The Integration of Machine Learning into Automated Test Generation: A Systematic Literature Review

Afonso Fontes, Gregory Gay*

Department of Computer Science and Engineering, Chalmers | University of Gothenburg, Sweden

Abstract

Context: Machine learning (ML) may enable effective automated test generation.

Objectives: We characterize emerging research, examining testing practices, researcher goals, ML techniques applied, evaluation, and challenges.

Methods: We perform a systematic literature review on a sample of 97 publications.

Results: ML generates input for system, GUI, unit, performance, and combinatorial testing or improves the performance of existing generation methods. ML is also used to generate test verdicts, property-based, and expected output oracles. Supervised learning—often based on neural networks—and reinforcement learning—often based on Q-learning—are common, and some publications also employ unsupervised or semi-supervised learning. (Semi-/Un-)Supervised approaches are evaluated using both traditional testing metrics and ML-related metrics (e.g., accuracy), while reinforcement learning is often evaluated using testing metrics tied to the reward function.

Conclusion: Work-to-date shows great promise, but there are open challenges regarding training data, retraining, scalability, evaluation complexity, ML algorithms employed—and how they are applied—benchmarks, and replicability. Our findings can serve as a roadmap and inspiration for researchers in this field.

Keywords: Automated Test Generation, Test Case Generation, Test Input Generation, Test Oracle Generation, Machine Learning

*Corresponding author

Email addresses: afonso.fontes@chalmers.se (Afonso Fontes), greg@greggay.com (Gregory Gay)
1. Introduction

Software testing is invaluable in ensuring the reliability of the software that powers our society [1]. It is also notoriously difficult and expensive, with severe consequences for productivity, the environment, and human life if not conducted properly. New tools and methodologies are needed to control that cost without reducing the quality of the testing process.

Automation has a critical role in controlling costs and focusing developer attention [2]. Consider test generation an effort-intensive task where sequences of program input and oracles that judge the correctness of the resulting execution are crafted for a system-under-test (SUT) [1]. Effective automated test generation could lead to immense effort and cost savings.

Automated test generation is a popular research topic, and outstanding achievements have been made in recent years [2]. Still, there are critical limitations to current approaches. Major among these is that generation frameworks are applied in a general manner—techniques target simple universal heuristics, and those heuristics are applied in a static manner to all systems equally. Parameters of test generation can be tuned by a developer, but this requires advanced knowledge and is still based on the same universal heuristics. Current generation frameworks are largely unable to adapt their approach to a particular SUT, even though such projects offer rich information content in their documentation, metadata, source code, or execution logs [3]. Such static application limits the potential effectiveness of automated test generation.

Advances in the field of machine learning (ML) have shown that automation can match or surpass human performance across many problem domains. ML has advanced the state-of-the-art in virtually every field. Automated test generation is no exception. Recently, researchers have begun to use ML either to directly generate input or oracles [4] or to enhance the effectiveness or efficiency of existing test generation frameworks [5]. ML offers the potential means to adapt test generation to a SUT, and to enable automation to optimize its approach without human intervention.

We are interested in understanding and characterizing emerging research around
the integration of ML into automated test generation. Specifically, we are interested in which testing practices have been addressed by integrating ML into test generation, the goals of the researchers using ML, how ML is integrated into the generation process, which specific ML techniques are applied, how such techniques are trained and validated, and how the whole test generation process is evaluated. We are also interested in identifying the emerging field’s limitations and open research challenges.

To that end, we have performed a systematic literature review. Following a search of relevant databases and a rigorous filtering process, we have examined 97 relevant studies, gathering the data needed to answer our research questions.

We observed that ML supports generation of input and oracles for a variety of testing practices (e.g., system or GUI testing) and oracle types (e.g., verdicts and expected values). During input generation, ML either directly generates input or improves the efficiency or effectiveness of existing generation methods.

The most common types of ML are supervised (59%) and RL (36%). A small number of publications also employ unsupervised (4%) or semi-supervised (2%) learning. Supervised learning is the most common ML for system testing, Combinatorial Interaction Testing, and all forms of oracle generation. Neural networks are the most common supervised techniques, and techniques are evaluated using both traditional testing metrics (e.g., coverage) and ML-related metrics (e.g., accuracy).

RL is the most common ML for GUI, unit, and performance testing. It is effective for practices with scoring functions and when testing requires a sequence of input steps. It is also effective at tuning generation tools. Reinforcement learning techniques are generally based on Q-Learning, and approaches are evaluated using testing metrics (often tied to the reward function). Finally, unsupervised learning is effective for filtering tasks such as discarding similar test cases.

The publications show great promise, but there are significant open challenges. Learning is limited by the required quantity, quality, and contents of training data.

1We focus specifically on the use of ML to enhance test generation, as part of the broader field of AI-for-Software Engineering (AI4SE). There has also been research in automated test generation for ML-based systems (SE4AI). These studies are out of the scope of our review.
Models should be retrained over time. Whether techniques will scale to real-world systems is not clear. Researchers rarely justify the choice of ML technique or compare alternatives. Research is limited by the overuse of simplistic examples, the lack of standard benchmarks, and the unavailability of code and data. Researchers should be encouraged to use common benchmarks and provide replication packages and code. In addition, new benchmarks could be created for ML challenges (e.g., oracle generation).

Our study is the first to thoroughly summarize and characterize this emerging research field. We hope that our findings will serve as a roadmap for both researchers and practitioners interested in the use of ML in test generation and that it will inspire new advances in the field.

2. Background and Related Work

2.1. Software Testing

It is essential to verify that software functions as intended. This verification process usually involves testing—the application of input, and analysis of the resulting output, to identify unexpected behaviors in the system-under-test (SUT) [1].

During testing, a test suite containing one or more test cases is applied to the SUT. A test case consists of a test sequence (or procedure)—a series of interactions with the SUT—with test input applied to some SUT component. Depending on the granularity of testing, the input can range from method calls, to API calls, to actions within a graphical interface. Then, the test case will validate the output against a set of encoded expectations—the test oracle—to determine whether the test passes or fails [1]. An oracle can be a predefined specification (e.g., an assertion), output from a past version, a model, or even manual inspection by humans [1].

An example of a test case, written in the JUnit notation, is shown in Figure 1. The test input is a string passed to the constructor of the TransformCase class, then a call to getText(). An assertion then checks whether the output matches the expected output—an upper-case version of the input.

---

2 This study extends an initial SLR on test oracle generation [6]. Our extended study also includes input generation, updates the sample of publications, and features an extended analysis and discussion.
@Test
public void testPrintMessage() {
    String str = "Test Message";
    TransformCase tCase = new TransformCase(str);
    String upperCaseStr = str.toUpperCase();
    assertEquals(upperCaseStr, tCase.getText());
}

Figure 1: Example of a unit test case written using the JUnit notation for Java.

Testing can be performed at different granularity levels, using tests written in code or applied by humans. The lowest granularity is unit testing, which focuses on isolated code modules (generally classes). Module interactions are tested during integration testing. Then, during system testing, the SUT is tested through one of its defined interfaces—a programmable interface, a command-line interface, a graphical user interface, or another external interface. Human-driven testing, such as exploratory testing, is out of the scope of this study, as it is often not amenable to automation.

2.2. Machine Learning

Machine learning (ML) constructs models from observations of data to make predictions [3]. Instead of being explicitly programmed like in traditional software, ML algorithms “learn” from observations using statistical analyses, facilitating the automation of decision making. ML has enabled many new applications in the past decade. As computational power and data availability increase, such approaches will increase in their capabilities and accuracy.

ML approaches largely fall into four categories—supervised, semi-supervised, unsupervised, and reinforcement learning—as presented in Figure 2. In supervised learning, algorithms infer a model from the training data that makes predictions about newly encountered data. Such algorithms are typically used for classification—prediction of a label from a finite set—or regression—predictions in an unrestricted format, e.g., a continuous value. If a sufficiently large training dataset with a low level of noise is used, an accurate model can often be trained quickly. However, a model is generally static once trained and cannot be improved without re-training.
Unsupervised algorithms do not use previously-labeled data. Instead, approaches identify patterns in data based on the similarities and differences between items. They model the data indirectly, with little-to-no human input. Rather than making predictions, unsupervised techniques aid in understanding data by, e.g., clustering related items, extracting interesting features, or detecting anomalies.

Semi-supervised algorithms start with training data, then employ feedback mechanisms to automatically retrain the model. Adversarial networks refine accuracy by augmenting the training data with new input by putting two supervised algorithms in
competition. One of the algorithms creates new inputs that mimic training data, while the second predicts whether these are part of the training data or impostors. The first refines its ability to create convincing fakes, while the second tries to separate fakes from the originals. Semi-supervised approaches require a longer training time, but can achieve more optimal models, often with a smaller initial training set.

Reinforcement learning (RL) algorithms select actions based on an estimation of their effectiveness towards achieving a measurable goal [5]. RL often does not require training data, instead learning through sequences of interactions with its environment. RL “agents” use feedback on the effect of actions taken to improve their estimation of the actions most likely to maximize achievement of their goal (their “policy”). Feedback is provided by a reward function—a numeric scoring function. The agent can also adapt to a changing environment, as estimations are refined each time an action is taken. Such algorithms are often the basis of automated processes, such as autonomous driving, and are effective in situations where sequences of predictions are required.

Recent research often focuses on “deep learning”. Deep approaches make complex and highly accurate inferences from massive datasets. Many DL approaches are based on complex many-layered neural networks—networks that attempts to mimic how the human brain works [7]. Such neural networks employ a cascade of nonlinear processing layers where one layer’s output serves as the successive layer’s input. Deep learning requires a computationally intense training process and larger datasets than traditional ML, but can learn highly accurate models, extract features and relationships from data automatically, and potentially apply models across applications. “Deep” approaches exist for all four of the ML types discussed above.

2.3. Related Work

Other secondary studies overlap with ours in scope. We briefly discuss these publications below. Our SLR is the first focused specifically on the application of ML to automated test generation, including both input and oracle generation, and no related study overlaps in full with our research questions. We have also examined a larger and more recent sample of publications.

Durelli et al. performed a systematic mapping study on the application of ML to
software testing [3]. Their scope is broad, examining how ML has been applied to any aspect of the testing process. They mapped 48 publications to testing activities, study types, and ML algorithms employed. They observe that ML has been used to generate input and oracles. They note that supervised algorithms are used more often than other ML types, and that Artificial Neural Networks are the most used algorithm. Jha and Popli also conducted a short review of literature applying ML to testing activities [8], and note that ML has been used for both input and oracle generation.

Ioannides and Eder conducted a survey on the use of AI techniques to generate test cases targeting code coverage—known as “white box” test generation [9]. Their survey focuses on optimization techniques, such as genetic algorithms, but they note that ML has been used to generate test input.

Barr et al. performed a survey on test oracles [11]. They divide test oracles into four types, including those specified by humans, those derived automatically, those that reflect implicit properties of programs, and those that rely on a human-in-the-loop. Approaches based on ML fall into the “derived” category, as they learn automatically from project artifacts to replace or augment human-written oracles. They discuss early approaches to using ML to derive oracles.

Balera et al. conducted a systematic mapping study on hyper-heuristics in search-based test generation [10]. Search-based test generation applies optimization algorithms to generate test input. A hyper-heuristic is a secondary optimization performed to tune the primary search strategy, e.g., a hyper-heuristic could adapt test generation to the current SUT. A hyper-heuristic can apply ML, especially RL, but can also be guided by other algorithms. We also observe the use of ML-based hyper-heuristics.

3. Methodology

Our aim is to understand how researchers have integrated ML into automated test generation, including generation of input and oracles. We have investigated publications related to this topic and seek to understand their methodology, results, and insights. To gain understanding, we performed a Systematic Literature Review (SLR).

We are interested in assessing the effect of integrating ML into the test generation...
### Table 1: List of research questions, along with motivation for answering the question.

| ID | Research Question                                                                 | Objective                                                                 |
|----|-----------------------------------------------------------------------------------|---------------------------------------------------------------------------|
| RQ1| Which testing practices have been supported by integrating ML into the generation process? | Highlights testing scenarios and systems types targeted for ML-enhanced test generation. |
| RQ2| What is the goal of using machine learning as part of automated test generation?  | To understand the reasons for applying ML techniques to enable or enhance test generation. |
| RQ3| How was machine learning integrated into the process of automated test generation? | Identifies the type of ML applied, how it was integrated, and how it was trained and validated. |
| RQ4| Which machine learning techniques were used to perform or enhance automated test generation? | Identify specific ML techniques used in the process, including type, learning method, and selection mechanisms. |
| RQ5| How is the test generation process evaluated?                                     | Describe the evaluation of the ML-enhanced test generation process, highlighting common metrics and artifacts (programs or datasets) used. |
| RQ6| What are the limitations and open challenges in integrating ML into test generation? | Highlights the limitations of enhancing test generation with ML and future research directions. |

To answer these questions, we have performed the following tasks:

1. Formed a list of publications by querying publication databases (Section 3.1).
2. Filtered this list for relevance (Section 3.2).
3. Extracted data from each study, guided by properties of interest (Section 3.3).
4. Identified trends in the extracted data in order to answer each research question (described along with results in Section 4).

#### 3.1. Initial Study Selection

To locate publications for consideration, a search was conducted using four databases: IEEE Xplore, ACM Digital Library, Science Direct, and Scopus. To narrow the results, we created a search string by combining terms of interest on test generation and machine learning. The search string used was:
These keywords are not guaranteed to capture all publications on the ML in test generation. However, they are intended to attain a relevant sample. Specifically, we combine terms related to test generation and terms related to machine learning, including common technologies. Our focus is not on any particular form of test generation. To obtain a representative sample, we have selected ML terms that we expect will capture a wide range of publications. These terms may omit some in-scope ML techniques, but attain a relevant sample while constraining the amount of manual inspection.

