5-2022

Testing of Neural Networks

Devan Lad
dpl2047@rit.edu

Testing of Neural Networks

by

Devan Lad

A Thesis Submitted in Partial Fulfillment of the Requirements for the Degree of Master of Science in Software Engineering

Supervised By
Dr. Mohamed Wiem Mkaouer

Department of Software Engineering
B. Thomas Golisano College of Computing and Information Sciences
Rochester Institute of Technology
Rochester, NY

May 2022

Committee Approval:

_____

Dr. J Scott Hawker Date
SE Graduate Program Director

_____

Dr. AbdelRahman Essaied Date
Assistant Professor (Advisor)

_____

Dr. Mohamed Wiem Mkaouer Date
Assistant Professor (Advisor)

To my family and friends, for their continued support and encouragement.

Acknowledgments

Accomplishing this work would not have been possible single-handedly. I am grateful to the many individuals who supported, advised, and encouraged me throughout this process.

My thanks to my advisor, Dr. Mohamed Wiem Mkaouer and Ahmed Ahmed Elsaid, for their dedication and guidance with this research. This work would not have been possible without their knowledge and experience.

Thank you to all the friends and colleagues who provided me with encouragement and support throughout my Maser's program. Many thanks also to my family for their continued support and encouragement in pursuing my Maser of Science in Software Engineering.

Abstract

Research in Neural Networks is becoming more popular each year. Research has introduced different ways to utilize Neural Networks, but an important aspect is missing: Testing. There are only 16 papers that strictly address Testing Neural Networks with a majority of them focusing on Deep Neural Networks and a small part on Recurrent Neural Networks. Testing Recurrent neural networks is just as important as testing Deep Neural Networks as they are used in products like Autonomous Vehicles. So there is a need to ensure that the recurrent neural networks are of high quality, reliable, and have the correct behavior. For the few existing research papers on the testing of recur- rent neural networks, they only focused on LSTM or GRU recurrent neural network architectures, but more recurrent neural network architectures exist such as MGU, UGRNN, and Delta-RNN. This means we need to see if ex- isting test metrics works for these architectures or do we need to introduce new testing metrics. For this paper we have two objectives. First, we will do a comparative analysis of the 16 papers with research in Testing Neural Networks. We define the testing metrics and analyze the features such as code availability, programming languages, related testing software concepts, etc. We then perform a case study with the Neuron Coverage Test Metric. We will conduct an experiment using unoptimized RNN models trained by a tool within EXAMM, a RNN Framework and optimized RNN Models trained and optimized using ANTS. We compared the Neuron Coverage Outputs with the assumption that the Optimized Models will perform better.

Contents

1 Introduction **1**

2 Background **4**
2.0.1 Software Testing 4
2.0.2 EXAMM Framework 5

3 Research Objective **6**
3.1 Motivation 6
3.2 Contribution 6
3.3 Research Questions 6

4 Research Findings **8**
4.1 RQ1: What Neural Network testing tools/approaches are available to the community and what types of neural networks are testing(DNN, RNN, CNN, etc)? 8
4.1.1 DNN Related Testing tools/approaches 8
4.1.2 RNN Related Testing tools/approaches 11
4.1.3 Programming Languages 13
4.2 RQ2: What are the Software testing-related concepts used(Coverage testing, Mutation testing, etc)? 14
4.2.1 Coverage Testing 14
4.2.2 Mutation Testing 17
4.3 RQ3: What data sets are used to train the neural network? . . 18
4.3.1 Modal and Data sets Used 18
4.4 Discussion 19

5 Case Study **20**
5.1 Neuron Coverage 20
5.2 Implementation 21
5.3 Data Set and Experiment Design 24
5.4 Results 24
5.4.1 RQ1: Does the optimized RNNs perform better then the unoptimized RNN? 25
5.4.2 RQ2: Is coverage-based testing appropriated for Δ-RNN, GRU, MGU, and UGRNN recurrent neural network architectures? Do we need to design new coverage metrics for the non-LSTM recurrent neural networks? 26

6 Threats to Validity **28**
6.1 Internal Validity 28
6.2 External Validity 28

7 Conclusion & Future Work **30**

List of Figures

5.1 Example of a RNN model trained by EXAMM with the layers labeled 22
5.2 Code for Creating the Dictionary of Nodes 23
5.3 Code for determining coverage and calculating Neuron Coverage 23
5.4 Unoptimized and Optimized Neuron Coverage results 24

List of Tables

4.1 Corpus of Papers related to Testing Neural Networks 15

Introduction

Nowadays deep neural networks(DNN) are prevalent in many areas such as autonomous driving [1], medical diagnosis [2], computer vision [3], and automatic speech recognition [4]. The applications have deep neural networks as important components, meaning there is a need to ensure that neural networks behave as intended. Similar to traditional software systems, deep neural network systems have to deal with erroneous behavior. Neural network systems have bugs or vulnerabilities that occur, leading to problems such as unexpected crashes. These bugs can be the DNN performing incorrect/unexpected behavior. This problem becomes bigger, especially when you have gone deep into training thousands of neural networks, and it crashes unexpectedly. Traditional and Neural Network testing behave differently due to different programming paradigms and development processes, which means traditional software testing practices are hard to use for Neural Network testing. As a result, research in testing deep neural networks has become popular in recent years as approaches are introduced to test and identify defects [11]– [19]. These approaches help ensure the correctness and quality of the neural networks. These approaches can be seen as part of software quality assurance in general, where there are plenty of studies targeting the improvement and refactoring of existing software systems [5–84].

Despite the growing popularity of testing approaches for Neural networks specifically DNN, this area of research is still in its infant stage. There are only a few papers in this research area compared to the general research area of Neural Networks. Most existing approaches are not designed to be applied

to other Neural Networks, such as Recurrent Neural Networks(RNN) which is the Neural Network we will be focusing on in this paper. Similar to DNN, Recurrent neural networks have to deal with bugs, so testing is needed. In addition, recurrent neural networks need to be reliable, high quality, and have the correct behavior. Testing helps ensure this, but recurrent neural networks are much more complex than DNN. This makes designing testing approaches much more difficult. Recurrent neural networks are a network of nodes that are structured into successive iterations [17]. These iterations form temporal behavior, which is an essential behavior of recurrent neural networks. They use sequential data or time-series data when creating neural networks. In a recurrent neural network, the connections between the nodes can go in any direction to another node and can also skip over nodes to other nodes. The output of these nodes has have affect surrounding nodes whereas DNN assumes the input and outputs are independent. The studies introduce several testing approaches adapted from traditional software testing methods like coverage testing and mutation testing. As mentioned before, these approaches focus on DNN, but some approaches do focus on RNN. These approaches have only focused on long-short memory networks(LSTM) [85] and gated recurrent unit(GRU) [86] which are popular architectures of recurrent neural networks. There has been no literature review for these papers that look at the approaches specifically the testing metrics and what metrics are suitable for a specific type of Neural Network or multiple types. In this paper, we will be doing a literature review on the papers found in testing Neural networks where we will be studying the testing metrics introduced and used in each paper and how they are used to test neural networks. We will be comparing and contrasting multiple parts

of these approaches such as target Neural networks, data set testing, coding availability, testing metrics, etc. Later on in the paper, we will go into the implementation of the Neuron coverage testing metric to see if that metric is adaptable with the EXAMM framework [87]zza which is a neuro-evolution Recurrent Neural Neural Network.

Paper Structure Section 2 provides a background of testing and EXAMM Framework. Section 3 discusses the research objectives. Section 4 details the papers and compares and contrasts to answer research questions. Section 5 goes into detail of a selected testing metric and the reasoning for the choice. The implementation and the results are also discussed here as well. Section 6 goes over the threats to the paper. The paper is concluded in section 7 with a summary of the results and a discussion of future work.

Background

2.0.1 Software Testing

Testing has always been an integral part of software development. Testing is an activity or group of activities performed to determine the quality of software. This means checking whether the software fulfills the set requirements and finding bugs in the software. 2 types of general testing approaches can be done: white box testing and black-box testing. White box testing focuses on the internal code whereas black block testing focuses on the functionality. From here there are specific testing levels: Unit Testing, Integration Testing, System Testing, and Acceptance Testing. Unit testing checks the basic level of input and outputs and the function's behavior. Integration testing test that different parts of the software function together. System testing tests the entire software, and acceptance testing is customer testing like a beta test. These are the basic level testing, but other testing methods like coverage testing help with making sure the test themselves are of quality. Some of these testing concepts can be adapted easily to Neural Networks and some can't. A big factor is developers build traditional software through logic consisting of control flow statements whereas Neural Network-based software automatically learns its logic with little to no human interaction through weights and activation functions [88].

