Neuroevolution-Based Generation of Tests and Oracles for Games

Patric Feldmeier
University of Passau
Germany
patric.feldmeier@uni-passau.de

Gordon Fraser
University of Passau
Germany
Gordon.Fraser@uni-passau.de

ABSTRACT

Game-like programs have become increasingly popular in many software engineering domains such as mobile apps, web applications, or programming education. However, creating tests for programs that have the purpose of challenging human players is a daunting task for automatic test generators. Even if test generation succeeds in finding a relevant sequence of events to exercise a program, the randomized nature of games means that it may neither be possible to reproduce the exact program behaviour underlying this sequence, nor to create test assertions checking if observed randomized game behavior is correct. To overcome these problems, we propose Neatest, a novel test generator based on the *NeuroEvolution of Augmenting Topologies* (NEAT) algorithm. Neatest systematically explores a program's statements, and creates neural networks that operate the program in order to reliably reach each statement—that is, Neatest learns to play the game in a way to reliably reach different parts of the code. As the networks learn the actual game behavior, they can also serve as test oracles by evaluating how surprising the observed inputs of a program under test are compared to the inputs obtained on a supposedly correct version of the program. We evaluate this approach in the context of Scratch, an educational programming environment. Our empirical study on 25 non-trivial Scratch games demonstrates that our approach can successfully train neural networks that are not only far more resilient to random influences than traditional test suites consisting of static input sequences, but are also highly effective with an average mutation score of more than 70%.

CCS CONCEPTS

• **Software and its engineering** → **Software testing and debugging**; • **Computing methodologies** → *Reinforcement learning*; *Neural networks.*

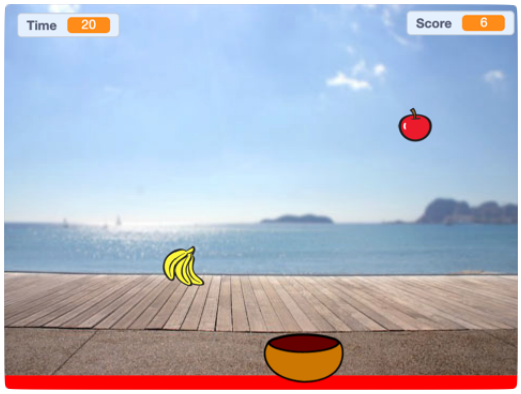
KEYWORDS

Neuroevolution, Game Testing, Automated Testing, Scratch

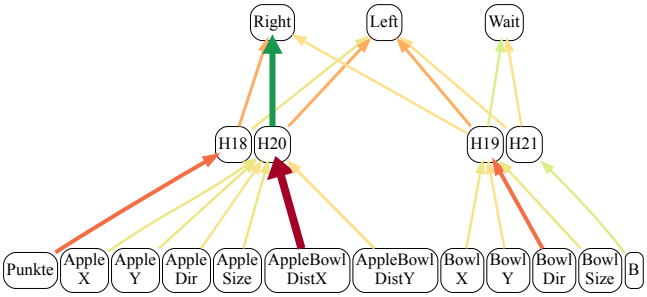
ACM Reference Format:
Patric Feldmeier and Gordon Fraser. 2018. Neuroevolution-Based Generation of Tests and Oracles for Games. In *Proceedings of The 37th IEEE/ACM International Conference on Automated Software Engineering (ASE 2022).* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/1122445.1122456

ASE 2022, 10-14 October, 2022, Ann Arbor

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/1122445.1122456



(a) FruitCatching example game.



(b) Example of an optimized network with excluded regression head. The width and brightness of connections represent their importance.

Figure 1: FruitCatching game and corresponding neural network that is capable of winning the game.

1 INTRODUCTION

Game-like programs allow players to interact with them using specific input actions, and challenge them to solve dedicated tasks while enforcing a set of rules. Once a player successfully completes the task of a game, the game usually notifies the player of his/her victory, leading to an advanced program state (*winning state*). Nowadays, game-like programs can be found in many software engineering domains. For example, the most common category of mobile apps in Google's Playstore[1] as well as in Apple's App Store[2] are games by far, and games are equally common in web and desktop applications. Programming education is also heavily reliant on learners building game-like programs in order to engage them with the challenges of programming. Fig. 1a shows the *FruitCatching* game, a typical program created by young learners of the Scratch programming language [32]: The player has to control a bowl at

[1] [May 2022] https://www.statista.com/statistics/279286/google-play-android-app-categories/
[2] [May 2022] https://www.businessofapps.com/data/app-stores/

the bottom of the screen using the cursor keys to catch as much dropping fruit with the bowl as possible within a given time limit.

Games are intentionally designed to be challenging for human players in order to keep them hooked and entertained. Unfortunately, precisely this aspect also makes game-like programs very difficult to test, and classical approaches to automated test generation will struggle to interact with games in a successful way. In Fig. 1a a test generator would need to successfully catch all apples for 30 seconds without dropping a single one in order to win the game and reach the winning state.

While winning the game is out of question for a traditional automated test generator, it may nevertheless succeed in producing tests for partial aspects of a game, such as catching an individual banana or apple for the game shown in Fig. 1a. Such tests typically consist of static, timed input sequences that are sent to the program under test to reproduce a certain program execution path. This causes a problem: Games are inherently randomized, and playing the same game twice in a row may lead to entirely different scenarios. It may be possible to instrument the test execution environment such that the underlying random number generator produces a deterministic sequence [31]. The Whisker [46] test framework for Scratch, for example, allows setting the random number generator to a fixed seed, such that the fruit in Fig. 1a always appears at the same location in the same program. However, even the slightest change to the program may affect the order in which these random numbers are consumed, thus potentially invalidating the test. For example, if the banana sprite is removed from the game in Fig. 1a, even though the code of the apple sprite is unchanged the position at which the apple spawns changes, as the order in which the random numbers are consumed is different, and as a result a test intended to catch the apple sprite would likely drop the apple. Even if by chance the test would succeed in catching the apple despite the randomness, the exact program state would likely be different. For example, an assertion on the exact position of the apple or bowl in Fig. 1a would fail, rendering the use of classical test assertions impossible.

To tackle the test generation and oracle problem for game-like programs, we propose the use of neural networks as *dynamic* tests. For instance, Fig. 1b shows a network structure adapted to reaching code related to winning the *FruitCatching* game (Fig. 1a). It uses aspects of the program state such as sprite positions as input features, and suggests one of the possible ways to interact with the program. The architecture of the network puts strong emphasis on the horizontal distance between the bowl and the apple which reflects that catching apples rather than bananas is necessary to win the game, and indeed the network is able to win the *FruitCatching* game by catching all apples for 30s. In contrast to conventional test suites consisting of static input sequences, such dynamic tests are capable of reacting to previously unseen program states, which is particularly useful for game-like programs that are heavily randomized. The same dynamic tests can also serve as test oracles in a regression testing scenario by comparing node activations of a supposedly correct program version against a modified one. For instance, Fig. 2 shows the activation distribution of node *H20* from Fig. 1b at time-step 50 when executed 100 times on a correct version of *FruitCatching*. Since the most influential input of node *H20* is the horizontal distance between the bowl and the apple, the network's

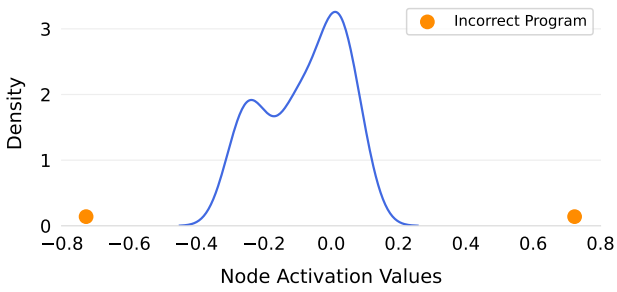


Figure 2: Activation distribution of node *H20* from Fig. 1b at time-step 50 after 100 *FruitCatching* execution. Orange dots correspond to activations from two faulty program versions.

effort to catch the fruit can clearly be observed from the high density around the node activation of zero. The two incorrect program versions depicted as orange dots disallow the player to move to the left or right, respectively, resulting in suspicious activation values that will be reported as invalid program behavior.

We introduce Neatest, an approach based on *neuroevolution* [12] to automatically generate artificial neural networks that are able to test games reliably, such as the network shown in Fig. 1b. Neatest targets games created in Scratch [32], a popular block-based programming environment with almost 90 million registered users and even more shared projects at the time of this writing[3] and is implemented on top of the Whisker [46] testing framework for Scratch. Given a Scratch program, Neatest iteratively targets statements based on the control dependence graph of the program, and evolves networks (i.e., dynamic tests) to reach those statements guided by an adapted reachability-related fitness function. Notably, by including statements related to the winning state of a game, the networks implicitly learn to meaningfully play and eventually win those games without requiring any domain-specific knowledge. Using the dynamic tests it produces, Neatest introduces a novel test oracle approach based on *Surprise Adequacy* [24] to detect incorrect program behavior by analyzing the networks' node activations during the execution of a program under test.

