58%	11%	16%	5%
Missing stepping the environment	63%	16%	5%	11%	5%
Missing terminal state	42%	16%	16%	21%	5%
Missing reset/close environment	37%	53%	0%	11%	0%
Missing exploration	11%	5%	5%	16%	63%
Suboptimal exploration rate	11%	16%	16%	16%	42%
Wrong update rule	11%	21%	11%	26%	32%
Suboptimal network update frequency	16%	0%	26%	21%	37%
Wrong network update	16%	0%	26%	21%	37%
Wrong calculation of gradients	32%	32%	5%	16%	16%
Wrong output	53%	32%	0%	11%	5%
Wrong activation for output	42%	32%	5%	21%	0%

to defining reward systems for new environments for efficient learning.").

VI. Discussion

The number of research works in software engineering that use SO as a source of information is growing which raises some concerns about quality [27]. The problem is the validity of its utility and reliability which has not been investigated yet. Humbatova et al. [2] also reported the views of some of their SO interviewees stating that Q&A over SO does not reflect the problems developers/researchers faced when developing DL programs. To compensate for this issue, we have excluded non-relevant artifacts during our manual analysis, have investigated each artifact by participation of more than one evaluator and have conducted a survey to validate the results. Participants in our online survey have provided us with some comments. We present and discuss common problems among these comments:

1) Reproducibility of result : Stochastic nature of RL problems make it difficult to reproduce the results and investigate the correctness of implementation. One of them mentioned that: *"finding a good random seed is annoying, even worse is the high sensitivity to different seeds: performance may greatly vary from one seed to another"*. Another participant stated that sometimes one would not get good results just because of bad luck in a run.
2) Complexity of DRL frameworks: some participants complained that DRL frameworks are quite complicated due to the modular design. So, they prefer to to write all the code from scratch, one participants noted that *"Writing all code from scratch seems to be the only way to have full control of what's going on. I believe we can do better"*. This makes it difficult to develop, debug and test DRL algorithms. They stated the advantages of single-file implementations to be their easy development and debugging. Moreover, they expected to see new frameworks including such features (e.g., *"I am a huge advocate for single-file implementations. Easier to debug, inspect, and develop new algorithms."*).
3) Scalability challenge: based on participant views, it seems that scalable experiment is a challenge in DRL projects. More effort is required to address this issue, for example to handle a reasonable number of agents in multi-agent problems.

VII. Threats to Validity

There are internal and external threats to the validity of this research. Our results could be affected by biased when mining and labelling the artifacts. To address this issue, a clear systematic approach is followed in our study and mining/labeling is performed and evaluated by at least two of the authors through various rounds. We have extracted only "closed" issues from GitHub and with "at least one accepted" answer from SO ensuring that we analyzed only issues that were solved. Moreover, we conducted a survey to validate the proposed taxonomy with participants who have not been involved in the process of constructing it. Although we constructed the taxonomy using artifacts produced when developing four frameworks, we kept the title and description of fault types as general as possible. Therefore, popular frameworks were selected and we kept the categories as framework-independent as possible during labelling. Participants with different levels of expertise and background have validated the taxonomy as a means to address this threat. It is also possible that our questions and presented categories in the survey affected participant's view directing them toward our proposed types of faults. To address this concern, we asked participants at the end of our survey to freely comment on our results and mention potential missing faults/issues in our study.

VIII. Conclusion

In this paper, a taxonomy of real faults in the DRL project has been proposed. The methodology was manual analysis of faulty software artifacts from SO and GitHub projects developed using four selected DRL frameworks (OpenAI Gym, Dopamine, Keras-rl, Tensorforce). We manually analyzed the artifacts and identified 761 issues. Then, we categorized the relevant issues through a multi-round labelling process. Finally, we obtained 12 distinctive types of faults that contain28 faulty artifacts. We have merged this taxonomy with the most recent taxonomy of DL faults to cover not only DRL-specific faults but all faults that could happen in DRL projects. The validation of the taxonomy has been performed by conducting a survey with 19 participants who have various backgrounds and levels of expertise in RL. The results have confirmed the relevance of identifying those types of faults in DRL projects.