2.0.2 EXAMM Framework

For some discussions in the literature review and the subject for testing for the implementation part, we will be using the Evolutionary eXploration of Augmenting Memory Models(EXAMM) framework [87]. EXAMM evolves Recurrent Neural Networks using memory structures like Δ-RNN, GRU, LSTM, MGU, and UGRNN. The Recurrent Neural network is evolved through mutation and Recombination operations. Examples of these operations are "Disable edge" which disables a random edge, Add Edge which adds an edge to a random set of nodes, Merge Node which combines two random nodes, and many more. These mutations are done randomly as the model is generated. We will be using a tool inside of EXAMM that will train unoptimized RNN models. These unoptimized models are use as comparison vs optimized models.. For optimize RNN models, we use an ANTS(Ant-based Neural Topology Search) algorithm that optimizes recurrent neural networks [89] [90]. This algorithm uses ants to walk a path through a Recurrent Neural Network model and selects the best performing paths to generate a better neural network. Paths are selected based on a pheromone value which is the probability of it getting selected by an ant. This is done with several neural networks where the best performing networks paths are rewarded with a higher pheromone for a higher chance of being selected. The worse performing networks receive a lower pheromone. This is done until a final Recurrent Neural Network is outputted. ANTS trains and optimizes RNNs models.

Research Objective

3.1 Motivation

Testing is an important part of building software as it ensures the software is of quality and working. This means that Neural Network-based software should be tested as well. Because this area of research is still new, there are no literature reviews on the existing testing approaches for Neural Networks. In addition, there are no studies on if the current testing approaches are adaptable to different types of Neural Networks outside of the initial test subject.

3.2 Contribution

The main contribution of this study is the creation of a literature review of the existing testing approaches for Neural Networks. These test approaches are compared and contrasted to help future researchers determine what metrics might work for their research. In addition, we attempt to adapt neuron coverage a testing metric to the EXAMM Framework to study if the metric is useful.

3.3 Research Questions

Literature Review Research Question

- **RQ1:** What Neural Network testing tools/approaches are available to the community and what types of neural networks are testing(DNN,

RNN, CNN)? Makes note of the available papers that address testing neural networks and what specific neural networks the papers force on.

- **RQ2:** What are the Software testing-related concepts used(Coverage testing, Mutation testing, etc)? Takes a look at the software testing concepts used in the approaches presented by the papers. This means how the testing is done, whether they looked at edge cases if they tried to copy the concepts or adapt them, etc.

- **RQ3:** What data sets are used to train the neural network? just looks at what data sets are used to train the neural networks, EXAMM mainly focuses on time sets/numbers, but there are different types of data sets like images.

Case Study Research Questions

- **RQ1:** Does the optimized RNNs perform better than the unoptimized RNNs? We also analyze if the optimized RNN models will perform better than the unoptimized RNN models at a given threshold.

- **RQ2:** Is coverage-based testing appropriated for Δ-RNN, GRU, MGU, and UGRNN recurrent neural network architectures? Do we need to design new coverage metrics for the non-LSTM recurrent neural networks? Analyzes the implementation process and/or data to determine whether the coverage base testing is appropriate or not for the Δ-RNN, GRU, LSTM, MGU, and UGRNN classes. We are answering whether adapting the test coverage metrics is enough for the recurrent neural networks, or do we need to introduce new test metrics for the different classes.

Research Findings

In this section, we present the findings for the proposed RQs based on a set of 15 Neural Network testing-related publications mixed with publications that introduced testing approaches and publications that adopted testing approaches. The papers were found manually and were found with the restriction of 2018 to current. A crawler was used as well, but we found no relevant papers in regard to our topic.

4.1 RQ1: What Neural Network testing tools/approaches are available to the community and what types of neural networks are testing(DNN, RNN, CNN, etc)?

This RQ will consist of three parts. Part one will provide an overview of the DNN related publications and test metric definitions. Part two will provide an overview of the RNN-related publications and test metric definitions. Part three will look at the programming languages used to develop these testing approaches and the code base availability.

4.1.1 DNN Related Testing tools/approaches

This part, we will provide an overview of the publications that tested DNN with their test approach. The publications are shared in no particular order. There is only one paper that tested multiple Neural Networks, but DNN was

the main focus.

One of the starting points for Neural Network testing is DeepXplore [91]. Here they introduce a white box differential testing algorithm that finds inputs that trigger problems between multiple DNN. They also introduce one of the first coverage metrics for DNN, Neuron Coverage(NC) which is a metric that measures how much of a DNN has been tested. This is done by looking at the ratio of active neurons to all neurons. Active neurons are determined based on a predefined threshold. From here, other approaches have integrated Neuron Coverage or have discovered other metrics for testing Neural Networks. DeepTest [88] is a testing tool for detecting erroneous behavior in vehicles that use DNN. Neuron Coverage is used to create test cases to maximize Neuron Coverage. They tested against convolutional neural networks(CNN) and LSTM recurrent neural networks. DeepTest tries to address DNN, CNN, and recurrent neural networks at once.

There are some proposals for new metrics for DNN for test coverage after Neuron Coverage was published. DeepGuage [92] built upon Neuron Coverage [91]. Here they introduce more test coverage criteria: k-Multisection Neuron Coverage(KMNC), Neuron Boundary Coverage(NBC), Strong Neuron Activation Coverage(SNAC), Top-k Neuron Coverage, and Top-k Neuron Patterns. k-multisection Neuron Coverage measures the ratio of all covered sections of all neurons of a DNN. A neuron has a range of values that are split into sections. Input covers a section if the output values fall in the value range. Neuron Boundary Coverage measures the extent corner case regions are covered outside of the main functional range. Strong Neuron Activation Coverage measures how the upper(above maximum value) corner cases are

covered. Top-k Neuron Coverage measures the ratio of neurons with the most active neurons on each layer given a test set. Top-k Neuron Patterns look at the neuron patterns(different activated scenarios) for the top active neurons of each layer. Their proposed criteria help develop insight on the testing quality of DNN guiding effective testing. [93] introduces four test criteria that are inspired by MC/DC test from traditional software testing. These are created specifically for the structures of DNN. The test cases are generated based on the symbolic approach and gradient-based heuristic search. Their approach shows the importance of balancing between finding bugs and the computation cost of generating test cases. This paper's research is done based on the internal structure of DNN, which means we could not consider these metrics or approaches due to recurrent neural network behaviors being different.

DeepHunter [94] proposes a coverage-guided fuzz testing framework for DNN where they implement the test criteria mentioned in [91, 92]. The DeepHunter tool implements the test criteria and seed selection strategies. They test how effective the tool is for coverage. A white-box testing approach exist called called ADAPT [95] for DNN. This approach uses an adaptive neuron selection strategy instead of a predetermined neuron selection. This allows the tool to be more effective as neuron selection determines the effectiveness of white box testing. This allows the white-box testing to adapt to different neural network models and coverage metrics where in this case they test Neuron Coverage and Top-k Neuron Coverage. DeepMutation [96] similarly does mutation testing for DNN. FFor this approach, they specialize in mutation testing to determine test data quality. This is done following the same concepts of traditional mutation testing where faults are injected and effectiveness

is determined by if these faults are detected. There are two metrics introduced in the paper. MutationScore is defined by the set of classes mutant killed by the test data. "Killed" means the test input data used on the mutant causes different behavior from the original. AverageErrorRate measures the overall difference in behavior introduced by the mutation operators.

Another paper that takes a separate path called Concolic Testing to help determine coverage [97]. They still use previous coverage metrics such as Neuron Coverage, MC/DC coverage, and Neuron Boundary coverage. In this paper the focus in generating a test suite to ensure coverage for a DNN, this is done by analyzing the activation patterns of the DNN and symbolic generation of new inputs. DLFuzzing [98] introductions the first differential fuzzing testing framework for DNN. It mutates the input to ensure that Neuron Coverage is maximized and generates adversarial inputs without needing an outside source like a similar DNN system or manual labeling. Another paper uses fuzzing called SENSEI [99]. They use fuzzing to data augment the training data which means adding some noise to the data to help the neural network become more robust. Tensorfuzzing [100] uses coverage guide fuzzing to determine coverage based on Approximate Nearest Neighbor Algorithm(ANN). ANN looks at neighboring inputs to see if they are covered and determines based on that if the current input is covered

4.1.2 RNN Related Testing tools/approaches

Three papers investigate the testing of recurrent neural networks directly. The first paper is TestRNN [101]. This is a coverage guide testing approach for LSTM class recurrent neural networks. They design LSTM related cover-

age metrics: boundary coverage, step-wise coverage, and temporal coverage. Boundary Coverage covers the boundary values of the LSTM data flow. Step-wise Coverage looks at the temporal changes between connected cells. Finally, temporal Coverage exploits temporal patterns of bounded length. Their paper aims to show that their testing approach is efficient and effective at achieving high coverage and detecting erroneous behavior for LSTM.

DeepStellar [102] is a general-purpose quantitative analysis framework for RNN-based systems. They introduce several coverage metrics such as Trace similarity, basic state coverage, and basic transition coverage based on an abstract model created from RNN internal characteristics. DeepStellar detects adversarial samples and does coverage guide testing based on the metrics and test criteria. Here their coverage metrics are based on a state level and transition level where they are looking at how much of the internal states and how thoroughly the recurrent neural network is tested. Trace Similarity Metrics quantify the prediction differences of different inputs at state-based and transition-based levels. For state-level coverage, Based State Coverage measures how thoroughly the test inputs cover the major function region visited while training where it treats every state as equal. Weighted State Coverage is similar to Base State Coverage, but instead, a weight can be assigned to emphasize some states more than others. N-Step State Boundary Coverage measures how well test inputs cover the corner case regions. Basic Transition Coverage compares the transitions exercised during the training and testing for transition-level coverage. Weighted Transition Coverage is similar to Basic Transition Coverage, but weight can be assigned to emphasize some transitions.