We limited our search to peer-reviewed publications in English. Our set of articles was gathered in December 2021, containing an initial total of 2,805 articles. This is shown as the first step in Figure 3.

To evaluate the search string’s effectiveness, we conducted a verification process. First, we randomly sampled ten entries from the final publication list. Then we looked in each article for ten citations that were in scope, resulting in 100 citations. We checked whether the search string also retrieved these citations, and all 100 were retrieved. Although this is a small sample, it indicates the robustness of the string.

3.2. Selection Filtering

We next applied a series of filtering steps to obtain a focused sample. Figure 3 presents the filtering process and the number of entries after applying each filter.

To ensure that publications are relevant, we used keywords to filter the list. We first searched the title and abstract of each study for the keyword “test” (including, e.g., “testing”). We then searched the remaining publications for either “learning” or “neural”—representing application of ML. We merged the filtered lists, and removed all duplicate entries. We then removed all secondary studies. This left 944 publications.

We examined the remaining publications manually, removing all publications not in scope following an inspection of the title and abstract. We removed any publications not related to test generation or that do not apply ML during the generation process (e.g., ML is used in a separate activity such as test reduction, a non-ML approach is
used, or ML is being tested). This determination was made by first reading the abstract and introduction. Then, if the publication seemed in scope, we proceeded to read the entire study. Both authors independently inspected publications during this step to prevent the accidental removal of relevant publications. In cases of disagreement, the authors discussed the study.

This process resulted in a final sample of 97 articles. Due to the size and scope of the sample, we have not performed snowballing as part of this study. We believe that
The specific type of testing scenarios or application domain focused on by the approach. It helps to categorize the publications, enabling comparison between contributions.

A short description of the approach proposed or research performed.

Highlights the differences between expectations and conclusions of the proposed approach.

Covers how ML techniques have been integrated into the test generation process. It is essential to understand what aspects of generation are handled or supported by ML.

Name, type, and description of the ML technique used in the study.

The reasons stated by the authors for choosing this ML technique.

How the approach was trained, including the specific data sets or artifacts used to perform this training. This property helps us understand how each contribution could be replicated or extended.

External tools or libraries used to implement the ML technique.

This attribute covers the objective of the ML technique (e.g., reward function or validation metric), and how it is validated, including data, artifacts, and metrics used (if any).

Covers how the ML-enhanced oracle generation process, as a whole, is evaluated (i.e., how successful are the generated input at triggering faults or meeting some other testing goal?). Allows understanding of the effects of ML on improving the testing process.

Notes on the threats to validity that could impact each study.

This property is used to understand the general strengths and limitations of enhancing a generation process with ML by collecting and synthesizing these aspects for both the ML techniques and entire test generation approaches.

Any future extensions proposed by the authors, with a particular focus on those that could overcome the identified limitations.

Table 2: List of properties used to answer the research questions. For each property, we include a name, the research questions the property is associated with, and a short description.

this sample is sufficiently broad to characterize research in this field.

The publications are listed in Section 4.2 associated with the specific testing practice addressed. Figure 6 shows the growth of interest in this topic since 2002 (one study was previously published in 1993). We can see modest, but growing, interest until 2010. The advancements in ML in the past decade have resulted in significantly more use of ML in test generation, especially starting in 2018. Over two-thirds of the
publications in our sample were published in the past five years alone. This is an area of growing interest and maturity, and we expect the number of publications to increase significantly in the next few years.

3.3. Data Extraction

To answer the questions in Table 1, we have extracted a set of key properties from each study, identified in Table 2. Each property listed in the table is briefly defined and is associated with the research questions. Several properties may collectively answer a RQ. For example, RQ2—covering the goals of using ML—can be answered using property P2. However, P1 provides context and the testing practice addressed may dictate how ML is applied.

Data extraction was performed primarily by the first author of this study. However, to ensure the accuracy of the extraction process, the second author performed a full independent extraction for a sample of ten randomly-chosen publications. We compared our findings, and found that we had near-total agreement on all properties. The second author then performed a verification of the findings of the first author for the remaining publications. A small number of corrections were discussed between the authors, but the data extraction was generally found to be accurate.

4. Results and Discussion

In this section, first, we identify the testing practices addressed by ML-enhanced test generation (RQ1, Section 4.1). We then note observations for individual testing practices (Section 4.2). Finally, we present answers to RQ2-6 (Section 4.3).

4.1. RQ1: Testing Practices Addressed

The purpose of RQ1 is to give an overview of which testing practices have been targeted by the publications. Figure 5 provides an overview. In this chart, we divide articles into layers, with each layer representing finer levels of granularity. The total number of publications in each category is reported below.

The specific formulation of a test case depends on the product domain and technologies utilized by the SUT [11]. However, broadly, a test case is defined by a set of
input steps and test oracles \[1\], both of which can be the target of automated generation. Therefore, input and oracles constitute our first division.

A majority of articles focus on input generation (67% of the sample). Automated input generation has become a major research topic in software testing over the past
20 years [2], and many different forms of automated generation have been proposed, using approaches ranging from symbolic execution [12] to optimization [5].

Oracle generation has long been seen as a major challenge for test automation research [1, 2]. However, ML is a realistic route to achieve automated oracle generation [6], and publications have started to appear (33%).

**RQ1 (Testing Practices):** ML supports generation of both test input and oracles, with a greater focus on input generation (67% of the sample).

Figure 6(a) shows the growth in both topics since 2002. Both show a similar trajectory until 2017, with a sharp increase in input generation after. New ML technologies, such as deep learning, and the growing maturity of open source learning frameworks, such as OpenAI Gym, have potentially contributed to this increase.

### 4.1.1. Test Input Generation

In the second layer of Figure 5, we divide test input generation by the source of information used to create test input:

- **Black Box Testing:** Also known as functional testing [11], approaches use information about the program gleaned from documentation, requirements, and other project artifacts to create test inputs.

- **White Box Testing:** Also known as structural testing [11], approaches use the source code to select test inputs (e.g., generating input that covers a particular outcome for an if-statement). Approaches do not require domain knowledge.

Of the 65 publications addressing input generation, 53 propose Black Box and 12 propose White Box approaches. White Box approaches are traditionally common in input generation, as the “coverage criteria”—checklists of goals [13]—that are the focus of White Box testing offer measurable optimization targets [5]. Such approaches can benefit from the inclusion of ML [5]. However, ML may have greater potential to enhance Black Box testing. Such approaches are based on external data about how the system
should behave. ML opens new opportunities to exploit that data, as shown by 82% of input generation publications proposing Black Box approaches.
The third layer of Figure 5 further subdivides approaches based on testing practice:

- **System Test Generation (26 publications):** Tests target a external subsystem or system interface and verify high-level functionality.

- **GUI Test Generation (18 publications):** Tests target a GUI to identify incorrect functionality or usability/accessibility issues [14].

- **Unit Test Generation (9 publications):** Tests target a single class and exercise its functionality in isolation from other classes.

- **Performance Test Generation (7 publications):** Tests focus on the speed, throughput, and responsiveness of the SUT, often examining variation of computational resources like CPU or disk capacity [15].

- **Combinatorial Interaction Testing (5 publications):** A system-level practice that produces tests covering important interactions between inputs [16].

System-level testing is the most common category (40% of input generation), followed by GUI (28%), then unit testing (14%). GUI, performance (11%) and combinatorial interaction testing (CIT) (8%) represent specialized forms of system testing.

**RQ1 (Testing Practices):** Input generation practices include system testing, specialized types of system testing (GUI, performance, CIT), and unit testing. The majority of these are Black Box approaches, with White Box approaches primarily restricted to unit testing.

Figure 6(b) shows the growth in publications in each area of input generation. We see a particularly strong growth in system and GUI testing since 2017. In addition to the emergence of open-source ML frameworks, we also hypothesize that this is partially driven by the emergence of mobile and web applications and autonomous vehicles. Mobile applications are tested primarily through a GUI, as are many web applications—leading to increased interest in GUI testing. Other web applications are tested through REST APIs, and are included in system testing. Autonomous vehicles also require new approaches, as they are tested in complex simulators [17].
RQ1 (Testing Practices): There has been an increase in publications on system and GUI input generation since 2017, potentially related to the emergence of web and mobile applications and autonomous driving, as well as to the availability of robust, open-source ML and deep learning frameworks.

4.1.2. Test Oracle Generation

The second layer under test oracle generation in Figure 5 divides approaches based on the type of test oracle produced:

- **Expected Output (15 publications):** The oracle predicts concrete behavior that should result for an input. Often, this will be abstracted (i.e., a class of output).

- **Metamorphic Relations and Other Properties (10 publications):** A metamorphic relation is a property, relating input to output [18]—e.g., \( \sin(x) = \sin(\pi - x) \). Such properties, as well as other property types, can be applied to many inputs. Violations identify potential faults.

- **Test Verdicts (8 publications):** The oracle predicts the final test verdict for a given input (i.e., a “pass” or “fail”).

ML supports decision processes. A ML technique makes a prediction, which can either be a decision, or it can offer information that supports make a decision. Test oracles follow a similar model, consisting of information used to issue a verdict and a procedure to arrive at a verdict [6]. ML offers a natural means to replace either component. Test verdict oracles replace the procedure, while expected output and property oracles support arriving at a verdict. Figure 6(c) shows steady growth for all types.

RQ1 (Testing Practices): ML supports generation of test verdict, metamorphic (and other property-based), and expected output oracles.

4.2. Examining Specific Practices

Before answering the remaining research questions, we examine how ML has supported test generation for each practice identified.
| Ref | Year | ML Approach | Technique | Training Data | ML Objective | Evaluation Metrics | Evaluated On |
|-----|------|--------------|-----------|---------------|--------------|-------------------|--------------|
| 19  | 2016 | RL           | Q-Learning| N/A           | Reward (Plan Coverage) | Code Coverage, Assertion Coverage | Robotic Systems |
| 20  | 2021 | RL           | Q-Learning| N/A           | Reward (Criticality) | Faults Detected | Autonomous Vehicles |
| 20  | 2013 | RL           | Delayed   | Q-Learning    | Reward (Test Improvement) | % of Runs Where Requirements Met | Ship Logistics |
| 21  | 2021 | RL           | Q-Learning| N/A           | Reward (Code Coverage) | Code Coverage | Triangle Classification, Nesting Structure, Complex Conditions |
| 22  | 2020 | RL           | Deep RL   | N/A           | Reward (Transition Coverage) | Not Evaluated | OpenAPI APIs |
| 23  | 2020 | RL           | Monte Carlo Control | N/A       | Reward (Input Diversity) | Input Diversity, Code Coverage | XML, JavaScript Parsing |
| 24  | 2021 | Semi-supervised | GAN, CNN | Image Input     | Regression (Speed) | Faults Detected | Autonomous Vehicles |
| 25  | 1993 | Supervised  | Not Specified | System Executions | Regression (Output) | Not Evaluated | N/A |
| 26  | 2018 | Supervised  | Backpropagation NN | System Executions | Regression (Output) | Output Coverage | Train Controller |
| 27  | 2021 | Supervised  | Gaussian Process, Decision Trees, AdaBoostedTree, Random Forest, SVM, ANN | System Executions | Regression (Output) | Accuracy | Power Grid Control |
| 28  | 2019 | Supervised  | LSTM NN   | Existing Inputs | Regression (Valid Input) | Accuracy, Code Coverage | FTP Programs |

Table 3: Publications 1-11 under System Test Generation (Black Box) with publication date, ML type, ML technique, training data, objective of the ML, evaluation metrics, and applications used to evaluate.

4.2.1. System Test Generation

26 publications target system testing. Tables 4-4 outline Black Box approaches, while Table 5 outlines White Box approaches. Each table is sorted by ML type, then by the first author’s name. When discussing the objective, we indicate both type of prediction and the purpose of the prediction.

Input Generation (Supervised, Semi-Supervised): Supervised approaches generally train models that associate input with qualities of interest. [25, 26, 33] infer a model from execution logs containing inputs and resulting output. The model is used to predict input leading to output of interest. For example, [26] identify small changes in input that lead to large differences in output, indicating boundary areas where faults are likely to emerge. [25] and [26] suggest comparing predictions with real output, and
Table 4: Publications 12-21 under **System Test Generation (Black Box)** with publication date, ML type, ML technique, training data, objective of the ML, evaluation metrics, and applications used to evaluate.

| Ref | Year | ML Approach  | Technique               | Training Data   | ML Objective                          | Evaluation Metrics          | Evaluated On                  |
|-----|------|--------------|-------------------------|-----------------|---------------------------------------|----------------------------|-------------------------------|
| 29  | 2019 | Supervised   | CRF                     | Test Descriptions | Regression (Requirement Associations) | Accuracy                   | Telecom Systems               |
| 30  | 2019 | Supervised   | LSTM NN                 | Existing Inputs  | Regression (Failing Input)            | Faults Detected, Efficiency | Smart TV                      |
| 31  | 2021 | Supervised   | Parallel Distributed Processing | System Executions | Regression (Output)                  | Efficiency, Faults Detected, Model Size | Autonomous Vehicles          |
| 32  | 2021 | Supervised   | MLP                     | System Executions | Classification (Input Validity)       | Accuracy                   | REST APIs (GitHub, LanguageTool, Stripe, Yelp, YouTube) |
| 33  | 2015 | Supervised   | C4.5                    | Existing Inputs  | Regression (Output)                  | Mutation Score             | Triangle, BMI, Air Traffic Control |
| 34  | 2020 | Supervised   | LSTM NN                 | Simulink models  | Regression (Validity Rules)           | Input Validity, Faults Detected | Simulink tools                |
| 35  | 2021 | Supervised   | CRF                     | Specifications   | Regression (Requirement Associations) | Accuracy                   | Unspecified                   |
| 36  | 2020 | Supervised,  | Decision Trees, Gradient Boosting, K-Nearest Neighbor, MeanShift | System Executions | Regression (Validity Rules), Clustering (Covered Input) | Num. Clusters, Accuracy, Event Coverage | Bus System, Supply Chain |
| 37  | 2007 | Supervised   | Backpropagation NN      | System Executions | Regression (Output)                  | Accuracy, Efficiency       | Fault Tolerant System, Arc Length |
| 38  | 2019 | Supervised   | SVM                     | Existing Inputs  | Regression (Validity Rules)           | Tests Generated, Tests Executed, Test Size, Faults Detected | Domain-Specific Compiler |

using misclassifications to indicate the need to re-train.

