Mock Deep Testing: Toward Separate Development of Data and Models for Deep Learning

Ruchira Manke *, Mohammad Wardat †, Foutse Khomh ‡, Hridesh Rajan §
* Dept. of Computer Science, Tulane University, Louisiana, USA, rmanke@tulane.edu
† Dept. of Computer Science and Engineering, Oakland University, Michigan, USA, wardat@oakland.edu
‡ SWAT Lab., Polytechnique Montréal, Montréal, Canada, foutse.khomh@polymtl.ca
§ School of Science and Engineering, Tulane University, Louisiana, USA, hrajan@tulane.edu

***Abstract*—While deep learning (DL) has permeated, and become an integral component of many critical software systems, today software engineering research hasn't explored how to separately test data and models that are integral for DL approaches to work effectively. The main challenge in independently testing these components arises from the tight dependency between data and models. This research explores this gap, introducing our methodology of *mock deep testing* for unit testing of DL applications. To enable unit testing, we introduce a design paradigm that decomposes the workflow into distinct, manageable components, minimizes sequential dependencies, and modularizes key stages of the DL, including data preparation and model design. For unit testing these components, we propose modeling their dependencies using mocks. In the context of DL, mocks refer to mock data and mock model that mimic the behavior of the original data and model, respectively. This modular approach facilitates independent development and testing of the components, ensuring comprehensive quality assurance throughout the development process. We have developed *KUnit*, a framework for enabling mock deep testing for the Keras library, a popular library for developing DL applications. We empirically evaluated *KUnit* to determine the effectiveness of mocks in independently testing data and models. Our assessment of 50 DL programs obtained from *Stack Overflow* and *GitHub* shows that mocks effectively identified 10 issues in the data preparation stage and 53 issues in the model design stage. We also conducted a user study with 36 participants using *KUnit* to perceive the effectiveness of our approach. Participants using *KUnit* successfully resolved 25 issues in the data preparation stage and 38 issues in the model design stage. We also found that mock objects provide a lightweight emulation of the dependencies for unit testing, facilitating early bug detection. Lastly, to evaluate the usability of *KUnit*, we conducted a post-study survey. The results reveal that *KUnit* is helpful to DL application developers, enabling them to independently test each component (data and model) and resolve issues effectively in different stages.**

***Index Terms*—deep learning, mocks, testing**

I. Introduction

Deep Learning (DL) is a sub-class of machine learning algorithms that has gained a lot of attention from the industry and academia due to its successful adoption in many domains [1], [2], [3]. The popularity of DL applications has drawn the interest of the software engineering community and the community has responded by conducting several studies [4], [5], [6], [7], [8], [9], [10] to understand the development process of these applications. These studies found that DL application developers usually focus on building and optimizing models using the training data, focusing less on modern software engineering practices such as modular design, unit testing, *etc*. [5]. DL application development follows a workflow that is different from the traditional software development [4], [11], [12] - where data is prepared first followed by model design and training, establishing a tight dependency between data and model. Therefore, incorporating software engineering practices, such as independent testing of data and models necessitates decomposing the workflow, *i.e.*, separating the data and model, and mimicking their dependencies to facilitate unit testing.

Inspired by the fundamental practice of unit testing in traditional software development [13], and the notion of creating mock objects that mimic the minimum expected behavior of dependencies for unit testing [14], we ask: *can we apply the concept of mock objects, commonly used in unit testing traditional software, to test DL applications?* Unit testing with mock objects not only allows for early bug detection in the development cycle but also facilitates the development of modules and verifying their functionality by deferring the dependencies. To the best of our knowledge, unit testing using mocks for DL applications — wherein the data and DL model are tested independently — has not been explored before.

This paper introduces the idea of mock testing in the context of *Deep Neural Networks (DNNs)*. In current DL application development, data is typically prepared by data scientists, while models are designed by machine learning engineers [5]. Each group focuses on distinct stages of the DL pipeline, creating a natural separation of responsibilities. To align with this practice, we recommend treating the data preparation and model design stages as independent modules or units and propose employing mocks for their independent testing. In the context of DL, mocks refer to mock data and mock model that mimic the behavior of the original data and model, respectively. Using mocks for the independent testing of these modules simplifies debugging, facilitates early bug detection, and ensures that the resulting code meets certain quality aspects, *i.e.*, good-quality data, a model that conforms with the requirements, and high operational reliability.

To introduce our notion of mock testing, we have focused on two types of DL architectures, Fully-Connected Neural Networks (FCNNs) and Convolution Neural Networks (CNNs)

Fig. 1: A buggy DL program and mocks in action.

for regression and classification problems. These architectures are commonly employed to handle high-dimensional data due to their ability to capture nonlinear relationships within datasets [15]. To facilitate unit testing, we introduce a design paradigm that considers each stage of DL, *i.e.*, data preparation and model design, as separate modules. The unique challenge in independently testing these components arises from the tight dependency between data and models. To handle the inherent dependencies among these modules, we propose defining clear interfaces to decouple them. These interfaces specify key elements of each stage, such as the 'number of features' in data preparation and the 'DNN architecture type' in model design, which influence each other's configuration. These interfaces are then leveraged to automatically create mock data or models that replicate the behavior of real components. This proposed approach allows for isolated testing of each module by substituting the original data or model with the automatically generated mock versions, ensuring independent quality assurance at each stage. To achieve this, we utilized Python's built-in unit testing framework, `unittest`, and developed, *KUnit* specifically for Keras. *KUnit* comprises 15 distinct methods with assertions aimed at verifying the expected behavior of specific sections of the code under test, leveraging the generated mocks. *KUnit* is open-sourced [16] and can be extended to incorporate more assertions and support other frameworks.

We have evaluated *KUnit* through empirical and user evaluation. Empirical evaluation is performed on 50 programs obtained from *Stack Overflow* and *GitHub*. We separated the data preparation and model design steps into two distinct modules, which were then tested in isolation using mocks. We compared the issues detected using mocks with issues detected when analyzing the two stages together. We observed that, for the data preparation stage, the mock model helped identify 10 issues, whereas, for the model design stage, the mock data assisted in identifying 53 issues during the empirical evaluation. Our results demonstrate that mocks effectively detect issues that cause abnormal behavior during training. We also performed a user study with 36 participants using *KUnit* testing 15 programs. Participants using *KUnit* successfully resolved 25 issues in the data preparation stage and 38 issues in the model design stage. Our findings indicate that mock objects provide an effective, lightweight simulation of dependencies for unit testing. In the post-study survey, we found that *KUnit* is helpful to the developers for testing each component independently and resolving issues early in the development process.

In summary, our work makes the following contributions:

1) **Originality:** First, we introduce mock deep testing for independent testing of data and model. Next, we identify the elements of each stage that are used to decouple the data preparation and model design stages. Leveraging these elements defined in an interface, we propose a method for automatically generating mock objects for each stage. These mock objects support unit testing and help identify issues early in the development process.
2) **Usefulness:** We develop a framework, *KUnit*, that is extensible and generalized to different classes of DL bugs. We specified 15 different bug types and the conditions necessary for their detection. These conditions are incorporated as assertions in the test methods to identify various bugs and repair strategies are proposed to provide actionable fixes in *KUnit*.
3) **Evaluation:** The empirical and user evaluation exhibit mock objects' potential in unit testing DL applications. Our results show that mock objects provided a lightweight

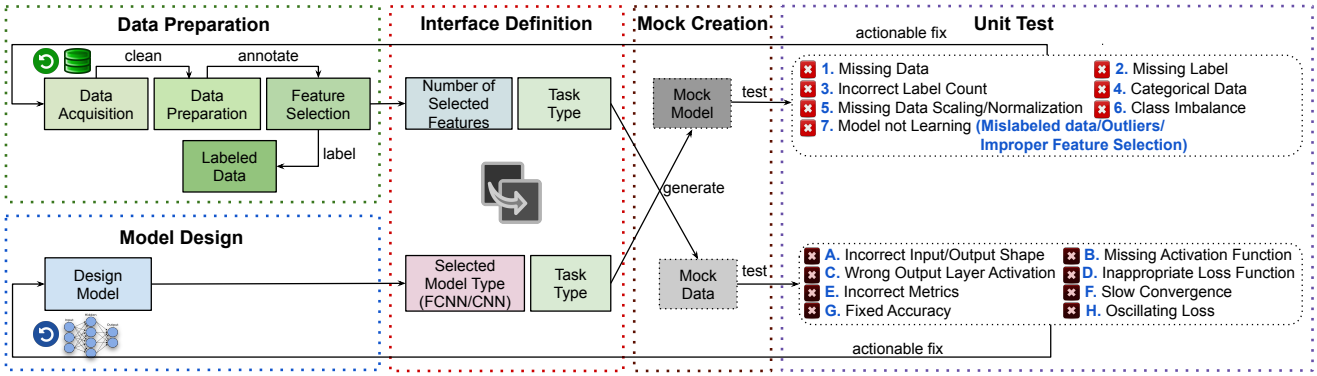

Fig. 2: Workflow of *KUnit*.

emulation of dependencies, allowing early bug detection. The user evaluation provides evidence that mocks are very helpful to developers in testing each component independently and resolving issues effectively.