Another approach is RNN-Test [103] which expands the adversarial testing research area for RNN. They focus on RNN systems with sequential outputs, where the RNN-test focuses on the main sequential structure without any limitations of tasks. They introduce two coverage metrics Hidden state coverage and cell state coverage. Hidden State Coverage captures RNN prediction logic. Cell State Coverage(design for LSTM) looks at more sections to explore more context space. Their evaluation showed that their coverage testing outperformed Neuron Coverage. DeepMutation++ [104] is written by the same authors as DeepMutation [96]. Here they focus on mutation testing RNN and Feed-forward Neural networks(FNN). The metrics used here are Kscore1 and Kscore2. Kscore1 calculates the killing score for whole data(DNN). Larger the value of Kscore1 the less robust the model is on the input. Kscore2 calculates the killing score for a segment(RNN model). Indicates the prediction probability divergence on the output. The larger the value of kscore2, the less robust the model is against the segment.

4.1.3 Programming Languages

The programming languages used to develop the testing approaches were consistent when looking at all the papers. The most commonly used language was python as shown in 4.1. There is no surprise with this discovery as several python libraries exist for Neural Network development that researchers can use. There are a couple of outliers as the adapt [95] paper uses Jupyter Notework to develop the adapt framework. TensorFuzz [100] in addition to Python also uses C++. EXAMM Framework that will be used for generating models later on in the paper is built using C++. This is taken into consideration

when we decide on a testing approach to adapt as we want to make sure we can translate the python code to c++. Majority of the papers have the code base available except for DeepGuage [92], Testing Deep Neural Network [93], DeepMutation [96], and DeepHunter [94].

4.2 RQ2: What are the Software testing-related concepts used(Coverage testing, Mutation testing, etc)?

This RQ will have three parts. Part one will discuss the Coverage related concepts. Part two will discuss the mutation-related concepts.

4.2.1 Coverage Testing

In figure 4.1, the testing metric column shows all the testing metrics that the papers use. A common word that is used is coverage which is related to coverage testing. Coverage testing in the traditional software setting has two parts: Code Coverage and Test Coverage. Code Coverage ensures that all parts of the code are tested at least once. Test Coverage is making sure each requirement is tested at least once. 4.1 shows that 13 of 15 papers are coverage related metric. The coverage concept that these papers are adapted are all code coverage related, but they take a different approach than looking at the code coverage. They adapt the idea by looking at the coverage of the Neural Network. This means traversing the neural network looking at something specific in the neurons or edges. In addition to this new concept, there

Table 4.1: Corpus of Papers related to Testing Neural Networks

Study	Year	Type of NN tested	Testing Metrics	Tool Availability	Test Data Set	Language Written	Target Neural Networks models/data sets
DeepXplore: Automated Whitebox Testing of Deep Learning Systems	2017	General DNN	Neuron Coverage	https://github.com/peikexin9/deepxplore	"MNIST, ImageNet,Driving, Contagio/VirusTotal, and Drebin"	Python	"MNIST: LeNet-1, LeNet-4, LeNet-5;ImageNet: VGG-16, VGG-19, ResNet50;Driving: DAVE-orig, DAVE-norminit, DAVE-dropout;Contagio/VirusTotal: 3;DNN based on 135 static features for PDFrate;Drebin: 3 out 36 DNN from Grozze et al"
DeepTest: Automated Testing of Deep-Neural-Network-driven Autonomous Cars	2018	"DNN, RNN, CNN"	Neuron Coverage	https://github.com/ARiSE-Lab/deepTest	"Rambo, Chauffeur, Epoch"	Python	Rambo: 3 CNN;Chauffeur: CNN and LSTM;Epoch: CNN CH2_002 dataset
DeepGauge: Multi-Granularity Testing Criteria for Deep Learning Systems	2018	DNN	"k-multisection Neuron Coverage, Neuron Boundary Coverage, Strong Neuron Activation Coverage, Top-k Neuron Coverage, Top-k Neuron Patterns"	N/A	"MNIST, ImageNet"	N/A	"MNIST: LeNet-1, LeNet-4, LeNet-5;ImageNet: VGG-19, ResNet-50"
Testing Deep Neural Networks		DNN	"Sign-Sign Coverage/SS Coverage, Value-Sign Coverage/VS Coverage, Sign-Value Coverage/SV Coverage, Value-Value Coverage/VV Coverage"	N/A	"MNIST, CIFAR-10, ImageNet"	N/A	ImageNet: VGG16
DeepMutation: Mutation Testing of Deep Learning Systems	2018	DNN	"MutationScore, AveErrorRate(AER)"	N/A	"MNIST, CIFAR-10"	Python	popular models in previous work
DeepHunter: A Coverage-Guided Fuzz Testing Framework for Deep Neural Networks	2019	DNN	"Neuron Coverage, k-multisection neuron coverage, neuron boundary coverage, strong neuron activation coverage, top-k neuron coverage"	N/A	"MNIST, CIFAR-10, ImageNet"	Python	"MNIST: LeNet-1, LeNet-5;CIFAR-10: ResNet-20, VGG-16;ImageNet: MobileNet"
RNN-Test: Towards Adversarial Testing for Recurrent Neural Network Systems	2019	RNN	"Hidden State Coverage, Cell State Coverage"	https://github.com/wxlsummer/RNN-Test	"PTB language model, Spell checker model, DeepSpeech ASR model"	Python	PTB language model: Two layer LSTM;Spell checker model: Two later bi-direction LSTM;DeepSpeech ASR model: One-layer bi-direction LSTM with CNN layers;MNIST-LSTM model: One-layer LSTM
Coverage Guided Testing for Recurrent Neural Networks	2019	RNN	"Boundary Coverage, Step-wise Coverage, Temporal Coverage"	https://github.com/xiaoweih/testingRNN	"MNIST, IMDB, Lipophilicity, UCF101"	Python	"MNIST: LSTM;Lipophilicity prediction: LSTM model;IMBD: LSTM;UCF101: VGG16+LSTM, VGG16 is a CNN for imageNet"
Effective White-Box Testing of Deep Neural Networks with Adaptive Neuron-Selection Strategy	2020	DNN	"Neuron Coverage, Top-k Neuron coverage"	https://github.com/kupl/adapt	"MNIST, ImageNet"	Jupyter Notebook	"MNIST: LeNet-4, LeNet-5;ImageNet: VGG-19, ResNet-50"
DeepStellar: Model-Based Quantitative Analysis of Stateful Deep Learning Systems	2019	RNN	"Trace Similarity metrics: State and transition based Coverage: Basic State Coverage, Weighted State Coverage, n-Step State Boundary Coverage, Basic Transition Coverage, Weighted Transition Coverage."	https://github.com/xiaoningdu/deepstellar	"DeepSpeech 0.1.1 Bi-LSTM, DeepSpeech 0.3.0 LSTM, MNIST-LSTM LSTM, MNIST-GRU GRU"	Python	"DeepSpeech 0.1.1 Bi-LSTM;DeepSpeech 0.3.0 LSTM;MNIST: MNIST-LSTM LSTM, MNIST-GRU GRU"
Concolic Testing for Deep Neural Networks	2018	DNN	"Lipschitz Continuity, Neuron Coverage, Sign-Sign Coverage/SS Coverage, Neuron Boundary Coverage"	https://github.com/TrustAI/DeepConcolic	"MNIST, CIFAR-10"	Python	N/A
Fuzz Testing based Data Augmentation to Improve Robustness of Deep Neural Networks	2020	DNN	"Classification-based robustness, Loss-based robustness"	https://github.com/gaoxiang9430/sensei	"GTSRB, FM, CIFAR-10, IMDB, SVHN"	Python	"GTSRB: 4 models;Fashion-MNIST: 3 models;CIFAR-10: WideResnet, 3 ResNet models;SVHN: VGG model and another model from a paper;IMDB: VGG16, VGG19"
DLFuzz: Differential Fuzzing Testing of Deep Learning Systems	2018	DNN	Neuron Coverage	https://github.com/turned2670/DLFuzz	"MNIST, ImageNet"	Python	"MNIST: LeNet-1, LeNet-4, LeNet-5;ImageNet: VGG-16, VGG-19, ResNet50"
TensorFuzz: Debugging Neural Networks with Coverage-Guided Fuzzing	2019	NN	Approximate nearest neighbr(ANN) to determine coverage	https://github.com/tensorflow/tensorflow	MNIST	C++ and Python	N/A
DeepMutation++: a Mutation testing framework for Deep Learning System	2019	RNN FNN	"KScore1, Kscore2"	https://sites.google.com/view/deepmutationpp/home	IMDB	Python	LeNet-model

is a change in terms of unit testing. Unit testing is the usual way to ensure code coverage, but due to the adaption, a new approach is needed, which is

where test data comes in. Test data is used to test neural network models, but only the neural network's final output is looked at. Because of the new coverage concept, papers looked at what happens at each node when input is entered into the neural network model. So test data becomes important here as researchers can investigate how a node functions when an adversarial input is entered into the model. An adversarial input is an input that is similar to a working input, but the model generates a different output. The more adversarial inputs one has, the more tested/covered your neural network. Some papers also introduce algorithms to generate test data or adversarial inputs to help increase coverage. For example, Neuron Coverage [91] traverses the hidden layers of the neural networks, looks at the node's output, and compared it to a user input threshold to determine whether the node is activated/covered. From here their algorithm looks at the nodes not coverage and tries to generate a test input that would result in the node being covered. Overall Neuron Coverage is a good example of a general coverage metric for a neural network without the algorithm. Other coverage test metrics go into more detail when traversing the Neural network. DeepGuage [92] introduces several coverage test metrics that look at more edge cases and patterns. For example Neuron, Boundary Coverage looks at the edge cases outside of the expected outputs and Strong Neuron Activation Coverage looks passed the maximum limit output cases. The coverage test metrics from DeepGuage and DeepXplore(Neuron Coverage) are used as references or adapted into a different framework for most papers found. In addition, most coverage test metrics are only tested on DNN.