In detail, the contributions of this paper are as follows:

- We propose Neatest, a novel test generation approach that uses the *NeuroEvolution of Augmenting Topologies* (NEAT) [49] technique to evolve artificial neural networks capable of testing game-like programs reliably.
- Our testing tool implements this approach in the context of the Scratch educational programming environment and the Whisker test generator for Scratch.
- We empirically demonstrate that the proposed framework is capable of producing tests for 25 Scratch games that are robust against randomized program behavior.
- We empirically show through mutation analysis on 25 Scratch games that the behavior of neural networks can serve as a test oracle and detect erroneous program behavior.

[3] [May 2022] https://scratch.mit.edu/statistics

2 BACKGROUND

In this paper we combine the ideas of search-based software testing with neuroevolution, considering the application domain of games implemented in the Scratch programming language.

2.1 Search-based Software Testing

Search-based software testing (SBST) describes the process of applying meta-heuristic search algorithms, such as evolutionary algorithms, to the task of generating program inputs. The search algorithm is guided by a fitness function, which typically estimates how close a test is towards reaching an individual coverage objective, or achieving 100% code coverage. The most common fitness function derives this estimate as a combination of the distance between the execution path followed by an execution of the test and the target statement in the control dependence graph (*approach level* [54]) and the distance towards evaluating the last missed control dependency to the correct outcome (*branch distance* [28]). The approach is applicable to different testing problems by varying the representation used by the search algorithm. For example, SBST has been used to evolve parameters for function calls [29], sequences of method calls [4, 13, 50], sequences of GUI interactions [16, 34], or calls to REST services [3]. The search only derives test inputs to exercise a program under test, and test oracles are typically added separately as *regression assertions*, which capture and assert the behavior observed on the program from which the tests are derived [14, 57]. Since the tests derived this way cannot adapt to different program behavior, we call them *static* tests.

2.2 Neuroevolution

Artificial neural networks (ANNs) are computational models composed of several interconnected processing units, so-called *neurons*. An ANN can be interpreted as a directed graph where each node corresponds to a *neuron model*, which usually implements a nonlinear static *transfer function* [9]. The edges of the graph, which connect nodes with each other, are represented by weights that determine the connection strength between two neurons. Altogether, the number and type of neurons and their interconnections form an artificial neural network's *topology* or *architecture*.

Before an ANN can solve specific problems, it has to be trained in the given problem domain. During training, *learning algorithms* are used to find suitable network parameters, such as the weights of the connections within a network. Learning algorithms commonly apply *backpropagation* [41], which updates the weights by calculating the gradient for each connection individually. However, backpropagation on its own is only capable of tuning the weights of a neural network. Developing an appropriate network architecture remains a tedious task for the developer. A promising approach to eliminate these time-consuming tasks is to apply evolutionary algorithms that can optimize both the weights of networks and their architecture.

Instead of using a conventional learning algorithm, *neuroevolution* encodes ANNs in genomes and gathers them in a *population*. In each iteration, a subset of the current population is selected and evolved using *mutation* and *mating* procedures inspired by the principles of Darwinian evolution. Then, each genome's performance is evaluated using a *fitness function*, which reflects how close a given genome is to solving the problem at hand. Finally, following the paradigm of survival of the fittest, a new generation of the population is formed by selecting the fittest genomes of the population. By exploring the search space through many generations of mutating and mating these genomes, it is expected that eventually a neural network with optimal topology and weights for the given problem will be found. With this learning approach the weights, the number of neuron layers, and the interconnections between them, can be genetically encoded and co-evolved at the same time.

The popular *NeuroEvolution of Augmenting Topologies* [49] (NEAT) algorithm solves several challenges of *Topology and Weight Evolving Artificial Neural Networks*, such as the *competing conventions* problem [39]. NEAT solves this problem by introducing innovation numbers that enable the comparison of genes based on their historical origin. Another crucial aspect of NEAT is the use of *speciation* [15] to protect innovative networks. This is necessary since innovative topologies tend to initially decrease the fitness of a network, leading to architectural innovations having a hard time surviving even a single generation. Speciation ensures that neural networks with similar structures have to primarily compete against other networks in the same species, which gives them time to optimize their weights. Two networks are assigned to the same species if the number of mismatching genes and the average connection weight difference falls below a threshold, which is adjusted in each generation of the algorithm based on the number of species that are present in the current generation: if there are fewer species than desired, the threshold is reduced (e.g., by 0.3) and vice versa.

NEAT's mutation operators can be divided into two groups: *structural mutations* and *non-structural mutations*. The former modify a parent network's topological structure, while the latter operate on the attributes of connection genes. The crossover operator randomly selects two parents and aligns their connection genes using the assigned innovation numbers. Then, both connection gene lists are sequentially traversed, and certain rules are applied to each gene pair: Similar genes are either inherited randomly from one parent or combined by averaging the connection gene weights of both parents. On the other hand, genes with differing innovation numbers are always inherited from the more fit parent.

2.3 Testing Scratch Programs

The block-based programming language Scratch [32] provides predefined command blocks encoding regular programming language statements as well as domain-specific programming instructions that make it easy to create games. Scratch programs are created by visually arranging these command blocks according to the intended program behavior. Scratch programs consist of a *stage* and a collection of *sprites*. The former represents the application window and displays a background image chosen by the user. Sprites, on the other hand, are rendered as images on top of the stage and can be manipulated by the user. Both the stage and the sprites contain *scripts* formed by combining command blocks to define the functional behavior of the program. Besides control-flow structures or variables, the blocks represent high-level commands to manipulate sprites. Scratch programs are controlled via user inputs such as keypresses, text, or mouse movement. To engage learners, the Scratch programs they create are usually interactive games.

Testing and dynamic analyses are important for Scratch programs: Even though the Scratch blocks ensure syntactic correctness, functional errors are still feasible, raising the need for tests to validate the desired behavior of programs. In particular, tests are a prerequisite for effective feedback [5, 17, 22, 58] and hints [23] to learners, as well as for automated assessment [1, 19]. Whisker [46] is a framework that automates testing of Scratch programs by sending inputs to Scratch programs and comparing the system's observed behavior against a predefined specification.

In order to discover as many system states as possible and thereby enhance the chances of finding violated properties, it is crucial to have a widespread set of test inputs. All Scratch inputs used to stimulate the given program under test are assembled in the so-called *test harness*. Whisker offers two types of test harnesses: a manually written test harness and an automatically generated test harness using a random test generation approach.

However, random test generation is often not powerful enough to cover advanced program states, especially in game-like projects. In order to increase the effectiveness of automatic test generation, we realize a novel test generation approach in the presented framework by extending the Whisker testing tool. The proposed test generator uses neuroevolution to produce neural networks that are capable of extensively exercising game-like programs.

3 TESTING WITH NEUROEVOLUTION

Traditional tests of static input sequences cannot react to deviating program behavior properly and therefore fail to test randomized programs reliably. As a robust alternative to static test suites, we introduce Neatest, an extension of the Whisker test generation framework [46] that is based on the NEAT [49] algorithm. Neatest generates dynamic test suites consisting of neural networks that are able to withstand and check randomized program behavior.

Algorithm 1 shows the test generation approach of Neatest, which starts in line 5 with the selection of a target statement s_t based on the CDG and a list of yet uncovered program statements (Section 3.1). Next, a new population of networks is created depending on the search progress: If we have not covered any statements yet, the first population consists of networks where each sprite's feature group (Section 3.2.1) is fully connected to a single hidden node which is then connected to each output node. Otherwise, a new population is produced by cloning and mutating networks that proved to be viable solutions for previously targeted statements. By selecting prior solutions, we focus on networks capable of reaching certain program states since those may be prerequisites for reaching more advanced states. Furthermore, to avoid unnecessary complex network structures, at least 10% of every new population consists of freshly generated networks. We require hidden layer neurons to determine the correctness of a program (Section 3.4).

The following loop over all networks in line 9 starts with the *play loop* in which a network simulates a human player by repeatedly sending input events to a given Scratch program (Section 3.2.2). The inputs for the networks are extracted from the current state of the program, while the possible outputs are determined by the actions a player can take in the game (Section 3.2.1).