II. Motivation

The current practice in DL application development involves sequentiality, where the data is prepared first followed by model design and training. The designed model is tested for crash bugs and silent bugs using the data by monitoring and identifying abnormal behavior during training [17], [18], [19], [20], [21], [22], [23]. Bugs can originate from any stage of the DL pipeline, such as data preparation or model design, and often exhibit similar symptoms during training. Therefore, determining the exact root cause of these abnormalities is particularly challenging, thereby requiring several iterations to identify the stage of origin of the bug correctly [17], [18], [19], [20], [23]. For example, exploding gradients, a common issue that can arise from the data preparation stage due to improper training data or from the model design stage due to high learning rate, improper weight or bias initialization, and large batch size [19]. This overlap in the symptoms makes it challenging to pinpoint the root cause of the bugs [24], highlighting the need for a systematic approach for testing and debugging each stage in isolation.

In traditional software development, unit testing has proven useful for conducting a lightweight evaluation of each functionality in isolation. It holds the potential to identify areas for improvement before integrating the two functionalities [25]. Due to the dependency of the model design stage on the data preparation stage in the DL applications, unit testing cannot be applied directly. Our insight is that by decoupling these stages and using the concept of mocks, each module can be tested independently before integration. In the context of DL, *mock data* refers to synthetic data designed to replicate the key characteristics of real data, while a *mock model* is a simplified version of the real model that mimics its behavior without incorporating the complexities of the full model architecture.

To illustrate, consider a DL program shown in Fig. 1 designed for predicting the selling price of the trucks based on 7 features (year, present price, miles driven, fuel type, seller type, transmission, owner). The code from `Lines 1-8` represents the preparation of the data, `Lines 9-13` represents the model design, and finally the model is trained (`Line 14`) using the data obtained from `Lines 5 & 8`. During model training, the program behaved erratically, resulting in NaN values for both the metrics, *i.e.*, mean_squared_error and mean_absolute_error. NaN loss during training can arise from either of the two stages: the data preparation stage, due to NaN values in data, or the model design stage, due to too high learning rate causing model parameters to update too aggressively, divide by zero error during learning or incorrect weight initialization [18]. For instance, this behavior occurred because of the NaN value in the data for the DL program shown in Fig. 1, as the developer forgot to remove or replace missing values during data preparation. Even if the issues in the data preparation stage are addressed, silent bugs in the model design stage can still occur. Isolating the two stages and testing them independently using mocks can help the developer identify and address the issue at the correct stage, thereby reducing the overall debugging effort required during the training process. This motivates the development of *KUnit*, a novel approach for facilitating unit testing of DL applications using mock objects. The fundamental idea of *KUnit* is based on the observation that the behavior of the original data on the mock model and the original model on the mock data remains consistent. To illustrate, for the example in Fig. 1, the unit testing of the data preparation stage using a mock model resulted in NaN values for both the metrics, mean_squared_error, and mean_absolute_error (consistent with original model behavior). *KUnit* reported that the issue occurred because of missing values in the data which can be addressed in the data preparation stage. Similarly, for the model design stage, the oscillating loss on the mock data reported by *KUnit* (consistent with the model behavior on original data after replacing the missing values in the original dataset), indicates incorrect hyperparameter selection, which can be refined before combining two stages. Unit testing of these stages and addressing the issues in the error-inducing stage helps identify potential problems early before integrating them and initiating the training process. The rest of this work describes our approach, *KUnit*, for enabling mock testing for DL applications.

```python
# Interface of Data Preparation Module
class DataInterface():
    @property
    def features(self):
        return 7
    @property
    def dataType(self):
        return 'structured'
    @property
    def task(self):
        return 'regression'
    def preprocess():
        pass
# Model Design Module implements the ModelInterface
# DataPreprocess Module depending on the ModelInterface
class DataPreprocess(DataInterface):
    def __init__(self):
        modelI = ModelInterface()
        modelType = modelI.architecture
        taskType = modelI.task
    def preprocess():
        # Code for preprocessing steps
```

(a) Interface of data preparation stage

```python
# Interface of Model Design Module
class ModelInterface():
    @property
    def architecture(self):
        return 'fcnn'
    @property
    def task(self):
        return 'regression'
    def design():
        pass
# Data Preprocessing Module implements the DataInterface
# Model Design Module depending on the DataInterface
class DNNModel(ModelInterface):
    def __init__(self):
        dataI = DataInterface()
        featureNumbers = dataI.features
        dataType = dataI.dataType
        taskType = dataI.task
    def design():
        # Code for designing model
```

(b) Interface of model design stage

Fig. 3: Interface definition and class description.

III. Approach

In this section, we provide an overview of our approach for unit testing DL applications using mocks. Inspired by the decomposition criteria proposed by Parnas [26], we suggest making each major step in the DL program a module. In a DL pipeline, these major steps correspond to the different stages, *i.e.*, *data preparation* and *model design*. Due to coupling between these stages, we decouple them by defining interfaces, allowing the data preparation and model design stages to depend on the interface, ensuring their independence. These interfaces facilitate the automatic generation of mocks, which are used for unit testing of each stage. The workflow of our approach, *KUnit* is shown in Fig. 2. We collected issues for each stage from various sources outlined in Section IV-A1 and established the conditions for identifying them. In total, we obtained 7 issues (1-7) for the data preparation and 8 issues (A-H) for the model design stage shown in Fig. 2. We leveraged the Python's built-in unit testing framework, `unittest`, to develop, *KUnit*; a testing framework for Keras. We defined each condition as an assertion in the test method aimed at verifying the expected behavior of the code under test leveraging the generated mocks. Once a failure is detected by *KUnit*, the user is notified with an assertion error and a workable solution. Our approach for unit testing, DL applications have two main steps: *interface definition* and *mock object creation and verification*. Below, we discuss each step in detail.

A. Interface Definition

Due to dependencies between the data preparation and model design stages, the primary task for independent testing of these stages is to design interfaces that decouple them. For decoupling, it's crucial to identify the elements of one stage that impact the design decisions of the other stage. Understanding these dependencies enables better modularization and facilitates smoother integration between stages. We propose interfaces that allow data preparation and model design stages to depend on the interface, ensuring their independence. Below, we elaborate on our approach to defining interfaces for each stage in detail.

a) Interface for Data Preparation Stage: In the data preparation stage, feature engineering is a common activity carried out intending to select informative features that the DL model learns during training. These features play a crucial role in the design decisions of the model design stage. For example, in the model design stage, most of the decisions, such as which neural network architecture to choose and its hyperparameters depend on the characteristics of data and the features selected in the data preparation stage. For instance, consider a model in Fig. 1 designed for predicting the selling price of trucks. As the dataset is structured (each row representing a different record), the developer selected a FCNN model, which is known to perform well for structured data [27]. Since this is a regression task, choosing the appropriate hyperparameters is another design decision in the model design stage. For example, the output layer activation function depends on the type of task, *i.e.*, regression or classification. For the data preparation stage, we propose an interface that incorporates the number of features selected during the feature selection step, the type of data, and the type of task. For the task in Fig. 1, Fig. 3(a) shows the interface and class description of the data preparation stage. The data preparation module implements this interface, exposing its behavior to other classes that depend on it for their design decisions.

b) Interface for Model Design Stage: In the model design stage, selecting an appropriate neural network architecture corresponding to the task is crucial for achieving optimal performance. For instance, FCNNs are a good choice for tasks involving structured data [27], where each feature is independent and there are no inherent spatial or temporal relationships to consider, whereas CNNs are well-suited for image classification tasks due to their ability to capture spatial hierarchies in images [28]. In the data preparation stage, feature selection is influenced by the neural network architecture chosen in the model design stage. Since different architectures have different learning processes, the informative features are usually refined based on the model's performance during evaluation [11]. For instance, in FCNNs there is no inherent weight sharing whereas, CNNs have a key feature of weight sharing through the convolution filters. As a result, these two architectures may perform differently even with the same data. Therefore, for the model design stage, we propose an interface comprising the architecture chosen in this stage and the type of task. Fig. 3(b) shows the interface and class description of the model design stage for the task in Fig. 1.