As mentioned before there are only four papers that focus on RNN with

one using a different testing concept, this means that the coverage metrics are made with RNN inner workings in mind. RNN-Test [103] created their coverage metrics hidden state and cell state coverage with the LSTM RNN in mind. There was also a focus on generating adversarial inputs due to other works having only focused on adversarial inputs for DNN. Their goal was to be able to generate adversarial inputs that were outside of the threshold of the training data whereas TestRNN [101] and Deepstellar [102] created their adversarial input generation within the thresholds of the training data they used. TestRNN coverage metrics are more temporal focused which is a big part of RNN. DeepStellar coverage metrics are based on the major functional regions and corner case regions. In general, all the test metrics follow similar patterns of introducing a test metric with the support of test cases or adversarial input generations. A common trend was papers reusing coverage test metrics and adding their algorithm to test the coverage.

4.2.2 Mutation Testing

Besides coverage testing, there was only one other type of testing: Mutation testing. DeepMutation [96] and DeepMutation++ [104] are both written by the same author and follow the same concept. DeepMutation is focused on DNN whereas DeepMutationn++ is focused on RNN. The traditional Mutation testing definition is creating programs with faults to see if a test can detect those faults. The faults are generated using Mutation Operator. The papers both introduce a neural network adapted mutation testing concept. For the DNN-related mutation testing, The data and training program are mutated using mutant operators and used to generate a mutated model. The

model is then tested with a test data set of successful inputs from the original model. The RNN-related mutation testing generates the RNN model and then mutates the model using mutant operators, then similar to the DNN testing, a test data set of successful inputs is used. Both papers have similar test metrics, where they are looking for the metric for how many mutations were caught.

4.3 RQ3: What data sets are used to train the neural network?

This RQ will have one part where we briefly discuss the models and data sets used by the papers.

4.3.1 Modal and Data sets Used

For all the papers two consistent names came up when looking at 4.1: MNIST and ImageNet. MNIST is a database with handwritten digits as images. ImageNet is a database that consists of millions of pictures for visual-based training. As they are both image-based data sets, some of the code for the paper is specifically image-based such as deepXplore where their adversarial generation is strictly image-based, but their Neuron Coverage code is generalized. In addition, there are some cases where MNIST was used for the testRNN [101] paper for testing. Outside of these common datasets, Papers used a variety of data sets such as CIRFAR-10, DeepSpeech, IMDB, etc.

4.4 Discussion

Our findings provide an understanding of the basics of all the papers. As shown by the RQ, this area of research is still new and there is still more potential to explore other testing concepts outside of coverage and mutation testing. Exploring more traditional testing concepts is an open playing field. We see that the early papers introducing coverage metrics had set a foundation and basic level concept for coverage testing a Neural network. There are some limits to considering what coverage test metric to adapt as code bases are missing For the existing codebases, there may be a learning curve due to the complexity and early stage of the research area. With several papers reusing old coverage test metrics, future researchers have an idea of what coverage test metrics may be easier to pick up since they get different perspectives on how the coverage testing metric is implemented. The Data sets do not have a big impact, but caution will be needed as testing metrics may be created specifically a data set.

Case Study

We did an experiment using the testing metric Neuron Coverage on several unoptimized RNN models generated by EXAMM and several optimized RNN models generated using the ANTS Algorithm. The case study will be split into fours part. We will discuss Neuron Coverage, the test metric that we will be implementing, how Neuron Coverage works, and why we chose it over other reviewed test metrics. Next, we will talk about the implementation process(language, testing, code, etc). We will discuss the experiment and, lastly, we will go through the experiment results. Coming into this case study, we assumed that the optimized model will perform better than the unoptimized one as the optimized model is more efficient in using the nodes than the unoptimized model's nodes.

5.1 Neuron Coverage

As mentioned in the literature review, Neuron Coverage was the test metric introduced in the deepXplore [91] paper. Neuron Coverage is measured by the ratio of unique nodes that have been activated for an input and the total amount of nodes in the Neural Network shown in figure5.3. A node is considered activated when its output is greater than a user input threshold. Neuron Coverage was selected as the test metric to adapt to the EXAMM framework

$$NeuronCoverage = \frac{|ActivatedNode|}{|TotalNode|}$$

because it was the most basic level test metric and coverage check among the test metrics introduced earlier in this paper. This means we wouldn't have to do any significant changes to the EXAMM framework code to access the node outputs in the neural network. Another reason was this test metric was adapted to several other papers showing how adaptable the test metric is. This also means that we had multiple code bases to study if we run into any problems with the original code base. Due to the simplicity of the codebase the adaption from python the language used in the original paper would be an easy conversion to C++ the language used in EXAMM.

5.2 Implementation

Our implementation Neuron Coverage was built using C++ on Linux. Our implementation consist of 27 lines of C++ code. We leverage several classes available in EXAMM that gave us access to parts of the Recurrent Neural Network model that was generated. For the Neuron Coverage implementation, we had to first study EXAMM and understand where the best place would be to put the code. We took advantage of a method in the RNN class called forward_pass(). This method loops the time series data and calculates the output for the nodes and edges(connections between nodes). For Neuron Coverage, we needed access to the output value for each node for each time series input and this method has the exact information that we need.

We took the next step of making a dictionary of all the hidden layer nodes. Here the key was a tuple with the hidden layer and innovation number. In EXAMM, each node is assigned a unique innovation number when a model

is generated. For a neural network, there are three-layer: input layer, hidden layer, and output layer. EXAMM assigns a number to each layer where input layer one is always 0, hidden layer is 1 to the number less than the output layer, and output layer is the max count of layers. An example of a Neural Network with the layers labeled is shown in figure 5.1. The value for the tuple key is a type Boolean where it's set as FALSE. We looped through the list of nodes and checked if the node was not in the input and output layers before adding it to the dictionary.

$$[(layer, innvoNum)] = false;$$

We then created a function called update_neuron_coverage(), where the Neuron Coverage is calculated for each node. This method is called for each time series input that is looped over. In the method, we loop the list of nodes

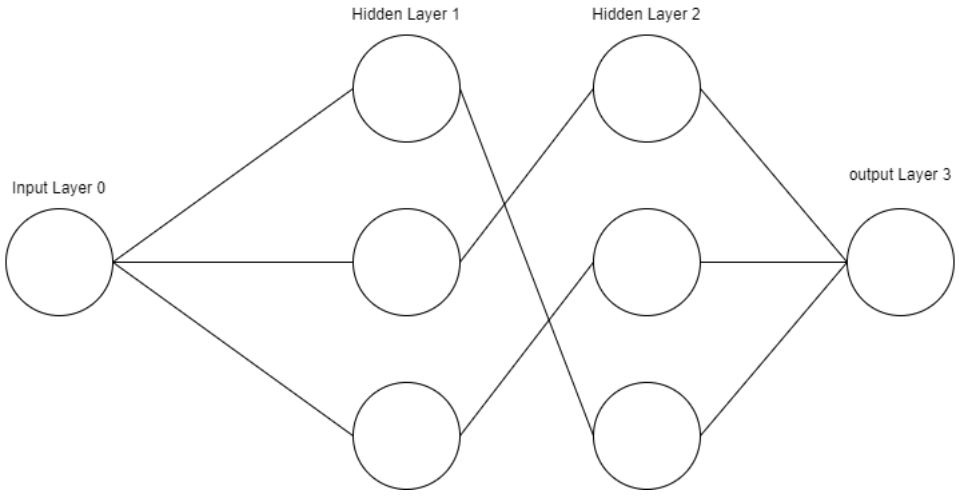

Figure 5.1: Example of a RNN model trained by EXAMM with the layers labeled

to have access to the node object. The node object has the layers, innovation number, and a list of the outputs for that particular node. We create a tuple representing the layer and innovation number from the node object. Here we check if the key exists in the dictionary. If the key exists, we check if the output for the node is greater than the threshold and the value has not been set to true. The output is found in the list based on the time series index in the for loop from the forward_pass() method. We then calculate the Neuron Coverage using the formula mentioned earlier. The coverage node amount was found based on how many keys had the TRUE value. The total node count was found by getting the size of the dictionary of nodes.

```
map<std::tuple<double,int32_t>,bool> init_model_dict(vector<RNN_Node_Interface*> &nodes){
    map<std::tuple<double,int32_t>,bool>model_dict;
    double out_depth = nodes[nodes.size()-1]->get_depth();
    for(uint32_t i = 0; i < nodes.size(); i++){
        if(nodes.at(i)->get_depth() != 0 &&nodes.at(i)->get_depth() != out_depth){
            std::tuple<double, int32_t>key(nodes.at(i)->get_depth(),nodes.at(i)->get_innovation_number());
            model_dict[key] = false;
        }
    }

    return model_dict;

}
```