Another concern is achieving code coverage. [13] use neural networks (NNs) for input and oracle generation. A NN associates inputs with paths through the source code, then generates inputs that execute uncovered paths. [36] cluster log files—gathered from customer reports—then compare clusters to logs from executing existing test cases to identify weakly-tested areas of the SUT. Supervised learning is used to fill in these gaps. Traces are formatted as vectors of actions using a bag-of-words algorithm, and the model predicts the next input in the sequence.

Others predict input that will fail. [30] train a NN to identify usage behaviors likely to lead to failures using existing test cases. [27] randomly-generate a set of inputs and
| Ref | Year | ML Approach | Technique | Training Data | ML Objective | Evaluation Metrics | Evaluated On |
|-----|------|--------------|-----------|---------------|--------------|-------------------|--------------|
| [39] | 2021 | RL | ReLU Q-Learning | Constraints | Reward (Solving Cost) | Code Coverage, Queries Solved | GNU coreutils |
| [40] | 2021 | RL | Deep Q-Network | N/A | Reward (Code Coverage, Path Length) | Code Coverage | Sorting |
| [12] | 2021 | Supervised | LSTM NN, Tree-LSTM, K-Nearest Neighbour | Constraints | Regression (Solving Time) | Accuracy, Constraint Solving Time | GNU coreutils, Busybox utils, SMT-COMP |
| [13] | 2014 | Supervised | Backpropagation NN | Existing Inputs, Code Coverage | Regression (Code Coverage) | Code Coverage | Binary Search, Sorting, Median, GCD, Triangle Class |
| [41] | 2011 | Supervised | Backpropagation NN | Existing Inputs, Code Coverage | Regression (Code Coverage) | Code Coverage | Triangle Classification |

Table 5: Publications under **System Test Generation (White Box)** with publication date, ML type, ML technique, training data, objective of the ML, evaluation metrics, and applications used to evaluate.

label them on whether they failed, then cluster failing instances to enhance accuracy. They train a model using several algorithms, then compare their ability to generate failing input. They propose an iterative process where more training data is added over time, and predictions are verified by developers. They find that Gaussian Process regression was the most accurate.

Several authors generate input using models inferred from behavioral specifications. These tests can show that specifications are met. [29] use Conditional Random Fields (CRF) to associate test cases and requirements, creating a dataset where requirements are tagged with output that should follow. CRF associates actions, conditions, and outputs in the requirements, then generates new tests with inputs, conditions, and expected output. [35] also use CRF to classify elements of a specification. Their method takes specifications in natural language and transforms them into a structured abstract test recipe that can be concretely instantiated with different input.

[31] learn a behavioral model. Use cases are modeled in a constraint language and used to generate input from the model based on the constraints. [24] use properties written by human testers to generate new input intended to violate those properties. They present an adversarial scenario where the output of one NN is applied to a second, which offers feedback that enhances the performance of the first. This allows the use
of a small initial training set. They use three Generative Adversarial Networks (GANs) and a Convolutional Neural Network (CNN) to manipulate image data used as input to an autonomous driving system. Collectively, these models predict which input will violate properties by, e.g., changing day to night or adding rain.

Finally, multiple authors generate complex input for particular system types. E.g., [34] train a NN to generate valid Simulink models for testing tool-chains based on the language—a visual language for modelling and simulation.

For compilers, an input is a full program, resulting in a large space of inputs. [38] restrict the range of inputs to avoid wasted effort. They focus on domain-specific compilers, and generate input appropriate for those domains. They extract features from the code, such as number of loops or matrix operations, then train a model to predict whether a new test case belongs to that domain. Test cases not belonging are discarded.

Protocols require textual input that conforms to a specified format. Often, determining conformance requires manual construction of a grammar. [28] generate protocol test input without a pre-defined grammar. They use seq2seq-attention—an encoder-decoder model that uses a NN to transform an input sequence of indeterminate length into semantic features and decode it. This model learns the probability distributions of every character of a message, enabling generation of new valid text sequences.

**Input Generation (Reinforcement Learning):** Both [19] and [22] use RL to generate input to cover states of a model. [19] generate input for robots. The agent operates on beliefs (facts about the environment), desires (goals), and intentions (interaction plans). Q-Learning explores the robot’s environment, using coverage of plan models as the reward function. [22] model APIs as stateful systems—where requests trigger transitions—and use ML to generate API calls to cover all states. They target APIs in the OpenAPI format, using transition coverage as the reward.

[17] use RL to select input for autonomous driving that violates critical requirements. The reward function encapsulates headway time, time-to-collision, and required longitudinal acceleration to avoid a collision.

[23] use RL to generate valid complex input (e.g., structured documents). They use a tabular, on-policy RL technique—Monte Carlo Control—where the reward function
favors unique and valid input. As uniqueness depends on previously-generated input, this is not a problem that can easily be solved with supervised learning.

**Enhancing Test Generation:** ML can improve efficiency or effectiveness of other test generation methods. A common target for improvement are Genetic Algorithms (GAs). A GA generates test cases intended to maximize or minimize a fitness function—a domain-specific scoring function, like the reward function in RL.

- [20] use RL to modify the fitness function, adding and tuning sub-objectives that assist in optimizing the core objective of the search. [37] replace the fitness function with a NN that predicts which input will cover unseen output behaviors. [41] also replace the fitness function, training a model to predict statements that will be covered by input. This model is used when there is no tool support to measure coverage, or in cases where measuring coverage would be expensive.

- [21] use RL to manipulate tests within the GA by modifying input. [40] similarly use RL to improve the effectiveness of a fuzzing tool. RL modifies input selected by the fuzzing algorithm to improve either code coverage or longest execution path length.

- [32] use a NN to predict input validity for REST APIs. This approach would allow a generation framework to filter invalid input before applying it. [12] and [39] both enhance symbolic execution. [12] improve efficiency of constraint solving. Normally, a fixed timeout is used. They used offline learning, based on LSTM and Tree-LSTM, as well as online K-Nearest Neighbours to predict time needed to solve a constraint. [39] also examine constraint solving, using RL to identify the optimal solving strategy for a constraint. Constraint solving is modeled as a Markov Decision Process and the RL agent is first trained offline, then applied online.

4.2.2. **GUI Test Generation**

Table 6 details the 18 GUI testing publications. GUI test generation often focuses on a state-based interface model that formulates display changes as transitions taken following input. Fifteen publications used ML to generate input covering this model.

Almost all publications adopted RL, as it can learn from feedback after applying an action to the GUI, and many GUIs require a sequence of actions to access certain elements. The most common RL technique is Q-Learning [52], [50], [47], [42], [53], [43].
| Ref | Year | ML Approach | Technique | Training Data | ML Objective | Evaluation Metrics | Evaluated On |
|-----|------|-------------|-----------|---------------|--------------|--------------------|--------------|
| 42  | 2018 | RL          | Q-Learning| N/A           | Reward (State Cov.) | State Coverage  | Android Apps  |
| 43  | 2021 | RL          | Monte Carlo Tree Search, Sarsa | N/A | Reward (Test Goal Coverage) | Faults Detected | 2D Games      |
| 44  | 2021 | RL          | Q-Learning| N/A           | Reward (State Cov.) | Qualitative | Resource Planning |
| 45  | 2013 | RL          | Not Specified | N/A | Reward (State Coverage, Loop Interactions) | State Coverage | Android Apps  |
| 46  | 2021 | RL          | Deep Q-Network | N/A | Reward (State Change Magnitude) | Code Coverage, Faults Detected | F-Droid      |
| 47  | 2019 | RL          | Not Specified | N/A | Reward (State Cov., Element Interaction) | State Coverage | F-Droid      |
| 48  | 2018 | RL          | Q-Learning | N/A | Reward (State Cov., Specifications) | State Coverage | F-Droid      |
| 49  | 2020 | RL          | Double Q-Learning | N/A | Reward (State Cov., Specifications) | State Coverage | F-Droid      |
| 50  | 2021 | RL          | Double Q-Learning | N/A | Reward (Specifications) | Faults Detected | F-Droid      |
| 51  | 2012 | RL          | Q-Learning | N/A | Reward (State Cov., Calls) | State Coverage | Password Manager, PDF Reader, Task List, Budgeting |
| 52  | 2020 | RL          | Q-Learning + LSTM | N/A | Reward (State Cov., Curiosity) | State Coverage | Android Apps  |
| 53  | 2018 | RL          | Q-Learning | N/A | Reward (State Cov.) | State Coverage | Android Apps  |
| 54  | 2021 | RL          | Double Q-Learning | N/A | Reward (State Cov., Specifications) | Code Coverage, Faults Detected | Android Apps  |
| 55  | 2021 | RL          | Q-Learning | N/A | Reward (State Cov., Curiosity) | Code Coverage, Faults Detected, Scalability | Web Apps (Research, Real-World, Industrial) |
| 56  | 2019 | Supervised | Deep NN    | System Executions | Regression (Action Probability) | State Coverage | Android Apps  |
| 57  | 2018 | Supervised | Recurrent NN | Existing Inputs | Regression (Test Flows) | State Coverage | Unspecified Web App |
| 58  | 2019 | Supervised | Random Forest | Web Pages | Classification (Page Elements) | Mutation Score | Task List, Job Recruiting Web Apps |
| 59  | 2019 | Supervised | Feedforward ANN | Generated Inputs | Regression (Output) | State Coverage | Login Web App |

Table 6: Publications under GUI Test Generation with publication date, ML type, ML technique, training data, objective of the ML, evaluation metrics, and applications used to evaluate.

53.54, which associates the value of an action with particular states, but can handle stochastic transitions. Q-Learning uses the same reward function to evaluate actual and projected actions. This can exaggerate estimations of future actions. Double Q-Learning, as used by 48.49, corrects this by using a different function to estimate
future rewards. \cite{44} adopted a deep convolutional NN to guide RL.

The main difference between publications lies in the reward function. Many base the reward on state coverage (e.g., \cite{53}), while incorporating additional information to bias state selection. Additional factors include magnitude of the state change \cite{44}, usage specifications \cite{47,48}, unique code functions called \cite{50}, a curiosity factor—focusing exploration of new elements \cite{51,54}—coverage of interaction methods (e.g. click, drag) \cite{46}, and avoidance of navigation loops \cite{44}.

Rather than state coverage, \cite{49} base reward on finding violations of specifications. \cite{14} also apply RL to select input for grid-based 2D games. The game state is represented as a graph, and “test goals” are synthesized from the graph. The reward emphasizes test goal coverage. \cite{7} use supervised learning, training a model to mimic patterns from interaction logs. \cite{7} used a Deep Neural Network (DNN), which associates GUI elements with a probability of usage—using probabilities to bias action selection. \cite{57} filter redundant test cases as part of enhancing a search-based test generation framework. Their model associates input and output, then uses predicted output to decide if tests are redundant.

\cite{55,56} use supervised ML to generate sequences of interactions. They trained using human-written interaction sequences spanning several webpages. \cite{56} extends the approach to interact with forms by extracting feedback messages from forms. The framework learns constraints for form input, and a constraint solver creates input that meets those constraints. A Random Forest is used to classify page components. This helps control how different component types are processed. Their approach requires a complex training phase and a large human-created dataset. However, models can be used for multiple websites, decreasing the training burden.

4.2.3. Unit Test Generation

Because unit testing focuses on individual classes—making domain concerns less applicable—the majority of publications in Table 7 are “White Box” approaches and are not tied to particular system types.

\cite{61,4} use RL to generate input, with code coverage as the reward. In \cite{61}, RL generates input directly. In \cite{4}, a Double Deep Q-Network generates optimization-based
| Ref | Year | Test Gen. Approach | ML Approach | Technique | Training Data | ML Objective | Evaluation Metrics | Evaluated On | Applications |
|-----|------|-------------------|-------------|-----------|--------------|--------------|-------------------|--------------|--------------|
| 58  | 2017 | Black             | Supervised  | Query Strategy Framework | System Executions | Regression (Output) | Mutation Score | Math Library, Time Library |
| 59  | 2018 | Black             | Unsupervised | Backpropagation NN | Existing Inputs | Clustering (Input Similarity) | Not Evaluated | N/A |
| 5   | 2020 | White             | RL          | UCB, DSG-Sama | N/A | Reward (Num Exceptions) | Num. Exceptions, Faults Detected | Compiler, Math Library, String Library, Time Library, Spreadsheet, Mocking Library |
| 60  | 2020 | White             | RL          | UCB, DSG-Sama | N/A | Reward (Input Diversity) | Input Diversity, Faults Detected | JSON Parser |
| 61  | 2011 | White             | RL          | Not Specified | N/A | Reward (Code Coverage) | Code Coverage | Data Structures |
| 62  | 2015 | White             | RL          | Q-Learning | N/A | Reward (Code Coverage) | Code Coverage | Data Structures, Collection Library, Primitives Library, Java/XML Parsers |
| 63  | 2018 | White             | RL          | Double Deep Q-Network | N/A | Reward (Code Coverage) | Code Coverage, Efficiency | GCD, EXP, Remainder |
| 64  | 2021 | White             | Supervised  | Gradient Boosting | Code Metrics | Classification (Fault Prediction) | Accuracy, Faults Detected | Compression, Imaging Library, Math Library, NLP, String Library |
| 65  | 2019 | White             | Supervised  | Backpropagation NN | Existing Inputs | Regression (Code Coverage) | Not Evaluated | N/A |

Table 7: Publications under Unit Test Generation with publication date, generation approach, ML type, ML technique, training data, objective of the ML, evaluation metrics, and applications used to evaluate.

input generation algorithms. RL manipulates heuristics controlling the algorithms. [58] use a supervised approach to generate input for system parts that have only been weakly tested. A model is trained to predict output. The model will have more confidence in prediction accuracy for input similar to the training data. Input with low certainty is retained, as they are likely to test parts of the system ignored in the training data. These inputs can be fed into the training data to re-train the model, shifting focus to other parts of the system.