TABLE I: KUnit's mock model generation process.

			Rules					
Conditions	Problem Type	Regression	Yes	Yes	No	No	No	No
		Binary Classification	No	No	Yes	Yes	No	No
		Multiclass Classification	No	No	No	No	Yes	Yes
	Model Type	FCNN	Yes	No	Yes	No	Yes	No
		CNN	No	Yes	No	Yes	No	Yes
	Classes	Number of classes = 1	Yes	Yes	No	No	No	No
		Number of classes = 2	No	No	Yes	Yes	No	No
		Number of classes >2	No	No	No	No	Yes	Yes
Actions	Hyperparameters	Hidden Layer Neurons	# of features					
		Output Layer Neurons	1	1	2	2	# of classes	# of classes
		Output Layer Activation	'linear'	'linear'	'sigmoid'	'sigmoid'	'softmax'	'softmax'
		Loss Function	'mse'	'mse'	'binary_ crossentropy'	'binary_ crossentropy'	'categorical_ crossentropy'	'categorical_ crossentropy'
		Metrics	'mae'	'mae'	'accuracy'	'accuracy'	'accuracy'	'accuracy'
	Mock Model	FCNN Model	✓		✓		✓	
		CNN Model		✓		✓		✓

The model design module implements this interface and its behavior is exposed to other classes that depend on it. The defined interfaces are utilized for creating useful mocks and testing each module independently.

B. Mock Object Creation and Verification

To ensure the correctness of each module, it is essential to verify that each module exhibits the correct functionality depending on the task. Our insight is that to evaluate the expected behavior of each module, useful mocks can be constructed that approximate the behavior of the original data or model using the information exposed in interfaces. In the mock implementation, the primary goal is to achieve simplicity instead of aiming for completeness [14]. To that end, we propose a systematic approach for creating mocks for the data preparation and model design stages, detailing how these mocks are utilized for verification. Below we discuss the process for each stage in detail.

1) Data Preparation Stage: The data preparation module intends to produce good-quality data and involves various tasks such as data cleaning, handling missing values, *etc*. Typically, the features are selected and refined based on the model's performance during evaluation, establishing a feedback loop from the model evaluation stage to the feature engineering stage [11]. Our insight is that by creating a mock model that approximates the behavior of the original model, data quality can be assessed and enhanced through early-stage evaluation. While the mock model doesn't have to preserve every semantic detail, it is essential to generate a model with the appropriate hyperparameters that yield the correct output tailored to a task. The mock model can be automatically generated using the interface of the model design stage as illustrated in Fig. 4(a). Below, we discuss the process of mock model creation.

a) Process for Mock Model Creation: Generating a mock model involves several key steps to ensure the correctness and reliability of the generated model.

Adaptive Mock Model Generation: A DL model has a lot of hyperparameters, which are provided at the time of model design by the developer. The choice of the hyperparameters depends on several factors, such as the type of task and complexity of the dataset [29], [30]. Any incorrect hyperparameter can be misleading, giving rise to bugs due to inaccuracies in the model. Therefore, for automatic mock model generation, it is necessary to construct adaptive mocks that change based on the task at hand and adapt to different testing conditions without manual intervention. This adaptability ensures that the mock model aligns with testing requirements. Furthermore, as the paper introduces a design paradigm that supports independent development, developers can utilize automatically generated mock models for testing the data without delving into the intricacies of designing them.

Our approach for the automatic mock model generation is described in Decision Table I. For initializing the mock model's hyperparameters, we reviewed the AI literature [31], [32], [33] and Keras documentation [34]. We utilized the hyperparameters suggested by the literature for a given task. For example, for the DL program in Fig. 1, the conditions as outlined in Decision Table I are problem type - 'regression', model type - 'FCNN', and classes - 1 (set to 1 for regression). The corresponding actions generate a mock model with hidden layer neurons equal to the '# of features', an output layer with 1 neuron and a 'linear' activation function, and a compilation layer with the 'mse' loss function and 'mae' as the metrics.

Complexity of Mock Model: The DL models are usually complex with several parameters and their complexity increases with the type of task at hand. Determining the complexity of the model requires careful consideration during mock model generation. Since the mock model aims to aid feature engineering, creating a complex mock model for the unit testing could lead to excessive resource usage without serving the primary goal of unit testing.

To optimize the model's complexity for unit testing while managing resource consumption, we propose creating a mock model consisting of only three layers: the input layer, one hidden layer, and the output layer. The rationale for opting for the simplest network is influenced by the principle of "Start Simple", as recommended in the machine learning literature [35], [36]. If a simple network struggles to learn from the training data, it suggests that the training data requires further refinement [29]. Next, we explain how this mock model is utilized for the verification of data.

b) Verification of Data using Mock Model: After generating a mock model that simulates the behavior of the original DNN model, verification is performed by inputting the preprocessed data into the mock model, rather than using the original model. Certain data preparation issues, like missing data, can be detected through data property assertions. In contrast, issues like mislabeled data, outliers, or improper feature selection require in-depth analysis. These issues often manifest as subtle errors that impact model performance and require a thorough examination of data-model interactions for effective identification. Therefore, we propose an integrated approach that combines data property assertions with analysis of the mock model's behavior on preprocessed data. The data property assertions are used to identify fundamental issues (1-6 in the data preparation stage shown in Fig. 2). Moreover, more complex issues labeled as 'Model not Learning' in Fig. 2 are identified by observing the mock model's behavior. Symptoms such as high loss, frequent misclassifications, or consistently low confidence on specific samples point to these issues, highlighting areas that require further investigation. This approach allows for the verification and refinement of data before it is used to train the original model.

```python
# Generate Mock Model using ModelInterface and test
DataPreprocess Class
class DataPreprocessTest(tensorflow.test.TestCase):
      @classmethod
      def setUpClass(self):
            super(DataPreprocessTest,self).setUpClass()
            modelI = ModelInterface()
            modelType = modelI.architecture
            taskType = modelI.task
            gm = GenerateMockModel()
            model = gm.mockModel(modelType,taskType)
      def test1():
            # Code for testing
```

(a) Mock model creation for data preparation stage

```python
# Generate Mock Data using DataInterface and test
DNNModel Class
class DNNModelTest(tensorflow.test.TestCase):
      @classmethod
      def setUpClass(self):
            super(DNNModelTest,self).setUpClass()
            dataI = DataInterface()
            features = dataI.features
            dataType = dataI.dataType
            taskType = dataI.task
            gd = GenerateMockData()
            X,y = gd.mockData(features,dataType,taskType)
      def test1():
            # Code for testing
```

(b) Mock data creation for model design stage

Fig. 4: Mock object creation for different stages.

2) Model Design Stage: The goal of the model design stage is to generate a model with suitable hyperparameters and correct API usage, appropriate for the given task. This facilitates the model in learning features from the training data. Since DNNs are data-driven, training data produced by the data preparation stage is typically used to evaluate the model's performance and tune its hyperparameters. Although the mock data might not help detect all the training time issues, our insight is that leveraging mock data enables the early detection of numerous bugs, including tensor shape mismatches, inappropriate hyperparameter selection, *etc*. It allows for improving the model's quality before assessing its performance on original training data. Fig. 4(b) illustrates the mock data creation process using the interface of the data preparation stage. While the mock data doesn't have to preserve every semantic detail, it is crucial to ensure that it does not contain missing values or outliers. Below, we discuss the process of mock data generation.

a) Process for Mock Data Creation: Generating mock data requires careful adherence to key steps to ensure the correctness of the resulting data.