After a network has finished its playthrough, we evaluate within lines 11 to 16 how reliable the network can cover the current target

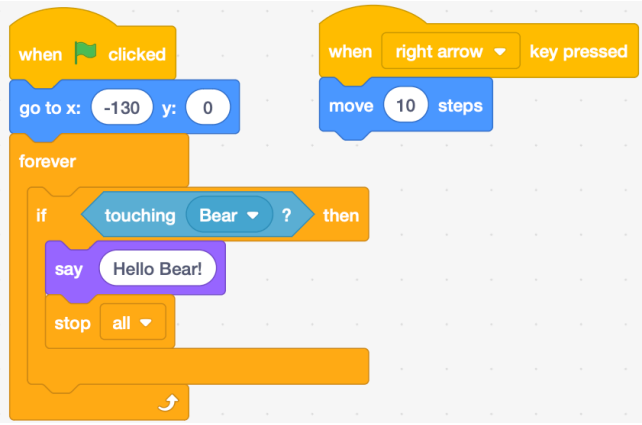
Algorithm 1: Neatest

```
   input  : control dependence graph CDG
   input  : list of uncovered program statements S
   input  : desired robustness count r_d
   output : dynamic test suite D
 1 function Neatest (CDG, S, r_d):
 2     requireNextStatement ← true;
 3     while stopping condition not reached do
 4         if requireNextStatement then
 5             s_t ← selectTargetStatement(CDG, S);
 6             P ← generatePopulation(D);
 7             requireNextStatement ← false;
 8         end
 9         foreach network n ∈ P do
10             initiatePlayLoop(n);
11             f ← calculateFitness(n);
12             if f == 1 then    // Statement was covered
13                 r_c ← validateCoverage(n, h);
14                 f ← f + r_c
15             end
16             network.fitness ← f;
17             if r_c ≥ r_d then
18                 D ← D + n;
19                 S_u ← S_u − s_t;
20                 requireNextStatement ← true
21             end
22         end
23         if !requireNextStatement then
24             P ← NEAT(P);
25         end
26     end
27     return D
28 end
```

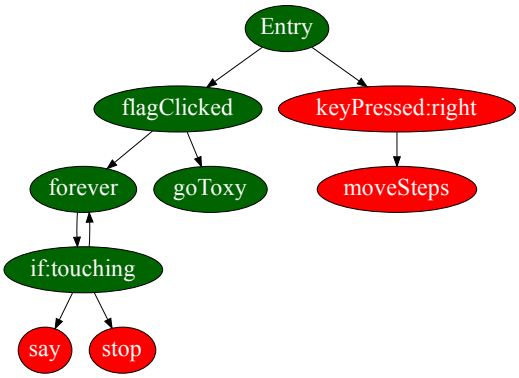
statement s_t (Section 3.3). If no network manages to pass the robustness check in line 17, the algorithm executes line 24 and evolves a new generation of networks using the NEAT algorithm Section 2.2. This new generation is then the starting point for the next iteration of the algorithm. Once a network has been optimized to cover s_t reliably for a user-defined number of times r_d, the network is added as a test to the dynamic test suite, and the next target statement is selected (lines 18-20). This process is repeated until the search budget is exhausted or all statements have been covered, and the dynamic test suite is returned to the user. The dynamic tests serve not only to exercise the programs, but also as test oracles for validating the functionality in a regression testing scenario (Section 3.4).

3.1 Explorative Target Statement Selection

Neatest iteratively selects increasingly more difficult target statements to guide the creation of incrementally more complex networks. In order to achieve this, targets are selected from the test program's control dependence graph (CDG) [7], an acyclic directed graph that shows which program statements depend on which



(a) Two Scratch scripts that implement the movement of the hosting sprite and a check if the hosting sprite touches a bear sprite.



(b) Generated CDG graph for the depicted script with green and red nodes representing covered and non-covered Scratch statements.

Figure 3: Example of a Scratch script and the corresponding CDG graph.

control locations. Since Scratch programs typically contain many small scripts that communicate via events and messages, we create an interprocedural CDG that covers the control flow of the entire program. For example, Fig. 3b shows the generated CDG that combines the two Scratch scripts depicted in Fig. 3a into one graph.

Target statements s_t are selected as children of already covered program statements, as advanced states can only be reached if previous control locations can be passed appropriately. If no statements have been covered yet, the root node of the CDG (*Entry*) is selected. In case more than one viable child exists, Neatest favors program statements that have accidentally been reached while focusing on other program states and did not yet pass the robustness check. Considering the example of Fig. 3b and assuming that the search was able to trigger the right key-press once, the next target statement would be set to the *keyPressed:right* statement. Otherwise, if none of the three statements that have a direct covered parent (*say*, *stop*, *keyPressed:right*) had been reached once, s_t would be randomly set to one of the three statements. To avoid getting stuck trying to cover hard-to-reach statements while other, perhaps easier-to-reach states are still present, Neatest changes its currently selected target after a user-defined number of consecutive generations in which no improvement of fitness values was observable.

3.2 Play Loop

Within the Play Loop, the networks govern the execution of a program by repeatedly sending input events until the execution halts, a pre-defined timeout has been reached, or the targeted statement s_t was covered. To make a decision on which user event to select at a given moment, we provide the networks with input features extracted from the current program state. The selected event is then sent to the Scratch program to advance the program execution.

3.2.1 Feature Extraction. Input features are fed to the input neurons of networks and play a crucial role during a playthrough by serving as the neural networks' eyes. The features are collected by filtering attributes of visible sprites by their relevance to the given Scratch program, as determined by analyzing the source code:

- **Position** of a sprite on the Scratch canvas is always added and defined through its x- and y-coordinates in the range $x \in [-240, 240]$ and $y \in [-180, 180]$.
- **Size** of a sprite is always added. The size bounds depend on the currently selected *costume* (i.e., visual representation).
- **Heading direction** iff the rotation style is set to *all around*. Defined through an angle α in the range $\alpha \in [-180, 180]$.
- **Selected costume** iff the sprite contains code to change its costume. Defined as $i \in \mathbb{N}$, indexing the current costume.
- **Distance to sensed sprite** iff the sprite contains code that checks for touching another sprite. Defined through the x and y distance to the sensed sprite in the range $[-600, 600]$.
- **Distance to sensed color** iff the sprite contains code that checks for touching a given color. Colors are measured by four rangefinders, which measure the distance on the canvas to the sensed color in the directions $[0, 90, 180, -90]$, with 0 representing the current *heading direction* of the sprite. Admissible values are again restricted to $[-600, 600]$.

In addition to relevant sprite attributes, we also add all program variables, for example, the current score of a game, to the set of input features. For each extracted feature, we add an input node to the networks and group them by the sprite they belong to. Since we only include relevant features for which corresponding code exists, the input features may not only differ between games but also within the same game, for example whenever a sprite creates a clone that has its own attributes. However, neuroevolution allows the framework to adapt to these input feature changes dynamically by adding input nodes to the networks whenever a new behavioral pattern is encountered. Finally, to ensure all extracted input features have the same weight, they are normalized to the range $[-1, 1]$.

Output features of a network resemble the set of user events a human player can send to the Scratch application at a given point in time. The framework simulates human inputs using the following Scratch events:

- **KeyPress**: Presses a specific key on the keyboard for a parameterized amount of time.
- **ClickSprite**: Clicks on a specific sprite.

- **ClickStage**: Clicks on the stage.
- **TypeText**: Types text using the keyboard.
- **MouseMove**: Moves the mouse to a parameterized position.
- **MouseMoveTo**: Moves the mouse to a position inferred by static analysis of the program code.
- **MouseDown**: Presses the left mouse button for a parameterized amount of time.
- **Sound**: Sends simulated sound to the program.
- **Wait**: Waits for a parameterized amount of time.

Neatest analyzes the source code of all actively running scripts for a given program state in order to select the subset of event types for which active event handlers exist. Then, after minimizing the set of event types, we create one output node for each remaining event. In case there are parameterized events, such as *MouseMove* events requiring coordinates the mouse pointer should be moved to, the classification problem is further extended to a combination of a classification and regression problem. These parameters are produced via a multi-headed network model [21] in which regression nodes are mapped to parameterized events. Similar to input features, the set of feasible output features and the number of required output nodes may change throughout the course of a game.

3.2.2 Network Activation & Event Selection. The problem of deciding which action to take at the current state of a game is cast to a multi-class single-label classification problem. Classification steps start by loading the input features of the current program state into the corresponding input nodes. Then, the neural network is activated in order to produce an output based on the fed input features. During network activation, values residing in the input nodes flow through the network in several timesteps. In every timestep, each node adds up all the activation values from the previous timestep and calculates the result of its own activation using the *tanh* function [20] on the summed-up values.

To not perturb values gathered during *feature extraction*, all input nodes send the normalized input features directly into the network without activation functions. In a single timestep, activation only flows from one neuron to the next, thus several network activations are required for the input nodes' signals to reach the output nodes.