Many authors use ML to enhance existing test generation approaches—often based on GAs. [5] [60] use RL to adapt GA’s test generation strategy by selecting the fitness functions optimized by the GA to identify functions that trigger exceptions [5] and input diversity [60]. [62] use RL to improve coverage of private and inherited methods by augmenting generated tests. The RL can make two types of changes—it can replace
| Ref | Year | ML Approach | Technique | Training Data | ML Objective | Evaluation Metrics | Evaluated On |
|-----|------|-------------|-----------|---------------|--------------|--------------------|--------------|
| 65  | 2019 | RL          | Dueling Deep Q-Network | N/A | Identified Bottlenecks | Reward (Execution Time) | Auction Website |
| 66  | 2019 | RL          | Q-Learning | N/A | Paths Explored, Efficiency | Reward (Path Length, Feasibility) | Biological Computation, Parser, Sorting, Data Structures |
| 67  | 2019 | RL          | Q-Learning | N/A | Not Evaluated | Reward (Response Time Deviation) | N/A |
| 68  | 2019 | RL          | Q-Learning | N/A | Not Evaluated | Reward (Response Time Deviation) | N/A |
| 69  | 2021 | Semi-Supervised | Conditional GAN | System Executions | Identified Bottlenecks, Accuracy, Labeling and Training Effort | Regression (Perf. Requirements), Classification (Test Realism) | Auction Website |
| 70  | 2016 | Supervised | RIPPER | System Executions | Identified Bottlenecks | Regression (Rule Learning) | Insurance, Online Stores, Project Management |
| 71  | 2021 | Supervised | Multivariate Time Series | Session Logs | Accuracy | Regression (Load) | Student Information |

Table 8: Publications under **Performance Test Generation** with publication date, ML type, ML technique, training data, objective of the ML, evaluation metrics, and applications used to evaluate.

A method call with one whose return type is a subclass of the original method’s, and it can replace a call to a public method with a call to a method that calls a private method. The reward is focused on private method coverage.

[63] predict whether a class is likely to be faulty. This can improve generation efficiency by determining which classes to target. They learn using source code metrics, labelled on whether a class had faults. They use Gradient Boosting—an ensemble of multiple learners. [64] use supervised learning to replace a fitness evaluation in a GA. They focus on data-flow coverage, which is very expensive to calculate. The model replaces the need to actually measure data-flow. [59] use ML to improve GA efficiency. A model clusters test cases. When new tests are generated, those too close to a cluster centroid are rejected.

### 4.2.4. Performance Test Generation

Table 8 details the 7 performance testing publications. Performance can be measured, which offers feedback for subsequent rounds of generation. Thus, the majority
of approaches are based on iterative processes, including reinforcement learning [65, 66, 15, 67], rule [69], and adversarial learning [68].

[65] use Dueling Deep Q Networks to expose performance bottlenecks. Q-Learning has trouble adapting to too many state/action combinations. Dueling DQN uses a NN to generalize states. The reward is based on maximized execution time. The authors note room for improvement by integrating other performance indicators into the reward.

Rather than generating input, [15, 67] apply Q-Learning to control the execution environment. They identify resource configurations (CPU, memory, disk) where timing requirements are violated, with reward based on response time deviation.

[68] uses a Conditional GAN to dynamically learn to generate input violating performance requirements using two competing NNs. The generator produces input, and the discriminator classifies whether input violates requirements. This feedback improves the generator. [69] uses the RIPPER rule learner to identify input classes that trigger intensive computations. When tests are executed, executions are clustered based on execution time. RIPPER learns and iteratively refines rules differentiating the clusters, which are then used to generate new input.

[70] generate workloads for load testing. The model generates realistic load levels on a system at various times and scenarios. Past session logs are clustered, and a multivariate time series is applied to predict system load during a scenario. Finally, [66] use RL to improve symbolic execution during stress testing. They identify input that triggers worst-case execution time, defined as inputs that trigger a long execution path. RL controls the exploration policy used by symbolic execution to identify a policy that favors long paths. The reward is based on path length and feasibility of generating input for that path.

4.2.5. Combinatorial Interaction Testing

Table 9 shows the 5 publications that use ML as part of CIT. [71, 72, 73] use supervised learning (Artificial Neural Networks) to generate covering arrays—input sets that cover pairwise interactions between input variables. [73] predict interaction coverage by an input. They use this model to identify a covering array.
Table 9: Publications under Combinatorial Interaction Testing with publication date, ML type, ML technique, training data, objective of the ML, evaluation metrics, and applications used to evaluate.

| Ref | Year | ML Approach | Technique | Training Data | ML Objective | Evaluation Metrics | Evaluated On |
|-----|------|-------------|-----------|---------------|--------------|--------------------|--------------|
| 16  | 2015 | RL          | SOFTMAX   | N/A           | Reward (Input Combinations) | Covering Array Size, Efficiency | Misc. Synthetic, Real Systems |
| 71  | 2014 | Supervised  | ANN       | Pairwise Input Combinations | Other (Structure Input Space) | Covering Array Size | Web Apps |
| 72  | 2018 | Supervised  | ANN       | Specifications | Other (Structure Input Space), Regression (Output) | Covering Array Size | Temperature Monitoring |
| 73  | 2018 | Supervised  | ANN       | Pairwise Input Combinations | Regression (Input Coverage) | Covering Array Size, Efficiency | Unspecified |
| 74  | 2013 | Unsupervised| Expectation-Maximization | System Executions | Clustering (Code Coverage) | Qualitative Analysis | Bubble Sort, Math Functions, HTTP Processing, Banking |

[71] map each hidden layer of the ANN to a variable, and each node represents a value class. The values are connected by their connection to other variables. They do not use the network for prediction, but as a structuring mechanism to generate a covering array. Code coverage prunes redundant test cases. In a follow-up study [72], they manually construct a ANN using requirements, linking outputs to input values, with each input node mapping to an input variable, hidden layers linked to conditions from the requirements, and output nodes linked to predicted SUT output. The NN again provides structure—a covering array is generated based on paths through the network.

[16] use RL to tune the generation strategy of a Simulated Annealing generation framework. The RL selects how Simulated Annealing mutates a covering array. The reward is based on the change in coverage of combinations after imposing a mutation. Their framework recognizes and exploits policies that improve coverage.

CIT assumes that input values are divided into classes. Division is generally done manually, but identifying divisions is non-trivial. [74] use clustering to identify value classes, based on executed code lines (and how many times lines were executed).

4.2.6. Test Oracle Generation

Tables [10][12] summarize the 33 oracle generation publications. Almost all approaches adopt supervised learning. These train oracles using previous system exe-
Ref | Year | ML Approach | Technique | Training Data | ML Objective | Evaluation Metric | Evaluated On
--- | --- | --- | --- | --- | --- | --- | ---
74 | 2018 | Supervised | Adaptive Boosting | System Executions | Classification (Verdict) | Mutation Score | Shopping Cart
76 | 2021 | Supervised | CNN | Screenshots | Classification (Verdict) | Accuracy, Faults Detected | Games (Android, iOS)
77 | 2018 | Supervised | Backpropagation NN | System Executions | Classification (Verdict) | Mutation Score | Embedded Software
78 | 2017 | Supervised | Not Specified | System Executions | Classification (Verdict) | Faults Detected | Automotive Applications
79 | 2012 | Supervised | L* | System Executions | Classification (Verdict) | Faults Detected, Efficiency | Platoon Simulator
80 | 2016 | Supervised | MLP | System Executions | Classification (Verdict) | Accuracy | User Creation
81 | 2010 | Supervised | Backpropagation NN | System Executions | Classification (Verdict) | Mutation Score | Student Registration
82 | 2021 | Supervised | MLP + LSTM | System Executions | Classification (Verdict) | Accuracy, Training Data Size | Blockchain Module, Deep Learning Module, Encryption Library, Stream Editor

Table 10: Publications under Test Verdicts Test Oracle Generation with publication date, ML type, ML technique, training data, objective of the ML, evaluation metrics, and applications used to evaluate.

cutions, screenshots, or metadata about source code features. The model then predicts the correctness of output or properties of expected output.

**Test Verdicts:** [80] [81] [77] [82] employ various NNs to train a model that predicts verdicts. [82] train models for complex programs using a deep NN with long-short term memory (LSTM). [75] use adaptive boosting, an ensemble technique.

[76] train a model to identify rendering errors in video games by training on screenshots of previous faults. [78] combine ML and model checking. A model is learned from system executions that predicts output. Given the model and specifications, a model checker assesses whether each specification is met, yielding a verdict. For each violation, a test is generated that can be executed to confirm the fault. If the fault is not real, the test and its outcome can be used to retrain the model. In a follow-up study [79], the authors demonstrate their technique on systems-of-systems.

**Expected Output:** The approaches train on system executions, and then predict output given a new input. Output is often abstracted to representative values or limited to functions with enumerated values, rather than specific output. A common application is “triangle classification”—a classification of a triangle as scalene, isosceles,
| Ref | Year | ML Approach | Technique                   | Training Data | ML Objective                  | Evaluation Metric | Evaluated On                  |
|-----|------|-------------|-----------------------------|---------------|-------------------------------|-------------------|-----------------------------|
| [13] | 2004 | Supervised  | Backpropagation NN          | System Executions | Classification (Output)       | Correct Classifications | Triangle Classification |
| [21] | 2021 | Supervised  | Regression Tree, SVM, Ensemble, RGP, Stepwise Regression | System Executions | Regression (Time) | Accuracy | Elevator |
| [13] | 2016 | Supervised  | SVM                         | System Executions | Classification (Output) | Mutation Score | Image Processing |
| [21] | 2021 | Supervised  | Regression Tree, SVM, Ensemble, TRGP, Stepwise Regression | System Executions | Regression (Time) | Accuracy | Elevator |
| [17] | 2008 | Supervised  | Backpropagation NN          | System Executions | Classification (Output)       | Correct Classifications | Triangle Classification |
| [13] | 2014 | Supervised  | Backpropagation NN          | System Executions | Classification (Output)       | Faults Detected | Static Analysis |
| [21] | 2019 | Supervised  | Deep NN                     | System Executions | Classification (Output)       | Mutation Score | Mathematical Functions |
| [21] | 2011 | Supervised  | RBF NN                      | System Executions | Classification (Output)       | Correct Classifications | Triangle Classification |
| [21] | 2011 | Supervised  | MLP                         | System Executions | Classification (Output)       | Mutation Score | Insurance Application |
| [21] | 2012 | Supervised  | MLP                         | System Executions | Classification (Output)       | Mutation Score | Insurance Application |
| [21] | 2016 | Supervised  | Backpropagation NN + Cascade | System Executions | Classification (Output)       | Accuracy | Credit Analysis |
| [21] | 2002 | Supervised  | Not Specified               | System Executions | Classification (Output)       | Mutation Score | Credit Analysis |
| [21] | 2014 | Supervised  | Backpropagation NN, Decision Tree | System Executions | Classification (Output)       | Mutation Score | Triangle Classification |
| [21] | 2006 | Supervised  | MLP                         | System Executions | Classification (Output)       | Mutation Score | Mathematical Functions |
| [21] | 2019 | Supervised  | Probabilistic NN            | System Executions | Classification (Output)       | Correct Classifications | Prime, Triangle Class |

Table 11: Publications under Expected Output Test Oracle Generation with publication date, ML type, ML technique, training data, objective of the ML, evaluation metrics, and applications used to evaluate.

equilateral, or not-a-triangle. This is a challenging function, with branching behavior, but it has limited outputs—making it a common target for oracle generation. [96] model a function that judges whether an integer is prime—a binary classification problem. [85, 90, 91, 92, 93] also generate oracles for applications with enumerated output. Recently, however, [84, 86, 88, 95] generate oracles for functions with unconstrained—e.g., integer—output.