Figure 5.2: Code for Creating the Dictionary of Nodes

```
void RNN::update_node_coverage(int32_t time, map<std::tuple<double,int32_t>,bool>model_dict, const double &threshhold){
    for(int32_t i = 0; i < nodes.size() ; i++){
        RNN_Node_Interface* node = nodes.at(i);
        std::tuple<double, int32_t>key(nodes.at(i)->get_depth(),nodes.at(i)->get_innovation_number());
        if(model_dict.count(key) == 1){
            if( abs(node->output_values.at(time)) > threshhold && !model_dict[key]){
                model_dict[key] = true;
            }
        }
    }
    int covered_neurons = 0;
    for(auto&& p: model_dict){
        if(p.second){
            ++covered_neurons;
        }
    }
    int total_nodes = model_dict.size();
    float neuron_coverage = float(covered_neurons)/float(total_nodes);//number of coverage nodes over total nodes
    print_node_coverage(total_nodes, covered_neurons, neuron_coverage);
}
```

Figure 5.3: Code for determining coverage and calculating Neuron Coverage

5.3 Data Set and Experiment Design

The Data Set we will be using is from a coal-fired power plant which was used in the EXAMM paper [87]. This data set will be used to train and test the unoptimized and optimized RNN model. We will be generating 10 unoptimized RNN models and 10 optimized RNN models for the experiment. For the threshold to determine the Neuron Coverage we will be using 0.25.

5.4 Results

Here we will discuss the findings based on testing the unoptimized models and optimized models with Neuron Coverage and the learning process of applying Neuron Coverage to the EXAMM framework.

Unoptimized(Avg %)	Optimized(Avg %)
44.9894	63.4814
44.9220	52.9300
54.3738	28.6258
49.9823	49.9385
49.9743	45.8867
20.0483	45.0803
15.0178	62.4572
55.0724	39.5560
49.9247	39.5560
44.9067	33.3795

Figure 5.4: Unoptimized and Optimized Neuron Coverage results

5.4.1 RQ1: Does the optimized RNNs perform better then the unoptimized RNN?

Coming into this experiment there was a hypothesis set that optimized RNNs will have better Neuron Coverage than the unoptimized RNNs. The results shown in 5.4. Looking at the Neuron Coverage for Unoptimized RNNs, There were some noticeable outliers on the low ends like 20.05% and 15.02% Neuron Coverage. This means that those particular models weren't very efficient with their nodes, meaning the output values of the nodes were on the lower end. This typically means that the output values did not affect the neighboring nodes as much compared to higher output values. With the threshold being 0.25 which is low, the output values should be passing the activation test. So it was interesting to see that there were low percentages for the Neural Networks. Some Neuron Coverage measured on the higher end as well: 55.07% and 54.37%. This shows that these particular models were the best performing models. These RNNs have more nodes that output values high than 0.25. In general, the average Neuron Coverage for the Unoptimized RNNs was 42.92% with the majority of the Neuron Coverage percentages hovering in the high 40s. For optimized RNNs, it was more spread out compared to the unoptimized Neuron Coverages, but there were some outliers. On the lower end, there was 28.63%, which is still higher than the unoptimized low-end outliers. So we can see that there is a sign that optimized RNNs are performing better than the unoptimized RNNs. For the upper outliers, we have 63.48% and 62.46%, which are similar to the lower ones and higher than the unoptimized RNN's higher-end outliers. In general, the average Neuron Coverage for the

optimized RNNs was 46.09% with the Neuron Coverages being more spread out.

With the average Neuron Coverage for the unoptimized RNNs being 42.92% and the average Neuron Coverage for the optimized RNNs being 46.09%. We can see that the optimized RNNs had higher Neuron Coverage for the threshold of 0.25 than the unoptimized RNNs. This shows that optimized was more efficient as the nodes output higher values affect the neighboring nodes. Even though optimized performed better, there wasn't a huge difference with the gap being 3% between the averages.

5.4.2 RQ2: Is coverage-based testing appropriated for Δ-RNN, GRU, MGU, and UGRNN recurrent neural network architectures? Do we need to design new coverage metrics for the non-LSTM recurrent neural networks?

For our experiment, the unoptimized RNN was generated using LSTM, but the optimized RNN was generated using all architecture types: Δ-RNN, GRU, LSTM, MGU, and UGRNN. Neuron Coverage was able to perform on both models successfully. This shows that Neuron Coverage is a viable option for delta-RNN, GRU, MGU, and UGRNN along with LSTM. This result is expected to some extent because of how generalized Neuron Coverage is. Neuron Coverage just looks at the output value of a node and does not dive deeper into the node. This means there is no need for consideration of the specific working of the architectures.

From here there is a possibility to create testing metrics to cater towards Δ-

RNN, GRU, LSTM, MGU, and UGRNN. During the implementation, there was no scenario where we were required to access anything related to the architecture types. This can be seen as a good and bad thing. If you were hoping to test the specific architecture Neuron Coverage will not give you useful information. This means that if you are interested in testing particular architectures you will need to explore other testing metrics and/or create your own. For this RNNTest [103] would be a good example/reference for this as their test metrics were explicitly made for LSTM. This shows that it is possible to create testing metrics for the individual architectures, but a question of if it's worth time as LSTM is the most popular of all architectures for RNNs.

Threats to Validity

This section goes over factors that may impact the applicability of the observations to the real world. It is split into 3 sections: construct, internal, and external validity.

6.1 Internal Validity

Internal validity pertains to the uncontrolled factors that interfere with the study results. The main threat here is the threshold was a big variable on whether neuron coverage was successful. If the threshold was 0 neuron coverage would 100%, if the threshold was 1, neuron coverage would be almost 0%. Finding a good threshold difficult, but a way to deal with this is to present a range of thresholds. Another threat is the unoptimized and optimized RNN Models are never a perfect match, this means a comparison on performance should be looked at with skepticism. The major difference was the architecture use where unoptimized used LSTM and optimized randomly used -RNN, GRU, LSTM, MGU, and UGRNN to train the model. This can be dealt with by specifying the architectures that the optimized Model is trained with, but this would warrant for a new research area as you would be exploring the specific Architectures.

6.2 External Validity

The main external threat to validity with this study was not all papers had a available code base. This means that papers no code bases were not

considered when a test metric was selected. This can be dealt with by emailing specific authors or attempting to implement the test metric based on the paper. Another threat is that the papers found were manually collected, here using crawlers to search for related papers using keywords could potentially find more papers, but the current papers were the main ones. Another threat was due to the limited time at the end, the sample size for RNNs was not big enough to get a more thorough experiment. Ideally 20 to 30 models for each optimized and unoptimized would give us a more accurate results and be able to tell the significant difference.

Conclusion & Future Work

The objectives of this work were to create and analyze the papers that exist for the research of testing Neural Networks and a case study with the Neuron Coverage test metric. To do this we documented different features of the paper such as defining test metrics, looking at the targeted neural network, programming language used for implementation, availablitity of code base, etc. We also implemented neuron coverage in the EXAMM framework and used the EXAMM framework and ANTS to train and test unoptimized and optimized neural networks. Here we assume the optimized RNNs will have a better performance than the unoptimized RNNs. For the literature review results we determined that Testing Neural Network papers are mainly focused on DNNs with a few on RNNs. Most testing metrics were focused on coverage testing where they were mixed between generalize test metrics or catered to specific Neural Networks. The case study results showed that the optimized RNNs had a higher Neuron Coverage than the unoptimized RNNs for the threshold of .25 which confirms our assumption. We also learned that Neuron Coverage is a good testing metric for generalize test for all types of RNNs.

Our paper is the start of exploring the existing testing metrics for Neural Networks. For Neuron Coverage, we need to explore the testing metric more to see if the metric can find erroneous behavior based on the Neuron Coverage results. This includes seeing improvements to the models to improve Neuron Coverage. We can also explore the results with an array of threshold between 0 and 1 to see how Neuron Coverage performs. For future EXAMM work, looking at other testing metrics and adapting them will be important. We are

still looking to see if all the test metrics are adaptable to EXAMM as we want to if they will give us more useful information to test RNNs generated from EXAMM. This includes considering new testing metrics for the architecture outside of LSTM.

Bibliography

[1] Tadashi Onishi, Toshiyuki Motoyoshi, Yuki Suga, Hiroki Mori, and Tsuya Ogata. End-to-end learning method for self-driving cars with trajectory recovery using a path-following function. In *2019 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8, 2019.