After the input signal has reached the output nodes, a probability distribution is calculated over the set of available events using the traditional *softmax function* [10]. The classification task is extended into a regression problem whenever a network selects a parameterized event. Parameters for the selected event are gathered by querying the responsible nodes in the network's regression head.

Selected events are passed on to the Scratch VM, where they may trigger the activation of previously inactive scripts. Upon receiving an input event, the Scratch VM executes a Scratch step by sequentially executing blocks of currently active scripts until specific halting blocks or the end of the script is reached. The Scratch step finishes by updating the state of the VM to match the effects of the executed blocks. The playing loop then starts anew by extracting the input and output features of the updated VM.

3.3 Network Evaluation

Given a selected target statement s_t, the neuroevolution is guided by a fitness function f which estimates how close a network is to reaching s_t, and once reached, how reliably it is executed. The distance is calculated as a linear combination of the following:

- **Approach level (AL)**: Integer defining the number of control dependencies between the immediate control dependency of the target and the closest covered statement.
- **Branch distance (BD)**: Indicates how close a network is to satisfying the first missed control dependency. Zero if the control location has been passed towards the target statement. Scratch often allows for a precise definition of the branch distance. For example, in Fig. 3b, the branch distance for the *if:touching* statement is defined as the normalized distance between the corresponding sprites.
- **Control flow distance (CFD)**: As execution may stop in the middle of a sequence of blocks (e.g., because of timed statements), the CFG counts the number of blocks between the targeted statement and the nearest covered statement.

To avoid deceiving fitness landscapes, we normalize the branch distance and the control flow distance in the range $[0, 1]$ using the normalization function $\alpha(x) = x/(1+x)$ [2], and multiply the approach level with a factor of two:

$$f = 2 \times AL + \alpha(BD) + \alpha(CFD)$$

In case $f > 0$, the target statement was not covered and we set the fitness to $f = 1/f$ to create a maximization objective with values in the range $(0, 1)$. Otherwise ($f = 0$), the network managed to cover the target once and we evaluate whether it can also cover the same target in randomized program scenarios. For this purpose, we generate a pre-defined number (r_d) of random seeds and count for how many seeded executions (r_c) the network is able to reach the target statement again. The resulting fitness value is $f = 1 + r_c$.

The fitness function ensures that networks that are closer or more robust in covering a specific statement are prioritized during the evolution. If a network eventually passes the robustness check $r_c \geq r_d$, the network is added to the dynamic test suite as test for target statement s_t and the next target is selected by querying the CDG as described in Section 3.1. Since covering one statement usually leads to reaching other related statements as well, we simultaneously evaluate whether the network may also satisfy the robustness check for other previously uncovered program statements.

3.4 Neural Networks as Test Oracles

In order to determine whether a test subject is erroneous, we observe how surprised a network behaves when confronted with the program under test and regard highly surprising observations as suspicious. As a metric, we use the *Likelihood-based Surprise Adequacy* (LSA) [24], which uses kernel density estimation [53] to determine how surprising the activations of selected neurons are when the network is presented with novel inputs. In the application scenario of regression testing, node activations originating from a supposedly correct program version serve as the ground truth and will be compared to activations occurring during the execution of the test subject. To incorporate many randomized program scenarios, we collect the ground truth activations by executing the networks on the sample program using a series of different random seeds. Defective programs are then detected if the LSA value surpasses a pre-defined threshold, or for the special case that all ground

truth traces have a constant value, if the surprise value is above zero since in this scenario already tiny deviations are suspicious.

Neatest only evaluates activation values occurring within hidden nodes because these nodes strike a nice balance between being close to the raw inputs that represent the program state while already offering an abstraction layer by incorporating a network's behavior. This allows differentiating between suspicious and novel program behavior, an essential characteristic for testing randomized programs that is missing in traditional static test assertions. For example, in the FruitCatching game (Fig. 1a), the apple and bananas may spawn at entirely new x-coordinates, but the program state is still correct as long as they spawn at a certain height. Successfully trained networks can differentiate between novel and suspicious states because they are optimized to catch falling fruit by minimizing the distance between the bowl and the fruit, as shown by the zero-centered activation distribution in Fig. 2. Since both defective programs, represented as orange dots, are far off the distribution center, they are rather surprising, with LSA values of 50 and 102. Therefore, given an appropriately set threshold below these values, Neatest correctly classifies these program versions as faulty.

In contrast to the original definition of LSA [24], we do not compare groups of nodes that may reside in the same layer but compare the activation values of hidden nodes directly. Furthermore, we classify activation values by the Scratch step in which they occurred. This precise approach of comparing node activations offers two major advantages: First, we can detect tiny but nonetheless suspicious program deviations that may only occur within a single node by calculating the LSA value for each node separately. Second, if a suspicious activation within a node was detected, we can identify the exact point in time during the execution and the rough reason for the suspicious observation by following the suspicious node's incoming connections back to the input nodes.

Changes to the structure of a network, which may occur throughout an execution, can also provide strong evidence of suspicious program behavior. For instance, if the actions a network can take in a game are different between the sample program and the test subject, the networks automatically add new output nodes for the novel user events, as explained in Section 3.2. Therefore, we can deduce from a change in a network's structure that the test subject exposes diverging and maybe even erroneous program behavior.

4 EVALUATION

In order to evaluate the effectiveness of the proposed testing framework we aim to answer the following three research questions:

- **RQ1**: Can Neatest optimize networks to exercise Scratch games reliably?
- **RQ2**: Are dynamic test suites robust against randomized program behavior?
- **RQ3**: Can neural networks serve as test oracles?

Neatest, the experiment dataset, parameter configurations, raw results of the experiments, and scripts for reproducing the experiments are publicly available on GitHub[4].

[4]https://github.com/FeldiPat/ASE22-Neatest-Artifact

4.1 Dataset

To establish a diverse dataset for the evaluation, we collected 187 programs that represent the educational application domain of Scratch from an introductory Scratch book[5], tutorial websites (Code Club[6], Linz Coder Dojo[7], learnlearn[8], junilearning[9]), and prior work [46]. Three selection criteria were used to determine if a project occurring in one of these sources should be considered for evaluation: First, a given project has to resemble a game that processes user input supported by the Whisker framework and challenges the player to achieve a specific goal while enforcing a set of rules. Second, the game must include some form of randomized behavior. Finally, we excluded games that do not have a challenging winning state and can thus be easily covered by sending arbitrary inputs to the program. Applying all three selection criteria to the mentioned sources and skipping programs that are similar to already collected ones in terms of player interaction and the overall goal of a game resulted in a total of 25 programs (including the *FruitCatching* game shown in Fig. 1a); statistics of these games are shown in Table 1. Overall, each of the 25 games offers unique challenges and ways of interacting with the program.

4.2 Methodology

All experiments were conducted on a dedicated computing cluster containing nine nodes, with each cluster node representing one AMD EPYC 7443P CPU running on 2.85 GHz. The Whisker testing framework[10] allows users to accelerate test executions using a customizable acceleration factor. In order to speed up the experiments, all conducted experiments used an acceleration factor of ten.

RQ1: We evaluate whether Neatest can train neural networks to cover program states *reliably* by comparing the proposed approach against a random test generation baseline. While Neatest produces dynamic tests in the form of neural networks, the random testing approach randomly selects input events from a subset of processable events and saves the chosen events as static test cases. Neatest maintains a population of 300 networks, and both methods are allowed to generate tests until a search budget of 23 hours has been exhausted.

The combination of 300 networks per population and the search budget of 23 hours allows the framework to conduct a reasonable amount of evolutionary operations in order to sufficiently explore the search space of the most challenging test subjects in the dataset. For smaller and easier programs, 23 hours and a population of 300 networks are far too much, and satisfying results may already be found with a population size of 100 networks and a search budget of only one hour. In general, the easier a winning state can be reached and the fewer control dependencies a Scratch program has, the fewer resources are required by Neatest.

We set the number of non-improving generations after which a target is switched to five because this value provides the evolution enough time to explore the search space while also not wasting too

[5][May 2022] https://nostarch.com/catalog/scratch
[6][May 2022] https://projects.raspberrypi.org/en/codeclub
[7][May 2022] https://coderdojo-linz.github.io/uebungsanleitungen/programmieren/scratch/
[8][May 2022] https://learnlearn.uk/scratch/
[9][May 2022] https://junilearning.com/blog/coding-projects/
[10][May 2022] https://github.com/se2p/whisker

Table 1: Evaluation games.