The majority of approaches use of some form of NN [83, 87, 13, 88, 89, 90, 91, 92, 94, 95, 96]. [83] used a Support Vector Machine (SVM) with label propagation—a
| Ref | Year | ML Approach | Technique | Training Data | ML Objective            | Evaluation Metric | Evaluated On     |
|-----|------|--------------|-----------|---------------|-------------------------|-------------------|------------------|
| 97  | 2020 | RL           | Not Specified | N/A           | Reward (Relations)      | Not Evaluated     | Ocean Modeling   |
| 98  | 2021 | RL           | Not Specified | N/A           | Reward (Relations)      | Not Evaluated     | Ocean Modeling   |
| 99  | 2020 | RL           | Contextual Bandit | N/A           | Reward (Faults Detected) | Faults Detected  | Object Detection |
| 100 | 2018 | Supervised   | SVM        | Code Features | Classification (Property) | Accuracy          | Misc. Functions  |
| 100 | 2013 | Supervised   | SVM, Decision Trees | Code Features | Classification (Property) | Mutation Score    | Misc. Functions  |
| 101 | 2016 | Supervised   | SVM        | Code Features | Classification (Property) | Mutation Score    | Misc. Functions  |
| 102 | 2021 | Supervised   | Decision Trees | System Executions | Regression (Conditions) | Accuracy          | Android Apps     |
| 103 | 2019 | Supervised   | SVM        | Code Features | Classification (Property) | ROC               | Matrix Calculation |
| 104 | 2007 | Supervised   | L*         | System Executions | Classification (Violation) | Training Data Size | Handshake Protocols |
| 105 | 2017 | Supervised   | RBF NN     | Code Features | Classification (Property) | Accuracy          | Misc. Functions  |

Table 12: Publications under Metamorphic Properties Test Oracle Generation with publication date, ML type, ML technique, training data, objective of the ML, evaluation metrics, and applications used to evaluate.

technique where labeled and unlabeled training data are used, and the algorithm propagates labels to similar, unlabeled data to reduce the quantity of required labeling. [84][86] compared regression trees, SVM, an ensemble model, a Regression Gaussian Process (RGP), and a stepwise regression. [84] found regression tree to be the best, while [86] found regression tree, ensemble, and RGP valid.

Metamorphic Relations: Several build on [100], whose approach (a) converts code into control-flow graphs, (b) selects code elements as features for a data set, and (c), trains a model that predicts whether a feature exhibits a particular metamorphic relation from a list. This requires training data where features are labeled with a classification based on whether or not they exhibit a particular relation. [101] extended this work by adding a graph kernel. [18] adapted this approach for label propagation. [105] extended the approach to a multi-label classification that can handle multiple metamorphic relations at once. [103] demonstrated how data augmentation can enlarge the training dataset using mutants as the source of additional training data.
Table 13: ML goals and the number of publications pursuing each goal.

| Type of Goal          | Goal                              | # of Publications |
|-----------------------|-----------------------------------|-------------------|
| Generate Input        | Maximize Coverage                  | 25                |
|                       | Expose Performance Bottlenecks    | 6                 |
|                       | Show Conformance to (or Violation of) Specifications | 5 |
|                       | Generate Complex Input             | 5                 |
|                       | Improve Input or Output Diversity  | 4                 |
|                       | Predict Failing Input              | 2                 |
| Generate Oracle       | Predict Output                     | 15                |
|                       | Predict Properties of Output       | 9                 |
|                       | Predict Test Verdict               | 8                 |
| Enhance Existing Method | Improve Effectiveness            | 11                |
|                       | Improve Efficiency                 | 7                 |

[102] predict the conditions on screen transitions in a GUI. Their model is trained using past system execution and potential guard conditions. [104] use ML to assess security properties of protocols. A protocol is specified using a state machine, and message confidentiality is assessed on message reachability. A model is inferred, then assessed for violations. If a violation is found, input is produced to check against the implementation. If the violation is false, the test helps retrain the model.

[97, 98] predict metamorphic relations for ocean modeling. The RL approach poses relations, evaluates whether they hold, and attempts to minimize a cost function based on the validity of the set of proposed relations. [99] use RL to select metamorphic relations from a superset of potentially-applicable relations. Their approach evaluates whether selected relations can discover faults in an image classification algorithm.

4.3. Answering the Research Questions

4.3.1. RQ2: Goals of Applying ML

Table 13 lists the goals of authors in adopting ML, sorted into three broad categories. In the first two, ML is used directly to generate input or an oracle. As previously discussed, oracle generation uses ML to predict output, to properties of output, or a test verdict. Regarding input generation, the most common goal is to use ML to increase coverage of some criterion associated with effective testing. This includes
coverage of code, states or transitions of models, or input interactions. Other uses of ML include generating input that exposes performance bottlenecks, demonstrates conformance to—or violation of—specifications, or increases input/output diversity. Others generate input for a complex data type or input likely to fail.

In the final category, ML tunes the performance or effectiveness of a generation framework—often a GA. To improve efficiency, ML clusters redundant tests, replaces expensive calculations with predictions, chooses generation targets, or checks input validity. To improve effectiveness, ML manipulates test cases (e.g., replaces method calls) or tunes the generation strategy (e.g., selects fitness functions, mutation heuristics, or timeouts).

**RQ2 (Goal of ML):** ML generates input (49%)—particularly to maximize some form of coverage—or oracles (33%)—particularly to predict output. It also improves efficiency or effectiveness of existing generation methods (19%).

### 4.3.2. RQ3: Integration into Test Generation

RQ3 highlights where and how ML has been integrated into the testing process. This includes types of ML applied, training data, and how ML was used (regression, classification, reward functions).

**RQ3 (Integration of ML):** The most common ML types are supervised (59%) and RL (36%). Some publications also employ unsupervised (4%) or semi-supervised (2%) learning.

Supervised techniques were the first applied to input and oracle generation, and remain the most common. Supervised techniques are—by far—the most common for oracle generation. They are also the most common for system and combinatorial interaction testing. The predictions made by models are either from pre-determined options (classification) or open (regression). Classification is often used in oracle generation, e.g., to produce a verdict (pass/fail) or output from a limited range. Regression is common in input generation, where complex predictions must be made.
Both training time and quantity of training data need to be accounted for when considering a supervised technique. After being trained, a model will not learn from new interactions, unlike with RL. A model must be retrained with new training data to improve its accuracy. Therefore, it is important that supervised methods be supplied with sufficient quantity and quality of training data. Supervised techniques generally learn from past system executions, labeled with a measurement of interest. If the label can be automatically recorded, then gathering sufficient data is often not a major concern. However, if the SUT is computationally inefficient or information is not easily collectible (e.g., a human must label data), it can be difficult to use supervised ML.

Adversarial learning may help overcome data challenges. This strategy forces models to compete, creating a feedback loop where performance is improved without the need for human input. Two publications adopted adversarial networks, both in cases where input was associated with a numeric quality (performance, vehicle speed). Neither case requires human labeling, so models can be automatically retrained.

**RQ3 (Integration of ML):** Supervised learning is the most common ML for system testing, CIT, and all forms of oracle. Models perform regression (often for input generation) or classification (often for oracle generation). Quantity and quality of training data are concerns, especially when human input is required. Adversarial approaches can improve accuracy.

RL is the second most common type of ML. Both supervised and RL have seen a sharp increase in after 2017. RL was even used more often than supervised in 2020, and almost as often in 2021. RL has been used in all input generation problems, and is the most common technique for GUI, unit, and performance generation.

RL is appealing because it does not require pre-training and automatically improves accuracy through interactions. RL is most applicable when effectiveness can be judged using a numeric metric, i.e., where a measurable assessment already exists. This includes performance measurements—e.g., resource usage—or code coverage. RL is also effective when the SUT has branching or stateful behavior—e.g., in GUI testing, where a sequence of input may be required. Similarly, performance bottlenecks often
emerge as the consequence of a sequence of actions, and code coverage may require multiple setup steps.

Outside of individual tests, RL is also effective at enhancing test generation algorithms. GAs, for example, evolve test suites over a series of subsequent generations. RL can tune aspects of this evolution, guided by feedback from the same fitness functions targeted by the optimization. If a test suite attains high fitness, RL may be able to improve that score by manipulating the test cases of the algorithm parameters. RL can, of course, generate input effectively in a similar manner to an optimization algorithm. However, it also can often improve the algorithm such that it produces even better tests.

RQ3 (Integration of ML): RL is the most common ML for GUI, unit, and performance testing. It is effective for practices with measurable scores, when a sequence of input is required, and at tuning existing generation tools.

Others applied unsupervised learning to cluster test cases to improve generation efficiency or to identify weakly tested areas of the SUT. Clustering has not been used often in generation, but is common in testing practices (e.g., to identify tests to execute [3]). It has potential for use in filtering tasks during generation, especially to improve efficiency.

RQ3 (Integration of ML): Clustering is effective for filtering tasks such as discarding similar test cases or identifying uncommon input.

4.3.3. RQ4: ML Techniques Applied

RQ4 examines specific ML techniques. Table[4] lists techniques employed, divided by ML type. Neural networks are the most common techniques used in supervised learning. In particular, Backpropagation NNs are used most (12%). Support vector machines are also employed often, as are forms of decision trees.

Backpropagation NNs are a classic technique where a network is composed of multiple layers. In each layer, a weight value for each node is calculated. In such networks,
| Type         | Family                  | Technique                                                                 | Publications |
|--------------|-------------------------|----------------------------------------------------------------------------|--------------|
| Supervised   | Neural Networks         | Backpropagation NN                                                        | 12           |
|              |                         | Multi-Layer Perceptron                                                    | 5            |
|              |                         | Artificial NN, Long Short-Term Memory (LSTM) NN                           | 4            |
|              |                         | Deep NN, Radial-Basis Function NN                                         | 2            |
|              |                         | Backpropagation NN + Cascade, Convolutional NN, Feedforward ANN, MLP + LSTM, Probabilistic NN, Recurrent NN | 1            |
|              | Trees                   | Decision Tree                                                             | 5            |
|              |                         | Gradient Boosting, Random Forest                                          | 2            |
|              |                         | Ada-Boosted Tree, C4.5, Regression Tree, Tree-LSTM                        | 1            |
|              | Others                  | Support Vector Machine                                                   | 9            |
|              |                         | L*, Conditional Random Fields, K-Nearest Neighbors                        | 2            |
|              |                         | Adaptive Boosting, Ensemble, Gaussian Process, Multivariate Time Series, Parallel Distributed Processing, Query Strategy Framework, Regression Gaussian Process, RIPPER, Stepwise Regression | 1            |
| Q-Learning   |                         | Q-Learning                                                                | 14           |
|              |                         | Deep Q-Network, Double Q-Learning                                        | 2            |
|              |                         | Delayed Q-Learning, Dueling Deep Q-Network, Double Deep                   | 1            |
|              |                         | Q-Network, Q-Learning + LSTM, ReLU Q-Learning                            | 1            |
| RL           |                         | Differential Semi-Gradient Sarsa (DSG-Sarsa), Upper Confidence Bound (UCB) | 2            |
|              |                         | Contextual Bandit, Deep RL, Monte Carlo Control, Monte Carlo Tree Search, Sarsa, SOFTMAX | 1            |
| Semi-supervised |                         | CNN, Generative Adversarial Network (GAN), Conditional GAN               | 1            |
| Unsupervised |                         | Backpropagation NN, Expectation-Maximization, MeanShift                  | 1            |

Table 14: ML techniques adopted—divided by ML type and family of ML techniques—ordered by number of publications where the technique is adopted.

information is fed forward—there are no cyclic connections to earlier layers. However, the backpropagation feature propagates error backwards, allowing earlier nodes to adjust weights if necessary. This leads to less complexity and faster learning rates. In recent years, more complex neural networks have continued to implement backpropagation as one (of many) features.

 Recently, NNs utilizing Long Short-Term Memory (LSTM) have also become quite common. Unlike in traditional feedforward NNs, LSTM has feedback connections. This creates loops in the network, allowing information to persist. This adaptation allows such networks to process not just single data points, but sequences where one data point depends on earlier points. LSTM networks and deep NNs are likely to
become more common.

RL is dominated by forms of Q-Learning—variants are used in 22% of publications. Q-Learning is a prototypical form of off-policy RL, meaning that it can choose either to take an action guided by the current “best” policy—maximizing expected reward—or it can choose to take a random action to refine the policy. Many other RL techniques are also off-policy, and follow a similar process, with various differences (e.g., calculating reward or action decisions in a different manner).

RQ4 (ML Techniques): Neural networks, especially Backpropagation NNs, are the most common supervised techniques. Reinforcement learning is generally based on Q-Learning.

Some authors have chosen algorithms because they worked well in previous work (e.g., [48,101]). Others saw algorithms work on similar problems outside of test generation (e.g., [16]), or chose algorithms thought to represent the state-of-the-art for a problem class (e.g., [30]). However, most authors do not justify their choice of algorithm, nor do they often compare alternatives.

In older publications, authors either implemented ML algorithms or adapted unspecified implementations. In recent years, mature open-source ML frameworks have emerged. These frameworks accelerate the pace and effectiveness of research by making robust algorithms available. ML frameworks used in the sampled publications include keras-rl [4], OpenAI Gym [22,499], PyTorch [99], scikit-learn [18], Theano [66], and WEKA [69,94]. The use of a framework constrains algorithm choice. However, all of these frameworks offer many options, and may allow researchers to compare results across algorithms.

RQ4 (ML Techniques): Algorithm choice is often not explained, but may be inspired by insights from previous or related work, an algorithm having performed well on a similar problem, or algorithms available in open-source frameworks (e.g., OpenAI Gym or WEKA).
Table 15: Evaluation metrics adopted (similar metrics are grouped), divided by ML approach and ordered by number of publications using each metric. Metrics in bold are related to ML.

| Type         | Metric                                                                 | Publications |
|--------------|------------------------------------------------------------------------|--------------|
| Supervised   | Faults Detected (Inc. Mutants, Performance Issues)                      | 25           |
|              | Accuracy (Inc. Correct Classifications, ROC)                           | 24           |
|              | Coverage (Code, State, etc.)                                           | 7            |
|              | Efficiency (Inc. Scalability, # Tests Gen or Executed, Time)           | 7            |
|              | Test Size (Case, Suite, Covering Array)                                | 4            |
|              | Training Data Size                                                     | 2            |
|              | Input/Output Diversity, Input Validity, Model Size                      | 1            |
| RL           | Coverage (Code, State, etc.)                                           | 20           |
|              | Faults Detected (Inc. Mutants, Performance Issues)                      | 10           |
|              | Efficiency (Inc. Scalability, # Tests Gen or Executed, Time)           | 4            |
|              | Input/Output Diversity                                                 | 2            |
|              | # Exceptions, Qualitative Analysis, Queries Solved, Requirements Met, Test Case Size | 1            |
| Semi-supervised | Faults Detected (Inc. Mutants, Performance Issues)                | 2            |
|              | Accuracy, Labeling and Training Effort                                 | 1            |
| Unsupervised | # Clusters, Qualitative Analysis                                       | 1            |

4.3.4. RQ5: Evaluation of the Test Generation Framework

RQ5 examines how authors have evaluated their work—in particular, how ML affects evaluation. The metrics adopted by the authors are listed in Table 15. We group similar metrics (e.g., coverage metrics, notions of fault detection, etc.).

