Neural Networks and Search Landscapes for Software Testing

Leonid Joffe

A report submitted in fulfilment of the requirements for the degree of **Doctor of Philosophy** of **UCL**.

Department of Computer Science
UCL

July 9, 2020

This thesis was made possible thanks to my father's persistent efforts – and despite mine. Thanks, dad.

Declaration

I, Leonid Joffe, confirm that the work presented in this thesis is my own. Where information has been derived from other sources, I confirm that this has been indicated in the work.

Summary of Publications

Parts of the work presented in this thesis have been published in the following papers. For each of these papers, the author of this thesis contributed to developing the concept and to writing, conducted all experimental work, and presented the papers at venues.

- Leonid Joffe and David Clark. *Constructing Search Spaces for SBST using Machine Learning*, published at the International Symposium on Search Based Software Engineering (SSBSE 2019). This paper forms the basis of Chapter 3, and outlines the concepts upon which further chapters build.

- Leonid Joffe and David Clark. *Directing a Search Towards Execution Properties with a Learned Fitness Function*, published at the 12th IEEE International Conference on Software Testing, Validation and Verification (ICST 2019). Chapter 4 is based on this paper.

- Leonid Joffe. *Machine Learning Augmented Fuzzing*, published at the International Symposium on Software Reliability Engineering Workshops (ISSREW 2018). This PhD symposium short paper is an overview of the whole thesis, as implemented and planned at the time of publication.

In addition, the author of this thesis co-authored the following publications that are not related to the work described in this thesis.

- David R White, Leonid Joffe, Edward Bowles, Jerry Swan. *Deep Parameter Tuning of Concurrent Divide and Conquer Algorithms in Akka*, published at the European Conference on the Applications of Evolutionary Computation (EvoStar 2017). Initially intended as a submission to the SSBSE challenge track, this short paper is a proof of concept for automatic deep parameter tuning for a Java concurrency framework. The author of this thesis participated in implementing and conducting

the experiments, and writing.

- Akhil Mathur, Tianlin Zhang, Sourav Bhattacharya, Petar Velickovic, Leonid Joffe, Nicholas D Lane, Fahim Kawsar, Pietro Lió. *Using Deep Data Augmentation Training to Address Software and Hardware Heterogeneities in Wearable and Smartphone Sensing Devices*, published at the 17th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN 2018). For this paper, the author of this thesis participated in the implementation of the experiments.

Abstract Search-Based Software Testing (SBST) methods are popular for improving the reliability of software. They do, however, suffer from challenges: poor representation, ineffective fitness functions and search landscapes, and restricted testing strategies. Neural Networks (NN) are a machine learning technique that can process complex data, they are continuous by design, and they can be used in a generative capacity. This thesis explores a number of approaches that leverage these properties to tackle the challenges of SBST.

The first use case is for defining fitness functions to target specific properties of interest. This is showcased by first training an NN to classify an execution trace as crashing or non-crashing. The estimate is then used to prioritise previously unseen executions that are deemed more likely to crash by the NN. This fitness function yields more efficient crash discovery than a baseline. The second proposition is to use NNs to define a search space for a diversity driven testing strategy. The space is constructed by encoding execution traces into an n-dimensions, where distance represents the degree of feature similarity. This thesis argues that this notion of similarity can drive a diversification driven testing strategy. Finally, an application of a generative model for SBST is presented. Initially, random inputs are fed to the program and execution traces are collected and encoded. Redundant executions are culled, distinct ones are kept and the loop is repeated. Over time, this mechanism discovers new program behaviours and these are added to the ever more diverse training dataset. Although this approach does not yet compete with existing tools, experiments show that the notion of similarity is meaningful, the generated program inputs are sensible and faults are found. Much of the work presented in this thesis is exploratory and is meant to serve as basis for future research.

Impact Statement

Testing is an integral part of a software development process. Many popular approaches use feedback to guide an automatic generator of program inputs. These methods fall under a discipline known as Search-Based Software Testing (SBST). The effectiveness of any SBST process depends on choices of representation, fitness function and search strategy. A poor representation restricts the choice of a fitness function and the resulting search landscape. A poor search landscape in turn limits the choice of search strategies. As a result, the effectiveness of an SBST process is hampered.

This thesis proposes the use of Neural Networks (NN) to address these challenges.

The first part of the thesis shows how an NN can be used to process observations of program execution to construct convenient, continuous search landscapes – for targeted and diversification-driven strategies alike. For targeted strategies, observations are used to train an NN to predict a execution property of interest, and the output of the NN is then interpreted as a fitness. For diversification-driven strategies, execution traces are encoded into a "latent" space which defines an ordering over program behaviours using apparently non-orderable observations. Experimental results show that these search spaces are continuous and large, and that they have a strong predictive ability for various properties of interest. To the best of the author's knowledge, constructing search landscapes for SBST in this manner is novel.

The second part of the thesis shows that an NN-generated search landscape for a property targeting strategy is effective. An NN is first trained to predict the presence of a crash given an execution trace. During testing, the output of the NN is used as a fitness of executions that have not yet crashed. Empirical results show that a search using this fitness consistently outperforms benchmarks.

The final portion of the thesis is a framework for diversity driven testing. The approach combines two NNs: one generates program inputs, and the other encodes execution traces. As the networks train, they generate new program inputs that explore the program's behaviours. Uninteresting datapoints are periodically discarded, so the training is conducted with a dataset comprised of the most interesting datapoints. Experimental results show that such a framework can be trained, that it does produce diverse program inputs, that the notion of similarity in the latent space is meaningful and that it can actually discover crashes.

The work presented does not attempt to outperform state of the art testing tools. Instead, it is intended to propose and explore a number of approaches for alleviating the challenges of SBST by using NNs. The presented results reflect these aims; they show that although the tools' immediate efficacy for automated testing is limited, the approaches are viable and can be used as a basis for various directions of future work.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Software is prevalent in the modern world and its reliability is undeniably paramount. The consequences of faulty software can be anywhere from unpleasant to catastrophic. Ensuring that software is fault-free is therefore of crucial importance. By and large, faults are unintended and unexpected, so their discovery is not a straightforward task. Indeed faults often need to be found, that is, searched for.

In principle, searching for faults in software is very much like searching for anything else... in an enormous, discrete, unordered space, often with local optima and indistinguishable intermediate solutions. It is a needle in a huge and confusing haystack, and the person looking for the needle knows nothing of hay, nor needles[1].

Luckily, this seemingly daunting situation can be alleviated by observing and evaluating the progress and using that to adjust the search. That is, instead of grabbing bunches of hay and checking them for a needle in a random manner, one can inspect the sampled bunch for clues, and then use the clues

[1]This analogy is not to be confused with the Dirac delta function, sometimes referred to as a "needle function" because of its shape.

to decide where to grab the next bunch.

The principle of evaluation of intermediate solutions for guiding a search forms the essence of feedback-driven, search-based automated testing methods (Sauder, 1962; Boyer et al., 1975; Harman et al., 2015). The work presented in this thesis proposes a number of novel ideas and techniques for improving these mechanisms. The proposed approaches are based on neural networks – machine learning mechanisms behind the recent deep learning boom (Goodfellow et al., 2016).

NNs possess several characteristics that make them particularly suitable for tackling some of the limitations of typical search methods in software testing (discussed below, in Section 1.2 and Section 1.3). First, they can process arbitrary data without manual involvement. Second, they can find patterns and signals in complex data which is not otherwise readily amenable to analysis. Third, they are differentiable by nature, which allows them to represent data in a smooth, continuous way. Fourth, their continuity also allows them to order data meaningfully according to the prevalence of features. Finally, they can be used as generators for new inputs. In terms of the "needle in a haystack" analogy, NNs can do the following.

i Eat up bunches of hay without flinching, as they are resilient to noise (Li et al., 2018a).

ii Identify features of the hay that are relevant to discovering the needle, and distinguishing one straw from another. By construction, NNs identify and focus on features of data that makes the learning task easier (Goodfellow et al., 2016).

iii Provide a quantification of a bunch of hay, rather than just a "yes" / "no" answer: e.g. "this bunch of hay is suspicious, keep looking around it" or "this bunch of hay is very different from the previous, but somewhat similar to the one before it".

iv Say where to draw the next bunch of hay from. That is, they can be used for generative purposes.

The rest of this introductory chapter is structured as follows. Basic concepts are covered next. Then, the challenge of software testing is described in more detail. This is followed by a more specific motivation of the challenges in search-based testing. Finally, the chapter concludes with an overview of the thesis.

1.1 Basic Concepts

This section covers the basic terms and concepts used throughout this thesis. The first instance of key terms is in boldface.

Programs consume **inputs** and produce **outputs**. In software testing, the standard term for a program that is undergoing testing is **Software Under Test (SUT)**.

An input to a program may be a string, a numeric value, a file, a stream, a timer, a random number generator, a series of clicks and so on. For generality, consider programs that do not take explicit inputs to take a `null` input. By definition, inputs are only meaningful when used by a program. Without a program, an input is, as it were, inert: "If a tree falls in a forest and no one is around to hear it, does it make a sound?"

An output of a program is typically understood to be something that was intended for a user to observe. It can be a printed statement on the screen, an image on the screen, a sound, a received and observed network data packet and so forth. As a program executes however, it produces other outputs that were not necessarily intended to be observed. These include resource usage patterns, memory traces, electromagnetic emanations or any other profiling information.

To be more general, we may talk about observable and non-observable outputs; synonymously external and internal **states**. Collectively, internal and external states of a program or its surroundings under execution are known as **behaviours** (Ammann and Offutt, 2016). The opening statement of the second paragraph of this section could thus actually be *"programs consume inputs and produce behaviours"*.

Programs are implicitly assumed to contain **faults** – underlying problems that need to be amended. The purpose of testing is to find faults in a program by execution (Myers et al., 2011). A fault is discovered by observing its manifestation: either an **error** or a **failure** (Ammann and Offutt, 2016). Errors refer to behaviours of non-observable, internal states, while failures are manifested in the behaviours of observable, external program states. A fault is defined with respect to an **expected behaviour**. An expected behaviour is something that a program *should* be doing, and an error or a failure is something it *should not* be doing.

It is not always clear what an expected, correct behaviour ought to be. There are implicit, obvious expected behaviours, such as an absence of crashes but in general, the definition of an expected behaviour is arbitrary, and depends on a program's specification. Other behaviours are less obviously undesirable. For instance, a low memory usage may be specified as an expected behaviour. A suboptimally high memory usage should thus be referred to as a "fault" – and an observed behaviour of high memory consumption as an "error" or a "failure". This is intuitively somewhat misleading however. After all, the program still works.

This thesis considers various examples of expected behaviours, some of them more undesirable than others. This indifference to undesirability warrants an alteration to standard terminology.

First, the proposed approaches do not explicitly target faults. Instead, any properties that testing may wish to ascertain – whether they represent prob-

lems or not – are considered. The notion of a fault is therefore replaced with a more general term **"property of interest"**.

Second, it is assumed that a user has the ability to observe both internal and external states, i.e. behaviours. Such a powerful user is interchangeably also referred to as a **tester** and an **observation** refers to any behaviours the tester (as it were) observes. Thus, given the powerful user, the distinction between an error and a failure is unnecessary. These are therefore collectively referred to as **behaviours of interest**.

The tester is not all-powerful however: they have full control over inputs and an ability to make observations, but they cannot, at will, force a program to exhibit a specific behaviour. That is, they can only provoke a behaviour of interest by providing an input that triggers it. The tester's task boils down to finding and crafting the input that triggers a behaviour of interest, which, in turn, is indicative of a property of interest. The discovery of inputs that trigger behaviours of interest is the overarching challenge this thesis aims to address.

1.2 Challenge of Testing in General

In the above definitions, the fact that testing involves *finding* properties of interest is critical. It implies that properties of interest are rare, and that discovering them is not trivial; that they are not immediately apparent given regular use of a program, or a lucky guess. To discover these non-trivial properties of interest, the tester requires an informed approach.

Consider the following sequence of testing approaches, in order of increasing amount of information available to the tester – from most to least informed. In the degenerate case where the tester knows where an issue is, they will execute the program under an input that triggers the behaviour of interest. This means running a manually constructed test suite and does not, in fact,

require any search. This however requires an in depth knowledge of the program and where its problems are. As noted above however, properties of interest are typically rare and the inputs that trigger them are unknown to the tester. They cannot thus construct a test suite manually.

The tester then turns to the next best thing – an educated guess. They try executing the program under **magic value** inputs that are known to typically trigger the behaviours of interest. For instance, a common way to attack a password checker (i.e. exercise a behaviour of interest – granting access) is to brute force a list of common passwords. A widely used penetration testing (Arkin et al., 2005) framework called Metasploit (Maynor, 2011) actually comes with such a list[2]. The common passwords are more likely to be correct than random, unstructured strings. Much like user-defined passwords, programs in general also tend to have structure, so the magic value approach is applicable.

In software testing, there are various typical magic values. For instance, for programs that take a numeric input, maximum integer value `MAX_INT + 1` is a magic value. This input might cause an integer overflow, resulting in unexpected, undesired behaviour. Another example is an empty string for a program that takes a string input. Depending on the implementation, a missing empty string check could result in unexpected behaviour. This was the case in a recent update to the login screen implementation of Mac OS where entering a blank password three times gave an attacker root access (Guardian, 2017). The problem with magic values is that there are only so many of them – the tester will eventually run out of educated guesses. What next?

The tester begins flinging uneducated guesses at the program; they start random testing. This approach is surprisingly effective (Arcuri et al., 2010) and (unsurprisingly) easy. In many cases however, it is grossly inefficient.

[2]Browsing the list of most common passwords really makes one think less of humanity...

Consider a program which crashes under a single string input of unknown length. The probability of producing it randomly is vanishingly small as each additional character of the string expands the number of possibilities by a factor of 128 (for ASCII strings). Attempting to find the single crash inducing string with random generation is infeasible.

In reality of course, not all possible inputs are equally likely to trigger a behaviour of interest – magic values are a manifestation of this fact. Indeed, most inputs will not be consumed by the program at all because the program imposes restrictions on what is a viable input.

Imagine a tester who attempts to use a program by clapping their hands at the computer. That is, they, for some reason, believe that a sequence of claps is a viable input. A typical program (one that does not happen to consume claps) would not recognise the input, and would exhibit a behaviour of "no reaction". Clapping hands in front of a computer is thus not a very productive testing method.

Consider a more educated tester who knows that programs typically do not respond to clapping. They are testing an XML parser which consumes a string one character at a time. Under most random input strings, only one behaviour is exercised: the program exits with an "invalid input" message. Only inputs with a valid XML prefix will trigger various behaviours beyond this.