Project	# Sprites	# Scripts	# Statements	Project	# Sprites	# Scripts	# Statements
BirdShooter	4	10	69	FlappyParrot	2	7	37
BrainGame	3	18	76	Frogger	8	22	105
CatchTheDots	4	10	82	FruitCatching	3	4	55
CatchTheGifts	3	7	68	HackAttack	6	19	93
CatchingApples	2	3	25	LineUp	2	4	50
CityDefender	10	12	97	OceanCleanup	11	22	156
DessertRally	10	27	212	Pong	2	2	15
DieZauberlehrlinge	4	14	87	RioShootout	8	26	125
Dodgeball	4	10	78	Snake	3	14	60
Dragons	6	33	381	SnowballFight	3	6	39
EndlessRunner	8	29	163	SpaceOdyssey	4	13	116
FallingStars	4	4	91	WhackAMole	10	49	391
FinalFight	13	48	286	Mean	5.5	16.6	118.3

much time on difficult program states. To distribute the 300 networks of a population across the species such that each species can meaningfully evolve the weights of its networks, we set the targeted species number to ten. For both test generators, the maximum time for a single playthrough was set to ten seconds, which translates to a play duration of up to 100 seconds due to the used acceleration factor of ten. The remaining neuroevolution hyperparameters are defined according to the values in the original NEAT approach [49].

As a metric to compare Neatest to a random test generator, we consider coverage values across all experiment repetitions with the *Vargha & Delaney* ($\hat{A}_{12}$) effect size [52], and the *Mann-Whitney-U-test* [33] with $\alpha = 0.05$ to determine statistical significance. To determine whether Neatest can train networks to reliably reach challenging program states, we also report the number of reached winning states per experiment repetition. For each game we manually determined the code representing a winning state, and consider it reached if the corresponding code is reliably covered. Since we are interested in tests that are robust against randomized program behavior, we treat a statement only as covered if a given test passes the robustness check (Section 3.3), for ten differently seeded program executions. To account for randomness, we repeat each experiment 30 times with different random seeds.

RQ2. In order to explore if the generated dynamic test suites are truly robust against diverging program behavior, we randomly extract for each program ten dynamic and static test suites that were produced during RQ1. For the purpose of eliminating differences originating from the test generation process, both suite types are extracted from the Neatest approach. The static test suites correspond to the input sequences successful networks produced when they entered the robustness check. All extracted test suites are then executed on the corresponding programs using seeds different from those during the test generation phase.

We evaluate the robustness of both suite types by reporting the difference in coverage between the test generation and test execution phase and investigate the effectiveness of dynamic test suites against static suites by comparing the achieved coverage and $\hat{A}_{12}$ values. Similar to RQ1, we use the *Mann-Whitney-U-test* to report statistically significant results. To match the application scenario of test suites, we do not use robustness checks and treat a statement as covered if it has been reached at least once. Within RQ2, we compensate for random influences by applying every extracted test suite to ten new random seeds, which results in 100 test runs for each suite type on every dataset project.

Table 2: Mutation operators.

Operator	Description
Key Replacement Mutation (KRM)	Replaces a block's *key* listener.
Single Block Deletion (SBD)	Removes a single block.
Script Deletion Mutation (SDM)	Deletes all blocks of a given script.
Arithmetic Operator Replacement (AOR)	Replaces an arithmetic operator.
Logical Operator Replacement (LOR)	Replaces a logical operator.
Relational Operator Replacement (ROR)	Replaces a relational operator.
Negate Conditional Mutation (NCM)	Negates boolean blocks.
Variable Replacement Mutation (VRM)	Replaces a variable.

RQ3. To answer the question if networks can serve as test oracles, we extended Whisker with a mutation analysis feature implementing the eight different mutation operators shown in Table 2, and applied mutation analysis to all 25 projects. These mutation operators were selected based on the traditional set of sufficient mutation operators [37]. After generating mutant programs, we execute the dynamic test suites of RQ2 on the respective mutants and measure how surprised the networks are by the modified programs. The ground truth activation traces required for calculating the LSA are generated by executing each suite 100 times on the unmodified program using randomly generated seeds. A program is marked as mutant if the LSA value of a single node activation surpasses an experimentally defined threshold of 30. Besides calculating the LSA value, mutants are also killed if the network structure changes while being executed on the mutant.

We answer RQ3 by reporting the distribution of killed mutants across the eight applied operators. Furthermore, we verify that the networks do not simply mark every program as a mutant by calculating the false-positive rate on unmodified programs. Similar to RQ2, we account for randomness within the mutant generation and network execution process by repeating the experiment for all extracted suites ten times using varying seeds. To avoid an explosion of program variations, each experiment repetition was restricted to 50 randomly selected mutants per operator.

4.3 Threats to Validity

External Validity: The dataset of 25 projects covers a broad range of game types and program complexity. Even though high priority was given to establish a dataset of games with many different objectives and ways to interact with them, we cannot guarantee that the results generalize well to other Scratch applications or games developed in different programming languages.

Internal Validity: Randomized factors within the experiments pose another threat to validity since repeated executions of the same experiments will, by definition, lead to slightly different outcomes. However, we expect 30 experiment repetitions for RQ1 and 100 test results for each project in RQ2 and RQ3 to suffice for assuming robust experiment results. The threshold that determines at which surprise value a program is marked as a mutant was determined experimentally and may not work well for other programs.

Construct Validity: For evaluating the effectiveness of Neatest in generating robust test suites, we used block coverage which is

Table 3: Mean coverage, coverage effect size (A12), and number of reached winning states (Wins) of the random test generator (R) and NEATEST (N) during the test generation process. Coverage values refer to *reliable* coverage achieved by passing the robustness check. Boldface indicates strong statistical significance with $p < 0.05$.

	Coverage				Wins	
Project	R	N	A12	p	R	N
BirdShooter	98.60	100	**0.98**	< 0.01	1	30
BrainGame	89.34	100	**0.95**	< 0.01	3	30
CatchTheDots	97.64	100	**0.82**	< 0.01	15	30
CatchTheGifts	89.80	100	**0.67**	< 0.01	20	30
CatchingApples	98.53	100	**0.68**	< 0.01	19	30
CityDefender	74.30	70.62	**0.28**	< 0.01	0	3
DessertRally	93.58	93.90	**0.58**	0.02	0	0
DieZauberlehrlinge	75.50	92.17	**1.00**	< 0.01	0	28
Dodgeball	94.87	96.15	**1.00**	< 0.01	0	0
Dragons	88.30	89.01	0.61	0.17	11	12
EndlessRunner	73.95	90.52	**0.90**	< 0.01	5	11
FallingStars	97.80	100	**1.00**	< 0.01	0	30
FinalFight	97.50	99.65	**1.00**	< 0.01	0	30
FlappyParrot	92.61	100	**1.00**	< 0.01	4	30
Frogger	88.69	95.74	**1.00**	< 0.01	0	15
FruitCatching	69.09	100	**1.00**	< 0.01	0	30
HackAttack	86.02	97.38	**1.00**	< 0.01	0	20
LineUp	96.00	100	**1.00**	< 0.01	0	30
OceanCleanup	73.65	78.61	**0.69**	0.01	14	27
Pong	94.22	100	**0.72**	< 0.01	17	30
RioShootout	94.40	94.40	0.50	1.00	30	30
Snake	95.00	95.00	0.50	1.00	0	0
SnowballFight	94.87	96.32	**0.78**	< 0.01	0	17
SpaceOdyssey	92.70	95.17	**0.67**	< 0.01	0	9
WhackAMole	79.64	80.44	0.54	0.62	0	19
Mean	89.07	94.60	0.80	-	5	20

similar to statement coverage in text-based programming languages. However, block coverage is not always a good indicator because many SCRATCH programs are often already covered by sending arbitrary inputs to the program. For exactly this reason, we focused on selecting games that have a winning state and therefore pose a true challenge to test generators. We answer whether networks can serve as test oracles by reporting the ratio of killed mutants. However, not every mutant must necessarily exhibit faulty behavior and may be considered a valid program alternative by a human.

4.4 RQ1: Can NEATEST Optimize Networks to Exercise SCRATCH Games Reliably?

Our first research question evaluates whether neural networks can be trained to exercise randomized SCRATCH programs reliably. Table 3 summarizes the experiment results and shows that the random tester and NEATEST reach generally high coverage values of 89.07% and 94.60%. This is because the programs tend to be small in terms of their number of statements. However, SCRATCH programs consist of many domain-specific blocks specifically designed for game-behavior, such that it is possible to implement fully functional game-behavior with very few blocks, which in other programming languages might require hundreds of lines of code. Blindly sending inputs, as random testing does, leads to the execution of a large share of these blocks without, however, actually playing the game. This can be seen when considering specifically how often an actual winning state was reached. For example, in *FallingStars* both approaches obtain almost the same average coverage values (97.80% and 100%), although only NEATEST is able to cover statements related to the challenging winning state. Table 3 shows that NEATEST covers, on average, 20 winning states, while the random test generator has trouble meaningfully exercising the games and reaches only 5 winning states per game. For three games, NEATEST fails to reach the winning state because either the allotted play duration is too short (*DessertRally*), the game is too complex to evolve sufficiently optimized networks within 23 hours (*Snake*), or more sophisticated evolution strategies are required to solve advanced search-problems, such as the maze problem [42] (*Dodgeball*).