Preprocessed Mock Data Generation: During the data preparation stage, several preprocessing steps such as data cleaning, outlier removal, class balancing, *etc*., are performed to ensure the quality of the data. While creating mock data automatically, it is necessary to ensure that similar preprocessing steps are added. The absence of data preprocessing steps can be misleading and result in errors stemming from inaccuracies present in the mock data.

To ensure that the mock data accurately reflects the real-world scenarios, we utilized the *make_classification()* and *make_regression()* functions provided by *scikit-learn* [37] for synthetic data generation. The advantage of utilizing these functions is that these functions provide a level of control over the characteristics of the generated dataset. For example, we can generate mock data that is normally distributed, without outliers, is labeled and classes are balanced. These functions are also customizable, allowing users to specify the number of samples, features, *etc*., offering flexibility for specific testing scenarios. Although the data generated by these functions is normally distributed, however, it is not scaled or normalized. Scaling or normalization is a common preprocessing step that aids in faster convergence of DL models during training [38]. In our approach to mock data generation, we scaled the data generated by *make_classification()* and *make_regression()* functions. The scaled mock data is then utilized to verify the behavior of the model. The rationale for validating the model's behavior with mock data is that if the model struggles to learn from the mock data, the model's hyperparameters can be refined before proceeding to train it with the original dataset, which is usually more complex than the mock data.

Quantification of Mock Data: Determining the amount of data necessary for training a DL model is often a subject of debate. It is typically gauged by several factors, such as the complexity of the dataset and the model's performance during evaluation. In this paper, mock data is employed for unit testing to ensure the precise functioning of the mathematical functions within each layer of the designed model. It also verifies the transformation of input data into meaningful representations tailored to the specific task for which the model is designed. Therefore, determining the amount of data required for the intended purpose (*i.e.*, unit testing) poses a challenge. A small number of samples can lead to misleading results, while a large volume of samples can be resource-intensive.

To address this challenge, we consulted established machine learning literature [39], [40] to estimate the approximate dataset size. We also conducted a sensitivity analysis of the dataset sizes suggested in the literature [39], [40]. We varied the dataset size in increments $\pm 5\%$, $\pm 10\%$ and $\pm 20\%$ and observed the impact on the model's performance. Based on this analysis, the samples generated by *KUnit* are as follows: for the regression task, the number of samples generated is 10 times the number of features and for the classification task, 100 samples are generated for each class. In our experiments, we found that these samples were adequate for identifying various types of issues in the designed models. We now discuss how the mock data is utilized for the verification of the model.

b) Verification of Model using Mock Data: After generating mock data that mimics key characteristics of the original dataset, verification is done by feeding the mock data into the designed model. This ensures the model's correctness without depending on the original data. To facilitate this, we propose an integrated approach that combines the model property assertions with an analysis of the model's behavior on mock data. For verifying the model's structure, assertions are defined using the data properties defined in the interface of the data preparation stage and conditions obtained from the literature (Section IV-A1), which helps to identify issues A-E illustrated in Fig. 2 for the model design stage. Next, the model's response to the mock data is analyzed to ensure

it appropriately handles normalized/scaled data. This analysis facilitates the early detection and resolution of potential issues F-G, shown in Fig. 2. By detecting these issues early, the approach allows verification and refinement of the model structure before using the original data for training.

IV. Evaluation

A. Experimental Setup

In this section, we discuss the process for collecting assertions for issue identification, datasets used for the empirical evaluation and user study, task description, and details of participants involved in the user study.

1) Procedure for Collecting Assertions for Issue Identification: In this section, we detail our process for identifying the types of bugs supported by *KUnit* and explain how the corresponding assertions are developed to detect them. To identify these bugs, we conducted a thorough literature review. Islam *et al.* [6] investigated the type of DL bugs and categorized them into different categories, with data and model bugs being two key categories. Humbatova *et al.* [7] refined the investigation and further divided data bugs into two subcategories: training data quality and preprocessing of training data. And, the model bugs were categorized into subcategories such as wrong input, wrong tensor shape, *etc*. These classifications provided a structured foundation for understanding common pitfalls in DL workflow. While some bugs reported in these studies require comprehensive end-to-end analysis, others can be effectively detected through targeted testing of specific components, such as data and model. For instance, focused component-level testing can detect crash bugs caused by wrong preprocessing and silent bugs resulting from incorrect activation functions. In contrast, issues like overfitting and underfitting depend on evaluating the model's performance on the original dataset. Thus, we focus on bugs that can be detected at the component level while excluding those that require end-to-end analysis. Similar to the procedure outlined in [20], [23] we filtered out these bugs from the empirical studies [6], [7] and obtained 7 data-related issues (1-7 in Fig. 2) and 8 model-design-related issues (A-H in Fig. 2) currently supported by *KUnit*. We then adopted an approach similar to that of TheDeepChecker [20] and reviewed existing works on fault localization and repair techniques for DL programs [19], [22], [18], [21], [41], [24], contracts for DL programs [42], [43], and the Keras official documentation [44], [45]. This review enhanced our understanding of the root cause of the bugs and how these issues manifest in DL workflow, allowing us to establish the conditions necessary to identify and address them. These conditions are implemented as assertions in *KUnit*'s test methods to identify the bugs and repair strategies that are utilized to provide actionable fixes in *KUnit*.

2) Implementation: We implemented *KUnit* in Python on top of *Keras* 2.3.0 and *TensorFlow* 2.1.0. The conditions obtained in Section IV-A1 are implemented as test cases using Python's built-in unit testing framework, `unittest`.

TABLE II: Datasets used for user study.

Datasets		Portfolio Data = NU Labels = NU Regression	Grain Data = NU Labels = CA Multiclass Classification	Truck Data = MI Labels = NU Regression	Loan Data = MI Labels = NU Binary Classification	Train Data = MI Labels = NU Binary Classification
Data Checks		Bugs				
Data Quality	MV	N	N	Y	Y	Y
	ML	N	N	N	N	Y
	CI	N	Y	N	Y	N
Preprocessing of Data	ME	N	Y	Y	Y	Y
	MS	Y	Y	Y	Y	Y

MV = Missing/Infinite Value, ML = Missing Label, CI = Class Imbalance,
ME = Missing encoding of categorical data, MS = Missing Scaling/Normalization
NU = Numeric, CA = Categorical, MI = Mixed

TABLE III: Models used for user study.

Model #	Architecture Type	# of layers	# of neurons	# of parameters
M1	FCNN	5	21	120
M2	FCNN	6	131	5703
M3	FCNN	3	41	601
M4	FCNN	6	57	1153
M5	CNN	5	71	5201

3) Empirical Evaluation: To evaluate our approach, we collected DL programs developed using Keras. We examined the recently published DL fault localization benchmarks [18], [21], [23]. The DeepLocalize's [18] benchmark comprises of 41 executable Keras codes containing both buggy and correct versions of DL programs from *Stack Overflow* (30) and *GitHub* (11). The DeepFD's [21] benchmark has 58 DL programs with patches for buggy and correct versions obtained from *Stack Overflow* (47) and *GitHub* (11). deepmufl's [23] benchmark comprises 109 DL programs obtained from *Stack Overflow*. Currently, our approach *KUnit*, supports two types of DL architectures, FCNNs and CNNs designed for regression and classification problems for structural data. We used these criteria for filtering the programs from these benchmarks. We found that there is some overlap among the programs in these benchmarks. Therefore, we acquired 50 programs, 42 from *Stack Overflow* and 8 from *GitHub*. We considered the 50 programs in our benchmark as "unseen" because we have not seen these buggy and correct programs during the process of acquiring conditions described in Section IV-A1.

4) User Study: We also performed a user study to evaluate our approach. We followed the methodology of Biswas *et al.* [11] to collect real-world datasets and tasks from Kaggle competitions [46] and collected 5 real-world datasets that require preprocessing to meet data quality requirements. The details of these datasets are shown in Table II. We selected 5 sequential DL models from Kaggle competitions [46] and provided them as a reference model to the participants. The details of these models are shown in Table III.

5) Tasks: To avoid versioning issues during experimental setup and save participants time, we hosted tasks on *GitHub* Codespaces [47]. It has a web-based VS Code IDE and virtual machines that allow developers to edit, run, test, and debug code within a web browser. We used Zoom to monitor each participant's screen and ensured they used the program as intended. We shared with participants a document with instructions explaining the goal of the task and how to run it on *GitHub* Codespaces IDE. In our task design, each participant performed two tasks: one in the traditional way and one in the modular way. Since participants working on a task are likely to remember its details and the issues encountered, we adopted a between-subjects design [48] to mitigate the learning

TABLE IV: Summary of issues detected by *KUnit*.