Almost all are standard evaluation metrics for test generation. Some are specific to a testing practice (e.g., covering array size) or aspect of generation (e.g., number of queries solved), while others are applied across testing practices (e.g., fault detection). Naturally—whether ML is incorporated or not—a generation framework must be evaluated on its effectiveness at generating tests.

However, many authors evaluate the impact of ML. Supervised approaches were often evaluated using some notion of model accuracy—using various accuracy measurements, correct classification rate, and ROC. Approaches have also been evaluated
on training data/model size. Semi-supervised approaches were also evaluated using accuracy and labeling/training effort. One unsupervised approach was evaluated on number of clusters.

RL approaches were not evaluated using ML-specific metrics. This is reasonable, as RL learns how to maximize a numeric function. The reward is based on the goals of the overall generation framework. Rather than evaluating using an absolute notion of accuracy, the success of RL can be seen in improved reward measurements, attainment of a checklist of goals, or metrics such as fault detection.

**RQ5 (Evaluation):** ML-enhanced generation is still evaluated by traditional metrics (e.g., fault detection). (Semi-/Un-)Supervised approaches are also evaluated using ML metrics (accuracy, training data/model size, labeling/training effort, # of clusters). RL is evaluated using testing metrics tied to the reward.

### 4.3.5. RQ6: Limitations and Open Challenges

The sampled publications show great potential. However, we have observed multiple challenges that must be overcome to transition research into real-world use.

**Volume, Contents, and Collection of Training Data:** (Semi-)Supervised ML requires training data to create a model. There are multiple challenges related to the required volume of training data, the required contents of the training data, and human effort required to produce that training data.

Regardless of the testing practice addressed, the volume of required training data can be vast. This data is generally attained from labeled execution logs, which means that the SUT needs to be executed many times to gather the information needed to train the model. Approaches based on deep learning could produce highly accurate models, but may require thousands of executions to gather required training data. Some approaches also must preprocess the collected data. While it may be possible to automatically gather training data, the time required to produce the dataset can still be high and must be considered.

This is particularly true for cases where a regression is performed rather than a
classification—e.g., an expected value oracle or complex test input. Producing a complex continuous value is more difficult than a simple classification, and requires significant training data—with a range of outcomes—to make accurate predictions.

In addition, the contents of the training data must be considered. If generating input, the training data must contain a wide range of input scenarios with diverse outcomes that reflect the specific problem of interest and its different branching possibilities. Consider code coverage. If one wishes to predict the input that will cover a particular element, then the training data must contain sufficient information content to describe how to cover that element. That requires a diverse training set.

Models based on output behavior—e.g., expected value oracles or models that predict input based on particular output values—suffer from a related issue. The training data for expected value oracles must either come from passing test cases—i.e., the output must be correct—or labels must be applied by humans. A small number of cases accidentally based on failing output may be acceptable if the algorithm is resilient to noise in the training data, but training on faulty code can result in an inaccurate model. This introduces a significant barrier to automating training by, e.g., generating input and simply recording the output that results.

Models that make predictions based on failures—e.g., test verdict oracles or models that produce input predicted to trigger a failure or performance issue—require training data that contains a large number of failing test cases. This implies that faults have already been discovered and, presumably, fixed before the model is trained. This introduces a paradox. There may be remaining failures to discover. However, the more training data that is needed, the less the need for—or impact of—the model.

In some cases, training data must be labelled (or even collected) by a human. Again, oracles suffer heavily from this problem. Test verdict oracles require training data where each entry is assigned a verdict. This requires either existing test oracles—reducing the need for a ML-based oracle—or human labeling of test results. Judging test results is time-consuming and can be erroneous as testers become fatigued, making it difficult to produce a significant volume of training data. Metamorphic relation oracles face a similar dilemma, where training data must be labeled based on whether a particular metamorphic relation holds.
For some problems, these issues can be avoided by employing RL instead. RL will learn while interacting with the SUT. In cases where the effectiveness of ML can be measured automatically—e.g., code coverage, performance bottlenecks—RL is a viable solution. However, cases where a ground truth is required—e.g., oracles—are not as amenable to RL. RL also requires many executions of the SUT, which can be an issue if the SUT is computationally expensive or otherwise difficult to execute and monitor, such as when specialized hardware is required for execution.

Otherwise, techniques are required that (1) can enhance training data, (2) that can extrapolate from limited training data, and (3), that can tolerate noise in the training data. Means of generating synthetic training data, like in the work of Nair et al. [103], demonstrate the potential for data augmentation to help overcome this limitation. Adversarial learning also offers a way to improve the accuracy of a model—reducing the need for a large training dataset. Again, however, such approaches are of limited use in cases where human involvement is required.

RQ6 (Challenges): Supervised learning is limited by the required quantity, quality, and contents of training data—especially when human effort is required. Oracles particularly suffer from these issues. RL and adversarial learning are viable alternatives when data collection and labelling can be automated.

Retraining and Feedback: After training, models have a fixed error rate and do not learn from new mistakes made. If the training data is insufficient or inaccurate, the generated model will be inaccurate. The ability to improve the model based on additional feedback could help account for limitations in the initial training data.

There are two primary means to overcome this limitation—either retraining the model using an enriched training dataset, or adopting a reinforcement learning approach that can adapt its expectations based on feedback. Both means carry challenges. Retraining requires (a) establishing a schedule for when to train the updated model, and (b), an active effort on the part of human testers to enrich and curate the training dataset. Adversarial learning offers an automated means to retrain the model. However, there are still limitations on when it can be applied.
Enriching the dataset—as well as the use of RL—requires some kind of feedback mechanism to judge the effectiveness of the predictions made. This can be difficult in some cases, such as test oracles, where human feedback may be required. Human feedback, even on a subset of the decisions made, reduces the cost savings of automation.

**RQ6 (Challenges):** Models should be retrained over time. How often retraining occurs depends, partially, on the cost to gather and label additional data or on the amount of human feedback required.

**Complexity of Studied Systems:** Regardless of ML type, many of the proposed approaches are evaluated on highly simplistic systems, with only a few lines of code or possible function outcomes. While it is intuitive to start with simplistic examples to examine viability of an ML approach, real-world application requires accurate predictions for complex functions and systems with many branching code paths. If a function is simple, there is likely little need for a predictive model in the first place. Several recent studies feature more thorough evaluations (e.g., [39][5][54]), even on industrial systems (e.g., [32][16]). However, it largely remains to be seen whether the proposed techniques can be used on real-world production code.

Generation of models for arbitrary systems with unconstrained output may be prohibitively difficult for even sophisticated ML techniques. This is particularly the case for expected value oracles. In such cases, some abstraction should be expected. One possibility to consider is a variable level of abstraction—e.g., a training-time decision to cluster output predictions into an adjustable number of representative values (e.g., the centroid of each cluster). Training could take place over different settings for this parameter, and the balance between accuracy and abstraction could be explored.

In any evaluation, a variety of systems should be considered. The complexity of the systems should vary. This enables the assessment of scalability of the proposed techniques. Researchers should examine how prediction accuracy, training data requirements (for supervised learning), and time to convergence on an optimal policy (for RL) scale as the complexity of the system increases. This would enable a better understanding of the limitations and applicability of ML-based techniques.
RQ6 (Challenges): Scalability of many ML techniques to real-world systems is not clear. When modeling complex functions, varying degrees of abstraction could be explored if techniques are unable to scale. In all evaluations, a range of systems should be considered, and explicit analysis of scalability (e.g., of accuracy, training, learning rate) should be performed.

Variety, Complexity, and Tuning of ML Techniques: Authors rarely explain or justify their choice of ML algorithm—often stating that an algorithm worked well previously or that it is “state-of-the-art”, if any rationale is offered. It is even rarer that multiple algorithms are compared to determine which is best for a particular task. As the purpose of many research studies is to demonstrate the viability of an idea, the choice of algorithm is not always critically important. However, this choice still has implications, as it may give a false impression of the applicability of an approach and unnecessarily introduce a performance ceiling that could be overcome through consideration of alternative techniques.

One reason for this limitation may be that testing researchers are generally ML users, not ML experts. They may lack the expertise to know which algorithms to apply. Collaboration with ML researchers may help overcome this challenge. The use of open-source ML frameworks can also ease this challenge by removing the need for researchers to develop their own algorithms. Rather than needing to understand each algorithm, they could instead compare the performance of available alternatives. This comparison would also lead to a richer evaluation and discussion.

Many of the proposed approaches—especially earlier ones—are based on simple neural networks with few layers. These techniques have strict limitations in the complexity of the data they can model and have been replaced by more sophisticated techniques. Deep learning, which may utilize many hidden layers, may be essential in making accurate predictions for complex systems. Few approaches to date have utilized deep learning, but such approaches are starting to appear, and we would expect more to explore these techniques in the coming years. However, deep learning also introduces steep requirements on the training data that may limit its applicability.
Almost all of the proposed approaches utilize a single ML technique. An approach explored in many domains is the use of ensembles [27]. In such approaches, models are trained on the same data using a variety of techniques. Each model is asked for a prediction, and then the final prediction is based on the consensus of the ensemble. Ensembles are often able to reach stable, accurate conclusions in situations where a single model may be inaccurate. A small number of studies have applied ensembles [27, 63, 75, 84, 86], but such techniques are rare. Ensembles may be a way to overcome the fragility of some ML approaches.

Many ML techniques have parameters that can be tuned (e.g., learning rate, number of hidden units, or activation function). Parameter tuning can significantly impact prediction accuracy and enable significant improvements in the results of even simple ML techniques. The sampled publications do not explore the impact of such tuning. This is an oversight that should be corrected in future work.

**RQ6 (Challenges):** Researchers rarely justify the choice of ML technique or compare alternatives. The use of open-source ML frameworks can ease comparison. Deep learning and ensemble techniques, as well as hyperparameter tuning, should also be explored.

**Lack of Standard Benchmarks:** Research benchmarks have enabled sophisticated analyses and comparison of approaches for automated test generation. Such benchmarks usually contain a set of systems prepared for a particular type of evaluation. Bug benchmarks, in particular, contain real faults curated from a variety of systems, along with metadata on those faults. Such benchmarks ease comparison with past research, remove bias from system selection, and demonstrate the effectiveness of techniques.

System or bug benchmarks have been used in a small number of sampled publications (e.g., Defects4J [5, 60], F-Droid [45, 46]). However, the majority of studies do not use benchmarks. Some studies require their own particular evaluation. However, in cases where evaluation is over-simplistic, or where code or metadata is unavailable, this makes comparison and replication difficult.
Benchmarks are typically tied to particular system types or testing practices. In cases where benchmarks exist—unit, web app, mobile app, and performance testing in particular—we would encourage researchers to use these benchmarks. In other cases, the creation of benchmarks specifically for ML-enhanced test generation research could advance the state-of-the-art in the field, spur new research advances, and enable replication and extension of proposed approaches.

In particular, we recommend the creation of such a benchmark for oracle generation. Such a benchmark should contain a variety of code examples from multiple domains and of varying levels of complexity. Code examples should be paired with the metadata needed to support oracle generation. This would include sample test cases and human-created test oracles, at minimum. Such a benchmark could also include sample training data that could be augmented over time by researchers.

**Lack of Replication Package or Open Code:** A common dilemma is lack of access to research code and data. Often, a publication is not sufficient to allow replication or application in a new context. This applies to research in ML-enhanced test generation as well, as few authors provided replication packages. Some publications make use of open-source frameworks. This is positive, in that the tools are trustworthy and available. However, there still may not be enough information in the paper to enable replication. Further, these frameworks evolve over time, and the results may differ because the underlying ML technique has changed since the original study was published.

Researchers should include a replication package with their source code, execution scripts, and the versions of external dependencies used when the study was performed. This package should also include training data and the gathered experiment observations used by the authors in their analyses.

---

**RQ6 (Challenges):** Research is limited by overuse of simplistic examples, the lack of common benchmarks, and the unavailability of code and data. Researchers should be encouraged to use available benchmarks, and provide replication packages and open code. New benchmarks could be created for ML challenges (e.g., oracle generation).
5. Threats to Validity

External and Internal Validity: Our conclusions are based on the publications sampled. It is possible that we may have omitted important publications. This can affect internal validity—the evidence we use to make conclusions—and external validity—the generalizability of our findings. SLRs are not required to reflect all publications from a research field. Rather, their selection protocol (search string, inclusion/exclusion criteria) should be sufficient to ensure an adequate sample. We believe that our selection strategy was appropriate. We tested different search strings, and performed a validation exercise to test the robustness of our string. We have used four databases, covering the majority of relevant venues. Our final set of publications includes 97 primary publications, which we believe is sufficient to make informed conclusions.

Conclusion Validity: Subjective judgements are part of article selection, data extraction, and categorizing publications. To control for bias, protocols were discussed and agreed upon by both authors, and independent verification took place on—at least—a sample of all decisions made by either author.