CityDefender is the only game where NEATEST reaches fewer coverage than the random tester. The statements responsible for these results are linked to relatively easy program states that the random tester can easily reach through randomly firing inputs at the program under test. While NEATEST also sometimes performs the required actions, the networks must be specifically trained to send those inputs regardless of which randomized program state they are confronted with. However, the relatively long duration of a single playthrough in this game sometimes leads to experiment repetitions in which the respective statements never get targeted. The three reached winning states in *CityDefender* that correspond to experiment repetitions in which a winning statement was actually targeted demonstrates that using a more sophisticated statement selection technique would allow NEATEST to train robust networks that reliably perform the required actions.

Having execution times of 23 hours may be recognized as a limiting factor of the proposed approach. However, we selected this relatively high search budget in order to sufficiently explore those programs in the dataset that have exceptionally long fitness evaluation durations. Fig. 4 illustrates the achieved *reliable* coverage over time for both approaches across all evaluation subjects. The results show that NEATEST is more effective than random testing by covering more program statements at any stage of the search process. Furthermore, the proposed approach covers 90% of all program statements in under 3 hours (156 min), while the random testing is not able to reach the same level of coverage at all. Moreover, NEATEST fully covers 5/25 projects in all 30 repetitions within the first hour, indicating that the required search time is highly dependent on the complexity of the game. In order to reduce the required search time for the more challenging programs, advanced search strategies, such as combining NEATEST with backpropagation [6], have to be investigated in future work.

Summary RQ1: An average $\hat{A}_{12}$ value of 0.80 and 18 covered winning states per game indicates that NEATEST manages to train networks capable of mastering games without requiring any game-specific knowledge.

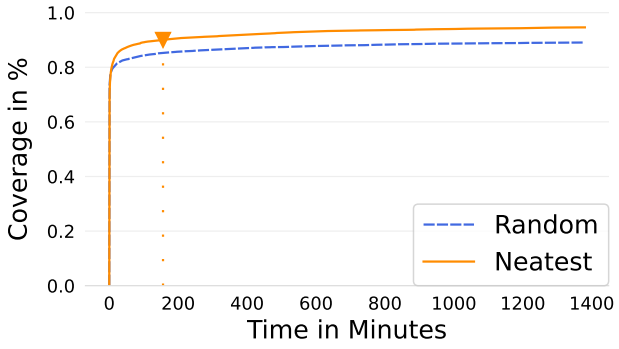


Figure 4: Coverage over time across all experiment repetitions and programs of the dataset with ▲ and ▼ representing achieved coverage values of 90% and 95%, respectively.

4.5 RQ2: Are Dynamic Test Suites Robust Against Randomized Program Behavior?

In order to evaluate the importance of training networks to *reliably* cover statements, RQ2 extracts static event sequences from the dynamic tests generated for RQ1, and compares them in terms of the coverage variation observed on executions with different random seeds. Each static test is the result of executing a dynamic test on the program under test once, and for the particular seed on which the static test is derived their coverage is by construction identical. Table 4 summarizes the effects of re-running the resulting tests for 100 different random seeds on each of the programs. The results show that dynamic tests are robust against randomized programs—coverage even increases slightly by 1.22% when compared to the test generation phase. In contrast, the static tests show a drop of 8.17% of coverage, which is substantial considering the generally high coverage observed for RQ1.

The slight increase in coverage for the dynamic test suites may seem surprising, but can be explained by the way coverage is measured: In contrast to RQ1, which measured how many statements were reliably reached by a test generator, here we count how many statements are covered on average over the 100 test executions. Thus, also statements that are only covered in some of the runs contribute to the overall coverage. The drop of 8.17% of the static tests demonstrates the need for robust test suites when testing randomized programs. This clear advantage of dynamic tests becomes even more apparent when looking at the $\hat{A}_{12}$ values, which indicate that for every program in the dataset, networks perform significantly better than static test sequences.

> **Summary RQ2**: Dynamic tests can adapt to unseen program behavior and exercise randomized programs reliably by achieving an average program coverage of 95.83%.

4.6 RQ3: Can Networks Serve as Test Oracles?

The randomized behavior of games makes it challenging to provide test oracles. The aim of this RQ is to investigate if the networks of the dynamic tests can also serve as test oracles by differentiating

Table 4: Mean coverage and effect size (A12) during the test execution phase of dynamic (D) and static (S) test suites. The column *Difference* shows coverage differences between the generation and execution phase. Values in boldface indicate strong statistical significance with $p < 0.05$.

	Coverage				Difference	
Project	S	D	A12	p	S	D
BirdShooter	94.76	99.59	**0.91**	< 0.01	-5.24	-0.41
BrainGame	23.53	95.76	**1.00**	< 0.01	-76.47	-4.24
CatchTheDots	99.40	100	**0.75**	< 0.01	-0.60	0.00
CatchTheGifts	99.47	100	**0.56**	< 0.01	-0.53	0.00
CatchingApples	96.17	100	**0.67**	< 0.01	-3.83	0.00
CityDefender	72.91	81.02	**0.87**	< 0.01	2.29	10.40
DessertRally	93.65	93.97	**0.62**	< 0.01	-0.25	0.07
DieZauberlehrlinge	78.66	94.92	**1.00**	< 0.01	-13.51	2.75
Dodgeball	95.17	95.83	**0.64**	< 0.01	-0.98	-0.32
Dragons	90.95	91.02	**0.53**	< 0.01	1.94	2.01
EndlessRunner	84.65	94.92	**0.97**	< 0.01	-5.87	4.40
FallingStars	98.57	99.69	**0.85**	< 0.01	-1.43	-0.31
FinalFight	96.07	99.69	**0.99**	< 0.01	-3.58	0.04
FlappyParrot	93.59	96.41	**0.79**	< 0.01	-6.41	-3.59
Frogger	93.95	95.72	**0.78**	< 0.01	-1.79	-0.02
FruitCatching	77.69	96.70	**0.93**	< 0.01	-22.31	-3.30
HackAttack	83.84	92.06	**0.75**	< 0.01	-13.54	-5.32
LineUp	71.03	99.46	**1.00**	< 0.01	-28.97	-0.54
OceanCleanup	92.38	98.51	**0.85**	< 0.01	13.77	19.90
Pong	89.93	98.00	**0.79**	< 0.01	-10.07	-2.00
RioShootout	85.10	96.67	**0.96**	< 0.01	-9.30	2.27
Snake	93.15	95.26	**0.79**	< 0.01	-1.85	0.26
Mean	86.43	95.83	0.82	-	-8.17	+1.22

randomized program scenarios from faulty behavior. For that purpose, we generated 243835 mutants using eight different mutation operators across all games in our dataset and checked whether the networks were able to identify those mutants. Table 5 demonstrates that the networks managed to achieve a mutation score of 70.76%, while maintaining a relatively low false-positive rate of 19.73%.

As Fig. 5 indicates, the average false-positive rate might be deceiving as there are two extreme outliers that have a strong influence on the mean, while the median is considerably lower with a value of only 10%. In those two programs, namely *Dragons* and *FinalFight*, the networks are confronted with many sprites having attributes that are distinct in almost every program execution. Therefore, the networks have to cope with activation traces that are entirely different from previously observed ones and thus seem suspicious. This problem could be solved by either increasing the number of recorded ground truth traces or through a more sophisticated abstraction layer by restricting the set of observed hidden nodes to nodes that reside in hidden layers that are further away from the input layer.

Neatest detects the least mutants for *CityDefender*, which is a direct consequence of the poorly trained networks, as discussed in Section 4.4. The lowest mutation kill rate can be observed in Fig. 5 for the *Single Block Deletion* mutation operator, because Scratch programs often contain statements that do not change the program behavior but instead offer purely visual effects. Depending on the scope of what is recognized as faulty behavior, these visual deviations could be detected by collecting more fine-grained input

Table 5: Generated (capped at 50 mutants per operator and game), killed and false-positive (FP) marked mutants.