Stage	Categories	# of issues in different categories	# of issues detected by KUnit
Data Preparation	Missing Scaling/Normalization	11	9
	Labels not matching problem definition	1	1
Model Design	Incorrect Input shape	1	1
	Incorrect Output shape	2	2
	Missing Activations	2	2
	Wrong Output Layer Activation	30	27
	Learning Rate out of Common Range	3	3
	Wrong Loss Function	8	8
	Incorrect Evaluation Metrics	7	7
	Oscillating Loss/Slow Convergence	9	3
Total		**74**	**63**

effect. Specifically, we assigned each participant two distinct problems. For the data preparation task, we provided participants with datasets and asked them to explore the data and add necessary preprocessing steps based on their experience. In the traditional setting, they did not test the code, whereas in the modular setting, participants tested the added preprocessing steps in isolation using an automatically generated mock model. For the model designing task, we provided participants with a reference DL model and asked them to make necessary structural changes according to the task requirements and their experience. In the traditional setting, participants were given preprocessed data from the data preparation task completed by another participant in the traditional setting and asked them to test the designed model using the original data. Whereas, in a modular setting, participants tested the designed model in isolation using automatically generated mock data. During the study, the participants were allowed to access the internet to confirm the syntax of different operations in Python and Keras. After completing the task, we requested participants to complete a survey to share their experience of *KUnit* on a 5-point Likert scale and provide open-ended feedback. We conducted a pilot study with 7 participants, which allowed us to refine the tasks and instructions. This study was reviewed by our Institutional Review Board.

6) Participants: We recruited participants via LinkedIn using direct messages and university mailing lists that described our study and a link to the screener survey. We screened participants (i) who were over 18, (ii) who had at least one year of programming experience, and (iii) who had experience with DL programming. To evaluate the applicability of our approach in everyday scenarios and industry settings, we recruited both graduate students and industry professionals. In total, we recruited 36 participants (21 male and 15 female), 24 graduate students from different universities, and 12 industry professionals working in Google, IBM, Neural Lab, *etc*. Participants were asked to self-classify their level of expertise from 1 - beginner to 5 - expert. The obtained expertise levels are: using existing DL programs ($\mu = 4.1, \sigma = 0.7$), developing new DL programs ($\mu = 4.0, \sigma = 0.8$), debugging DL programs ($\mu = 3.9, \sigma = 0.8$), and familiarity with preprocessing steps ($\mu = 4.0, \sigma = 0.8$).

In the study design, we aimed to have each task performed by participants with varying levels of expertise. This allowed us to investigate the mistakes made by developers across different skill levels and evaluate the utility of unit testing in DL applications. Participants were assigned tasks based on their self-reported expertise level. During self-assessment, we found that the participants rated themselves as either 3 (competent), 4 (proficient), or 5 (expert). When a participant scheduled a session, we assigned a task that no other participant with the same experience level had already been assigned. If a new participant with the same expertise level scheduled a time slot and all tasks had been completed by others with the same expertise level, we randomly assigned tasks to ensure that at least three different participants performed each task.

B. Results

In this section, we report on the efficiency of our technique for unit testing DL applications using mocks and answer our research questions.

1) Do mock objects aid in testing each functionality in isolation without reliance on external dependencies?: In this research question, we validate whether different functionalities of the DL programs can be tested without committing to a labeled dataset or model using mock objects, isolating the code being tested from external dependencies (dataset or model). To answer this research question, we first evaluated the performance of *KUnit* on 50 DL programs in our benchmark. The first author manually inspected each DL program and divided it into two parts: *Data Preparation:* contains all the steps related to data preparation and *Model Design:* contains all the steps related to designing the model including the compilation step. Each part is tested independently using the mocks automatically generated by *KUnit*. To address the challenges posed by the indeterministic nature of DL applications, we execute each test 3 times and consider issues reported in more than one run. The buggy version of the original program was examined, and the number of issues in each of the 50 programs was counted. Table IV reports the total number of issues found in 50 programs across different categories. The results shown in Table IV depict that the mock model facilitated testing of the data preparation steps, with *KUnit* identifying 10 out of 12 issues in this stage. Similarly, for the model design stage, mock data facilitated testing of the designed model, with *KUnit* identifying 53 out of 62 issues in this stage. Further investigation was done to determine the reason behind the issues missed by *KUnit*. In the data preparation stage, we found that the 2 missed issues were related to scaling the labels to the appropriate range to match the output layer activation. *KUnit* only verifies the scaling of the data, not the labels. Therefore, *KUnit* missed these issues. For the model design stage, the assertions used in *KUnit* are obtained from various sources (discussed in Section IV-A1) which cover various frequently occurring scenarios that might not account for some edge cases, such as models designed for datasets with specific label ranges; *KUnit* missed 9 issues. As *KUnit* is open-source, developers have the flexibility to refine these assertions or define new ones according to their needs.

Secondly, during the user study, participants used *KUnit* to independently test the data preparation steps and designed model using mocks. Tables V and VI present the results of testing the three solutions (S1, S2, and S3) provided by

TABLE V: Summary of issues detected by *KUnit* using mock model in data preparation stage.

Stage	Tests		Task 1 (Portfolio)			Task 2 (Grain)			Task 3 (Truck)			Task 4 (Loan)			Task 5 (Train)		
			S1 (Com)	S2 (Com)	S3 (Prof)	S1 (Prof)	S2 (Prof)	S3 (Exp)	S1 (Com)	S2 (Prof)	S3 (Exp)	S1 (Com)	S2 (Prof)	S3 (Exp)	S1 (Com)	S2 (Prof)	S3 (Exp)
Data Preparation (Testing using **Mock Model**)	Preprocessing steps are applied correctly	Check scaling/normalization is done correctly	✗	✗	✗	✓	✓	✓	✓	✓	✗	✗	✓	✓	✓	✓	✗
	Quality assurance	Missing values are removed/replaced	–	–	–	–	–	–	✗	✗	✗	✓	✓	✓	✓	✓	✓
		Missing label are removed/replaced	–	–	–	–	–	–	–	–	–	–	–	–	✗	✗	✓
		Classes are balanced	–	–	–	✗	✗	✗	–	–	–	✗	✗	✓	✗	✗	✓
		Labels are matching problem definition	–	–	–	–	–	–	–	–	–	–	–	–	✗	✗	✓
	Correct format of data	Categorical data is converted to numeric data	–	–	–	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	Performance of the mock model	Check mock model is able to learn from selected features	✓	✓	✓	✓	✓	✓	✗	✗	✗	✓	✓	✓	✗	✗	✓
Number of issues detected in data preprocessing			**1**	**1**	**1**	**1**	**1**	**1**	**2**	**2**	**3**	**2**	**1**	**0**	**4**	**4**	**1**
Number of issues resolved by participants			**1**	**1**	**1**	**1**	**1**	**1**	**2**	**2**	**3**	**2**	**1**	**0**	**4**	**4**	**1**
Number of issues dismissed as false alarms by participants			**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**

Expertise level of participants: Exp - Expert, Prof - Proficient, Com - Competent
✓**:** Steps are applied correctly and test case passed. ✗**:** Steps are either missed or applied incorrectly and test case failed. –**:** Steps not required for the dataset.

TABLE VI: Summary of issues detected by *KUnit* using mock data in model design stage.