[2] Pranav Rajpurkar, Jeremy Irvin, Kaylie Zhu, Brandon Yang, Hershel Mehta, Tony Duan, Daisy Ding, Aarti Bagul, Curtis Langlotz, Katie Shpanskaya, Matthew Lungren, and Andrew Ng. Chexnet: Radiologist-level pneumonia detection on chest x-rays with deep learning. 11 2017.

[3] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. *Commun. ACM*, 60(6):84–90, may 2017.

[4] Geoffrey Hinton, Li Deng, Dong Yu, George E. Dahl, Abdel-rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, Tara N. Sainath, and Brian Kingsbury. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal Processing Magazine*, 29(6):82–97, 2012.

[5] Christian D Newman, Michael J Decker, Reem Alsuhaibani, Anthony Peruma, Mohamed Mkaouer, Satyajit Mohapatra, Tejal Vishoi, Marcos Zampieri, Timothy Sheldon, and Emily Hill. An ensemble approach for annotating source code identifiers with part-of-speech tags. *IEEE Transactions on Software Engineering*, 2021.

[6] Yaroslav Golubev, Zarina Kurbatova, Eman Abdullah AlOmar, Timofey Bryksin, and Mohamed Wiem Mkaouer. One thousand and one stories: a large-scale survey of software refactoring. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1303–1313, 2021.

[7] Eman Abdullah AlOmar, Anthony Peruma, Mohamed Wiem Mkaouer, Christian D Newman, and Ali Ouni. Behind the scenes: On the relationship between developer experience and refactoring. *Journal of Software: Evolution and Process*, page e2395, 2021.

[8] Eman Abdullah AlOmar, Mohamed Wiem Mkaouer, Christian Newman, and Ali Ouni. On preserving the behavior in software refactoring: A systematic mapping study. *Information and Software Technology*, 140:106675, 2021.

[9] Eman Abdullah AlOmar, Ben Christians, Mihal Busho, Ahmed Hamad AlKhalid, Ali Ouni, Christian Newman, and Mohamed Wiem Mkaouer. Satdbailiff-mining and tracking self-admitted technical debt. *Science of Computer Programming*, 213:102693, 2022.

[10] Anthony Peruma, Steven Simmons, Eman Abdullah AlOmar, Christian D Newman, Mohamed Wiem Mkaouer, and Ali Ouni. How do i refactor this? an empirical study on refactoring trends and topics in stack overflow. *Empirical Software Engineering*, 27(1):1–43, 2022.

[11] Eman Abdullah AlOmar, Jiaqian Liu, Kenneth Addo, Mohamed Wiem Mkaouer, Christian Newman, Ali Ouni, and Zhe Yu. On the documentation of refactoring types. *Automated Software Engineering*, 29(1):1–40, 2022.

[12] Eman Abdullah Alomar, Tianjia Wang, Vaibhavi Raut, Mohamed Wiem Mkaouer, Christian Newman, and Ali Ouni. Refactoring for reuse: an empirical study. *Innovations in Systems and Software Engineering*, pages 1–31, 2022.

[13] Anthony Peruma, Emily Hu, Jiajun Chen, Eman Abdullah AlOmar, Mohamed Wiem Mkaouer, and Christian D Newman. Using grammar patterns to interpret test method name evolution. In *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*, pages 335–346. IEEE, 2021.

[14] Eman Abdullah AlOmar, Mohamed Wiem Mkaouer, and Ali Ouni. Mining and managing big data refactoring for design improvement: Are we there yet? *Knowledge Management in the Development of Data-Intensive Systems*, pages 127–140, 2021.

[15] Wajdi Aljedaani, Anthony Peruma, Ahmed Aljohani, Mazen Alotaibi, Mohamed Wiem Mkaouer, Ali Ouni, Christian D Newman, Abdullatif Ghallab, and Stephanie Ludi. Test smell detection tools: A systematic mapping study. *Evaluation and Assessment in Software Engineering*, pages 170–180, 2021.

[16] Hussein Alrubaye, Mohamed Wiem Mkaouer, and Ali Ouni. On the use of information retrieval to automate the detection of third-party java library migration at the method level. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*, pages 347–357. IEEE, 2019.

[17] Hussein Alrubaye, Mohamed Wiem Mkaouer, Igor Khokhlov, Leon Reznik, Ali Ouni, and Jason Mcgoff. Learning to recommend third-party library migration opportunities at the api level. *Applied Soft Computing*, 90:106140, 2020.

[18] Hussein Alrubaye, Stephanie Ludi, and Mohamed Wiem Mkaouer. Comparison of block-based and hybrid-based environments in transferring programming skills to text-based environments. *arXiv preprint arXiv:1906.03060*, 2019.

[19] Anthony Peruma, Mohamed Wiem Mkaouer, Michael John Decker, and Christian Donald Newman. Contextualizing rename decisions using refactorings and commit messages. In *2019 19th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 74–85. IEEE, 2019.

[20] Christian D Newman, Mohamed Wiem Mkaouer, Michael L Collard, and Jonathan I Maletic. A study on developer perception of transformation languages for refactoring. In *Proceedings of the 2nd International Workshop on Refactoring*, pages 34–41, 2018.

[21] Hussein Alrubaye and Mohamed Wiem Mkaouer. Automating the detection of third-party java library migration at the function level. In *CASCON*, pages 60–71, 2018.

[22] Hussein Alrubaye, Mohamed Wiem Mkaouer, and Anthony Peruma. Variability in library evolution: An exploratory study on open-source java libraries. In *Software Engineering for Variability Intensive Systems*, pages 295–320. Auerbach Publications, 2019.

[23] Montassar Ben Messaoud, Ilyes Jenhani, Nermine Ben Jemaa, and Mohamed Wiem Mkaouer. A multi-label active learning approach for mobile app user review classification. In *International Conference on Knowledge Science, Engineering and Management*, pages 805–816. Springer, 2019.

[24] Hussein Alrubaye, Mohamed Wiem Mkaouer, and Ali Ouni. Migrationminer: An automated detection tool of third-party java library migration at the method level. In *2019 IEEE international conference on software maintenance and evolution (ICSME)*, pages 414–417. IEEE, 2019.

[25] Deema Alshoaibi, Kevin Hannigan, Hiten Gupta, and Mohamed Wiem Mkaouer. Price: Detection of performance regression introducing code

changes using static and dynamic metrics. In *International Symposium on Search Based Software Engineering*, pages 75–88. Springer, 2019.

[26] Eman Abdullah AlOmar, Mohamed Wiem Mkaouer, Ali Ouni, and Marouane Kessentini. On the impact of refactoring on the relationship between quality attributes and design metrics. In *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–11. IEEE, 2019.

[27] Licelot Marmolejos, Eman Abdullah AlOmar, Mohamed Wiem Mkaouer, Christian Newman, and Ali Ouni. On the use of textual feature extraction techniques to support the automated detection of refactoring documentation. *Innovations in Systems and Software Engineering*, pages 1–17, 2021.

[28] Eman AlOmar, Mohamed Wiem Mkaouer, and Ali Ouni. Can refactoring be self-affirmed? an exploratory study on how developers document their refactoring activities in commit messages. In *2019 IEEE/ACM 3rd International Workshop on Refactoring (IWoR)*, pages 51–58. IEEE, 2019.

[29] Alex Bogart, Eman Abdullah AlOmar, Mohamed Wiem Mkaouer, and Ali Ouni. Increasing the trust in refactoring through visualization. In *2020 IEEE/ACM 4th International Workshop on Refactoring (IWoR)*, 2020.

[30] Eman Abdullah AlOmar, Anthony Peruma, Mohamed Wiem Mkaouer, Christian Newman, Ali Ouni, and Marouane Kessentini. How we refactor and how we document it? on the use of supervised machine learning algorithms to classify refactoring documentation. *Expert Systems with Applications*, 167:114176, 2021.

[31] Eman Abdullah AlOmar, Hussein AlRubaye, Mohamed Wiem Mkaouer, Ali Ouni, and Marouane Kessentini. Refactoring practices in the context of modern code review: An industrial case study at xerox. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 348–357. IEEE, 2021.

[32] Eman Abdullah AlOmar, Anthony Peruma, Christian D Newman, Mohamed Wiem Mkaouer, and Ali Ouni. On the relationship between

developer experience and refactoring: An exploratory study and preliminary results. In *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*, pages 342–349, 2020.

[33] Eman Abdullah AlOmar, Philip T Rodriguez, Jordan Bowman, Tianjia Wang, Benjamin Adepoju, Kevin Lopez, Christian Newman, Ali Ouni, and Mohamed Wiem Mkaouer. How do developers refactor code to improve code reusability? In *International Conference on Software and Software Reuse*, pages 261–276. Springer, 2020.

[34] Eman Abdullah AlOmar, Tianjia Wang, Raut Vaibhavi, Mohamed Wiem Mkaouer, Christian Newman, and Ali Ouni. Refactoring for reuse: An empirical study. *Innovations in Systems and Software Engineering*, pages 1–31, 2021.

[35] Anthony Peruma, Khalid Almalki, Christian D. Newman, Mohamed Wiem Mkaouer, Ali Ouni, and Fabio Palomba. Tsdetect: An open source test smells detection tool. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2020, New York, NY, USA, 2020. Association for Computing Machinery.