References

[1] V. François-Lavet, P. Henderson, R. Islam, M. G. Bellemare, and J. Pineau, *An Introduction to Deep Reinforcement Learning*, vol. 11. 2018.

[2] N. Humbatova, G. Jahangirova, G. Bavota, V. Riccio, A. Stocco, and P. Tonella, "Taxonomy of real faults in deep learning systems," in *The 42nd International Conference on Software Engineering (ICSE 2020)*, ACM, 2020.

[3] M. J. Islam, G. Nguyen, R. Pan, and H. Rajan, "A comprehensive study on deep learning bug characteristics," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 510–520, 2019.

[4] M. Trujillo, M. Linares-Vásquez, C. Escobar-Velásquez, I. Dusparic, and N. Cardozo, "Does neuron coverage matter for deep reinforcement learning? a preliminary study," in *DeepTest Workshop, ICSE 2020*, 2020.

[5] 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*, pp. 129–140, 2018.

[6] M. Morales, *Grokking Deep Reinforcement Learning*. Manning Publications, 2019.

[7] M. Lapan, *Deep Reinforcement Learning Hands-On: Apply modern RL methods, with deep Q-networks, value iteration, policy gradients, TRPO, AlphaGo Zero and more*. Packt Publishing Ltd, 2018.

[8] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[9] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, *et al.*, "Human-level control through deep reinforcement learning," *nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[10] D. Gandhi, L. Pinto, and A. Gupta, "Learning to fly by crashing," in *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 3948–3955, IEEE, 2017.

[11] M. Moravčík, M. Schmid, N. Burch, V. Lisỳ, D. Morrill, N. Bard, T. Davis, K. Waugh, M. Johanson, and M. Bowling, "Deepstack: Expert-level artificial intelligence in heads-up no-limit poker," *Science*, vol. 356, no. 6337, pp. 508–513, 2017.

[12] C. J. Watkins and P. Dayan, "Q-learning," *Machine learning*, vol. 8, no. 3-4, pp. 279–292, 1992.

[13] F. Agostinelli, G. Hocquet, S. Singh, and P. Baldi, "From reinforcement learning to deep reinforcement learning: An overview," in *Braverman Readings in Machine Learning. Key Ideas From Inception to Current State*, pp. 298–328, Springer, 2018.

[14] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," *arXiv preprint arXiv:1509.02971*, 2015.

[15] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," in *International conference on machine learning*, pp. 1928–1937, 2016.

[16] H. Borges, A. Hora, and M. T. Valente, "Understanding the factors that impact the popularity of github repositories," in *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 334–344, IEEE, 2016.

[17] "Openai gym." "https://github.com/openai/gym", 2016.

[18] P. S. Castro, S. Moitra, C. Gelada, S. Kumar, and M. G. Bellemare, "Dopamine: A Research Framework for Deep Reinforcement Learning," 2018.

[19] M. Plappert, "keras-rl." https://github.com/keras-rl/keras-rl, 2016.

[20] A. Kuhnle, M. Schaarschmidt, and K. Fricke, "Tensorforce: a tensorflow library for applied reinforcement learning." Web page, 2017.

[21] "Github official website." https://github.com/about. Accessed: 2020-8-25.

[22] C. B. Seaman, "Qualitative methods in empirical studies of software engineering," *IEEE Transactions on software engineering*, vol. 25, no. 4, pp. 557–572, 1999.

[23] G. Vijayaraghavan and C. Kaner, "Bug taxonomies: Use them to generate better tests," *Star East*, vol. 2003, pp. 1–40, 2003.

[24] "Google forms." https://www.google.ca/forms/about/, 2020.

[25] "Replication package." https://github.com/deepRLtaxonomy/drl-taxonomy, 2020.

[26] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.

[27] S. Meldrum, S. A. Licorish, and B. T. R. Savarimuthu, "Crowdsourced knowledge on stack overflow: A systematic mapping study," in *Proceedings of the 21st International Conference on Evaluation and Assessment in Software Engineering*, pp. 180–185, 2017.