An even more educated tester will be aware of a collection of rules that defines the validity of a program's input – the **grammar**. For instance, one rule of the XML grammar is "each document must start with an opening chevron, <". This tester can leverage their knowledge to generate new inputs that are constrained by the rules of the grammar, in an approach known as **grammar based testing** (Lämmel, 2001). Strings produced by a grammar based input generator would begin with <, which would successfully pass the first character check of the parser.

An ability to produce well-formed inputs is not the end of story however. Faults may manifest under both valid or invalid inputs. Imagine that the XML parser is faulty, e.g. it exhibits a behaviour of interest with an invalid input `xy<root>...`. This indicates a fault and needs to be corrected. A grammar based testing tool that produces only valid inputs would miss this fault. In other words, test inputs should not be "too correct". Furthermore, even if programs exhibited behaviours of interest only under well-formed inputs, the input space could still be enormous, so blind randomness would once again be non-viable. There are infinitely many valid XML strings and any of them may trigger behaviours of interest.

The tester's problem thus boils down to an informed search of inputs that trigger behaviours of interest – **inputs of interest**.

1.3 Search-Based Software Testing and its Challenges

The high level aim of the work described in this thesis is to improve the process of discovery of inputs of interest, by means of automated testing. The key element of the proposed techniques is a feedback loop that guides the generation of new inputs based on observations of a program's execution.

The use of a feedback loop for software engineering purposes places this thesis into a subdiscipline of software engineering called **Search Based Software Engineering (SBSE)**. SBSE applies search based optimisation to software engineering (Harman et al., 2012; Harman and Jones, 2001). Approaches include linear programming, genetic algorithms (Whitley, 1994; Holland, 1975), hill climbing (Harman et al., 2002a; Mitchell and Mancoridis, 2002) and simulated annealing (Metropolis et al., 1953). SBSE methods consider **candidate solutions** (i.e. intermediate solutions) to a problem, evaluate their

performance, and use the evaluation to guide subsequent search.

The instantiation of SBSE in software testing is known as **Search-Based Software Testing (SBST)** (McMinn, 2004). SBST applies search techniques to all manner of software testing problems including test suite optimisation, test case prioritisation and automatic test case generation (Korel, 1990). The techniques presented and investigated in this thesis deal with automated test generation and are thus part of SBST.

Typically, much like in testing in general, the purpose of automated test generation in SBST is to discover faults. In the context and terminology of this thesis however, the purpose of SBST is to discover inputs of interest.

A high level illustration of an SBST-driven automated testing workflow is shown in Figure 1.1. First, a program is executed under an input and its execution is observed **(1)**. Then, the observation is evaluated **(2)**. Finally, this evaluation is used to generate a subsequent input **(3)**. This loop – a feedback mechanism and repeated executions of a program under generated inputs – is the essence of SBST-driven automated testing. The tools proposed in this thesis follow this pattern.

Two choices characterise any SBSE (and therefore also SBST) process: that of representation and a fitness function (Harman and Jones, 2001; Harman, 2007). The **representation** is the description of a candidate solution, which is related to step **(1)** in Figure 1.1. A representation needs to be **relevant**, **interpretable** and sufficiently **rich**. Relevance means that the representation is correlated with the target of search; a property known as the **representation condition** (Shepperd, 1995). Interpretability means that the observations can be readily processed and analysed. Richness refers to the fact that the representation needs to have enough detail and granularity to yield a navigable search space.

A representation may be coarse or fine grained and its choice affects which

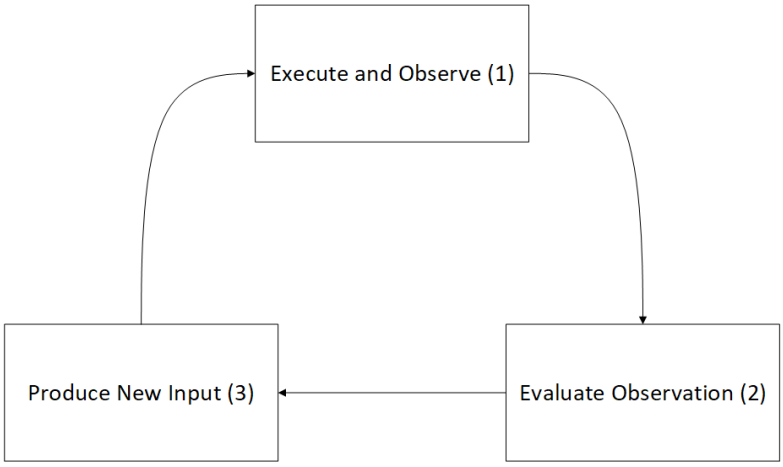
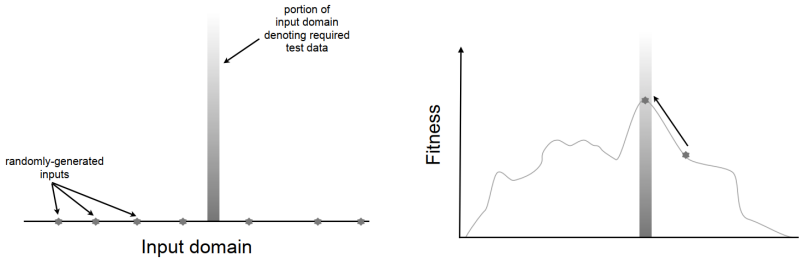


Figure 1.1: Feedback-Driven Automated Testing Workflow. A program is executed under an initial input and its behaviour is observed **(1)**. The observation is then evaluated **(2)**. Then a new input is generated **(3)**.

properties can be asserted. Consider a testing scenario with two testers and two properties of interest. The first tester uses a stop watch as a profiling tool, so the representation is execution time. This simple, coarse observation allows the tester to characterise an execution only in terms of its running time. The other tester uses a fine grained program profiler which records sequences of machine code operations. The former is easy to collect but is limited in its descriptive power, while the latter is harder to collect and process, but provides fine detail of a program's behaviour under execution.

These two testers are tasked with discovering two properties of interest. The first property is "the program leaks kernel memory". Observing the execution time is unlikely to be useful for discovering this property, whereas observations of machine code operations are more likely to be relevant. The second property of interest is whether a program hangs, i.e. exceeds a threshold execution time. In this case, execution timing is sufficient, whereas observing machine code operations is redundant; choosing an appropriate representation is critical.



(a) Randomly generated inputs are unlikely to trigger a rare behaviour.

(b) A search is guided towards a goal along a fitness landscape.

Figure 1.2: A fitness landscape (RH image) allows a search to be guided towards a behaviour of interest. Image source (McMinn, 2011).

A **fitness function** is the function that transforms a representation into a measure of quality in an SBSE process, point **(2)** in Figure 1.1. It needs to yield candidate solution values that are useful with respect to discovering a search goal. The output space of a fitness function is known as a **fitness landscape** or **search landscape**.

The choice of a representation is once again crucial, as it affects the quality of a fitness function and the search landscape. Consider a search landscape shown in Figure 1.2, perhaps resulting from a conditional statement such as `if (x=="abc") foo(); else bar();`. If a tester only observes the line coverage profile, the fitness function will have a plateau where all inputs appear identical. This will prevent a search from progressing effectively. If the tester considers a more detailed representation, they might be able to discover a useful signal which will lead the search towards the required input value `"abc"`.

Finally, the principle according to which the fitness landscape is navigated is known as a **search strategy**. In this thesis, two high level strategies are considered. A **targeted strategy** refers to a search which follows a signal towards a specific goal. A **diversification strategy** is one which has no

signal to follow, so instead it aims to discover candidate solutions that are unlike those previously seen. Consider an example of a canonical targeted strategy, the hill climbing algorithm. It uses the gradient of the hillside to arrive at an optimum at the top of the hill. The problem, of course, is that the peak may be only a local optimum (much like the smaller peaks in the illustration on the RH image of Figure 1.2). To avoid getting stuck in a local optimum, the search strategy can be improved by allowing it to occasionally take exploratory steps away from the gradient of the hill. That is to say, these two seemingly contradictory strategies can be, and indeed often are, complementary.

Targeted strategies are not readily applicable to arbitrary goals however, since gradients towards hilltops in the context of SBST can be very difficult to define.

For example, take an illegal memory write as a property of interest, i.e the optimum for a hill climb. In an illustrative pseudo-language, a legal memory write operation consists of a `malloc(addr_0, ...)` call, followed by a `write(addr_0, ...)` call. An illegal memory write occurs due to a missing first call to `malloc(addr_0, ...)` or there is a `free(addr_0)` call before `write(addr_0, ...)`. Defining a useful targeted fitness function is a manifestly difficult, perhaps even impossible task; it is very hard to manually construct a fitness function that indicates "how close" a search is to triggering an illegal memory write. In this case, one might, for instance, suggest that since an illegal memory write entails a memory write operation, the *number* of write operations is an indication of fitness: the more writes – the higher the fitness. This is an unprincipled and probably weak ad-hoc heuristic however.

A more principled, general approach for creating property targeting search strategies is needed. One of the contributions of this thesis is a demonstration of how such targeted strategies can be defined and implemented using

neural networks (NN). An NN is taught to identify execution trace features relevant to properties of interest. The network's estimate of the likelihood of a presence of the property is interpreted as a fitness.

In contrast to targeted strategies, diversification strategies prioritise candidate solutions that are less like those previously seen, rather than exploiting a gradient towards a peak. The question of defining (dis)similarity becomes central, and it is not obvious how this ought to be done.

Consider a program that exhibits a behaviour described by an input string, and a tester who measures the execution time. The first input x_0 is `"wait for 0 seconds"`. The next input is either `"do not wait, just terminate immediately"` (x_a) or `"wait for 60 seconds"` (x_b). Which one is better? If the search aims for diversity on input strings, x_a is preferable since it is syntactically less similar to x_0 than x_b is. However, in terms of behaviours as represented by run time, x_0 and x_a are identical – whereas x_b produces a new, previously unseen behaviour. Thus in terms of diversity of behaviours, x_b is the preferred candidate solution.

It is not apparent (nor agreed upon in literature, as will be discussed in Chapter 2), which notion of similarity is preferable with respect to fault discovery; many have been proposed. This brings us to another central contribution of this thesis: the use of NNs for defining and implementing search spaces for diversity driven search strategies. An NN is trained to identify the most discriminating features of execution traces and candidate solutions are evaluated on their "unusualness" with respect to previously encountered executions.

To conclude this section, let us summarise the introduced terminology in the context of automated test generation. A candidate solution is the combination of an input and the observations of an execution. A representation is what the tester chooses, or is practically able to observe about a program's execution. The observations of candidate solutions are processed into

a measure of quality with a fitness function. Finally, a search strategy determines how those fitness values are to be used in the context of a search process.

Let us also recap the fundamental choices of an SBST process – those of i) representation, ii) fitness function and iii) search strategy. All three are interrelated. Representation affects the fitness function, which affects the strategy. The intended strategy, in turn, affects the choice of a representation. This ensemble of issues represents the problems this thesis aims to address, discussed in detail next.

1.4 Overview, Contributions and Scope

This thesis proposes and explores methods that address fundamental challenges of SBST by using neural networks. Ultimately, the purpose of these techniques is to improve the fault discovery effectiveness of traditional SBST methods.

At this stage however, the scope is limited to exploration and prototyping. A thorough, systematic evaluation of all the new ideas is simply infeasible in the context of a single PhD thesis. This is particularly true when dealing with NNs that are notorious for relying on trial-and-error backed by considerable hardware resources.

This thesis proposes, implements and evaluates proofs of concepts, but not production ready products. The outcomes do not therefore represent ready alternatives for existing state of the art tools. That said, the presented approaches introduce novelty and are intended to be built upon in future research. These ideas appear to have been met with interest in the community; papers on which Chapters 3 and 4 are based, have been peer reviewed and published at conferences.

The rest of the thesis is arranged as follows. This introductory chapter is followed by a chapter on background and literature. After that, the three main content chapters explore the main idea of the thesis: using NNs to process execution traces to generate rich representations and consequently stronger fitness functions, as well as to generate new inputs for a diversification strategy. An overview and contributions of the main content chapters are given below. Finally, the thesis is concluded with a summary and an outline of potential future work in Chapter 6.

1.4.1 Producing Search Landscapes for SBST with Neural Networks

Chapter 3 introduces a methodology for creating search landscapes for both property targeting and diversification strategies using NNs. It presents the approach and describes the nature and quality of the generated landscapes, but does not evaluate their usefulness with respect to property discovery in an actual search process. That is done in further chapters; both Chapters 4 and 5 build on the principles introduced here.

For a property targeting strategy, an NN is trained on a large corpus of programs to map execution traces to a property of interest. During testing, traces of new executions that had not yet exhibited the property of interest are evaluated by the trained NN. The NN's output is a likelihood of whether the execution had the property of interest. This value, it is argued, can be interpreted as a fitness. It is shown that the landscapes are continuous and they satisfy the representation condition, i.e. they are strongly predictive of a number of properties of interest. These are all desirable properties of a fitness landscape.

For a diversification strategy, an NN called an Autoencoder (described in Chapter 2) is trained to identify the most discriminating features of execution traces, agnostic with respect to any particular property of interest.

The intermediate state of the NN is an n-dimensional encoding space which quantifies the similarity of these features. This space is likewise continuous, but also, crucially, it imposes a meaningful ordering on execution traces. It is suggested, that since the aim of a diversification strategy is to exercise behaviours that are most unlike previous ones, and an encoding represents a behaviour, *this* is the space over which diversity ought to be defined.

The contribution of this work is the proposition of a novel way of constructing convenient search landscapes for SBST. The use of NNs in this manner has not been considered before. Although this work does not evaluate the fault finding ability these landscapes yield, it does assess their properties: continuity, size, representativeness and ordering.

It also sets the foundation for future work. Chapters 4 and 5 build on the principles proposed in this chapter and evaluate these search landscapes in a search process. Furthermore, other instantiations of these ideas can be investigated in future work within the community.

1.4.2 Fitness Functions for Property Targeting Search

Chapter 4 builds on the idea of constructing a fitness function for a property targeting search introduced in Chapter 3. It goes on to present an implementation and an empirical evaluation. The work is composed of two parts. First, a fitness function is constructed by the same mechanism as in Chapter 3. Second, an existing SBST tool is modified to employ a targeted search strategy based on the fitness.

The fitness function is built by training an NN regressor to map standard C library call execution traces (the representation) to crash and non-crash labels (the property of interest), across a corpus of programs. The NN's output is thus the crash likelihood of an execution trace – an estimate of whether the execution resulted in a crash or not. During testing, the NN's

output is used as a new fitness function in the American Fuzzy Lop fuzzer (AFL, described in detail in Subsection 2.1.1) (Zalewski, 2007). AFL's search heuristics are modified to prioritise those candidate solutions that have *not yet crashed*, but have a high crash likelihood according to the NN.