Project	# Generated	Killed %	FP %	Project	# Generated	Killed %	FP %
BirdShooter	7200	96.68	22.00	FlappyParrot	3800	38.58	0.00
BrainGame	7800	40.40	4.00	Frogger	14600	92.97	2.00
CatchTheDots	9000	68.79	0.00	FruitCatching	6279	66.22	13.19
CatchTheGifts	8000	81.53	0.00	HackAttack	8200	40.44	3.00
CatchingApples	2500	76.92	1.00	LineUp	5600	98.61	10.00
CityDefender	8900	11.79	0.00	OceanCleanup	10600	63.17	44.00
DessertRally	14569	99.65	33.00	Pong	1300	34.00	10.00
DieZauberlehrlinge	12397	29.57	18.00	RioShootout	10900	54.43	16.00
Dodgeball	7821	84.22	26.26	Snake	7000	49.21	3.00
Dragons	23838	98.27	99.00	SnowballFight	4596	70.39	5.00
EndlessRunner	13199	46.21	5.00	SpaceOdyssey	7500	42.37	26.00
FallingStars	13162	87.41	13.13	WhackAMole	14424	78.31	42.00
FinalFight	20650	90.41	97.00	Mean	9753.40	70.76	19.73

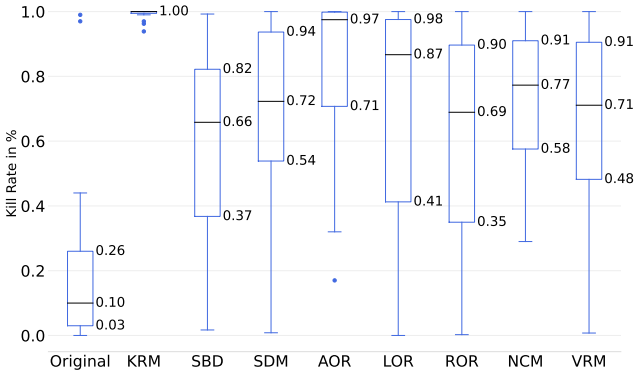


Figure 5: Mutant kill rates across all mutation operators.

features, such as the visual effects applied to the individual sprites. The highest mutation kill rate can be seen for the *Key Replacement* mutations (x%), in which an event handler's observed key gets replaced with a randomly selected one. In many of these cases, the mutant is detected due to the dynamic adaption of the networks to changes in the input and output space Section 3.2.1.

> **Summary RQ3**: A high mutation score of 70.53% and a low false-positive median of 10% show that networks can distinguish between novel and erroneous behavior and are therefore suitable test oracles for randomized programs.

5 RELATED WORK

Due to the increasing popularity of NEAT, many variations have been proposed [26, 40, 48, 55, 56] and applied to master games in many different gaming environments, such as Atari [18] and real-time based games [38, 47]. Since, to the best of our knowledge, no previous work exists that uses neuroevolution for block-based programming environments like Scratch, we applied the established variant of NEAT, and will evaluate improvements in future work.

More generally, reinforcement learning techniques have been shown to be capable of learning to play many different game genres such as strategy games [44, 45], arcade games [51], first-person shooter games [30, 35], and casual games [27]. With the goal of assisting developers in the level design process, Shin et al. [43] combine predefined game strategies of the popular *Match 3* mobile game genre with reinforcement learning to perform automated playtesting. We used a neuroevolution approach rather than alternative reinforcement learning techniques since NEAT allows easy adaptation to changes in the input space. Furthermore, prior reinforcement learning-based approaches reward an agent for achieving game-specific objectives. In contrast, Neatest explores the CDG and automatically derives objectives in order to cover specific target statements, thus implicitly learning to play the games.

The *Likelihood-based Surprise Adequacy* metric, which we use as a test oracle, has previously been used for prioritizing test inputs for image classification tasks [24]. Kim et al. [25] reduced the cost of calculating the *Surprise Adequacy* by replacing the kernel density estimation with the Mahalanobis distance [8]. In future research, we will explore other metrics that might be useful as test oracles, such as the certainty of a network in its predictions [11].

6 CONCLUSIONS

Games pose a considerable challenge for automated test generation because they usually contain states that require the test generator to master playing the given game. Furthermore, games tend to be heavily randomized to keep the player engaged, which prevents traditional static test suites from testing such programs reliably. Those randomized behaviors also make static test assertions infeasible since they can not differentiate between novel and erroneous program states. In an attempt to solve these challenges, this work combines the research areas of search-based software testing with neuroevolution to produce a novel type of test suite. Dynamic test suites consist of neural networks trained via neuroevolution to reach program states such as the winning state reliably, regardless of the encountered program scenarios. By observing the network structure and measuring how surprised the networks are by a given program state, our Neatest prototype can utilize the trained networks as test oracles to detect incorrect program behavior. Our experiments confirm that neuroevolution is a suitable technique to train networks that manage to master and test Scratch games.

The Neatest approach provides an initial foundation for combining automated testing with neuroevolution. The Scratch games we considered in our prototype are often similar to classic arcade games, a common target for neuroevolution research. In principle, Neatest is also applicable to other domains, such as Android or iOS games, but adaptation will require engineering effort for feature extraction and implementing the play loop. Integrating recent advances in neuroevolution, such as deep learning extensions of NEAT [36], will be of interest in order to generalize further beyond arcade games. Furthermore, it may be worth exploring if Neatest provides a way to overcome the problems caused by randomized program behavior outside the domain of games (i.e., flaky tests). Finally, even though our experiments demonstrate that the Surprise Adequacy metric is an interesting alternative to traditional test oracles, it originates from a highly active area of research, and further research with new and alternative metrics may further improve the detection rates for erroneous program behavior.

REFERENCES

[1] Kirsti M Ala-Mutka. 2005. A survey of automated assessment approaches for programming assignments. *Computer science education* 15, 2 (2005), 83–102.

[2] Andrea Arcuri. 2013. It really does matter how you normalize the branch distance in search-based software testing. *Software Testing, Verification and Reliability* 23, 2 (2013), 119–147.

[3] Andrea Arcuri. 2019. RESTful API automated test case generation with EvoMaster. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 28, 1 (2019), 1–37.

[4] Luciano Baresi, Pier Luca Lanzi, and Matteo Miraz. 2010. Testful: an evolutionary test approach for java. In *2010 Third International Conference on Software Testing, Verification and Validation*. IEEE, 185–194.

[5] Kevin Buffardi and Stephen H Edwards. 2015. Reconsidering Automated Feedback: A Test-Driven Approach. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*. 416–420.

[6] Lin Chen and Damminda Alahakoon. 2006. Neuroevolution of augmenting topologies with learning for data classification. In *2006 International Conference on Information and Automation*. IEEE, 367–371.

[7] Ron Cytron, Jeanne Ferrante, Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. 1991. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 13, 4 (1991), 451–490.

[8] Roy De Maesschalck, Delphine Jouan-Rimbaud, and Désiré L Massart. 2000. The mahalanobis distance. *Chemometrics and intelligent laboratory systems* 50, 1 (2000), 1–18.

[9] Włodzisław Duch and Norbert Jankowski. 1999. Survey of Neural Transfer Functions. *Neural computing surveys* 2, 1 (1999), 163–212.

[10] Rob A Dunne and Norm A Campbell. 1997. On the Pairing of the Softmax Activation and Cross-Entropy Penalty Functions and the Derivation of the Softmax Activation Function. In *Proc. 8th Aust. Conf. on the Neural Networks, Melbourne*, Vol. 181. Citeseer, 185.

[11] Yang Feng, Qingkai Shi, Xinyu Gao, Jun Wan, Chunrong Fang, and Zhenyu Chen. 2020. Deepgini: prioritizing massive tests to enhance the robustness of deep neural networks. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 177–188.

[12] Dario Floreano, Peter Dürr, and Claudio Mattiussi. 2008. Neuroevolution: From Architectures to Learning. *Evolutionary intelligence* 1, 1 (2008), 47–62.

[13] Gordon Fraser and Andrea Arcuri. 2011. Evosuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. 416–419.

[14] Gordon Fraser and Andreas Zeller. 2011. Mutation-driven generation of unit tests and oracles. *IEEE Transactions on Software Engineering* 38, 2 (2011), 278–292.

[15] NN Glibovets and NM Gulayeva. 2013. A Review of Niching Genetic Algorithms for Multimodal Function Optimization. *Cybernetics and Systems Analysis* 49, 6 (2013), 815–820.

[16] Florian Gross, Gordon Fraser, and Andreas Zeller. 2012. Search-based system testing: high coverage, no false alarms. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*. 67–77.

[17] Sumit Gulwani, Ivan Radiček, and Florian Zuleger. 2018. Automated Clustering and Program Repair for Introductory Programming Assignments. *ACM SIGPLAN Notices* 53, 4 (2018), 465–480.