Construct Validity: We used a set of properties to guide data extraction. These properties may have been incomplete or misleading. However, we have tried to establish properties that were informed by our research questions. These properties were iteratively refined, and we believe they have allowed us to thoroughly answer the questions.

6. Conclusions

Automated test generation is a well-studied research topic, but there are critical limitations to overcome. Recently, researchers have begun to use ML to enhance automated test generation. We have characterized emerging research on this topic through a systematic literature review examining testing practices that have been addressed, the goals of using ML, how ML is integrated into the generation process, which specific ML techniques are applied, how the full test generation process is evaluated, and open research challenges.

We observed that ML generates input for system, GUI, unit, performance, and combinatorial testing or improves the performance of existing generation methods. ML is
also used to generate test verdicts, property-based, and expected output oracles. Supervised learning—often based on neural networks—and reinforcement learning—often based on Q-learning—are common, and some publications also employ unsupervised or semi-supervised learning. (Semi-/Un-)Supervised approaches are evaluated using both traditional testing metrics and ML-related metrics (e.g., accuracy), while reinforcement learning is often evaluated using testing metrics tied to the reward function.

The work-to-date shows great promise, but there are open challenges regarding training data, retraining, scalability, evaluation complexity, ML algorithms employed—and how they are applied—benchmarks, and replicability. We hope that our findings will serve as a roadmap for both researchers and practitioners interested in the use of ML as part of test generation.

7. Acknowledgments

This research was supported by Vetenskapsrådet grant 2019-05275.

References

[1] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, S. Yoo, The oracle problem in software testing: A survey, IEEE transactions on software engineering 41 (2014) 507–525.

[2] A. Orso, G. Rothermel, Software testing: A research travelogue (2000–2014), in: Proceedings of the Future of Software Engineering, FOSE, ACM, New York, NY, USA, 2014, pp. 117–132. doi:10.1145/2593882.2593885

[3] V. H. Durelli, R. S. Durelli, S. S. Borges, A. T. Endo, M. M. Eler, D. R. Dias, M. P. Guimaraes, Machine learning applied to software testing: A systematic mapping study, IEEE Transactions on Reliability 68 (2019) 1189–1212.

[4] J. Kim, M. Kwon, S. Yoo, Generating test input with deep reinforcement learning, in: International Workshop on Search-Based Software Testing, SBST, Association for Computing Machinery, New York, NY, USA, 2018, p. 51–58. doi:10.1145/3194718.3194720
[5] H. Almulla, G. Gay, Learning how to search: Generating exception-triggering tests through adaptive fitness function selection, in: IEEE International Conference on Software Testing, Validation and Verification (ICST), 2020, pp. 63–73. doi:10.1109/ICST46399.2020.00017

[6] A. Fontes, G. Gay, Using machine learning to generate test oracles: A systematic literature review, in: International Workshop on Test Oracles, TORACLE, Association for Computing Machinery, New York, NY, USA, 2021, p. 1–10. doi:10.1145/3472675.3473974

[7] Y. Li, Z. Yang, Y. Guo, X. Chen, Humanoid: A deep learning-based approach to automated black-box android app testing, in: IEEE/ACM International Conference on Automated Software Engineering, ASE, IEEE Press, 2019, p. 1070–1073. doi:10.1109/ASE.2019.00104

[8] N. Jha, R. Popli, Artificial intelligence for software testing-perspectives and practices, in: International Conference on Computational Intelligence and Communication Technologies (CCICT), 2021, pp. 377–382. doi:10.1109/CCICT53244.2021.00075

[9] C. Ioannides, K. I. Eder, Coverage-directed test generation automated by machine learning – a review, ACM Trans. Des. Autom. Electron. Syst. 17 (2012). doi:10.1145/2071356.2071363

[10] J. M. Balera, V. A. de Santiago Júnior, A systematic mapping addressing hyper-heuristics within search-based software testing, Information and Software Technology 114 (2019) 176–189.

[11] M. Pezze, M. Young, Software Test and Analysis: Process, Principles, and Techniques, John Wiley and Sons, 2006.

[12] S. Luo, H. Xu, Y. Bi, X. Wang, Y. Zhou, Boosting symbolic execution via constraint solving time prediction (experience paper), in: ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA, Associ-
[13] N. Majma, S. M. Babamir, Software test case generation test oracle design using neural network, in: Iranian Conference on Electrical Engineering (ICEE), 2014, pp. 1168–1173. doi:10.1109/IranianCEE.2014.6999712.

[14] S. Ariyurek, A. Betin-Can, E. Surer, Automated video game testing using synthetic and humanlike agents, IEEE Transactions on Games 13 (2021) 50–67. doi:10.1109/TG.2019.2947597.

[15] M. H. Moghadam, Machine learning-assisted performance testing, in: Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ES-EC/FSE 2019, Association for Computing Machinery, New York, NY, USA, 2019, p. 1187–1189. doi:10.1145/3338906.3342484.

[16] Y. Jia, M. B. Cohen, M. Harman, J. Petke, Learning combinatorial interaction test generation strategies using hyperheuristic search, in: International Conference on Software Engineering, ICSE, IEEE Press, 2015, p. 540–550.

[17] D. Baumann, R. Pfeffer, E. Sax, Automatic generation of critical test cases for the development of highly automated driving functions, in: IEEE Vehicular Technology Conference (VTC), 2021, pp. 1–5. doi:10.1109/VTC2021-Spring51267.2021.9448686.

[18] B. Hardin, U. Kanewala, Using semi-supervised learning for predicting metamorphic relations, in: International Workshop on Metamorphic Testing, MET, Association for Computing Machinery, New York, NY, USA, 2018, p. 14–17. doi:10.1145/3193977.3193985.

[19] D. Araiza-Illan, A. G. Pipe, K. Eder, Intelligent agent-based stimulation for testing robotic software in human-robot interactions, in: Workshop on Model-Driven Robot Software Engineering, MORSE, Association for Computing Machinery, New York, NY, USA, 2021, p. 336–347. doi:10.1145/3460319.3464813.
[20] M. Buzdalov, A. Buzdalova, Adaptive selection of helper-objectives for test case generation, in: 2013 IEEE Congress on Evolutionary Computation, 2013, pp. 2245–2250. doi:10.1109/CEC.2013.6557836

[21] M. Esnaashari, A. H. Damia, Automation of software test data generation using genetic algorithm and reinforcement learning, Expert Systems with Applications 183 (2021) 115446. doi:10.1016/j.eswa.2021.115446

[22] S. Huurman, X. Bai, T. Hirtz, Generating api test data using deep reinforcement learning, in: IEEE/ACM International Conference on Software Engineering Workshops, ICSEW, Association for Computing Machinery, New York, NY, USA, 2020, p. 541–544. doi:10.1145/3387940.3392214

[23] S. Reddy, C. Lemieux, R. Padhye, K. Sen, Quickly generating diverse valid test inputs with reinforcement learning, in: ACM/IEEE International Conference on Software Engineering, ICSE, Association for Computing Machinery, New York, NY, USA, 2020, p. 1410–1421. doi:10.1145/3377811.3380399

[24] Y. Deng, G. Lou, X. Zheng, T. Zhang, M. Kim, H. Liu, C. Wang, T. Y. Chen, Bmt: Behavior driven development-based metamorphic testing for autonomous driving models, in: International Workshop on Metamorphic Testing (MET), 2021, pp. 32–36. doi:10.1109/MET52542.2021.00012

[25] F. Bergadano, Test case generation by means of learning techniques, SIGSOFT Softw. Eng. Notes 18 (1993) 149–162. doi:10.1145/167049.167074

[26] C. Budnik, M. Gario, G. Markov, Z. Wang, Guided test case generation through ai enabled output space exploration, in: International Workshop on Automation of Software Test, AST, Association for Computing Machinery, New York, NY, USA, 2018, p. 53–56. doi:10.1145/3194733.3194740.
[27] R. Eidenbenz, C. Franke, T. Sivanthi, S. Schoenborn, Boosting exploratory testing of industrial automation systems with ai, 2021, pp. 362–371. doi:10.1109/ICST49551.2021.00048.

[28] Z. Gao, W. Dong, R. Chang, C. Ai, The stacked seq2seq-attention model for protocol fuzzing, in: IEEE International Conference on Computer Science and Network Technology (ICCSNT), 2019, pp. 126–130. doi:10.1109/ICCSNT47585.2019.8962499.

[29] K. Kikuma, T. Yamada, K. Sato, K. Ueda, Preparation method in automated test case generation using machine learning, in: International Symposium on Information and Communication Technology, SoICT, Association for Computing Machinery, New York, NY, USA, 2019, p. 393–398. doi:10.1145/3368926.3369679.

[30] M. Kırac, B. Aktemur, H. Sözer, C. Gebizli, Automatically learning usage behavior and generating event sequences for black-box testing of reactive systems, Software Quality Journal 27 (2019) 861–883. doi:10.1007/s11219-018-9439-1.

[31] K. Meinke, H. Khosrowjerdi, Use case testing: A constrained active machine learning approach, Lecture Notes in Computer Science 12740 LNCS (2021) 3–21. doi:10.1007/978-3-030-79379-1_1.

[32] A. G. Mirabella, A. Martin-Lopez, S. Segura, L. Valencia-Cabrera, A. Ruiz-Cortés, Deep learning-based prediction of test input validity for restful apis, in: International Workshop on Deep Learning for Testing and Testing for Deep Learning (DeepTest), 2021, pp. 9–16. doi:10.1109/DeepTest52559.2021.00008.

[33] P. Papadopoulos, N. Walkinshaw, Black-box test generation from inferred models, in: International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering, RAISE, IEEE Press, 2015, p. 19–24.
[34] S. L. Shrestha, Automatic generation of simulink models to find bugs in a cyber-physical system tool chain using deep learning, in: ACM/IEEE International Conference on Software Engineering: Companion Proceedings, ICSE, Association for Computing Machinery, New York, NY, USA, 2020, p. 110–112. doi:10.1145/3377812.3382163

[35] K. Ueda, H. Tsukada, Accuracy improvement by training data selection in automatic test cases generation method, 2021, pp. 438–442. doi:10.1109/ICIET51873.2021.9419636

[36] M. Utting, B. Legeard, F. Dadeau, F. Tamagnan, F. Bouquet, Identifying and generating missing tests using machine learning on execution traces, in: IEEE International Conference On Artificial Intelligence Testing (AITest), 2020, pp. 83–90. doi:10.1109/AITEST49225.2020.00020

[37] R. Zhao, S. Lv, Neural-network based test cases generation using genetic algorithm, in: Pacific Rim International Symposium on Dependable Computing (PRDC), 2007, pp. 97–100. doi:10.1109/PRDC.2007.63

[38] J. Zhu, L. Wang, Y. Gu, X. Lin, Learning to restrict test range for compiler test, in: IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), 2019, pp. 272–274. doi:10.1109/ICSTW.2019.00064

[39] Z. Chen, Z. Chen, Z. Shuai, G. Zhang, W. Pan, Y. Zhang, J. Wang, Synthesize solving strategy for symbolic execution, 2021, pp. 348–360. doi:10.1145/3460319.3464815

[40] C. Paduraru, M. Paduraru, A. Stefanescu, Riverfuzzr - an open-source tool to experiment with reinforcement learning for fuzzing, 2021, pp. 430–435. doi:10.1109/ICST49551.2021.00055

[41] K. Mishra, S. Tiwari, A. Misra, Combining non revisiting genetic algorithm and neural network to generate test cases for white box testing, Ad-
[42] D. Adamo, M. K. Khan, S. Koppula, R. Bryce, Reinforcement learning for Android GUI testing, in: ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation, A-TEST, Association for Computing Machinery, New York, NY, USA, 2018, p. 2–8. doi:10.1145/3278186.3278187.

[43] M. Brunetto, G. Denaro, L. Mariani, M. Pezzè, On introducing automatic test case generation in practice: A success story and lessons learned, Journal of Systems and Software 176 (2021) 110933. doi:10.1016/j.jss.2021.110933.

[44] W. Choi, G. Necula, K. Sen, Guided GUI testing of Android apps with minimal restart and approximate learning, ACM SIGPLAN Notices 48 (2013) 623–639. doi:10.1145/2544173.2509552.

[45] E. Collins, A. Neto, A. Vincenzi, J. Maldonado, Deep reinforcement learning based Android application GUI testing, in: Brazilian Symposium on Software Engineering, SBES, Association for Computing Machinery, New York, NY, USA, 2021, p. 186–194. doi:10.1145/3474624.3474634.

[46] C. Degott, N. P. Borges Jr., A. Zeller, Learning user interface element interactions, in: ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA, Association for Computing Machinery, New York, NY, USA, 2019, p. 296–306. doi:10.1145/3293882.3330569.

[47] Y. Koroglu, A. Sen, O. Muslu, Y. Mete, C. Ulker, T. Tanriverdi, Y. Donmez, Qbe: Qlearning-based exploration of Android applications, in: IEEE International Conference on Software Testing, Verification and Validation (ICST), 2018, pp. 105–115. doi:10.1109/ICST.2018.00020.