The results suggest that guiding the testing process using the model's prediction of a crash likelihood improves the efficiency of crash discovery. A targeted strategy outperformed an unmodified AFL baseline in 80.1% and 63.5% of the conducted experimental runs on a testing dataset, in terms of total crashes and unique crashes respectively. The trained model was then applied to three real world applications with mixed results: the performance was not significantly better than the baseline. Upon further investigation, the underwhelming result was most likely due to the insufficiency of the chosen representation. Nevertheless, the work showcases the use of NNs to interpret program behaviour and guide an SBST method towards a specific property of interest.

This work is a proof of concept intended to illustrate the use of an NN-made property targeting fitness function in practice. It does not attempt to make claims of generalisability. However, it does show the validity of the approach in principle, and its applicability to other representations and properties of interest can be investigated in future work.

1.4.3 Deconvolutional Generative Adversarial Fuzzer

Chapter 5 presents a diversity driven testing framework based on a generative NN, called the Deconvolutional Convolutional Adversarial Fuzzer (DCGAF). There are two main contributions in the work on DCGAF. The first is that it implements a diversity driven strategy, as introduced in Chapter 3. Second, it proposes an architecture for self-feeding generative NNs for test generation.

The work does not attempt to show that the proposed diversification strategy or the generator are explicitly better than existing methods in SBST. It does however present promising results and a principally novel approach to using NNs in automated testing. Experimental results show that new, sensible program inputs are generated, varied program behaviours are exercised, crashes are found – and this is all done without feature engineering, training datasets or any knowledge of the underlying program.

The approach combines two NNs: one generates program input strings, and the other encodes execution traces into a Euclidean space over which a diversity driven search is conducted. As the networks train, they keep generating new program inputs that trigger novel execution traces. The novelty (or interestingness) of an execution is assessed by how its encoding compares to previously seen ones. Uninteresting datapoints are discarded, and training is resumed with a dataset comprised of the most interesting, representative datapoints.

The findings of this work are exploratory. They show that such a framework can be trained in principle, that it does produce diverse and sensible program inputs, that the notion of similarity in the encoding space is meaningful and that it can actually discover crashes. A more systematic, in depth study of the framework's performance ought to be conducted. In addition, the architecture represents two novel ideas for automated testing, and thus lends itself to two immediate directions of future work. First, a diversity driven strategy on an NN-generated search space has not been proposed before. Alternative representations, fitness functions and formulations of a search strategy can be investigated further. Second, the use of generative models without a training dataset has not been previously explored in automated testing. Thus, the details of trace encoding, generation of new inputs and dataset pruning may all be directions for future work.

Chapter 2

Background and Literature

This chapter outlines recent work that addresses the same problems this thesis deals with, and puts them into context of the proposed approaches.

First, the challenges and proposed solutions for SBST are discussed. The primary instantiation of SBST considered in this thesis is the American Fuzzy Lop Fuzzer (AFL). Therefore its functionality, as well as recent literature that has used AFL as its basis is described in detail.

Then, the functionality and recent applications of neural networks are presented. The NN architectures used in this thesis range from simple and well known to experimental, but their usage for software engineering is novel in every instance. The literature review of NNs thus primarily serves as background on their mechanics and designs – not as attempts to address the problems this thesis deals with.

This section closes by presenting works that use machine learning tools for software engineering tasks. Although the approaches are different from those of this thesis, the papers are nonetheless relevant in that they use ML to reason about programs and address challenges of software engineering.

2.1 Search-Based Software Testing

A fundamental feature of any search-based process is a feedback loop which evaluates progress, so that the search can be guided, as shown in Figure 1.1. Any SBSE process is characterised by choices of representation, fitness function and search strategy. Their quality determines the effectiveness of the approach.

A representation that is insufficiently rich, relevant or interpretable can lead to a poor fitness function, which in turn produces a poor search landscape with plateaus and local optima. This then limits the choice of a search strategy. On the other hand, the issue can be seen as that of a poor search strategy. When it comes to the two basic strategies concerned in this thesis – diversification and targeting – the problem is either in an inappropriate notion of diversity or in the infeasibility of defining a targeted fitness function. In the former case, various diversities have been considered to different degrees of success. In the latter case, characterising a specific property and targeting a search towards discovering it has been hardly explored at all.

This thesis aims to address the problem from both of these perspectives – by improving representations and fitness functions, and by proposing novel takes on search strategies. This is done by using NNs to process execution traces to create fitness functions and search landscapes, as well as to generate new program inputs. Recent literature in which these problems have been tackled by other means is summarised further in this section.

2.1.1 Fuzzing and the American Fuzzy Lop

The main instantiation of SBST considered throughout this thesis is **fuzzing** – an automated testing technique which attempts to crash a program by feeding it randomly generated inputs (Sutton et al., 2007). The original methodology (Miller et al., 1990) used purely random input generation but

modern **evolutionary fuzzing** techniques make extensive use of feedback and search, bringing fuzzing into the realm of SBST. There are numerous tools and approaches to automated test generation in SBST and for a comprehensive review the reader is referred to recent surveys (Li et al., 2018b; Ali et al., 2010; Anand et al., 2013).

A popular modern evolutionary fuzzer is the American Fuzzy Lop (AFL) (Zalewski, 2007). It has recently seen wide use in academia, and it is the main SBST tool employed throughout this thesis. This subsection is dedicated to describing its mechanics and principles. The reason for doing so this early is that many of the related works described in sequel are based on AFL.

The documentation of AFL is very clear about the fact that its design decisions are not principled, but based on empirical evidence. This can be seen as an invitation for academics and other contributors to improve on them. Indeed many have, as will be shown.

At its core, AFL implements a diversification strategy on observed program traces, based on several heuristics. Its basic workflow is shown in Figure 2.1, and a detailed description is presented below with numbers in brackets corresponding to the numbers in the figure.

2.1.1.1 Initialisation

AFL's fuzzing process begins with one or more initial input seeds provided by the user. This forms the initial queue **(1)**. Each queue element is calibrated: it is executed, and its running time and execution trace are recorded. Although the initial seed can be a blank dummy file, a set of inputs that are relevant to the program under test can drastically improve AFL's performance. This is the focus of a number of recent papers discussed further in Subsubsection 2.1.2.3.

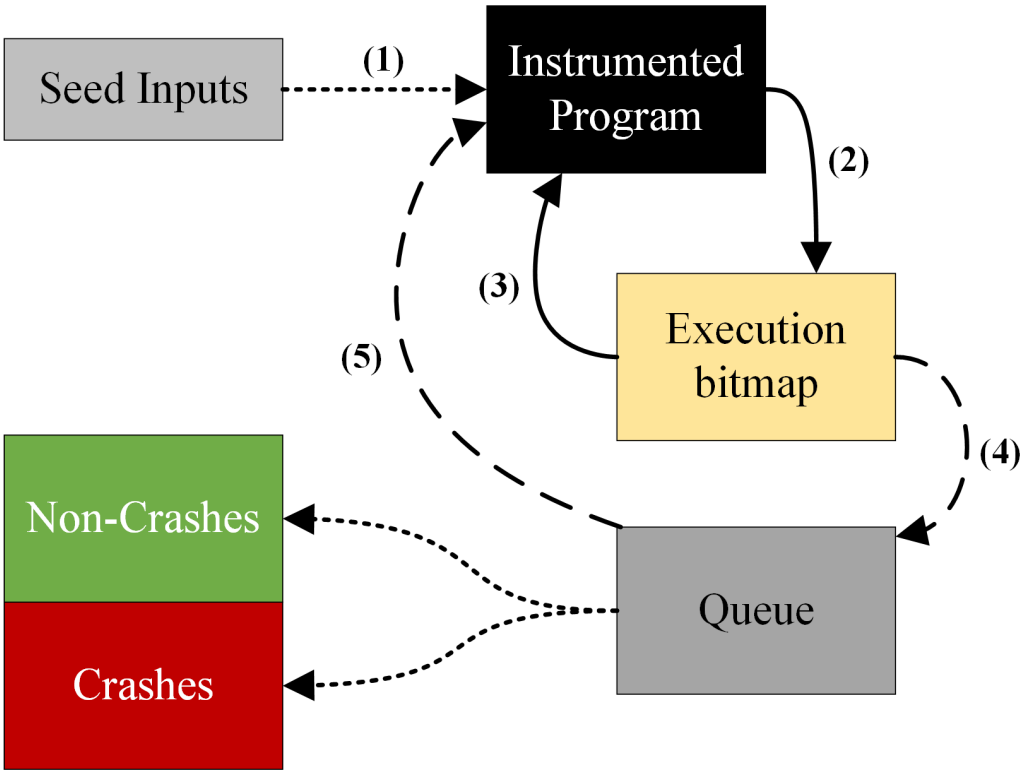


Figure 2.1: AFL workflow. Solid arrows indicate inner loop, dashed arrows outer loop, and dotted lines initialisation and termination steps. Initial seeds are provided by the user **(1)**. The instrumented program is executed with a queue element and the trace is recorded **(2)**. The element is then modified and the program re-executed **(3)**. Traces deemed interesting are appended to the queue **(4)**. Once a cycle completes, all queue elements are scored **(5)** and a new cycle begins. At termination, the queue and unique crashes remain available.

2.1.1.2 Trace Representation

Before fuzzing, AFL instruments a program at every decision point. The instrumentation is done with a drop-in replacement for GCC (or G++ or Clang) prior to testing. The transitions between these points form a hashmap (referred to as "bitmap" in AFL's documentation) of edges and their hit counts **(2)**. This trace is a representation of a program's execution; an execution path. To fit the bitmap fully into the L2 cache for efficiency, only a rough number of transition counts is kept: 0, 1, 2, 3, 4–7, 8–15, 16–31, 32–127, 128+. The bitmap has a static size of 64K, so the resulting vector of hit counts for small programs tends to be very sparse – most values are 0. There are two kinds of bitmaps: a persistent one which summarises the executions that have been observed, and a new one for each execution. Each new bitmap is compared with the summary bitmap of previous executions to determine its "interestingness" – as defined below.

The representation is somewhat coarse however. For example, it does not capture the order in which the edges were taken, i.e. it is context insensitive. Also, the bucketisation of trace hit counts reduces expressiveness. As a result, multiple input elements may produce the same execution path. Improvement of AFL's representation is the topic of a number of recent papers, discussed further in this chapter.

AFL's representation is convenient for use with NNs for two reasons. First, the bucketisation of the bitmap and the fixed size make it convenient for processing by an NN. It requires no normalisation or pre-processing. Second, thanks to AFL's blistering speed, it can produce vast numbers of datapoints for a data hungry network. This is relevant for the work in Chapter 5, where AFL's instrumentation harness is used to capture execution traces and feed them to an NN.

2.1.1.3 Fuzzing Operations

Each queue item is modified and executed repeatedly for a number of executions in accordance with its fuzzing budget **(2)** & **(3)**. The fuzzing budget is determined by a number of heuristics that are described below in Subsubsection 2.1.1.5. The modifications are deterministic or stochastic. The deterministic steps are bit and byte flips, simple arithmetic and "interesting integers" such as 0, -1, 1 and `MAX_INT`. The deterministic steps can be disabled with an option flag which allows AFL to proceed to stochastic steps – random, feedback driven search. The stochastic steps are combinations of the above, as well as block deletions, insertions and overwrites, memset and splicing, and standard genetic programming operations like mutation and cross-over. Further details of these operations can be found in AFL's documentation (Zalewski, 2014). The work in this thesis does not alter AFL's fuzzing operations but the mechanism is nonetheless important to understand for background.

2.1.1.4 Interestingness

Fuzzing, and typically software testing in general, aims to exercise program behaviours diversely: diversity is quality. AFL defines a notion of interestingness (i.e. diversity, or interchangeably "uniqueness") of an execution on the state transition bitmap by one of these three conditions.

i the execution produced a state transition that has not been seen before;

ii it changed the bucket count of a transition;

iii or it did not exercise a transition which had previously always been exercised.

For an input to be appended to the queue **(4)**, its execution trace needs to fulfil one of these conditions. Crashing inputs are *not* used as seeds for further fuzzing by default.

This definition of interestingness is heuristically chosen as a trade-off between efficiency of crash discovery and instrumentation overhead. While the heuristics of AFL are empirically effective, they are not principled and do not order nor quantify the similarity of executions and program behaviours. The approaches proposed in this thesis, on the other hand, do aim to quantify the quality of diversity for the purposes of defining novel, more principled search strategies.

2.1.1.5 Prioritisation Heuristics

A single pass through the queue is called a cycle. At the end of a cycle, AFL culls those old queue elements whose bitmaps are fully superseded by new ones, with respect to the above definition of uniqueness. Fuzzing then continues with an updated queue. At the start of a new cycle, each queue element is assigned a time budget α **(5)**. The time budget value determines how many times each element is to be modified and executed. This is AFL's fitness function. There are five heuristics to determine the fitness value.

- An element's execution time with respect to the average. Quicker executions get a higher priority simply to allow for more total executions.
- The size of the element's bitmap. The reasoning is, that larger traces correspond to exercising more complex behaviours, and are thus more likely to reveal faults.
- Recency of a queue element. Newer entries are prioritised, since they have gotten less "air time" (have been modified and executed fewer times) than older ones.
- An element's hierarchic novelty, a.k.a. its "depth". For instance, the depth of entries that were found using the initial queue is 1. Elements that were found by modifying depth 1 elements would have a depth of 2 and so on. Elements of greater depth (i.e. descendants of more

complex parents) get a higher priority as they are expected to produce ever more complex behaviours.

- Dynamic ad-hoc budget extension. If unique executions are found while fuzzing an element, its budget is dynamically increased until no more new behaviours are found. This heuristic effectively overrides the former ones while there is mileage in the queue element.

The work in Chapter 4 introduces a new heuristic to AFL, and re-prioritises the queue accordingly.

2.1.2 Representations and Fitness Landscapes

Choosing a representation and creating a fitness landscape is no trivial task and requires expertise from the tester. This thesis suggests that the task can be made easier with the use of neural networks, and with minimal human involvement.

This subsection first explains what constitutes a poor or a good search landscape. Then it presents some approaches of improved representations and hence better landscapes – in SBST at large, as well as in the case of AFL specifically. Finally, it broadens the notion of representations of program executions by introducing exotic representations with the example of side-channels.

2.1.2.1 Characteristics of Search Landscapes

There are two main undesirable characteristics of a search landscape: the presence of plateaus and local optima (McMinn, 2011; Aleti et al., 2017). A plateau is a situation where multiple, adjacent candidate solutions have the same fitness value, which makes them indistinguishable in terms of quality. The prevalence of plateaus in real world programs has been recently corroborated by Albunian et al. (Albunian et al., 2020). This prevents the search

mechanism from deciding how to proceed with the search. A local optimum is a solution which appears good, but is not globally optimal.