Stage	Tests		Task 1 (Portfolio)			Task 2 (Grain)			Task 3 (Truck)			Task 4 (Loan)			Task 5 (Train)		
			S1 (Com)	S2 (Prof)	S3 (Exp)	S1 (Com)	S2 (Prof)	S3 (Exp)	S1 (Prof)	S2 (Prof)	S3 (Exp)	S1 (Prof)	S2 (Exp)	S3 (Exp)	S1 (Prof)	S2 (Exp)	S3 (Exp)
Model Design (Testing using **Mock Data**)	Data and model input-output layer alignment	Check input shape in input layer is correct	✓	✗	✓	✓	✓	✓	✗	✓	✓	✓	✗	✓	✓	✓	✓
		Check output shape in input layer is correct	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	Correct operations	Activation functions are applied correctly in all hidden layers	✗	✗	✗	✓	✓	✓	✗	✗	✗	✓	✗	✗	✓	✓	✓
		Output layer format is correct depending on the task	✓	✓	✓	✓	✓	✓	✗	✓	✓	✓	✗	✗	✗	✓	✓
		Correct loss function is selected	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
		Correct metrics is selected	✗	✓	✓	✗	✗	✓	✗	✗	✓	✗	✗	✗	✓	✓	✓
	Performance of model on mock data	Model is learning and accuracy is changing	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	✓
		No oscillating loss	✓	✓	✓	✗	✗	✗	✓	✓	✓	✓	✗	✗	✗	✗	✗
		Model is not converging slowly	✓	✓	✓	✗	✗	✗	✓	✓	✓	✗	✗	✓	✗	✗	✗
Number of issues detected in model structure			**3**	**2**	**1**	**4**	**3**	**2**	**4**	**2**	**1**	**2**	**6**	**5**	**3**	**2**	**2**
Number of issues resolved by participants			**3**	**2**	**1**	**4**	**3**	**2**	**4**	**2**	**1**	**2**	**6**	**2**	**3**	**2**	**1**
Number of issues dismissed as false alarms by participants			**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**3**	**0**	**0**	**1**

Expertise level of participants: Exp - Expert, Prof - Proficient, Com - Competent
✓**:** Steps are applied correctly and test case passed. ✗**:** Steps are either missed or applied incorrectly and test case failed.

different participants for each task, with columns indicating the issues detected in each solution. For the data preparation stage, Table V demonstrate that a total of 25 issues were identified by *KUnit* using mock models for all the tasks. All these issues were acknowledged as valid findings by participants with varying levels of expertise and were subsequently resolved by them. For the model design stage, the participants tested the designed model using the mock data. In Table VI, for each task, we highlighted the issues identified and reported by *KUnit*. Across all tasks, *KUnit* detected a total of 42 issues using mock data, of which 38 were accepted and resolved by participants with varying levels of expertise. However, for 4 issues reported by *KUnit*, 2 participants mentioned that, based on their experience, these might be false alarms. For example, in Task 4, a binary classification task, *KUnit* verifies that the output layer yields a value between 0 and 1 depicting the positive class probability (property of the sigmoid function). However, for Task 4 (S3), the participant expressed a preference for using a probability distribution (softmax function) to represent the output layer results instead of class probabilities (sigmoid function), leaving room for extending the problem to multiclass classification in the future. Similarly, for detecting the oscillating loss issue, *KUnit* monitors the loss after every 5 epochs. However, for Task 5 (S3), the participant stated that they prefer to evaluate the model's stability every 10 epochs. Therefore, due to differing criteria and preferences used by different developers for evaluating model stability, 4 out of 42 issues were dismissed as a false alarm by 2 participants for Task 4 (S3) and Task 5 (S3). Given that *KUnit* is open-source, developers can customize these assertions to fit their problem requirements. *In summary, our results demonstrate that mock objects facilitated independent testing of each functionality and assisted in the early detection of issues.*

2) How efficient are mock objects in identifying issues compared to traditional deep learning testing approaches?: In this research question, we evaluate the efficiency of mock objects in uncovering issues in DL programs at an early stage, that in current practice, are detected after combining different stages, specifically during training. To compare the efficiency of *KUnit* with traditional DL testing approaches, we conducted an evaluation of 50 programs in our benchmark. The original program (with data and model stages combined) is tested using a state-of-the-art approach [19] and the results are compared with the issues identified by *KUnit* in each stage. DeepDiagnosis [19] is a fault localization tool that detects silent bugs in DL programs by monitoring for abnormal behavior during training. This tool is selected because it covers most of the silent bugs (8) encountered during model training compared to other existing fault localization tools [17], [18], [22], [21]. By comparing *KUnit* with DeepDiagnosis, we evaluate whether *KUnit* can effectively identify issues at an earlier stage using mocks before data-model integration that DeepDiagnosis identifies after integration during training. Our analysis shows that, for 22 programs with issues in only one stage, *i.e.*, data preparation or model design, *KUnit* identified the same problems with mocks that DeepDiagnosis identified using the original dataset during training. For the 25 programs with issues in both the data preparation and model design stages, DeepDiagnosis detected issues in the data preparation stage for 7 of these programs. However, uncovering bugs in

the model design stage with DeepDiagnosis requires fixing the data preparation issues first, necessitating multiple iterations to identify problems in the model design stage. Similarly, for 5 of these programs, DeepDiagnosis reported a numerical error in computation but could not determine the error-inducing stage. For the remaining 13 programs, DeepDiagnosis did not detect any issues. The main challenge for DeepDiagnosis stems from the programs with multiple bugs, as it detects one issue at a time. This requires retraining the model on an original training dataset after every modification, leading to inefficient use of computational resources [22]. In contrast, testing each stage separately with *KUnit* using mocks provides a lightweight emulation of dependencies, facilitating testing each stage before and after modifications, thereby saving resources and accurately identifying errors in the correct stage in all 25 programs. Some numerical computations in DNNs are highly data-dependent. For example, using activation functions that are not suitable for certain input ranges can lead to out-of-range problems, which cannot be detected during unit testing with mocks. Therefore, for 3 programs, *KUnit* missed these issues, whereas, testing using the original dataset helped DeepDiagnosis identify issues in 2 out of 3 programs. During the user study, participants utilized both *KUnit* and DeepDiagnosis for debugging. The results show that testing individual stages in isolation with mocks helped *KUnit* to efficiently pinpoint the root causes of bugs, streamlining the debugging process. In contrast, DeepDiagnosis, which detects issues after data-model integration, cannot identify the root cause of the bug in programs with multiple bugs. This is particularly evident in programs with bugs originating from both the data preparation and model design stages, as these bugs often exhibit overlapping symptoms during training, thereby complicating the debugging process. Due to this, in 10 programs with multiple bugs, DeepDiagnosis reported a numerical error but failed to pinpoint the root cause. Detailed results are reported in the supplementary material [49], [50]. We also analyzed the debugging time for each task in Tables V and VI, comparing results with and without KUnit. In the traditional setting, when data and model are integrated and tested using DeepDiagnosis, we observed that the quality of the preprocessed data significantly impacted the debugging time. Participants had to resolve data-related issues before addressing model-specific problems. In contrast, when using *KUnit*, the data and model are tested in isolation using mocks, allowing participants to focus on stage-specific issues. As DeepDiagnosis identifies issues after integration, for a fair comparison, we computed the total time, *i.e.*, summation of time taken by participants to resolve bugs in data and model stages using KUnit separately and compared it with DeepDignosis. Our analysis reveals that, on average, participants took 15 and 12 minutes to debug the DL programs using DeepDiagnosis, and *KUnit*, respectively. By isolating data and model, *KUnit* facilitates a more focused and efficient debugging process that can save time and resources during training. Detailed analysis is provided in our GitHub repository [51]. *In summary, our analysis shows that mock objects effectively mimic essential system behaviors, simplifying complexity for unit testing and assisting in identifying issues that lead to abnormal behavior during training. Mock testing is efficient and resource-friendly, especially for programs with issues in multiple stages.*

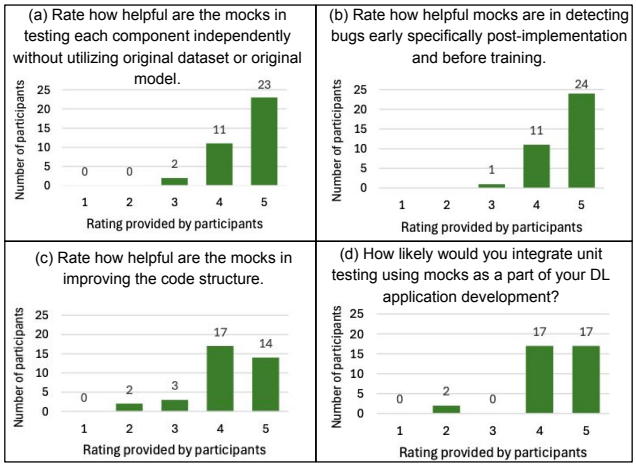

Fig. 5: Survey results with participants ratings.