[48] Y. Koroglu, A. Sen, Functional test generation from UI test scenarios using...
reinforcement learning for android applications, Software Testing Verification and Reliability (2020). doi:10.1002/stvr.1752

[49] Y. Koroglu, A. Sen, Functional test generation from ui test scenarios using reinforcement learning for android applications, Software Testing Verification and Reliability 31 (2021). doi:10.1002/stvr.1752

[50] L. Mariani, M. Pezze, O. Riganelli, M. Santoro, Autoblacktest: Automatic black-box testing of interactive applications, in: IEEE International Conference on Software Testing, Verification and Validation, 2012, pp. 81–90. doi:10.1109/ICST.2012.88

[51] M. Pan, A. Huang, G. Wang, T. Zhang, X. Li, Reinforcement learning based curiosity-driven testing of android applications, in: ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA, Association for Computing Machinery, New York, NY, USA, 2020, p. 153–164. doi:10.1145/3395363.3397354

[52] T. A. T. Vuong, S. Takada, A reinforcement learning based approach to automated testing of android applications, in: ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation, A-TEST, Association for Computing Machinery, New York, NY, USA, 2018, p. 31–37. doi:10.1145/3278186.3278191

[53] H. Yasin, S. Hamid, R. Yusof, Droidbotx: Test case generation tool for android applications using q-learning, Symmetry 13 (2021) 1–30. doi:10.3390/sym13020310

[54] Y. Zheng, Y. Liu, X. Xie, Y. Liu, L. Ma, J. Hao, Y. Liu, Automatic web testing using curiosity-driven reinforcement learning, 2021, pp. 423–435. doi:10.1109/ICSE43902.2021.00048

[55] D. Santiago, P. J. Clarke, P. Alt, T. M. King, Abstract flow learning for web application test generation, in: ACM SIGSOFT International Workshop
on Automating TEST Case Design, Selection, and Evaluation, A-TEST, Association for Computing Machinery, New York, NY, USA, 2018, p. 49–55. doi:10.1145/3278186.3278194

[56] D. Santiago, J. Phillips, P. Alt, B. Muras, T. King, P. Clarke, Machine learning and constraint solving for automated form testing, volume 2019-October, 2019, pp. 217–227. doi:10.1109/ISSRE.2019.00030

[57] M. M. Kamal, S. M. Darwish, A. Elfatatry, Enhancing the automation of gui testing, in: International Conference on Software and Information Engineering, ICSIE, Association for Computing Machinery, New York, NY, USA, 2019, p. 66–70. doi:10.1145/3328833.3328842

[58] N. Walkinshaw, G. Fraser, Uncertainty-driven black-box test data generation, in: IEEE International Conference on Software Testing, Verification and Validation (ICST), 2017, pp. 253–263. doi:10.1109/ICST.2017.30

[59] I. Hooda, R. Chhillar, Test case optimization and redundancy reduction using ga and neural networks, International Journal of Electrical and Computer Engineering 8 (2018) 5449–5456. doi:10.11591/ijece.v8i6.pp5449-5456

[60] H. Almulla, G. Gay, Generating diverse test suites for gson through adaptive fitness function selection, Lecture Notes in Computer Science 12420 LNCS (2020) 246–252. doi:10.1007/978-3-030-59762-7_18

[61] A. Groce, Coverage rewarded: Test input generation via adaptation-based programming, in: IEEE/ACM International Conference on Automated Software Engineering, ASE, IEEE Computer Society, USA, 2011, p. 380–383. doi:10.1109/ASE.2011.6100077

[62] W. He, R. Zhao, Q. Zhu, Integrating evolutionary testing with reinforcement learning for automated test generation of object-oriented software, Chinese Journal of Electronics 24 (2015) 38–45. doi:10.1049/cje.2015.01.007

[63] E. Hershkovich, R. Stern, R. Abreu, A. Elmishali, Prioritized test generation guided by software fault prediction, in: IEEE International Conference on Soft-
[64] S. Ji, Q. Chen, P. Zhang, Neural network based test case generation for data-flow oriented testing, in: IEEE International Conference On Artificial Intelligence Testing (AITest), 2019, pp. 35–36. doi:10.1109/AITest.2019.00-11

[65] T. Ahmad, A. Ashraf, D. Truscan, I. Porres, Exploratory performance testing using reinforcement learning, in: Euromicro Conference on Software Engineering and Advanced Applications (SEAA), 2019, pp. 156–163. doi:10.1109/SEAA.2019.00032

[66] J. Koo, C. Saumya, M. Kulkarni, S. Bagchi, Pyse: Automatic worst-case test generation by reinforcement learning, in: IEEE Conference on Software Testing, Validation and Verification (ICST), 2019, pp. 136–147. doi:10.1109/ICST.2019.00023

[67] M. Helali Moghadam, M. Saadatmand, M. Borg, M. Bohlin, B. Lisper, Machine learning to guide performance testing: An autonomous test framework, in: IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), 2019, pp. 164–167. doi:10.1109/ICSTW.2019.00046

[68] A. Sedaghatbaf, M. H. Moghadam, M. Saadatmand, Automated performance testing based on active deep learning, in: IEEE/ACM International Conference on Automation of Software Test (AST), 2021, pp. 11–19. doi:10.1109/AST52587.2021.00010

[69] Q. Luo, D. Poshyvanyk, A. Nair, M. Grechanik, Forepost: A tool for detecting performance problems with feedback-driven learning software testing, in: International Conference on Software Engineering Companion, ICSE, Association for Computing Machinery, New York, NY, USA, 2016, p. 593–596. doi:10.1145/2889160.2889164

[70] H. Schulz, D. Okanović, A. van Hoorn, P. Tůma, Context-tailored workload model generation for continuous representative load testing, in: ACM/SPEC
[71] L. Mudarakola, J. Sastry, C. Vudatha, Generating test cases for testing websites through neural networks and input pairs, International Journal of Applied Engineering Research 9 (2014) 11819–11831.

[72] L. Mudarakola, J. Sastry, A neural network based strategy (nnbs) for automated construction of test cases for testing an embedded system using combinatorial techniques, International Journal of Engineering and Technology(UAE) 7 (2018) 74–81.

[73] R. Patil, V. Prakash, Neural network based approach for improving combinatorial coverage in combinatorial testing approach, Journal of Theoretical and Applied Information Technology 96 (2018) 6677–6687.

[74] C. Duy Nguyen, P. Tonella, Automated inference of classifications and dependencies for combinatorial testing, 2013, pp. 622–627. doi:10.1109/ASE.2013.6693123.

[75] R. Braga, P. S. Neto, R. Rabêlo, J. Santiago, M. Souza, A machine learning approach to generate test oracles, in: Brazilian Symposium on Software Engineering, SBES, Association for Computing Machinery, New York, NY, USA, 2018, p. 142–151. doi:10.1145/3266237.3266273.

[76] K. Chen, Y. Li, Y. Chen, C. Fan, Z. Hu, W. Yang, Glib: Towards automated test oracle for graphically-rich applications, in: ACM Joint Meeting of European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE, Association for Computing Machinery, New York, NY, USA, 2021, p. 1093–1104. doi:10.1145/3468264.3468586.

[77] F. Gholami, N. Attar, H. Haghighi, M. V. Asl, M. Valueian, S. Mohamadyari, A classifier-based test oracle for embedded software, in: Real-Time and Embed-
[78] H. Khosrowjerdi, K. Meinke, A. Rasmusson, Learning-based testing for safety critical automotive applications, Lecture Notes in Computer Science 10437 LNCS (2017) 197–211. doi:10.1007/978-3-319-64119-5_13.

[79] H. Khosrowjerdi, K. Meinke, Learning-based testing for autonomous systems using spatial and temporal requirements, in: International Workshop on Machine Learning and Software Engineering in Symbiosis, MASES, Association for Computing Machinery, New York, NY, USA, 2018, p. 6–15. doi:10.1145/3243127.3243129.

[80] W. Makondo, R. Nallanthighal, I. Mapanga, P. Kadebu, Exploratory test oracle using multi-layer perceptron neural network, in: International Conference on Advances in Computing, Communications and Informatics (ICACCI), 2016, pp. 1166–1171. doi:10.1109/ICACCI.2016.7732202.

[81] S. Shahamiri, W. Wan Kadir, S. Bin Ibrahim, An automated oracle approach to test decision-making structures, volume 5, 2010, pp. 30–34. doi:10.1109/ICCSIT.2010.5563989.

[82] F. Tsimpourlas, A. Rajan, M. Allamanis, Supervised learning over test executions as a test oracle, in: ACM Symposium on Applied Computing, SAC, Association for Computing Machinery, New York, NY, USA, 2021, p. 1521–1531. doi:10.1145/3412841.3442027.

[83] K. K. Aggarwal, Y. Singh, A. Kaur, O. P. Sangwan, A neural net based approach to test oracle, SIGSOFT Softw. Eng. Notes 29 (2004) 1–6. doi:10.1145/986710.986725.

[84] A. Arrieta, J. Ayerdi, M. Illarramendi, A. Agirre, G. Sagardui, M. Arratibel, Using machine learning to build test oracles: an industrial case study on elevators dispatching algorithms, in: IEEE/ACM International Conference on Automat-
[85] J. Ding, D. Zhang, A machine learning approach for developing test oracles for testing scientific software, volume 2016-January, 2016, pp. 390–395. doi:10.18293/SEKE2016-137.

[86] A. Gartziandia, A. Arrieta, A. Agirre, G. Sagardui, M. Arratibel, Using regression learners to predict performance problems on software updates: A case study on elevators dispatching algorithms, in: ACM Symposium on Applied Computing, SAC, Association for Computing Machinery, New York, NY, USA, 2021, p. 135–144. doi:10.1145/3412841.3441894.

[87] H. Jin, Y. Wang, N. Chen, Z. Gou, S. Wang, Artificial neural network for automatic test oracles generation, in: International Conference on Computer Science and Software Engineering, volume 2, 2008, pp. 727–730. doi:10.1109/CSSE.2008.774.

[88] A. Monsefi, B. Zakeri, S. Samsam, M. Khashehchi, Performing software test oracle based on deep neural network with fuzzy inference system, Communications in Computer and Information Science 891 (2019) 406–417. doi:10.1007/978-3-030-33495-6_31.

[89] O. P. Sangwan, P. K. Bhatia, Y. Singh, Radial basis function neural network based approach to test oracle, SIGSOFT Softw. Eng. Notes 36 (2011) 1–5. doi:10.1145/2020976.2020992.

[90] S. Shahamiri, W. Kadir, S. Ibrahim, S. Hashim, An automated framework for software test oracle, Information and Software Technology 53 (2011) 774–788. doi:10.1016/j.infsof.2011.02.006.

[91] S. Shahamiri, W. Wan-Kadir, S. Ibrahim, S. Hashim, Artificial neural networks as multi-networks automated test oracle, Automated Software Engineering 19 (2012) 303–334. doi:10.1007/s10515-011-0094-z.
[92] A. Singhal, A. Bansal, A. Kumar, An approach to design test oracle for aspect oriented software systems using soft computing approach, International Journal of Systems Assurance Engineering and Management 7 (2016) 1–5. doi:10.1007/s13198-015-0402-2.

[93] M. Vanmali, M. Last, A. Kandel, Using a neural network in the software testing process, International Journal of Intelligent Systems 17 (2002) 45–62. doi:10.1002/int.1002.

[94] Vineeta, A. Singhal, A. Bansal, Generation of test oracles using neural network and decision tree model, in: Confluence The Next Generation Information Technology Summit (Confluence), 2014, pp. 313–318. doi:10.1109/CONFLUENCE.2014.6949311.

[95] M. Ye, B. Feng, L. Zhu, Y. Lin, Neural networks based automated test oracle for software testing, Lecture Notes in Computer Science 4234 LNCS - III (2006) 498–507.

[96] R. Zhang, Y.-W. Wang, M.-Z. Zhang, Automatic test oracle based on probabilistic neural networks, Advances in Intelligent Systems and Computing 752 (2019) 437–445. doi:10.1007/978-981-10-8944-2_50.

[97] D. J. Hiremath, M. Claus, W. Hasselbring, W. Rath, Automated identification of metamorphic test scenarios for an ocean-modeling application, in: IEEE International Conference On Artificial Intelligence Testing (AITest), 2020, pp. 62–63. doi:10.1109/AITEST49225.2020.00016.

[98] D. J Hiremath, M. Claus, W. Hasselbring, W. Rath, Towards automated metamorphic test identification for ocean system models, in: IEEE/ACM International Workshop on Metamorphic Testing (MET), 2021, pp. 42–46. doi:10.1109/MET52542.2021.00014.

[99] H. Spieker, A. Gotlieb, Adaptive metamorphic testing with contextual bandits, Journal of Systems and Software 165 (2020) 110574. doi:10.1016/j.jss.2020.110574.
[100] U. Kanewala, J. Bieman, Using machine learning techniques to detect metamorphic relations for programs without test oracles, 2013, pp. 1–10. doi:10.1109/ISSRE.2013.6698899

[101] U. Kanewala, J. Bieman, A. Ben-Hur, Predicting metamorphic relations for testing scientific software: A machine learning approach using graph kernels, Software Testing Verification and Reliability 26 (2016) 245–269. doi:10.1002/stvr.1594

[102] O. Korkmaz, C. Yilmaz, Sysmodis: A systematic model discovery approach, 2021, pp. 67–76. doi:10.1109/ICSTW52544.2021.00023

[103] A. Nair, K. Meinke, S. Eldh, Leveraging mutants for automatic prediction of metamorphic relations using machine learning, in: ACM SIGSOFT International Workshop on Machine Learning Techniques for Software Quality Evaluation, MaTeSQuE 2019, Association for Computing Machinery, New York, NY, USA, 2019, p. 1–6. doi:10.1145/3340482.3342741

[104] G. Shu, D. Lee, Testing security properties of protocol implementations - a machine learning based approach, in: International Conference on Distributed Computing Systems (ICDCS), 2007, pp. 25–25. doi:10.1109/ICDCS.2007.147

[105] P. Zhang, X. Zhou, P. Pelliccione, H. Leung, Rbf-mlmr: A multi-label metamorphic relation prediction approach using rbf neural network, IEEE Access 5 (2017) 21791–21805. doi:10.1109/ACCESS.2017.2758790