There are various ways of characterising and quantifying the complexity of a search landscape. These include the autocorrelation function and correlation length (Weinberger, 1990, 1991), information content (Vassilev et al., 2000), and the negative slope coefficient (Vanneschi et al., 2006). These approaches define adjacency and landscape neighbourhoods with respect to search operators; two candidate solutions that are one search operation away from each other, are adjacent. The landscapes proposed in this thesis are not defined by search operators however, but NNs' encodings of candidate solutions. These analyses are therefore inapplicable.

Symmetrically, there are also definitions of desirable characteristics of search landscapes. According to Harman and Clark, the search space of a good fitness function ought to have the following properties (Harman and Clark, 2004). It needs to be **i)** large and approximately continuous, the fitness function needs to have **ii)** low computational complexity and **iii)** not have known optimal solutions. Furthermore, they propose that various metrics can be used as fitness functions which implies two further characteristics. First, according to the representation condition, **iv)** a good metric needs to be truly representative of the property it seeks to denote (Shepperd, 1995). Second, a metric imposes an order relation over a set of elements by definition, and **v)** for a metric to be useful as a fitness function, the ordering needs to be meaningful.

Search landscapes proposed by approaches of this thesis aim to have these beneficial properties. First, NNs are trained by a process of backpropagation (Rumelhart et al., 1986) which means they must be differentiable and thus continuous by construction (Glasmachers, 2017). That is, if a neural network trains, its intermediate states must not have plateaus. NNs therefore appear to be suitable to tackle the issue of plateaus in search landscapes. Secondly, it

has been shown that given sufficient size, neural networks avoid local optima (Kawaguchi, 2016; Swirszcz et al., 2016; Nguyen and Hein, 2017, 2018). This property tackles the second central problem of classic SBST techniques – that of local optima. Thus with a sufficiently powerful representation and a large enough network, one ought to be able to produce a convenient landscape: there will be no plateaus nor local optima. Finally, NNs impose an ordering on the underlying data, which defines a notion of similarity which can be used to define novel search strategies.

2.1.2.2 Improving Search Landscapes in SBST

The proposed techniques aim to address the problem of poor search landscapes in principally novel ways. Approaches to address this problem in current SBST literature are described here for context.

One way of improving the landscape is to consider a more expressive representation. For instance, rather than simply looking at a coverage profile, Wegener *et al.* proposed using **branch distance** (Wegener et al., 2001). The fitness value is based on the distance of a numeric value for taking a path of interest and level of hierarchy at which the wrong path was taken. A fitness plateau resulting from only considering whether a path was taken or not is transformed into a more expressive landscape. This approach only handles numeric predicates however and still suffers from plateaus, e.g. in cases of boolean flags. If a branch predicate depends on a boolean, there is no difference between candidate solutions and search cannot proceed (Bottaci, 2002).

The presence of discrete, unorderable predicates like a boolean flag is, a common issue for SBST techniques. Consider the use of string distance measures in string literal predicates (Bottaci, 2003). For instance, simply checking whether a predicate such as `x=="abc"` is met using equality, will yield a plateau in the search landscape. A string similarity measure like Levenshtein

distance (Levenshtein, 1966) can be used to provide a quantifiable similarity measure which can be interpreted as a fitness (e.g. (Alshraideh and Bottaci, 2006)). A plateau inducing binary fitness is thus transformed into a richer landscape. However, much like the landscape analysis techniques mentioned above, this notion of similarity is defined over search operators – specifically those that define Levenshtein distance. That is, this notion of similarity is sensible only with respect to the search operators such as insertion, deletion and substitution. It is therefore inapplicable as a notion of similarity to any generators that use other search operations.

Another approach to address the problem of difficult predicates is the use non-search based methods in conjunction with SBST. For instance, a technique called testability transformation temporarily rewrites difficult conditionals (Harman et al., 2002b, 2004; McMinn et al., 2009), e.g. `if (x=="abc")` → `if(1)`. The behaviours that were guarded by this predicate can then be easily explored by a search tool. Once this is done, the modified program is discarded and the original one is restored. This method obviously requires access to the internals of a program, as well as an expert's involvement. Furthermore, the rewritten program may behave fundamentally differently from the original, so there is no guarantee of correctness of the testing. Nonetheless, it illustrates how a search process can be aided using non-search mechanisms – in this case, by temporarily ignoring the problem behaviours guarded by conditionals.

2.1.2.3 Improvements to AFL's Representation

As mentioned above, the main instantiation of SBST in this thesis is AFL. Its representation is a context insensitive execution trace of basic block transitions, as described in Subsection 2.1.1. While efficient, this representation sometimes fails to create a sufficiently rich search landscape. There is recent work that augments AFL's representation.

For instance, Chen *et al.* add context sensitivity in their tool called Angora (Chen and Chen, 2018). Rather than simply keeping track of counts of basic block transitions, this extended representation also records the order in which these transitions were made. Angora uses inequality constraints of predicates for intermediate fitness values which further enriches the representation. Furthermore, Angora uses taint tracking to target input locations of greatest interest, which not only further enhances the representation, but also aids the generation of new inputs.

As another example of improvement to AFL's representation, Gan *et al.* show that AFL's original profiling mechanism suffers from collisions – multiple edges get mapped to the same value (Gan et al., 2018). They introduce the use of a hash function which maps the traces into distinct values, which serves to alleviate the collision problem. As a result, the representation becomes more precise without considerable additional overhead. The above two works are examples of how an enriched representation outperforms AFL's original, more basic one.

Like for SBST in general, rare behaviours, such as those guarded by magic value predicates, pose a problem for AFL. One solution to this issue is the use of non-search-based mechanisms, such as on-demand **symbolic execution** (King, 1976). This was shown in a concolic (Sen, 2007) variant of AFL called Driller (Stephens et al., 2016). When AFL's own fuzzing heuristics appear to get stuck, an unexplored path is passed to a symbolic execution engine. The symbolic execution mechanism produces an input that triggers the unexplored path and returns it to AFL to resume fuzzing. The evaluation shows that the approach successfully finds paths guarded by magic value predicates. In this case the symbolic constraints are also an extension of the representation. Symbolic execution approaches tend to suffer from scalability issues however (Baldoni et al., 2018), which greatly limits their applicability. That is to say, the scalability of this work is questionable.

Another approach to address AFL's limited ability to discover rare behaviours was proposed by Rawat *et al.* with a tool called Vuzzer (Rawat et al., 2017). In fact, the modifications of Vuzzer deal with three aspects of AFL's functionality: an extension of representation, a change of search strategy, and improved input mutation. It introduces static analysis into the fuzzing process, which makes AFL aware of magic values and certain control flow features. The addition of knowledge of magic values can be thought of as enriching the representation. Although the techniques proposed in this thesis do not use magic values or static analysis, this is not out of the question for future work. For instance, restricting the mechanism described in Chapter 5 to only viable characters may be used to improve its performance. More immediately, this work is conceptually important as the authors use the extended representation to define an alternative search strategy.

2.1.2.4 Exotic Representations

The works and techniques described above focus on search-based methods and the AFL fuzzer specifically. However, what if one was to consider the notion of a representation in a broader context? Are there representations of program behaviours in fields other than software testing that are convenient, relevant and interpretable?

A perhaps surprising inspiration for this thesis is the work on **side-channel** attacks. A side-channel attack discovers something about the internal state of a program by observing the *physical* execution trace (Zhou and Feng, 2005; Spreitzer et al., 2018). The relevance of side-channel attacks with regards to this thesis is in their use of unusual representations, as well as their behaviour targeting strategies, particularly in the context of exploit generation (Avgerinos et al., 2014). They demonstrate that execution trace representations do not need to be limited to standard profiling information such as coverage.

For instance, Callan *et al.* show how electromagnetic emanations (EM) can be used for program profiling (Callan et al., 2016). EM traces and execution paths are recorded, and then used to train a classifier. Execution paths of subsequent program executions can then be identified based on their observed EM profile. Granted, this work is not an attack – merely profiling, but it does demonstrate the principle of using observations of a physical execution traces as a representation. Another example of a side-channel attack is the acoustic attack on an old printer (Backes et al., 2010). As and old printer operates, it emanates distinct sounds for each character. By identifying the mapping of sounds to the corresponding characters, the acoustic attack can deduce the content of a printed document. Both of these papers rely on machine learning techniques to interpret the representations, which does not require analysis by an expert. In that sense, they share a similarity with the works in this thesis.

There is even a case where observations of the physical execution are used to define a targeted search strategy, and it is a central motivating example for this thesis. It is the timing attack on the RSA encryption algorithm (Rivest et al., 1978), where an attacker searches for the correct key by observing the time it takes for a program to deny access to a guess (Kocher, 1996). The RSA algorithm relies on multiplication and division of large primes, which is a slow, $O(n^4)$ process. A number of optimisations are therefore typically employed. This reduces the time for decryption, crucially, by different amounts, depending on how close the guess of a key is. The closer the guess, the quicker the (negative) response. By timing the responses, an attacker can deduce the behaviour of interest, i.e. granted access, one bit at a time. The execution time thus acts as a fitness function. This example is a rare case where the underlying implementation happens to allow for a convenient fitness landscape: continuous, easily observable and strongly correlated with the target property. This, in turn, enables a property targeting search strategy, in this case granting access with the correctly guessed key. Of course,

arbitrary programs are not amenable to this particular fitness function – the use of the timing channel for discovering an arbitrary property of interest. Yet the concept of exploiting an unexpected representation for defining a good fitness function is very powerful and is exemplary of what this thesis ultimately aims to achieve.

Thanks to NNs' ability to find patterns in large, noisy data which would not be amenable to human interpretation, exotic representations, like those used in side-channel attacks, can be considered. Some representations might be "too rich" however; they may contain redundant information which amounts to noise. Luckily, modern deep NN architectures are somewhat resilient to noise (Li et al., 2018a), and classic methods like feature selection can be employed to alleviate the issue further (Verikas and Bacauskiene, 2002; Leray and Gallinari, 1999; Wang et al., 2014). The problem of noisy data is not immediate for the work in this thesis however, and is mentioned here pre-emptively – for future work.

Observations of physical execution are not, in fact, used in this thesis. Nonetheless, the intuition that arbitrary representations can be considered for construction of fitness functions, was a major driver in the design of the approaches. That said, the representations used in the work of this thesis are rather complex. Interpreting them manually would be difficult, while the proposed techniques deal with them without human involvement. In principle, in future work, the proposed approaches could be applied to arbitrary poorly interpretable representations.

2.1.3 Search Strategies

This thesis considers two high level search strategies of SBST – diversification and targeting.

Diversification refers to exploratory search where a specific target is not ex-

plicitly defined. Instead, the search is driven towards exercising program behaviours unlike those previously seen. This approach implies two fundamental questions.
One: what is the representation over which a behaviour ought to be defined?
Two: how is (dis)similarity of behaviours measured?

Targeted search strategies are those that have a specific property of interest as a driver of their fitness function. A targeted strategy requires that a signal which characterises the proximity to the property of interest is known. There are likewise two questions for these strategies.
First, what representation is relevant to the property of interest?
Second, how does one interpret those observations as a targeting fitness?

In this thesis, both of these questions are tackled with NNs. For targeted strategies, NNs are used to process representations to identify patterns indicative of properties of interest. Their estimate of presence of properties is interpreted as a fitness for a targeted strategy. For diversification strategies, NNs identify the most distinguishing features of execution traces. This understanding allows an NN to arrange execution traces into a continuous space where their location is determined by their distinguishing features, and it is argued that a diversification ought to be defined over this space.

However, there are numerous other methods for answering the above questions. These are discussed below, again, with special focus given to works using (or based on) AFL. Much like in the perspective that shortcomings of SBST stem from poor representations, as presented in Subsection 2.1.2, the examples below show that strategies in SBST are a debated topic with various solutions.

2.1.3.1 Lack of Consensus on Strategies in SBST

In testing, program behaviour is often represented by some notion of coverage – be that lines, basic blocks or branches. Coverage, in turn, is often taken as

the quality measure of a testing approach. Exercising a program's behaviours with maximal coverage corresponds to doing so with maximal diversity. Thus coverage is perhaps the most common instantiation of diversity.

There are numerous works with evidence to support coverage driven testing, e.g. (Hutchins et al., 1994; Namin and Andrews, 2009). However, there is also evidence to suggest that coverage driven SBST may be sub-optimal (Frankl and Weiss, 1991; Inozemtseva and Holmes, 2014; Kochhar et al., 2015; Whalen et al., 2013). In fact, it has been argued that as a sole target, coverage can even be detrimental (Heimdahl and George, 2004; Heimdahl et al., 2004; Staats et al., 2012; Gay et al., 2015). It has also been suggested that if the cost of instrumentation and engineering are considered, perhaps the best way to go about SBST is blind randomness (Arcuri et al., 2010; Arcuri and Briand, 2011).

Given the evidence that coverage and more generally diversity are not universally desirable, it is not immediately clear why they are nonetheless widely accepted as the gold standard of testing. It is perhaps based on the empirical findings that failure inducing inputs tend to be located in contiguous or regularly arranged regions of the input space, as adjacent or regularly spaced inputs are more likely to follow the same execution path and hence both trigger a fault (Anand et al., 2013). It may also be the case that targeted strategies are simply too difficult to construct – or they seem fundamentally inapplicable. Whatever the historical reason for preference of coverage and diversity, it is far more prevalent in SBST than targeted strategies, and a number of canonical SBST works aim at it.

2.1.3.2 Search Strategies in SBST

There are numerous tools and papers using diversity and coverage driven search strategies. A number of seminal works, along with their definitions of diversity and search strategies, as well as their relevance to the techniques

proposed in this thesis are described next.

There are several highly regarded and highly cited seminal testing frameworks that aim to maximise coverage. For instance, Directed Automated Random Testing (DART) automatically extracts the input interface of a program, constructs a test driver, and executes a program with randomly generated test inputs (Godefroid et al., 2005). It combines symbolic and concrete execution for input generation with the aim of maximising code coverage. Another testing tool which aims to reach coverage criteria is EvoSuite (Fraser and Arcuri, 2011). Originally it was implemented to generate JUnit test cases for a single coverage criterion. Since then, it has been modified to simultaneously optimise a number of coverage criteria (Fraser and Arcuri, 2013). EvoSuite is an ongoing project with multiple contributors and extensions, and the techniques of this thesis could potentially be adapted to using it as the underlying framework. Another seminal diversity driven framework is Automated Random Testing (ART) (Chen et al., 2004). It showed that diversifying the numeric input space during testing leads to better fault discovery. This idea was further developed in the ARTOO tool which aims for higher diversity of object inputs for OO programs (Ciupa et al., 2008).