3) How do developers perceive the effectiveness of unit testing using mocks compared to traditional deep learning testing approaches?: To answer this research question, we collected survey responses from 36 participants during the user study. In particular, on a 5-point Likert scale question, participants rated their experience about the usefulness of mocks for unit testing. Likewise, 35 participants (Fig. 5(b)) responded (rating $>$ 3) that mocks facilitate the early identification and resolution of issues during the development process. Regarding the usefulness of the mocks in improving code structure, 31 participants (Fig. 5(c)) responded positively (rating $>$ 3), while 5 participants (rating $\leq$ 3) expressed the concern that mocks might produce false alarms and mislead efforts to improve the code structure. Regarding the integration of the mocks into their DL development process, 34 participants (Fig. 5(d)) rated positively (rating $>$ 3) and expressed their interest in incorporating unit testing with mocks to enhance software reliability. We also asked participants to share open-ended feedback on the advantages and disadvantages of *KUnit*. The first two authors conducted an open coding phase over the qualitative responses [52] and grouped codes into different themes. For advantages, our inductive thematic analysis identified 6 repeated themes in participants' qualitative responses. The themes are: *bugs can be detected early on, makes testing easy/easier to manage, time efficient/saves a lot of time, automation reduces human efforts, saving resources, great experience/helpful/useful*. For disadvantages, we found 2 repeated themes: *implementation in the industry could be challenging/overhead to set up* and *incorrect reports/false alarms.* Detailed qualitative responses are provided in the supplementary material [53]. Fig. 5 shows the survey results with participants ratings. As illustrated in Fig. 5(a), 34 participants rated positively (rating $>$ 3) and found that mocks help test each component independently. The learning curve for *KUnit* depends on the user's familiarity with DL concepts and

experience with Keras. To help *KUnit* users easily understand the workflow, we provided detailed documentation and a running example in our GitHub repository [16]. Participants in the post-study feedback illustrated that the well-structured documentation helped them easily understand *KUnit*'s functionality and workflow. After familiarizing themselves with the workflow, which on average takes 10-15 minutes, they only need to adjust the interfaces to align with their task. In the data preparation stage, the user loads the original dataset, applies preprocessing steps, and executes the test file.*KUnit* generates a mock model and feeds preprocessed data into it to verify data quality. In the model design stage, users build the DNN model and run the test. *KUnit* generates mock data and feeds it into the designed model to verify its correctness and compatibility with expected data properties. In the post-study feedback, participants mentioned that they found *KUnit* easy to use with minimal manual effort required for setup and customization. They were able to adjust the interfaces to align with their task and used the framework for developing and unit testing their DL application. By making *KUnit* open-source, we provide flexibility to adapt the tool to developer needs and improve its accuracy and ease of use through community contributions. *In summary, we found that the developers view unit testing using mocks as a valuable addition to traditional DL testing techniques. It enables independent testing of each component, facilitates early problem detection during development, and contributes to improving the overall code structure.*

V. Threats to Validity

A potential threat to the internal validity of our study is the possibility of bugs in *KUnit*'s implementation, which could lead to inaccurate results. To mitigate this risk, we conducted a user study involving developers with diverse expertise levels and backgrounds. This approach provided multiple perspectives on *KUnit*'s functionality. Additionally, we have made *KUnit*'s source code publicly available, allowing other researchers to review and validate our work. In the user study, participants used DeepDiagnosis, which detects one bug at a time. Participants manually addressed each bug identified by DeepDiagnosis and repeated the process until no further issues were reported. To mitigate the risk due to incorrect tool usage, we had our protocol reviewed by the authors of DeepDiagnosis to ensure it aligned with its intended functionality [54]. Our proposed approach may be affected by external threats, such as imprecise conditions used as assertions and the effectiveness of the actionable fixes provided as solutions. To address this issue, we have adopted guidelines from previous works [24], [19], [22], [42], [43], [21], [23] and Keras documentation [44], [45]. In our empirical evaluation, we assessed 50 DL programs in our benchmark. This process involved manually separating each DL program into two parts: data preparation and model design, which could introduce human error. To mitigate this threat, one of the co-authors thoroughly examined each part to ensure its correctness.

VI. Related Work

A. Unit Testing using Mocks

Unit testing stands as a fundamental practice in software development, aimed at evaluating each functionality of the software independently and uncovering bugs early in the development cycle. Due to dependencies, testing the code in isolation becomes challenging. To tackle this challenge, a technique known as mock objects have been proposed in the past by Mackinnon *et al.* [14] for unit testing, involving the replacement of dependencies with dummy implementations. Their findings suggest that creating unit tests using mock objects leads to better tests and improves the structure of both domain and test code. Prior studies have illustrated the benefits of employing mock objects for unit testing different applications, ranging from servlet [55], multi-agent systems [56], mobile apps [57] to database applications [58]. However, the use of mock objects for testing DL applications has not been investigated before.

B. Fault Localization and Bug Repair in DL Programs

The rise in DL application usage has led researchers to adapt fault localization techniques to this field, aiming to validate various components of DL-based systems and pinpoint faulty behaviors. In the past, various static and dynamic analysis approaches have been proposed for DL programs. NeuraLint [41] is a static analysis approach for automatic fault detection in DL programs that uses predefined rules to identify bugs. UMLAUT [17] combines static and dynamic analysis by examining program structure before training and model behavior during training. DeepLocalize [18] is a dynamic fault localization approach for DL programs that identifies numerical errors during training. AutoTrainer [22] is a system designed to identify and repair 5 common training issues in DL models. DeepDiagnosis [19] is a dynamic technique that identifies various symptoms during training and suggests actionable fixes. DeepFD [21] is a learning-based fault localization framework for diagnosing faults in DL programs. [20] is a property-based debugging approach that detects bugs in DL programs before, during, and after training. deepmufl [23] is a mutation-based fault localization approach for DL programs that generates mutants of pre-trained models to detect bugs in these programs. Prior works treat data preprocessing steps and the model as a comprehensive DL program, and identify and localize bugs by monitoring the training process. In contrast, *KUnit* treats data and the model as independent entities and aims to detect bugs before integrating them.

VII. Conclusion

This paper introduces the concept of mock testing in the context of DNNs and presents a novel technique, *KUnit*. The technique is based on the idea of decoupling to reduce the dependencies between different stages of DL applications, specifically data preparation and model design. Decoupling is achieved by defining interfaces that facilitate the creation of mock objects for unit testing of each stage. The empirical evaluation using 50 DL programs shows that in the data

preparation stage, the mock model helped identify 10 issues, and mock data assisted in identifying 53 issues in the model design stage. In a user evaluation with 36 participants testing 15 programs, *KUnit* helped resolve 25 issues in the data preparation stage and 38 issues in the model design stage. Our results show that mock objects provided a lightweight emulation of the dependencies for unit testing and identified issues during unit testing that, in current practice, are typically identified after data and model integration, specifically during DL model training. Participants using *KUnit* found it helpful for identifying and resolving issues early in the development process.

VIII. Data Availability

Our evaluation results and the code for our framework, *KUnit*, are available in our replication package [16].

References

[1] C. Richard, "Deep learning based chatbot models," in *arXiv preprint arXiv:1908.08835*, 2019.

[2] I. Giancarlo, L. L. Bello, A. Nucita, and G. M. Grasso, "A vision and speech enabled, customizable, virtual assistant for smart environments," in *In 2018 11th International Conference on Human System Interaction (HSI)*, 2018, pp. 50–56.

[3] R. Abhimanyu, J. Sun, R. Mahoney, L. Alonzi, S. Adams, and P. Beling, "Deep learning detecting fraud in credit card transactions," in *In 2018 systems and information engineering design symposium*, 2018, pp. 129–134.

[4] T. Zhang, C. Gao, L. Ma, M. Lyu, and M. Kim, "An empirical study of common challenges in developing deep learning applications," in *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*, 2019, pp. 104–115.

[5] X. Zhang, Y. Yang, Y. Feng, and Z. Chen, "Software engineering practice in the development of deep learning applications," in *ICSE'20: The 42nd International Conference on Software Engineering*, 2020.

[6] M. J. Islam, G. Nguyen, R. Pan, and H. Rajan, "A comprehensive study on deep learning bug characteristics," in *ESEC/FSE'19: The ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, ser. ESEC/FSE 2019, August 2019.