[18] Matthew Hausknecht, Joel Lehman, Risto Miikkulainen, and Peter Stone. 2014. A Neuroevolution Approach to General Atari Game Playing. *IEEE Transactions on Computational Intelligence and AI in Games* 6, 4 (2014), 355–366.

[19] Petri Ihantola, Tuukka Ahoniemi, Ville Karavirta, and Otto Seppälä. 2010. Review of recent systems for automatic assessment of programming assignments. In *Proceedings of the 10th Koli calling international conference on computing education research*. 86–93.

[20] Bekir Karlik and A Vehbi Olgac. 2011. Performance Analysis of Various Activation Functions in Generalized MLP Architectures of Neural Networks. *International Journal of Artificial Intelligence and Expert Systems* 1, 4 (2011), 111–122.

[21] Shruti Kaushik, Abhinav Choudhury, Nataraj Dasgupta, Sayee Natarajan, Larry A Pickett, and Varun Dutt. 2020. Ensemble of Multi-headed Machine Learning Architectures for Time-Series. *Applications of Machine Learning* (2020), 199.

[22] Hieke Keuning, Johan Jeuring, and Bastiaan Heeren. 2016. Towards a Systematic Review of Automated Feedback Generation for Programming Exercises. In *Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education*. 41–46.

[23] Dohyeong Kim, Yonghwi Kwon, Peng Liu, I Luk Kim, David Mitchel Perry, Xiangyu Zhang, and Gustavo Rodriguez-Rivera. 2016. Apex: Automatic Programming Assignment Error Explanation. *ACM SIGPLAN Notices* 51, 10 (2016), 311–327.

[24] Jinhan Kim, Robert Feldt, and Shin Yoo. 2019. Guiding deep learning system testing using surprise adequacy. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 1039–1049.

[25] Jinhan Kim, Jeongil Ju, Robert Feldt, and Shin Yoo. 2020. Reducing dnn labelling cost using surprise adequacy: An industrial case study for autonomous driving. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1466–1476.

[26] Nate Kohl and Risto Miikkulainen. 2012. An Integrated Neuroevolutionary Approach to Reactive Control and High-level Strategy. *IEEE Transactions on Evolutionary Computation* 16, 4 (2012), 472–488.

[27] Naoki Kondo and Kiminori Matsuzaki. 2019. Playing Game 2048 with Deep Convolutional Neural Networks Trained by Supervised Learning. *Journal of Information Processing* 27 (2019), 340–347.

[28] Bogdan Korel. 1990. Automated software test data generation. *IEEE Transactions on software engineering* 16, 8 (1990), 870–879.

[29] Kiran Lakhotia, Mark Harman, and Hamilton Gross. 2010. AUSTIN: A tool for search based software testing for the C language and its evaluation on deployed automotive systems. In *2nd International symposium on search based software engineering*. IEEE, 101–110.

[30] Guillaume Lample and Devendra Singh Chaplot. 2017. Playing FPS Games with Deep Reinforcement Learning. In *Thirty-First AAAI Conference on Artificial Intelligence*.

[31] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. 2014. An Empirical Analysis of Flaky Tests. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 643–653.

[32] John Maloney, Mitchel Resnick, Natalie Rusk, Brian Silverman, and Evelyn Eastmond. 2010. The Scratch Programming Language and Environment. *ACM Transactions on Computing Education (TOCE)* 10, 4 (2010), 1–15.

[33] Henry B Mann and Donald R Whitney. 1947. On a Test of Whether One of Two Random Variables is Stochastically Larger than the Other. *The annals of mathematical statistics* (1947), 50–60.

[34] Ke Mao, Mark Harman, and Yue Jia. 2016. Sapienz: Multi-objective automated testing for android applications. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*. 94–105.

[35] Michelle McPartland and Marcus Gallagher. 2010. Reinforcement Learning in First Person Shooter Games. *IEEE Transactions on Computational Intelligence and AI in Games* 3, 1 (2010), 43–56.

[36] Risto Miikkulainen, Jason Liang, Elliot Meyerson, Aditya Rawal, Daniel Fink, Olivier Francon, Bala Raju, Hormoz Shahrzad, Arshak Navruzyan, Nigel Duffy, et al. 2019. Evolving deep neural networks. In *Artificial intelligence in the age of neural networks and brain computing*. Elsevier, 293–312.

[37] A Jefferson Offutt, Ammei Lee, Gregg Rothermel, Roland H Untch, and Christian Zapf. 1996. An experimental determination of sufficient mutant operators. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 5, 2 (1996), 99–118.

[38] Jacob Kaae Olesen, Georgios N Yannakakis, and John Hallam. 2008. Real-Time challenge balance in an RTS Game using rtNEAT. In *2008 IEEE Symposium On Computational Intelligence and Games*. IEEE, 87–94.

[39] Nicholas J Radcliffe. 1993. Genetic Set Recombination and its Application to Neural Network Topology Optimisation. *Neural Computing & Applications* 1, 1 (1993), 67–90.

[40] Aditya Rawal and Risto Miikkulainen. 2016. Evolving Deep LSTM-based Memory Networks using an Information Maximization Objective. In *Proceedings of the Genetic and Evolutionary Computation Conference 2016*. 501–508.

[41] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. 1986. Learning Representations by Back-Propagating Errors. *nature* 323, 6088 (1986), 533–536.

[42] Stefano Sarti, Jason Adair, and Gabriela Ochoa. 2022. Recombination and Novelty in Neuroevolution: A Visual Analysis. *SN Computer Science* 3, 3 (2022), 1–15.

[43] Yuchul Shin, Jaewon Kim, Kyohoon Jin, and Young Bin Kim. 2020. Playtesting in Match 3 Game using Strategic Plays via Reinforcement Learning. *IEEE Access* 8 (2020), 51593–51600.

[44] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. 2016. Mastering the Game of Go with Deep Neural Networks and Tree Search. *nature* 529, 7587 (2016), 484–489.

[45] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. 2018. A General Reinforcement Learning Algorithm that Masters Chess, Shogi, and Go through Self-play. *Science* 362, 6419 (2018), 1140–1144.

[46] Andreas Stahlbauer, Marvin Kreis, and Gordon Fraser. 2019. Testing Scratch Programs Automatically. *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (2019), 165–175.

[47] Kenneth O Stanley, Bobby D Bryant, and Risto Miikkulainen. 2005. Real-time Neuroevolution in the NERO Video Game. *IEEE transactions on evolutionary computation* 9, 6 (2005), 653–668.

[48] Kenneth O Stanley, David B D'Ambrosio, and Jason Gauci. 2009. A Hypercube-Based Encoding for Evolving Large-Scale Neural Networks. *Artificial life* 15, 2 (2009), 185–212.

[49] Kenneth O Stanley and Risto Miikkulainen. 2002. Evolving Neural Networks through Augmenting Topologies. *Evolutionary computation* 10, 2 (2002), 99–127.

[50] Paolo Tonella. 2004. Evolutionary testing of classes. *ACM SIGSOFT Software Engineering Notes* 29, 4 (2004), 119–128.

[51] Nikolaos Tziortziotis, Konstantinos Tziortziotis, and Konstantinos Blekas. 2014. Play Ms. Pac-man Using an Advanced Reinforcement Learning Agent. In *Hellenic Conference on Artificial Intelligence*. Springer, 71–83.
[52] András Vargha and Harold D Delaney. 2000. A Critique and Improvement of the CL Common Language Effect Size Statistics of McGraw and Wong. *Journal of Educational and Behavioral Statistics* 25, 2 (2000), 101–132.
[53] Matt P Wand and M Chris Jones. 1994. *Kernel smoothing*. CRC press.
[54] Joachim Wegener, André Baresel, and Harmen Sthamer. 2001. Evolutionary test environment for automatic structural testing. *Information and software technology* 43, 14 (2001), 841–854.
[55] Shimon Whiteson and Peter Stone. 2006. On-line Evolutionary Computation for Reinforcement Learning in Stochastic Domains. In *Proceedings of the 8th annual conference on Genetic and evolutionary computation*. 1577–1584.
[56] Shimon Whiteson, Peter Stone, Kenneth O Stanley, Risto Miikkulainen, and Nate Kohl. 2005. Automatic Feature Selection in Neuroevolution. In *Proceedings of the 7th annual conference on Genetic and evolutionary computation*. 1225–1232.
[57] Tao Xie. 2006. Augmenting automatically generated unit-test suites with regression oracle checking. In *European Conference on Object-Oriented Programming*. Springer, 380–403.
[58] Jooyong Yi, Umair Z Ahmed, Amey Karkare, Shin Hwei Tan, and Abhik Roychoudhury. 2017. A Feasibility Study of Using Automated Program Repair for Introductory Programming Assignments. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. 740–751.