In addition to coverage-driven testing frameworks, other notions of diversity have been proposed, albeit their popularity is not comparable to that of the frameworks above. For instance, output diversity has been considered: it was shown that increasing program output diversity improves fault discovery (Alshahwan and Harman, 2014). Other measures of input diversity, such as Normalised Compression Distance (NCD), have also been shown to improve the quality of test suites (Feldt et al., 2016). These examples illustrate the fact that diversity is not only defined in terms of coverage, and that various interpretations can be used. The work in this thesis suggests a novel notion of diversity, based on the representation of execution traces by an autoencoder. The idea is first explored in Chapter 3 and then implemented

in Chapter 5.

Despite the popularity of diversity driven methods, property targeting strategies are not unknown to SBST. For instance, SBST methods have been applied to produce a test suite which explicitly targeted defects, rather than maximising coverage (Fraser and Zeller, 2012). Fitness of intermediate steps was nonetheless evaluated on the coverage of fault inducing mutated code. That is to say, a targeted fitness function prioritised a very focused form of coverage.

Another example of a targeted strategy was proposed by Feldt and Poulding, where the aim is to produce test suites with specific properties (Feldt and Poulding, 2013). In this work, a grammar based input string generator uses the size of inputs as the representation. Beginning with an empty string, each subsequent token in a string depends on a probabilistic choice of an integer called a Gödel number (not to be confused with the original Gödel numbers (Gödel, 1931)). The sequence of Gödel numbers is given a fitness value with respect to the property of interest – the size of the input string. The inputs in the generated test suite are considerably larger than those in a baseline, as intended by the fitness function. This work is of fundamental relevance to this thesis as it shows how an arbitrary property can be set as a target of a generative SBST process. The same concept is behind the idea for a targeted search strategy as first presented in Chapter 3 and then realised in Chapter 4.

Finally, work on targeting **non-functional properties** is worth mentioning. For instance, Wegener and Grochtmann consider execution time as the target for testing (Wegener and Grochtmann, 1998). The aim is to discover program executions with minimal and maximal execution times. In the context of **genetic improvement** (Petke et al., 2017) properties like memory usage (Wu et al., 2015) and power consumption (Bruce et al., 2015) have also been considered. Non-functional properties are not the specific focus of work

in this thesis, save for the cache behaviour properties in Chapter 3. They are nonetheless examples of properties of interest, and in future work the techniques presented in this thesis can be applied to them as well.

2.1.3.3 Search Strategies in AFL

Search strategies have been the implicit and explicit focus of recent work on AFL. These papers are primarily relevant to the work in Chapter 4 which likewise modifies the search strategy of AFL. For reference, AFL's mechanics can be found in Subsection 2.1.1.

Böhme *et al.* propose a principled diversity prioritisation heuristic for AFL (Böhme et al., 2017b). The heuristic assigns a fuzzing budget that is proportional to the inverse of the frequency of the path's occurrence. In other words, candidate solutions that yield rarer execution traces get a higher fitness. This strategy results in a more diverse distribution of exercised paths and consequently higher path coverage than the baseline AFL. The intuition behind this work is wonderfully elegant: simply aim to make the distribution of frequency of occurrence of paths uniform – a quintessential instantiation of diversity.

The control flow analysis aspect of Vuzzer (Rawat et al., 2017) (introduced above, in Subsubsection 2.1.3.2) ought to be mentioned here again. The control flow analysis is used to calculate the probability of a path being taken, given random input generation. To increase the likelihood of rare paths being taken, they are given higher priority during fuzzing. Focusing on specific regions of input and the altered prioritisation yield better exploration and crash discovery. This work is conceptually important for this thesis as it is an example of a case where an enriched representation allows a targeted strategy to be defined. A similar idea is first explored in Chapter 3 and then implemented in Chapter 4.

Two more works that also identify and point mutation operations towards

specific areas of the input were produced by Rajpal *et al.* (Rajpal et al., 2017) and She *et al.* (She et al., 2018). The details of implementation are different but the principle is the same. First, run an unmodified AFL to gather information on which sections of the input strings are more likely to produce new paths. Then, train an NN to predict the likely path exploration gain of different sections of the input. Finally, direct a modified AFL to mutate regions of the input that are more likely to yield exploration gains.

The approach of these papers is very relevant to the work in Chapter 4 which uses NNs to identify relevant features of candidate solutions, and its output is used as a fitness function to direct a modified version of AFL. The work is different in that its aim is not to exploit the most interesting portions of the input, but to identify a generalisable signal in execution traces and create a gradient for an inter-program property of interest.

There are further examples of targeted strategies implemented for AFL. One is a work by Böhme *et al.* which adjusts the fitness function to target manually selected interesting regions of the source code, such as patch diffs (Böhme et al., 2017a). The results show that a source code line targeting strategy is possible. Another example is a work by Petsios *et al.* (Petsios et al., 2017). Rather than focusing on specific regions of the source code, they aim at discovering issues of algorithmic complexity. This is achieved by giving a greater fitness to executions with higher resource usage. The overhead of this method is too large to be practical, but the work illustrates how an execution property targeting fitness can be implemented with AFL. The directed fuzzing approach proposed in this thesis shares the concept of a property targeting search with these two papers. The definition of the target is fundamentally different however: it is defined by the trained NN, rather than by a specified code location or a human chosen property of an execution.

2.2 Neural Networks

This section presents the relevant background on neural networks (NN). Save for a few exceptions, NNs have hardly been used for aims similar to those of this thesis; for defining search landscapes and generating new program inputs in SBST. Thus most of the work mentioned in this section has served as inspiration, rather than comparable approaches to solving the problems this thesis aims to address. Most NN concepts presented here are described in Deep Learning by Goodfellow *et al.* (Goodfellow et al., 2016), so citations to it are not repeated for each mechanism in this section.

2.2.1 Neural Network Model

NNs are a staple tool in modern machine learning (Haykin, 1994). They aim to learn an approximate mapping of inputs to outputs. The inputs to an NN are known as **features** and outputs are called **labels**, and a single instance of data is an **example**. The labels output by the network are referred to as **predicted** labels while the ground truth labels are known as **true** labels; what the output actually *should* be, according to the dataset. NNs are composed of **neurons** – small constructs that output an **activation function** ϕ applied to a weighted sum of inputs x and weights w, as shown in Equation 2.1.

$$y = \phi(\sum^{n} w_i x_i) \tag{2.1}$$

Neurons are arranged in **layers**. The input and output layers are self-explanatory, and the intermediate layers are called **hidden** layers. The vector of outputs of neurons of a hidden layer is referred to as a **representation** of the data at that point in the network. The term "representation" is unfortunately overloaded with the SBSE term, but the meaning will be hopefully

clear by context.

The structure of individual neurons and their interconnectedness determines the **architecture** of an NN. Although individual neurons are simple, arranging them into a network yields a complex function which is able to approximate a wide range of interesting non-linear functions (Csáji, 2001). To this end, for NNs deeper than one level, their activation functions need to be **non-linear** (Goodfellow et al., 2016). The de facto default non-linear activation function is the **Rectified Linear Unit (ReLu)**: $f(x) = \max(0, x)$. It has been shown to significantly outperform other non-linear activation functions for deep NN architectures (Glorot et al., 2011).

Depending on the nature of the output, an NN can be either a **regressor** or a **classifier** (Google, 2018). When labels are numeric, it is called a regressor. When labels are categorical, the model is a classifier. Categorical variables like "blue" and "red" cannot be fed to an NN directly as NNs operate on numeric values. Categorical variables therefore have to be encoded, i.e. mapped to numeric integer values (although practically, even integers are still represented as floats in the NNs). The collection of integer values representing the classes is called a **dictionary**.

When categories are non-orderable, as they often are, integers need to be encoded as **one-hot** vectors; a vector of length $|dictionary|$ containing all zeros, with a single 1.0 at the index of the integer. Why is this encoding necessary for non-orderable categorical data? Imagine representing the letters "a", "b" and "c" as numbers $1, 2, 3$. To an NN this means that "a" is more similar to "b" than to "c", which is nonsensical, unless there is a reason to use lexicographic ordering. Representing letters as one-hot encodings however does not impose an order, so the NN is not "confused" into interpreting them as being ordered.

Since the numeric values of a one-hot encoding are between 0.0 and 1.0, an appropriate activation function is needed on the output of a classifier –

the **softmax** function. The softmax is simply a probability density function normalised over the exponents of a layer's outputs, as shown in Equation 2.2. For instance, true and predicted labels for a three class classifier NN could be `<1.0, 0.0, 0.0>` and `<0.6, 0.2, 0.2>` respectively.

$$softmax(x)_i = \frac{e^{x_i}}{\sum_{j=1}^{K} e^{x_j}} \tag{2.2}$$

2.2.2 Training

Three disjoint datasets are typically used in designing and implementing an NN: **training**, **development (dev)** and **testing**. The training dataset is used to train a model. Dev is an auxiliary dataset used during model design and training phases, for instance to define an early stopping condition. If the performance of a model starts to deteriorate on the dev set, while improving on the train set, this is a sign of overfitting (explained below) and training should be stopped. Once the model is trained, the test set is used to evaluate the performance of the model on previously unseen data – how well the model **generalises**.

An NN is trained by using the error signal given by a **loss function**. A loss function is a measure of error between the true label and the predicted label. The two loss functions (or their variants) used in this thesis are **Mean Squared Error (MSE)** between and **Cross-Entropy (CE)**, for numeric and categorical labels respectively. MSE and CE are shown in Equations 2.3 and 2.4, where y is the true label, $\hat{y}$ is the prediction, p is the probability that observation[1] o belongs to the c class, out of M total classes. These are then averaged across all N datapoints, indexed by i (amended in the formulas for clarity).

[1]Not to be confused with observations in the SBSE sense.

$$MSE = \frac{1}{N}\sum_{i}^{N}(\hat{y} - y)^2 \tag{2.3}$$

$$CE = -\frac{1}{N}\sum_{i}^{N}\sum_{c}^{M} y_{o,c}\log(p_{o,c}) \tag{2.4}$$

The gradient of the loss is calculated at output, and then passed through the network by a method known as **backpropagation** (Rumelhart et al., 1986) – a family of algorithms that exploit the chain rule to adjust the weights of an NN with respect to the loss. At each layer, the weights are adjusted so that the layer's output moves along its gradient to reduce the error.

The mechanism that governs the direction, rate and schedule of steps towards an optimum is called an **optimiser**. In addition to various parameters, optimisers' steps are determined by the error signal with respect to the error gradients; optimisers adjust a network's parameters in a direction which reduces the error. There are various optimisers, each with their own drawbacks and benefits, including the ability to train on a particular dataset to begin with, the speed of training, as well as stability. The two optimisers used throughout the empirical work in this thesis are RMSProp (Tieleman and Hinton, 2012) and Adam (Kingma and Ba, 2014). An explanation of their inner workings is not central to this thesis, suffice it to say that they are standard in modern NN work.

A common problem for NNs is **overfitting**. This is an effect where the network learns "too well". Essentially it memorises the training dataset. As a consequence, the NN predicts the training dataset elements very well, but fails to generalise.

There are various methods for addressing overfitting, two of which are used in this thesis. The first is **dropout** (Srivastava et al., 2014). During training,

a portion of neurons is randomly turned off. The network is thus forced to really learn the data patterns into its weights, rather than just memorising the datapoints, which yields better generalisation. The other technique is called **batch normalisation (BN)** (Ioffe and Szegedy, 2015). In simple terms, BN transforms the data between intermediate layers of the NN so that it follows a normal distribution. In effect, this keeps the internal states of an NN regularised, i.e. normalised. The reason why BN actually works is not fully understood, and is the subject of ongoing debate in the ML community, e.g. (Santurkar et al., 2018). It has been shown to work in practice however, both for mitigating overfitting and accelerating training.

Symmetrically with overfitting, NNs may also suffer from **underfitting**. In this case the network does not manage to approximate the training data – not to mention test data. Unlike overfitting, which is indicative of a problem in the training process, underfitting is a sign of a more fundamental issue with the NN. One obvious cause for underfitting is insufficient capacity (i.e. depth and width) of a network. In this case the NN's structure simply does not allow it to approximate the relationship between features of inputs and outputs correctly.

On the other hand, the architecture may be *too* strong. In deep architectures with many layers, the error signal used to drive backpropagation may disappear, in an effect known as the **vanishing gradient** problem. Modern optimisers and BN are ways for alleviating underfitting. Another method used in this thesis is the **residual block** (He et al., 2016a) – a shortcut between layers. The gradient signal is enhanced by an identity layer shortcut (pictured in Figure 2.2). This allows for the error signal to propagate to earlier layers of an NN, since it is not only passed through the weight layers, but also directly via the shortcut (*identity* in Figure 2.2). As a consequence, deeper NN architectures can be trained.

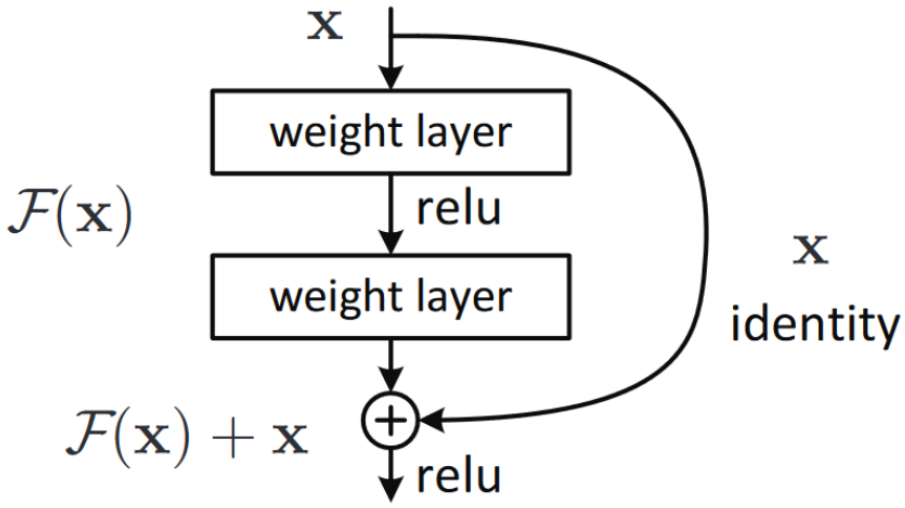


Figure 2.2: A Residual Block is a method for alleviating the vanishing gradient problem. The backpropagation error signal is enhanced with an identity bypass connection. Image source (Sahoo, 2018)

2.2.3 Inference

Once a network has been trained, it is used for predicting the labels of previously unseen data. This stage is known as **inference**. The quality of an NN is gauged by its ability to perform inference. Regressors can be evaluated simply by the numeric error, e.g. squared, absolute or percentage error. For evaluating classifiers, there are many measures, but they ultimately rely on notions of **true** (**T**) and **false** (**F**), **positives** (**P**) and **negatives** (**N**). A positive and negative refer to the predicted label of a class; whether the model's output was 1 or 0. True and false mean whether the output matched the true label. For instance, the **accuracy** (a common measure of performance) of a classifier is the sum of correct predictions over the total number of examples: $acc = (TP + TN)/(TP + TN + FP + FN)$.