[36] Anthony Peruma, Christian D. Newman, Mohamed Wiem Mkaouer, Ali Ouni, and Fabio Palomba. An exploratory study on the refactoring of unit test files in android applications. In *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*, ICSEW'20, page 350–357, New York, NY, USA, 2020. Association for Computing Machinery.

[37] Anthony Peruma, Khalid Almalki, Christian D. Newman, Mohamed Wiem Mkaouer, Ali Ouni, and Fabio Palomba. On the distribution of test smells in open source android applications: An exploratory study. In *Proceedings of the 29th Annual International Conference on Computer Science and Software Engineering*, CASCON '19, page 193–202, USA, 2019. IBM Corp.

[38] Sirine Gharbi, Mohamed Wiem Mkaouer, Ilyes Jenhani, and Montassar Ben Messaoud. On the classification of software change messages using multi-label active learning. In *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*, pages 1760–1767, 2019.

[39] Wiem Mkaouer, Marouane Kessentini, Adnan Shaout, Patrice Koligheu, Slim Bechikh, Kalyanmoy Deb, and Ali Ouni. Many-objective software

remodularization using nsga-iii. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 24(3):1–45, 2015.

[40] Mohamed Wiem Mkaouer, Marouane Kessentini, Slim Bechikh, Kalyanmoy Deb, and Mel Ó Cinnéide. Recommendation system for software refactoring using innovization and interactive dynamic optimization. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pages 331–336, 2014.

[41] Mohamed Wiem Mkaouer, Marouane Kessentini, Slim Bechikh, Kalyanmoy Deb, and Mel Ó Cinnéide. High dimensional search-based software engineering: finding tradeoffs among 15 objectives for automating software refactoring using nsga-iii. In *Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation*, pages 1263–1270, 2014.

[42] Mohamed Wiem Mkaouer, Marouane Kessentini, Slim Bechikh, Mel Ó Cinnéide, and Kalyanmoy Deb. On the use of many quality attributes for software refactoring: a many-objective search-based software engineering approach. *Empirical Software Engineering*, 21(6):2503–2545, 2016.

[43] Mohamed Wiem Mkaouer, Marouane Kessentini, Mel Ó Cinnéide, Shinpei Hayashi, and Kalyanmoy Deb. A robust multi-objective approach to balance severity and importance of refactoring opportunities. *Empirical Software Engineering*, 22(2):894–927, 2017.

[44] Rafi Almhana, Wiem Mkaouer, Marouane Kessentini, and Ali Ouni. Recommending relevant classes for bug reports using multi-objective search. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 286–295. IEEE, 2016.

[45] Anthony Peruma, Khalid Almalki, Christian D Newman, Mohamed Wiem Mkaouer, Ali Ouni, and Fabio Palomba. On the distribution of test smells in open source android applications: An exploratory study. In *Proceedings of the 29th Annual International Conference on Computer Science and Software Engineering*, pages 193–202, 2019.

[46] Islem Saidani, Ali Ouni, and Mohamed Wiem Mkaouer. Improving the prediction of continuous integration build failures using deep learning. *Automated Software Engineering*, 29(1):1–61, 2022.

[47] Wajdi Aljedaani, Mona Aljedaani, Eman Abdullah AlOmar, Mohamed Wiem Mkaouer, Stephanie Ludi, and Yousef Bani Khalaf. I cannot see you—the perspectives of deaf students to online learning during covid-19 pandemic: Saudi arabia case study. *Education Sciences*, 11(11):712, 2021.

[48] Islem Saidani, Ali Ouni, and Wiem Mkaouer. Detecting skipped commits in continuous integration using multi-objective evolutionary search. *IEEE Transactions on Software Engineering*, 2021.

[49] Marwa Daaji, Ali Ouni, Mohamed Mohsen Gammoudi, Salah Bouktif, and Mohamed Wiem Mkaouer. Multi-criteria web services selection: Balancing the quality of design and quality of service. *ACM Transactions on Internet Technology (TOIT)*, 22(1):1–31, 2021.

[50] Nuri Almarimi, Ali Ouni, Moataz Chouchen, and Mohamed Wiem Mkaouer. csdetector: an open source tool for community smells detection. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1560–1564, 2021.

[51] Islem Saidani, Ali Ouni, Moataz Chouchen, and Mohamed Wiem Mkaouer. Bf-detector: an automated tool for ci build failure detection. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1530–1534, 2021.

[52] Oumayma Hamdi, Ali Ouni, Mel Ó Cinnéide, and Mohamed Wiem Mkaouer. A longitudinal study of the impact of refactoring in android applications. *Information and Software Technology*, 140:106699, 2021.

[53] Oumayma Hamdi, Ali Ouni, Eman Abdullah AlOmar, Mel Ó Cinnéide, and Mohamed Wiem Mkaouer. An empirical study on the impact of refactoring on quality metrics in android applications. In *2021 IEEE/ACM 8th International Conference on Mobile Software Engineering and Systems (MobileSoft)*, pages 28–39. IEEE, 2021.

[54] Islem Saidani, Ali Ouni, Mohamed Wiem Mkaouer, and Fabio Palomba. On the impact of continuous integration on refactoring practice: An exploratory study on travistorrent. *Information and Software Technology*, 138:106618, 2021.

[55] Lobna Ghadhab, Ilyes Jenhani, Mohamed Wiem Mkaouer, and Montassar Ben Messaoud. Augmenting commit classification by using fine-grained source code changes and a pre-trained deep neural language model. *Information and Software Technology*, 135:106566, 2021.

[56] Fan Fang, John Wu, Yanyan Li, Xin Ye, Wajdi Aljedaani, and Mohamed Wiem Mkaouer. On the classification of bug reports to improve bug localization. *Soft Computing*, 25(11):7307–7323, 2021.

[57] Makram Soui, Mabrouka Chouchane, Narjes Bessghaier, Mohamed Wiem Mkaouer, and Marouane Kessentini. On the impact of aesthetic defects on the maintainability of mobile graphical user interfaces: An empirical study. *Information Systems Frontiers*, pages 1–18, 2021.

[58] Eman Abdullah AlOmar, Hussein AlRubaye, Mohamed Wiem Mkaouer, Ali Ouni, and Marouane Kessentini. Refactoring practices in the context of modern code review: An industrial case study at xerox. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 348–357. IEEE, 2021.

[59] Moataz Chouchen, Ali Ouni, Raula Gaikovina Kula, Dong Wang, Patanamon Thongtanunam, Mohamed Wiem Mkaouer, and Kenichi Matsumoto. Anti-patterns in modern code review: Symptoms and prevalence. In *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 531–535. IEEE, 2021.

[60] Xin Ye, Yongjie Zheng, Wajdi Aljedaani, and Mohamed Wiem Mkaouer. Recommending pull request reviewers based on code changes. *Soft Computing*, 25(7):5619–5632, 2021.

[61] Hussein Alrubaye, Deema Alshoaibi, Eman Alomar, Mohamed Wiem Mkaouer, and Ali Ouni. How does library migration impact software quality and comprehension? an empirical study. In *International Conference on Software and Software Reuse*, pages 245–260. Springer, 2020.

[62] Moataz Chouchen, Ali Ouni, Mohamed Wiem Mkaouer, Raula Gaikovina Kula, and Katsuro Inoue. Whoreview: A multi-objective search-based approach for code reviewers recommendation in modern code review. *Applied Soft Computing*, 100:106908, 2021.

[63] Moataz Chouchen, Ali Ouni, and Mohamed Wiem Mkaouer. Androlib: Third-party software library recommendation for android applications. In *International Conference on Software and Software Reuse*, pages 208–225. Springer, 2020.

[64] Nuri Almarimi, Ali Ouni, Moataz Chouchen, Islem Saidani, and Mohamed Wiem Mkaouer. On the detection of community smells using genetic programming-based ensemble classifier chain. In *Proceedings of the 15th International Conference on Global Software Engineering*, pages 43–54, 2020.

[65] Eman Abdullah AlOmar, Mohamed Wiem Mkaouer, and Ali Ouni. Toward the automatic classification of self-affirmed refactoring. *Journal of Systems and Software*, 171:110821, 2021.

[66] Islem Saidani, Ali Ouni, and Mohamed Wiem Mkaouer. Web service api anti-patterns detection as a multi-label learning problem. In *International Conference on Web Services*, pages 114–132. Springer, 2020.

[67] Bader Alkhazi, Andrew DiStasi, Wajdi Aljedaani, Hussein Alrubaye, Xin Ye, and Mohamed Wiem Mkaouer. Learning to rank developers for bug report assignment. *Applied Soft Computing*, 95:106667, 2020.

[68] Motaz Chouchen, Ali Ouni, Mohamed Wiem Mkaouer, Raula Gaikovina Kula, and Katsuro Inoue. Recommending peer reviewers in modern code review: a multi-objective search-based approach. In *Proceedings of the 2020 Genetic and Evolutionary Computation Conference Companion*, pages 307–308, 2020.

[69] Islem Saidani, Ali Ouni, Moataz Chouchen, and Mohamed Wiem Mkaouer. Predicting continuous integration build failures using evolutionary search. *Information and Software Technology*, 128:106392, 2020.