[7] N. Humbatova, G. Jahangirova, G. Bavota, V. Riccio, A. Stocco, and P. Tonella, "Taxonomy of real faults in deep learning systems," in *ICSE'20: The ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 1110–1121.

[8] Y. Zhang, C. Yifan, C. Shing-Chi, X. Yingfei, and Z. Lu, "An empirical study on tensorflow program bugs," in *27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2018, pp. 129–140.

[9] J. Cao, B. Chen, C. Sun, L. Hu, S. Wu, and X. Peng, "Understanding performance problems in deep learning systems," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 357–369.

[10] M. S. Rahman, F. Khomh, A. Hamidi, J. Cheng, G. Antoniol, and H. Washizaki, "Machine learning application development: practitioners' insights," *Software Quality Journal*, vol. 31, no. 4, pp. 1065–1119, 2023.

[11] S. Biswas, M. Wardat, and H. Rajan, "The art and practice of data science pipelines: A comprehensive study of data science pipelines in theory, in-the-small, and in-the-large," in *ICSE'22: The 44th International Conference on Software Engineering*, May 21-May 29 2022.

[12] 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 *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, 2019, pp. 291–300.

[13] R. Binder, *Testing object-oriented systems: models, patterns, and tools*. Addison-Wesley Professional, 2000.

[14] T. Mackinnon, S. Freeman, and P. Craig, "Endo-testing: unit testing with mock objects," in *Extreme programming examined*, 2000, pp. 287–301.

[15] I. Goodfellow, Y. Bengio, and A. Courville, *Deep learning*. MIT press, 2016.

[16] https://github.com/anon3173/KUnit, 2024.

[17] E. Schoop, F. Huang, and B. Hartmann, "Umlaut: Debugging deep learning programs using program structure and model behavior," in *Proceedings of the 2021 CHI Conference Extended Abstracts on Human Factors in Computing Systems*, 2021.

[18] M. Wardat, W. Le, and H. Rajan, "Deeplocalize: fault localization for deep neural networks," in *ICSE'21: The 43nd International Conference on Software Engineering*, 2021.

[19] M. Wardat, B. D. Cruz, W. Le, and H. Rajan, "Deepdiagnosis: Automatically diagnosing faults and recommending actionable fixes in deep learning programs," in *ICSE'22: The 44th International Conference on Software Engineering*, 2022.

[20] H. Ben Braiek and F. Khomh, "Testing feedforward neural networks training programs," *ACM Trans. Softw. Eng. Methodol.*, vol. 32, no. 4, may 2023. [Online]. Available: https://doi.org/10.1145/3529318

[21] J. Cao, M. Li, X. Chen, M. Wen, Y. Tian, B. Wu, and S.-C. Cheung, "Deepfd: Automated fault diagnosis and localization for deep learning programs," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 573–585.

[22] X. Zhang, J. Zhai, S. Ma, and C. Shen, "Autotrainer: An automatic dnn training problem detection and repair system," in *ICSE'21: The 43rd International Conference on Software Engineering*, 2021, pp. 359–371.

[23] A. Ghanbari, D.-G. Thomas, M. A. Arshad, and H. Rajan, "Mutation-based fault localization of deep neural networks," in *ASE'2023: 38th IEEE/ACM International Conference on Automated Software Engineering*, September 11–15 2023.

[24] M. J. Islam, R. Pan, G. Nguyen, and H. Rajan, "Repairing deep neural networks: Fix patterns and challenges," in *ICSE'20: The 42nd International Conference on Software Engineering*, May 23-May 29, 2020 2020.

[25] P. Runeson, "A survey of unit testing practices," *IEEE software*, vol. 23, no. 4, pp. 22–29, 2006.

[26] D. L. Parnas, "On the criteria to be used in decomposing systems into modules," *Communications of the ACM*, vol. 15, no. 12, pp. 1053–1058, 1972.

[27] V. Borisov, T. Leemann, K. Seßler, J. Haug, M. Pawelczyk, and G. Kasneci, "Deep neural networks and tabular data: A survey," *IEEE Transactions on Neural Networks and Learning Systems*, 2022.

[28] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[29] T. Yu and H. Zhu, "Hyper-parameter optimization: A review of algorithms and applications," *arXiv preprint arXiv:2003.05689*, 2020.

[30] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," in *Proceedings of IEEE*, vol. 11, 1998, pp. 2278–2324.

[31] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional networks," in *NIPS'12: The 25th International Conference on Neural Information Processing Systems*, vol. 1, 2012, p. 1097–1105.

[32] Y. LeCun, L. Bottou, G. B. Orr, and K.-R. Müller, "Efficient backprop," in *Neural networks: Tricks of the trade*. Springer, 2002, pp. 9–50.

[33] Y. Bengio, "Practical recommendations for gradient-based training of deep architectures," in *Neural Networks: Tricks of the Trade: Second Edition*. Springer, 2012, pp. 437–478.

[34] Francois Chollet, "Keras: the Python deep learning library," 2015, https://keras.io/api/losses/.

[35] A. Ng, "Machine learning course," https://www.coursera.org/learn/neural-networks-deep-learning.

[36] "Occam's razor," https://en.wikipedia.org/wiki/Occam%27s_razor.

[37] scikit-learn, "sklearn.datasets: Samples generator," 2007, https://scikit-learn.org/stable/modules/classes.html#module-sklearn.datasets.

[38] Y. A. L. Léon, B. B.Orr, and K.-R. Müller, "Efficient backprop." Berlin, Heidelberg: Springer, 2012.

[39] "Rule of 10," https://machinelearningmastery.com/much-training-data-required-machine-learning/, 2023.

[40] V. Lakshmanan, S. Robinson, and M. Munn, *Machine learning design patterns*. O'Reilly Media, 2020.

[41] A. Nikanjam, B. B. Houssem, M. M. Mohammad, and K. Foutse, "Automatic fault detection for deep learning programs using graph transformations," in *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 31, no. 1, 2021, pp. 1–27.

[42] S. Ahmed, S. M. Imtiaz, S. S. Khairunnesa, B. D. Cruz, and H. Rajan, "Design by contract for deep learning apis," in *ESEC/FSE'2023: The 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, December 03-December 09 2023.

[43] S. S. Khairunnesa, S. Ahmed, S. M. Imtiaz, H. Rajan, and G. T. Leavens, "What kinds of contracts do ml apis need?" *Empirical Software Engineering*, vol. 1, no. 1, March 2023.

[44] Francois Chollet, "Keras: the Python deep learning library," 2015, https://keras.io/.

[45] ——, "Keras: the Python deep learning library," 2015, https://keras.io/examples/.

[46] "Kaggle ," https://www.kaggle.com/competitions, 2024.

[47] G. Codespaces, "Github," https://github.com/features/codespaces.

[48] M. C. Davis, S. Choi, S. Estep, B. A. Myers, and J. Sunshine, "Nanofuzz: A usable tool for automatic test generation," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2023, pp. 1114–1126.

[49] https://github.com/anon3173/KUnit/blob/main/RQ2_Results.xlsx, 2024.

[50] https://github.com/anon3173/KUnit/blob/main/RQ2_UserStudy_Results.xlsx, 2024.

[51] https://github.com/anon3173/KUnit/blob/main/Analysis_of_Debugging_Time.pdf, 2024.

[52] R. S. Weiss, *Learning from strangers: The art and method of qualitative interview studies*. Simon and Schuster, 1995.

[53] https://github.com/anon3173/KUnit/blob/main/Participants_Response.pdf, 2024.

[54] Authors, "Personal Communication with Authors regarding DeepDiagnosis: Automatically Diagnosing Faults and Recommending Actionable Fixes in Deep Learning Programs ," Email, March, 2024.

[55] D. Thomas and A. Hunt, "Mock objects," *IEEE Software*, vol. 19, no. 3, pp. 22–24, 2002.

[56] R. Coelho, U. Kulesza, A. von Staa, and C. Lucena, "Unit testing in multi-agent systems using mock agents and aspects," in *Proceedings of the 2006 international workshop on Software engineering for large-scale multi-agent systems*, 2006, pp. 83–90.

[57] M. Fazzini, A. Gorla, and A. Orso, "A framework for automated test mocking of mobile apps," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, 2020, pp. 1204–1208.

[58] K. Taneja, Y. Zhang, and T. Xie, "Moda: Automated test generation for database applications via mock objects," in *Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering*, 2010, pp. 289–292.