A problem with accuracy as a measure of a model's performance is that it does not consider class sizes. For instance, if the proportion of class "B" and

class “A” in a test dataset is 1:9, a classifier would achieve an accuracy of 0.9 by always guessing “A”.

The **Receiver Operator Characteristic (ROC)** measure addresses this issue. The ROC is a plot of the false positive versus the true positive rate of a binary classifier. Its main benefit over the use of accuracy is that it is independent of the balance of labels (Cortes and Mohri, 2004; Huang and Ling, 2005). The Area Under Curve (AUC) of the ROC is a class size independent measure of a classifier's performance. Since class sizes are normalised, a degenerate classifier that always guesses the larger label achieves an AUC of 0.5. This is a more honest representation of a model's performance and it is used for evaluating classifiers in this thesis.

2.2.4 Embeddings

Data represented in NNs is always implicitly numerically ordered, which can be both a benefit and a drawback. For instance, working with pixel hues in images is relatively straightforward, since the values are continuous. Discrete, unorderable values like characters in a string are trickier however. For instance, “bat” is clearly *not* almost the same as “cat” although the two words only differ by a single (alphabetically adjacent) character – “cat” $\neq$ “bat” $+1.0$ (Goodfellow, 2016). On the other hand, in continuous variables such as pixel hue values, a slight offset does not dramatically change the meaning: a cat can be still identified even if the image is slightly blurry.

Because of this ordered, numeric representation, discrete values also need to be encoded as reals; as n-dimensional real vectors. These real vectors are called **embeddings**. Values for embeddings cannot be assigned arbitrarily, as that would, once again, imply an ordering. Instead, embeddings can be constructed manually, or much like any other representations within NNs they can be learned.

Embeddings are an inherent feature of NNs. The following four ought to be pointed out, as they keep coming up throughout this thesis. The first is the one-hot encoding (mentioned above in Subsection 2.2.1) – a degenerate embedding which simply expresses discrete values as a vector of reals with a single 1.0 and all other values of 0.0.

The second embedding is the non-degenerate variant of one-hot: a probability density function (PDF). These PDFs are the predicted labels produced by classifiers. In the above example, a predicted label of `<0.6, 0.2, 0.2>` corresponds to highest likelihood of an output being `a`.

As concrete example of representing discrete classes as probabilities, consider a model trained to approximate the function $Y = X^2$, with Y as inputs and X as outputs. A single value of Y will map to two values of X, as shown in Figure 2.3. The output is categorical (i.e. each value of X is a class), so the height of the peak in the plot is probability. Given an input value $y = 4$, the NN produces a probability estimate of $p(\hat{x} = 2.0) = 0.49, p(\hat{x} = -2.0) = 0.48, ...$[2].

The embedding of discrete data into PDFs is exploited in Chapter 5. The probabilities a classifier produces on its output are used for generating strings by sampling the PDF.

The third kind of embedding is contextual, like in **word2vec** (Mikolov et al., 2013). Word2vec is a seminal paper in which Mikolov *et al.* embed words into a space of reals through the context of their occurrence in a corpus of text. This allows algebraic operations over the meanings of words, a famous one being $king - man + woman = queen$. This contextual embedding is now part of Keras (Chollet et al., 2018) and Tensorflow libraries (Abadi et al., 2015) and used in this thesis to embed execution trace events.

Last but not least is the **autoencoder**, described in a separate subsection

[2]The values are just illustrative, not an exact correspondence to the plot.

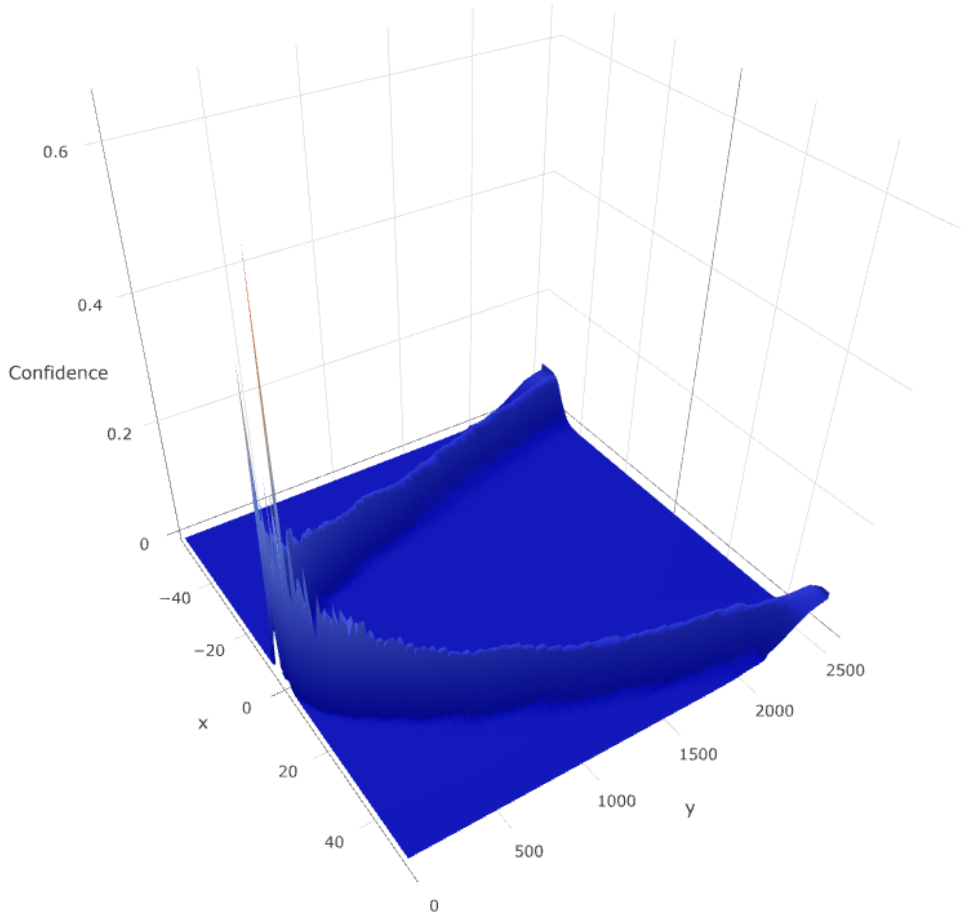


Figure 2.3: The output space of a Generative Neural Network that approximates the $y = x^2$ function. Given an x input, the network learns to generate values for y, with confidence represented by the height of the plane.

below.

Before describing autoencoders, a conceptual summary point ought to be made. The mapping of discrete values into embeddings (a vector space) is a fundamental property of NNs. As a consequence of training by backpropagation, intermediate representations (i.e. embeddings) of the NN "take on properties that make the [classification] task easier" (Goodfellow et al., 2016). This implies that numerical proximity in NNs' representations defines a similarity of features. Whether a particular embedding – an arbitrary embedding of intermediate representations – is useful, is not a given. The usefulness of an embedding needs to be evaluated empirically, with respect to a specific purpose. In this thesis, it is argued, that embeddings can be used as smooth, continuous and meaningful search landscapes. For targeted search strategies these are simply the outputs of classifiers or regressors, while for diversification strategies they are the embeddings produced by autoencoders.

2.2.5 Autoencoders

An autoencoder (AE) is an NN which attempts to reconstruct its inputs at the outputs, while arranging the datapoints into an n-dimensional encoding – a **latent space**. An AE is composed of an **encoder** and a **decoder**, with an intermediate encoding layer of a smaller dimensionality between them, as shown in Figure 2.4 (Dertat, 2017). The latent space is an embedding in which datapoints are arranged by similarity of their most salient features; those that are most useful for distinguishing one datapoint from another (Goodfellow et al., 2016).

What gives an AE this ability to identify salient features is its hourglass shape. The encoding layer is of a much lesser dimensionality than the original data so it is unable to represent datapoints perfectly. Instead, it only encodes the features that distinguish one datapoint from another, while the redundancy (i.e. features that are common in the corpus) gets encoded into

the structure of the decoder. Thus an autoencoder performs a dimensionality reduction into the most discriminating features of the data, with meaningful locality in the latent space.

Much like NNs in general, AEs do not naturally map particular features to distinct axes. It is the space itself – the whole representation, rather than individual cells (i.e. axes) that embeds the semantic information of data (Szegedy et al., 2013), and the distance in the latent space is a measure of similarity of features.

A classic illustration is an AE trained on the MNIST hand written digit dataset (LeCun and Cortes, 2017; Chollet, 2017b; Tiao, 2017). The interpolation space and the latent space of that AE are shown in Figure 2.5. An AE is trained on images of 10000 hand drawn digits and its 2-dimensional latent space forms is a representation of the most salient features of those images. The image on the RH side shows the dataset encoded into a 2-d latent space. The colours represent the actual digits of the hand written images. The image clearly shows that datapoints representing the same image tend to cluster in distinct regions of the latent space.

To be clear, the AE is not trained on the values of the digits – only on the hand drawn images. Thus the clusters are formed because of the *features* that hand written digits happen to share. The LH image shows the images generated by an interpolation of the latent space. These two images exemplify the nature of AEs: they arrange unlabelled, unordered data by their features and the resulting embedding captures the semantic information of the dataset.

The flavour of an autoencoder used in the above work is called the Variational Autoencoder (VAE) (Kingma and Welling, 2013). The datapoints in a VAE tend to be more evenly spread across the latent space and interpolation between them is smoother than in a regular autoencoder.

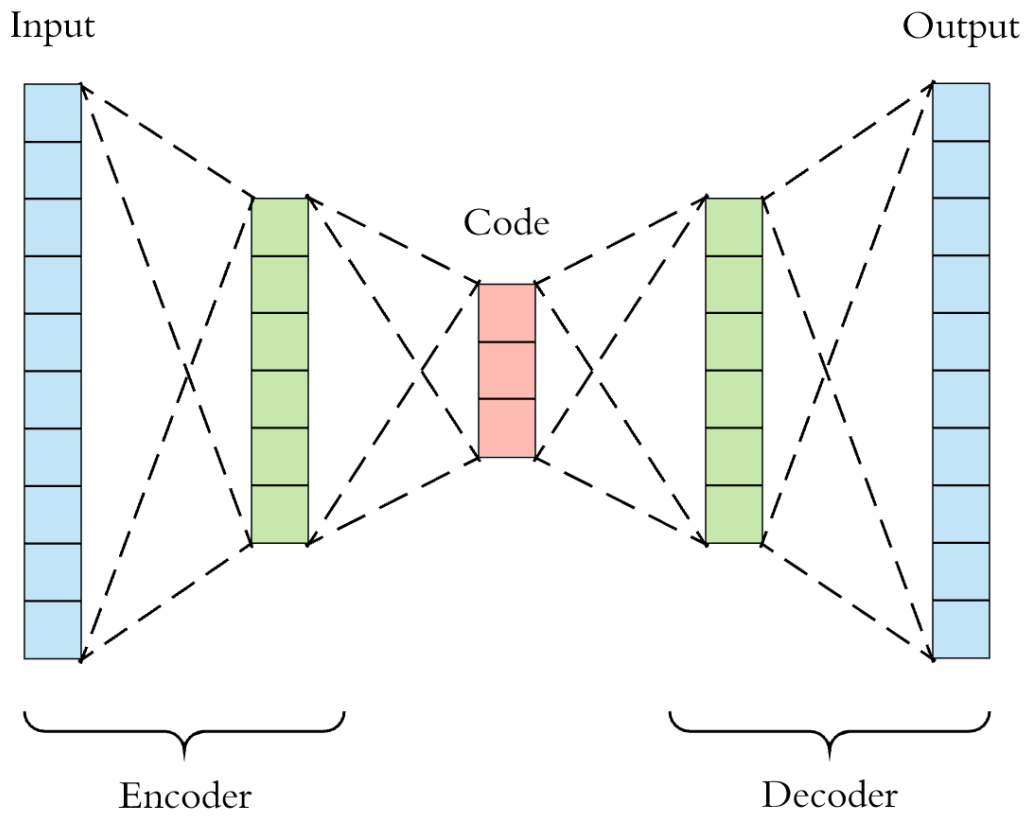


Figure 2.4: An illustrative structure of an autoencoder. Each column is a layer of the network, with individual boxes representing neurons. An autoencoder is trained with x as both input (fed into the network from the LH side on this illustration) and output (labelled x' on the output – the RH side). The z layer is the encoded latent representation. Image source (Dertat, 2017).

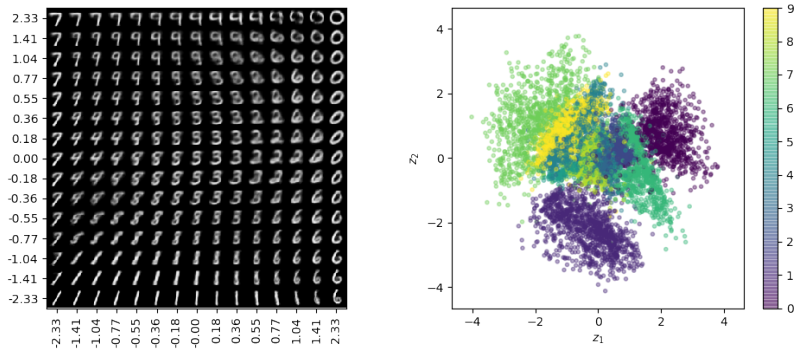


Figure 2.5: Latent space representation of the MNIST hand written digits dataset (LeCun and Cortes, 2017), used to train an autoencoder model. The right hand image shows the latent space itself, with the two latent space axes z_1 and z_2. The colours correspond to the actual digits of the hand drawn images. Labels clearly cluster in regions of the latent space due to shared features of the images. The left hand image is an interpolation of the latent space which generates a images from points in the latent space. The images show how the latent space encodes the dataset into a continuous embedding of features.