[70] Anthony Peruma, Khalid Almalki, Christian D Newman, Mohamed Wiem Mkaouer, Ali Ouni, and Fabio Palomba. tsdetect: an open source test smells detection tool. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1650–1654, 2020.

[71] Islem Saidani, Ali Ouni, Moataz Chouchen, and Mohamed Wiem Mkaouer. On the prediction of continuous integration build failures using

search-based software engineering. In *Proceedings of the 2020 Genetic and Evolutionary Computation Conference Companion*, pages 313–314, 2020.

[72] Nuri Almarimi, Ali Ouni, and Mohamed Wiem Mkaouer. Learning to detect community smells in open source software projects. *Knowledge-Based Systems*, 204:106201, 2020.

[73] Islem Saidani, Ali Ouni, Mohamed Wiem Mkaouer, and Aymen Saied. Towards automated microservices extraction using muti-objective evolutionary search. In *International Conference on Service-Oriented Computing*, pages 58–63. Springer, Cham, 2019.

[74] Nuri Almarimi, Ali Ouni, Salah Bouktif, Mohamed Wiem Mkaouer, Raula Gaikovina Kula, and Mohamed Aymen Saied. Web service api recommendation for automated mashup creation using multi-objective evolutionary search. *Applied Soft Computing*, 85:105830, 2019.

[75] Makram Soui, Mabrouka Chouchane, Mohamed Wiem Mkaouer, Marouane Kessentini, and Khaled Ghedira. Assessing the quality of mobile graphical user interfaces using multi-objective optimization. *Soft Computing*, 24(10):7685–7714, 2020.

[76] Nasir Safdari, Hussein Alrubaye, Wajdi Aljedaani, Bladimir Baez Baez, Andrew DiStasi, and Mohamed Wiem Mkaouer. Learning to rank faulty source files for dependent bug reports. In *Big Data: Learning, Analytics, and Applications*, volume 10989, page 109890B. International Society for Optics and Photonics, 2019.

[77] Vahid Alizadeh, Marouane Kessentini, Mohamed Wiem Mkaouer, Mel Ocinneide, Ali Ouni, and Yuanfang Cai. An interactive and dynamic search-based approach to software refactoring recommendations. *IEEE Transactions on Software Engineering*, 46(9):932–961, 2018.

[78] Anthony Peruma, Mohamed Wiem Mkaouer, Michael J Decker, and Christian D Newman. An empirical investigation of how and why developers rename identifiers. In *Proceedings of the 2nd International Workshop on Refactoring*, pages 26–33, 2018.

[79] Makram Soui, Mabrouka Chouchane, Ines Gasmi, and Mohamed Wiem Mkaouer. Plain: Plugin for predicting the usability of mobile user interface. In *VISIGRAPP (1: GRAPP)*, pages 127–136, 2017.

[80] Ian Shoenberger, Mohamed Wiem Mkaouer, and Marouane Kessentini. On the use of smelly examples to detect code smells in javascript. In *European Conference on the Applications of Evolutionary Computation*, pages 20–34. Springer, Cham, 2017.

[81] Mohamed Wiem Mkaouer. Interactive code smells detection: An initial investigation. In *International Symposium on Search Based Software Engineering*, pages 281–287. Springer, Cham, 2016.

[82] Mohamed Wiem Mkaouer, Marouane Kessentini, Slim Bechikh, and Mel Ó Cinnéide. A robust multi-objective approach for software refactoring under uncertainty. In *International Symposium on Search Based Software Engineering*, pages 168–183. Springer, Cham, 2014.

[83] Mohamed Wiem Mkaouer and Marouane Kessentini. Model transformation using multiobjective optimization. In *Advances in Computers*, volume 92, pages 161–202. Elsevier, 2014.

[84] Mohamed W Mkaouer, Marouane Kessentini, Slim Bechikh, and Daniel R Tauritz. Preference-based multi-objective software modelling. In *2013 1st International Workshop on Combining Modelling and Search-Based Software Engineering (CMSBSE)*, pages 61–66. IEEE, 2013.

[85] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Comput.*, 9(8):1735–1780, nov 1997.

[86] Junyoung Chung, Çaglar Gülçehre, KyungHyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. *CoRR*, abs/1412.3555, 2014.

[87] Alexander Ororbia, AbdElRahman ElSaid, and Travis Desell. Investigating recurrent neural network memory structures using neuro-evolution. GECCO '19, page 446–455, New York, NY, USA, 2019. Association for Computing Machinery.

[88] Yuchi Tian, Kexin Pei, Suman Jana, and Baishakhi Ray. Deeptest: Automated testing of deep-neural-network-driven autonomous cars. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 303–314, 2018.

[89] AbdElRahman ElSaid, Fatima El Jamiy, James Higgins, Brandon Wild, and Travis Desell. Optimizing long short-term memory recurrent neural

networks using ant colony optimization to predict turbine engine vibration. *Applied Soft Computing*, 73:969–991, 2018.

[90] AbdElRahman ElSaid, Alexander G. Ororbia, and Travis J. Desell. Ant-based neural topology search (ants) for optimizing recurrent networks. In Pedro A. Castillo, Juan Luis Jiménez Laredo, and Francisco Fernández de Vega, editors, *Applications of Evolutionary Computation*, pages 626–641, Cham, 2020. Springer International Publishing.

[91] Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. Deepxplore: Automated whitebox testing of deep learning systems. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 1–18, New York, NY, USA, 2017. Association for Computing Machinery.

[92] Lei Ma, Felix Juefei-Xu, Fuyuan Zhang, Jiyuan Sun, Minhui Xue, Bo Li, Chunyang Chen, Ting Su, Li Li, Yang Liu, Jianjun Zhao, and Yadong Wang. *DeepGauge: Multi-Granularity Testing Criteria for Deep Learning Systems*, page 120–131. Association for Computing Machinery, New York, NY, USA, 2018.

[93] Youcheng Sun, Xiaowei Huang, Daniel Kroening, James Sharp, Matthew Hill, and Rob Ashmore. Testing Deep Neural Networks. *arXiv e-prints*, page arXiv:1803.04792, March 2018.

[94] Xiaofei Xie, Lei Ma, Felix Juefei-Xu, Minhui Xue, Hongxu Chen, Yang Liu, Jianjun Zhao, Bo Li, Jianxiong Yin, and Simon See. *DeepHunter: A Coverage-Guided Fuzz Testing Framework for Deep Neural Networks*, page 146–157. Association for Computing Machinery, New York, NY, USA, 2019.

[95] Seokhyun Lee, Sooyoung Cha, Dain Lee, and Hakjoo Oh. Effective white-box testing of deep neural networks with adaptive neuron-selection strategy. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2020, page 165–176, New York, NY, USA, 2020. Association for Computing Machinery.

[96] Lei Ma, Fuyuan Zhang, Jiyuan Sun, Minhui Xue, Bo Li, Felix Juefei-Xu, Chao Xie, Li Li, Yang Liu, Jianjun Zhao, and Yadong Wang. Deepmutation: Mutation testing of deep learning systems. In *2018 IEEE 29th International Symposium on Software Reliability Engineering (ISSRE)*, pages 100–111, 2018.

[97] Youcheng Sun, Min Wu, Wenjie Ruan, Xiaowei Huang, Marta Kwiatkowska, and Daniel Kroening. *Concolic Testing for Deep Neural Networks*, page 109–119. Association for Computing Machinery, New York, NY, USA, 2018.

[98] Jianmin Guo, Yu Jiang, Yue Zhao, Quan Chen, and Jiaguang Sun. Dlfuzz: Differential fuzzing testing of deep learning systems. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2018, page 739–743, New York, NY, USA, 2018. Association for Computing Machinery.

[99] Xiang Gao, Ripon K. Saha, Mukul R. Prasad, and Abhik Roychoudhury. Fuzz testing based data augmentation to improve robustness of deep neural networks. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, page 1147–1158, New York, NY, USA, 2020. Association for Computing Machinery.

[100] Augustus Odena, Catherine Olsson, David Andersen, and Ian Goodfellow. TensorFuzz: Debugging neural networks with coverage-guided fuzzing. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 4901–4911. PMLR, 09–15 Jun 2019.

[101] Wei Huang, Youcheng Sun, Xingyu Zhao, James Sharp, Wenjie Ruan, Jie Meng, and Xiaowei Huang. Coverage-guided testing for recurrent neural networks. *IEEE Transactions on Reliability*, pages 1–16, 2021.

[102] Xiaoning Du, Xiaofei Xie, Yi Li, Lei Ma, Yang Liu, and Jianjun Zhao. Deepstellar: Model-based quantitative analysis of stateful deep learning systems. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019, page 477–487, New York, NY, USA, 2019. Association for Computing Machinery.

[103] Jianmin Guo, Quan Zhang, Yue Zhao, Heyuan Shi, Yu Jiang, and Jiaguang Sun. Rnn-test: Towards adversarial testing for recurrent neural network systems. *IEEE Transactions on Software Engineering*, pages 1–1, 2021.

[104] Qiang Hu, Lei Ma, Xiaofei Xie, Bing Yu, Yang Liu, and Jianjun Zhao. Deepmutation++: A mutation testing framework for deep learning systems. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering*, ASE '19, page 1158–1161. IEEE Press, 2019.