The essential detail of a VAE is in the structure of its encoding layer. Rather than encoding datapoints as single real number values, they are instead encoded as tuples of mean and variance (μ, σ^2). A VAE's encoding layer is composed of densely connected μ and σ^2 layers. In addition, the encoding layer includes a source of Gaussian noise. The construction of the encoding layer z is shown in Equation 2.5. When the VAE is queried for an encoding of a trace, it is μ layer's output that is fetched. That is, σ^2 and z layers are only used during training to give the latent space the desired characteristics of being evenly spread around origin, but the actual encoding is the value μ. The $\mathcal{N}$ term is a Gaussian noise drawn from a standard normal distribution.

$$z = \mu + e^{0.5\sigma^2} * \mathcal{N}(0, 1) \tag{2.5}$$

The loss function of the VAE is composed of two terms, as shown in Equation 2.6. The first term is the latent encoding regularisation loss and the second is the reconstruction loss. More technically, the loss components are the Kullback-Leibler divergence of the approximate (q) to the true (p) posterior distributions, given datapoints and model parameters, and the *lower bound* on the marginal likelihood of datapoint i respectively (Kingma and Welling, 2013). The purpose of the latter is to get the VAE to actually encode the data – whether categorical or continuous. This term is minimised when the VAE learns to reconstruct the input data on the output perfectly. The former term forces the latent space to approximate a normal distribution; it gives it a convenient shape. In the equation, the approximate posterior term q is the actual distribution of z and σ layers in the latent space, while p is the standard normal distribution. This term is minimised when the latent space encoding follows a standard normal distribution $\mathcal{N}(0, 1)$.

$$loss_{VAE} = D_{KL}(q_\phi(z|x^{(i)})\|p_\theta(z|x^{(i)})) + \mathcal{L}(\theta, \phi; x^{(i)}) \tag{2.6}$$

The main use for autoencoders in this thesis is to produce an embedding of execution traces. The benefit of encoding traces in this way is that it imposes an order relation on discrete, seemingly unorderable datapoints. Furthermore, the space is continuous and smooth by design. These are characteristics of a good search landscape (Harman and Clark, 2004), as described in Subsubsection 2.1.2.1.

Since execution traces are the representation of program behaviour, the latent space, in fact, constitutes a representation of program behaviours. A key proposition of this thesis is that since the aim of diversification strategies is to explore *diverse behaviours*, this latent space is what diversity ought to be defined over. The idea is explored in Chapters 3 and 5.

2.2.6 Recurrent Neural Networks

Some of the data considered in this thesis is not only discrete, but also dependent on temporal context: the order of elements matters in a sequence. For instance, the order of characters in strings is important. So is the sequence of events in an execution trace.

Recurrent Neural Networks (RNN) are a type of NNs used for processing sequence data (Goodfellow et al., 2016). They have been used in various NLP tasks like language modelling (Sundermeyer et al., 2012), sentence generation (Bowman et al., 2015), as well as voice recognition (Graves et al., 2013).

The key feature of RNNs is their dynamic nature. RNN cells "unroll" as an input sequence is consumed. Each cell within a layer is connected to itself, as illustrated in Figure 2.6. This way, as a sequence is consumed, each element of an input sequence affects the state of the cell. This gives recurrent

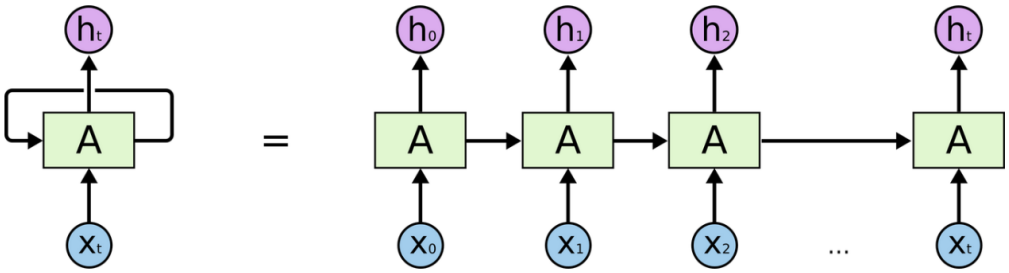


Figure 2.6: A Recurrent Neural Network (RNN) cell is dynamically unrolled to process a sequence of input elements. The internal output of a cell is passed to subsequent cells, so the network learns to represent the sequence in the final state. Image source (Olah, 2015).

architectures a built-in "memory". The final state thus encodes the input sequence into a collapsed form.

Although in theory an RNN can be trained on arbitrarily long sequences, this is not practically tractable due to the vanishing gradient problem (Melis et al., 2017). Long-Short Term Memory (LSTM) cells (Hochreiter and Schmidhuber, 1997) or Gated Recurrent Unit (GRU) cells (Chung et al., 2014) are recurrent cell structures that were proposed to alleviate this problem. Rather than simply passing the output of intermediate cells to subsequent ones, they also pass the state of the previous cell. Implementation details of these cells are not critical here, suffice it to say they propagate a stronger signal which allows them to handle longer sequences than vanilla RNNs. Even with these techniques however, RNN based architectures are not applicable to very long sequences.

2.2.7 Convolutional Neural Networks

A **Convolutional Neural Network (CNN)** is an NN design that can capture positional dependencies in multi-dimensional data. CNNs have been successfully used in a wide range of applications, including image, video, and

sentence classification (Krizhevsky et al., 2012; Karpathy et al., 2014; Kim, 2014; Wang et al., 2012; Kalchbrenner et al., 2014). In sequences, they can handle much longer dependencies than RNNs and therefore outperform them in sequence modelling tasks (Bai et al., 2018).

CNNs work by sliding a stack of **filters** (also sometimes called **kernels**) of a fixed size across an input, shown on the LH image in Figure 2.8 (Karpathy, 2016). The filters perform a convolution operation on the inputs (Goodfellow et al., 2016). The convolution encodes several adjacent elements of the input into a single output vector thereby capturing local dependencies. The collection of the series of outputs of a convolutional layer forms an abstract representation of the input data.

Stacking convolutional layers creates a deep CNN architecture where each layer learns to identify and encode features of the original input at different levels of abstraction. Figure 2.7 (Krizhevsky et al., 2012) shows the outputs of intermediate layers of a CNN, where each square is an output of an individual filter. The patterns correspond to visual features each filter learns, e.g. horizontal lines. Intermediate representations of CNNs are not inspected in this thesis. The purpose of presenting these images here is to illustrate how different features (i.e. abstractions) of data are identified and learned by filters at different layers of a CNN. This is the essence of the why *deep* NN architectures are so powerful: it is the collection of abstractions that allows them to learn complex patters in data.

The filters can be slid by more than a single step at a time with **strided** convolutions. Strided convolutions can be used for dimensionality reduction. For instance, applying a convolution with $stride = 2$ to an image with a width, a height and a depth of 3 (RGB channels) of $64, 64, 3$ will yield an output of size $32, 32, x$. The x is a manual choice for the number of filters and is typically larger than the depth of input. Intuitively, this increase is necessary for "describing the same data with fewer, *but more expressive*

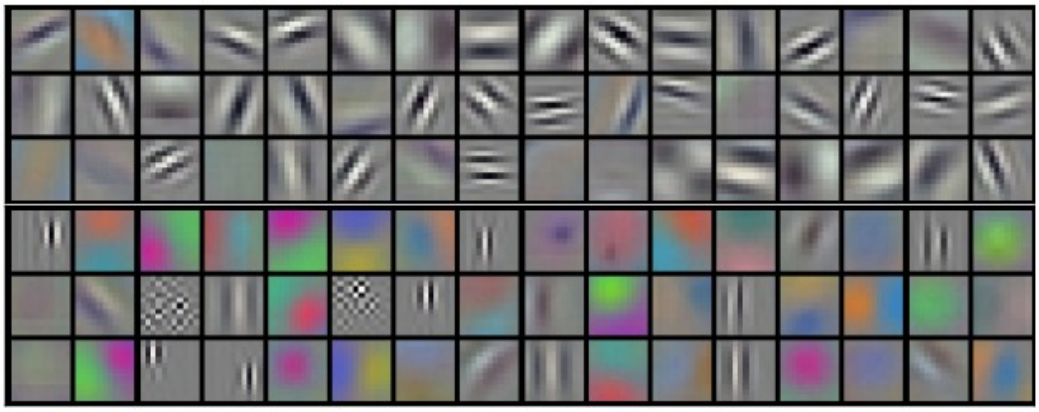


Figure 2.7: Outputs of individual filters of a CNN, where each square shows the output of a single filter. The different patterns show how each filter learns to identify and encode different features of an image. The combination of filters allows a CNN to represent data at various levels of abstraction. Image source (Krizhevsky et al., 2012).

words". Stacking strided convolutional layers can be used to produce an encoding of the whole image, as seen in Figure 2.8, RH side (Karpathy, 2016).

In this thesis, CNNs are used to process very long sequence data so that it is then amenable to RNNs. Combining CNNs with RNNs in this way was done by Donahue *et al.* for visual recognition and description (Donahue et al., 2015) which serves as an inspiration for the architecture presented in Chapter 3. In addition, CNNs are used as generators in Chapter 5, building on techniques described next.

2.2.8 Generative Neural Networks

Generative Neural Networks (GNN) are a class of NNs that produce new data. They have been used in various contexts, for instance in text (Kawthekar et al., 2017), image (Kingma and Welling, 2013; Radford et al., 2015) and audio generation (Van Den Oord et al., 2016).

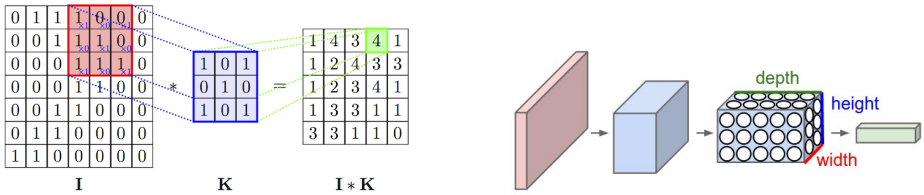


Figure 2.8: In a Convolutional Neural Network (CNN), a filter slides across an input sequence and outputs the product of a convolution operation. This captures and abstracts the short range dependency of inputs. Stacking CNN layers produces a high level abstraction of the input. Image source (Karpathy, 2016).

There are various GNN architectures, but the basic, unifying intuition is that they are trained in reverse of regular classifiers. That is, they are trained to model the output-to-input mapping, $gnn : Y \mapsto X$, as in the above example shown in Figure 2.3.

Much like regular NNs, GNNs are first trained, but rather than inference, their second phase is called **generation**. This is equivalent to inference in that the GNN outputs a value $\hat{x}$ given some input y. The inputs to a GNN are sometimes denoted z, reflecting the architecture of some GNNs (e.g. DCGAN (Radford et al., 2015)) that take a Gaussian noise as input. The input to a GNN is sometimes referred to as the **priming**.

The three works of relevance to the architectures used in this thesis, are the Deconvolutional Generative Adversarial Network (DCGAN) (Radford et al., 2015), a generative RNN by Sutskever *et al.* (Sutskever et al., 2011) and the variational autoencoder by Kingma *et al.* (Kingma and Welling, 2013).

In DCGAN, two neural networks – a generator and a discriminator – are pitched against each other. The generator takes Gaussian noise as input, passes it through a stack of convolutions and produces an image on the output. The discriminator is trained to predict whether an image is real or fake (generated). While the discriminator learns to be ever more effective

in telling fake images from real ones, the generator attempts to produce images that fool the discriminator into believing they are real. This tandem of networks ends up creating realistic images. A high level structure of a DCGAN is shown in Figure 2.9 (Perarnau, 2017).

DCGAN is a central inspiration for the Deconvolutional Generative Adversarial Fuzzer (DCGAF) architecture proposed in Chapter 5. DCGAN cannot itself be directly used for generating new program inputs, as its discriminator requires a dataset of real samples. Having a large, representative dataset of real samples defeats the purpose of a generative model, since that dataset could itself be used to test a program. DCGAF is of a fundamentally different nature however. Its discriminator relies on a notion of diversity, rather than true / false labels of a training dataset, and its generator does not require training data either because it (with the discriminator's aid) produces its own.

Another GNN model is one based on the VAE. It is simply the decoder portion of a complete VAE which was described in Subsection 2.2.5. The decoder is fed Gaussian noise and it produces an image on the output. An example is shown in the LH image of Figure 2.5. Each image is generated from a 2-d priming vector of the latent space coordinates. This architecture is not used in any final implementations in this thesis, but it had been considered, and it is therefore mentioned here for completeness.

The final GNN of interest is based on an RNN (Sutskever et al., 2011). This model basically predicts the next token given a sequence. First, the training corpus is parsed with a sliding window which collects substrings several characters long (n-grams), and the individual characters that follow them, e.g. ("abc", "d"); ("bcd", "e"), ("cde", "f") and so on. The input to the RNN is the n-gram and the true labels are the characters that follow them. The RNN thus learns to predict a character given an n-gram. During generation, the predicted character is appended onto the input string and the

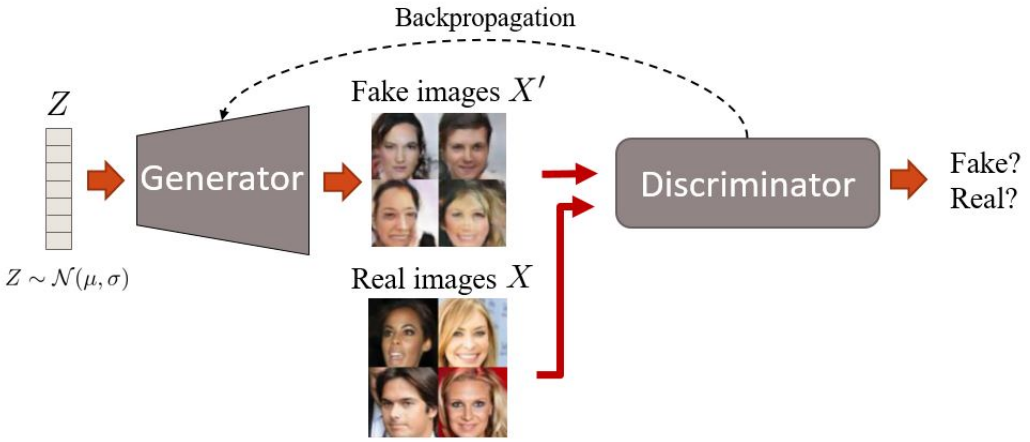


Figure 2.9: High Level Structure of DCGAN, the Deconvolutional Generative Adversarial Network (Radford et al., 2015). The model is composed of a generator and a discriminator. The generator takes a noise vector as input and produces an image using a stack of deconvolutional layers. The image is then fed to a discriminator for evaluation. The discriminator, in turn, is trained to determine whether an image is real or fake. The discriminator becomes better at telling apart real images from fake ones, and this ability forms the loss function for the generator: the generator attempts to create (fake) images that the discriminator labels as real. This tandem of NNs ends up generating ever more realistic images. DCGAN is a major inspiration for the Deconvolutional Generative Adversarial Fuzzer (DCGAF), presented in Chapter 5. Unlike DCGAF, DCGAN still requires a training dataset of real data samples. This makes its immediate applicability to test case generation limited. Image credit (Perarnau, 2017).

leading character of the resulting string is popped, e.g. ($y =$"pqr", $\hat{x} =$"s") $\rightarrow$ "pqrs" $\rightarrow$ "qrs"; ($y =$"qrs", $\hat{x} =$"t") $\rightarrow$ "qrst" $\rightarrow$ "rst". This architecture was briefly considered for DCGAF in Chapter 5 but quickly discarded due to being very very slow to train and to generate new strings, compared to CNN based models. Furthermore, due to the vanishing gradient, these models are unable to generate very long sequences.

Combinations of GNN architectures are also possible. An RNN based GNN was used in combination with a VAE for sentence generation by Bowman *et al.* (Bowman et al., 2015). This paper is a rare example of particular interest for this thesis as it attempts to produce discrete sequences from a continuous space. It demonstrates how a latent space can be used as priming for string generation, and that an interpolation of the latent space produces a gradual change in the syntax of generated strings. The same idea of combining an embedding of an autoencoder with a GNN, albeit with a different implementation (VAE + CNN, rather than VAE + RNN), forms the backbone of the architecture of the work in Chapter 5.

2.3 Machine Learning in Software Engineering

"Machine learning" (ML) is a broad term used to refer to anything from K-means clustering to deep neural networks. Although examples of use of ML are numerous (e.g. (Anderson et al., 1995; Vanmali et al., 2002; Mao et al., 2006; Briand et al., 2007, 2008; Ben Abdessalem et al., 2016) and survey (Durelli et al., 2019)), this section focuses on the most relevant applications of ML, namely processing of execution traces and the generation of program inputs. The first purpose is prevalent throughout all subsequent chapters of the thesis, while generative neural networks are used in Chapter 5.

2.3.1 Program Profiling

One of the use cases of ML in this thesis is the processing of representations whose analysis would otherwise be non-trivial. There is work using ML for this purpose, albeit not for the end goal of constructing fitness functions.

Control flow graph traces have been used to classify program behaviour (Bowring et al., 2004). The work uses hierarchical clustering on exercised branches, method calls and def-use pairs to classify execution traces. It also goes on to present a use case classification of test inputs for designing a testing strategy. Another example of classifying and detecting behaviours from program traces was presented by Lo *et al.* (Lo et al., 2009). Sequences of events in execution traces are used to predict failures. These works are examples of how ML can be used to create a space for representing program behaviour based on a collection of execution traces. The concept is similar to that proposed in this thesis.

There are also examples of program profiling with ML to identify properties of interest. Pacheco shows how execution traces can be used to define clusters of program behaviours, and to identify executions that lead to faults (Pacheco and Ernst, 2005). This information is then used to optimise test suites by sampling executions from different clusters. The work also implements tools for automated oracle and test input generation. This is an example of how an ML technique is used to identify an execution property, arranging execution into clusters and then sampling from those clusters to improve diversity. On a high level, these ideas are close to those presented in this thesis.

ML tools have been used in an instantiation of program behaviour analysis called **anomaly detection**. The purpose of anomaly detection, as the name suggests, is to identify behaviours that are atypical. In order for anomalous behaviours to be identified, a definition of typical behaviour is required.

For example, Forrest *et al.* present how sequences of system calls can be used to define normal and anomalous behaviour. The results show that abnormal behaviour of Unix processes, such as intrusions, can be identified by observing their system calls and matching them against a database of previously observed normal traces (Forrest et al., 1996). This work suggests that a pattern of calls is a meaningful representation with respect to identifying abnormal behaviour. A comprehensive survey on anomaly detection, where ML is often used, the reader is referred to a survey compiled by Chandola *et al.* (Chandola et al., 2009).

In this thesis, NNs are used to define abstract notions similar to "anomalous". For instance, in the work on targeted search strategies in Chapter 4, the fitness produced by the NN is the "suspiciousness" of an execution crash with respect to a property of interest. As for diversification driven strategies, the locality in the latent space as a representation of program behaviour, can be interpreted as "unusualness". This idea is first introduced in Chapter 3 and then implemented in Chapter 5.

2.3.2 GNNs in Software Engineering

Although GNNs have been used for image, sound and even text generation as outlined in Subsection 2.2.8, they appear to not have yet properly caught on in SBST. They are used in Chapter 5 of this thesis, and hopefully the paper on which the chapter is based will spark further interest in the community.

Their use in software engineering is not non-existent however. Godefroid *et al.* use an RNN based GNN as a fuzzer (Godefroid et al., 2017). First, the network is trained on a corpus of PDF snippets, to approximate the grammar of PDF files. It is then used to generate new PDF snippets to test the PDF reader functionality of Microsoft's Edge browser. The architecture is quite rudimentary in ML terms, dating back to 2011 (Sutskever et al., 2011). No faults were actually discovered, which is unsurprising, considering that the

Edge browser is well tested to begin with. Nonetheless, as a proof of concept, this work demonstrated that a GNN can be used to produce string inputs for a program in a testing setting.

GNNs have also been used to augment AFL. Although AFL can begin fuzzing from scratch (rather, from a single empty dummy file), the quality of initial seeds can drastically improve its performance. A number of recent papers have used RNN based GNNs to produce initial seeds for AFL (Wang et al., 2017; Lv et al., 2018). The GNNs were trained on *XSLT*, *XML*, *mp3*, *bmp* and *flv* file formats, and then used to produce corpora of initial seeds for AFL. The former paper uses a web crawler to collect its training data, while the latter uses existing corpora. They both report that using a GNN to synthesise a corpus of initial seeds improves AFL's performance.

The main interest of these papers with respect to this thesis is not their implementation, nor their use case. Instead, it is the implication that using GNNs for mere seed generation is a much simpler problem than their direct use as fuzzers, like in Godefroid's work above. This is because training a GNN to learn an input syntax, while also maintaining an appropriate degree of randomness, is a very complicated task. However, a GNN can be used synergistically, in conjunction with another tool like AFL. The GNN is used to only produce approximately interesting inputs, while AFL then uses its internal search operators to tweak the seeds. This concept of using a GNN and AFL together, although considered, was not eventually implemented in the work of this thesis.

The topic of Chapter 5 is the Deconvolutional Generative Adversarial Fuzzer (DCGAF), which is based on a GNN. The framework is principally different from those described above in several ways however. First, it is based on a CNN, rather than an RNN, which makes it much faster to train and generate inputs. Second, it does not require a training dataset as it produces its own. Third, it is grammar agnostic. Finally, being a fuzzer, it does indeed manage

to find crashes, unlike the GNN in Godefroid's work.

Chapter 3

Producing Search Landscapes for SBST with Neural Networks

This chapter presents an approach for constructing meaningful and convenient fitness landscapes using neural networks – for targeted and diversification strategies alike. Whether one follows a targeted or a diversity-driven strategy, a search landscape should be large, continuous, easy to construct and representative of the underlying property of interest (Harman and Clark, 2004). Constructing such a landscape is not a trivial task often requiring a significant manual effort by an expert (Shepperd, 1995). The techniques introduced in this work illustrate how these issues can be addressed with NNs.

3.1 Overview

3.1.1 Property Targeting Search Landscape

Any fitness landscape should ideally possess the desirable properties of continuity, size and lack of local optima (as discussed in Subsubsection 2.1.2.1). A fitness function for a property targeting search strategy also needs to indicate a "proximity" of a candidate solution to a property of interest. The fitness function therefore needs to be representative of the property of interest, i.e. it needs to meet the representation condition.

Consider an example where a tester aims to exercise a specific program point behind a numeric conditional statement. The numeric difference between the value of a variable and the predicate value of the if statement (branch distance) is the obvious fitness function here (Wegener et al., 2001). In many interesting "needle in a haystack" testing scenarios however, such an easy fitness function does not exist. For instance, a tester is looking for a crash, but the program has not crashed after a thousand executions produced by mutation of an original input. Can we argue that some of those executions are "closer" to a crash and are therefore better candidates for further mutation?

This work proposes that an output of an NN predictor can be interpreted as a fitness for a property targeting strategy. The NN is trained on a corpus of execution traces and various properties of interest, prior to searching. During search, the trained NN is queried to predict an estimate of a property given an execution trace. The outputs of the NN form a search space which is strongly representative of the property of interest. Such a search space could be useful for driving a search towards specific properties of interest.

This approach is conceptually similar to a surrogate fitness function. A surrogate fitness function substitutes the true fitness landscape with an ap-

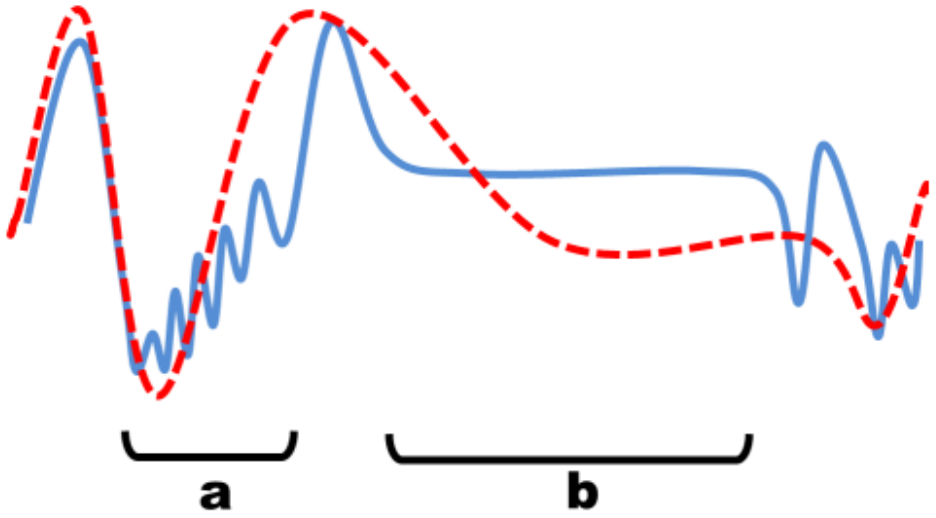


Figure 3.1: A Surrogate Fitness Function. The true fitness landscape (blue) has plateaus and many local optima. The surrogate landscape (dashed red) approximates the true landscape, while mitigating these inconvenient characteristics. Image source (Brownlee et al., 2015)

.

proximation that has beneficial characteristics (Brownlee et al., 2015). The example in Figure 3.1 shows how the use of a surrogate fitness function can ameliorate undesirable characteristics of a fitness landscape. In the work described next, the NN-generated fitness function acts as a replacement for the degenerate true fitness landscape with a large "no crash" plateau. In that sense, it can be seen as a surrogate fitness function.

This work does not evaluate the effectiveness of the fitness function, but simply shows how it can be constructed, and discusses the properties of the landscape. An implementation of such a search is presented further, in Chapter 4. There, the NN is trained on execution traces and crash / no crash labels as properties of interest. So rather than simply observing a "no crash" output, an NN is queried to say that some inputs exhibited a behaviour or

"looks suspiciously like a crash".

3.1.2 Diversity Driven Search Landscape

As discussed in Subsection 2.1.3, diversity is widely accepted as beneficial for testing, although various representations have been proposed as targets for diversification. Perhaps the most common manifestation is code coverage, yet the effectiveness of coverage driven testing strategies has been disputed. This suggests that preferring dissimilarity of candidate solutions as measured by code coverage is not ideal. Regardless of representation, the actual purpose of diversity driven testing is to exercise a maximally diverse range of *behaviours*.

To be able to exercise diverse (i.e. dissimilar) behaviours, given a representation that is thought to be a good abstraction of program behaviour, a notion of similarity is required. The definition of similarity can then be used to drive a search strategy. A similarity measure requires an order relation, which is a difficult task typically requiring an expert's input (Shepperd, 1995). For instance, is "*cat*" < "*dog*"? Lexicographically – yes. By average weight of the animal – usually. By preference as a pet – debatable.

This work proposes a method for constructing a search landscape using autoencoders (see Subsection 2.2.5). The autoencoder arranges data based on features that are most important in distinguishing one datapoint from another. The distance in the latent space is thus a measure of similarity of features. Since datapoints are observations of program behaviour, the landscape is a convenient, continuous representation of behaviours, which defines an order relation and thus a notion of similarity. Importantly, an autoencoder architecture can be applied to arbitrary data formats. This means that a tester is not restricted to any particular representation of execution traces. This work suggests that such a notion of similarity for diversification strategies may prove to be useful for SBST. Chapter 5 implements this idea

of an autoencoder-generated search space for a diversification strategy.

3.1.3 Contributions and Scope

This chapter introduces the approach for building search landscapes for SBST with NNs. This is one of the main contributions of this thesis as a whole. Further chapters build on the techniques first introduced here.

The approach relies on predictor and autoencoder neural networks for property targeting and diversification driven testing strategies respectively. The proof of concept is implemented here using a corpus of small C programs from *Codeflaws*, and several real world applications.

The findings suggest that the landscapes possess a number of useful characteristics. The first is that they are continuous and arbitrarily large. Second, they meet the representation condition. Third, they yield a meaningful order relation to seemingly non-orderable observations. Fourth, the order relation implies a notion of similarity. Lastly, they are created automatically, without analytical effort or domain knowledge.

The ideas presented here form the foundation on which the following chapters are built. Chapter 4 extends and implements the idea of a targeted search space and Chapter 5 uses the search landscape for a diversity driven search strategy. The scope of this chapter is to present the search landscapes themselves, along with an analysis of their characteristics – not to evaluate their effectiveness for discovering properties of interest.

3.2 Datasets and Tools

The experiments for this work explore multiple representations and a range of properties of interest. The properties of interest were collected from a large corpus of programs using Valgrind, and the traces with the PIN and

ftrace instrumentation tools.

3.2.1 Program Corpus

The main corpus used for this work is taken from the *Codeflaws* program repository (Tan et al., 2017). *Codeflaws* is a program repository of 7808 small C programs with lines of code ranging from 1 to 322, along with test cases and automatic fix scripts. Its stated purpose is "comprehensive investigation of the set of repairable defect classes by various program repair tools". The programs in *Codeflaws* are grouped into 40 defect classes, with defect triggering inputs and automatic repair scripts.

The tools used in this thesis require large training datasets, and there were not enough test inputs in the repository to begin with. The corpus was therefore augmented by generating additional inputs via fuzzing with AFL. The corpus used here consisted of a grand total of 349,360 executions across 4775 unique programs. After the work in this chapter, the dataset was reused and further augmented for the work in Chapter 4. AFL found numerous crashes in many of the programs, with a median of 3.09% of inputs per SUT resulting in a crash.

The dataset was divided into training, testing and validation datasets. To ensure that the test dataset was not tainted with training data, the partition was created as follows. The programs in the *Codeflaws* repository are grouped pairwise by defect IDs. That is, each element in the repository is a pair consisting of a faulty and a fixed version of a program. The difference between a fixed and a faulty program tends to be small, e.g. a change of an operator in one line. Having one program from a pair in the training set and the other in the test set would contaminate the training data: the model would get trained on an example which is very similar to one in the test set. Thus only one program in a fixed-faulty pair should be included in the training set.

The dataset split was therefore done on a granularity of defect IDs. For instance, there are seven pairs of programs with a 38-B defect ID. All seven pairs of the 38-B defect would be excluded from the training set and kept for testing. This ensured that the programs in the test set are sufficiently different from the ones in the training set. The number of unique programs and inputs respectively were 3978 and 303,233 for the training set, 617 and 35,829 for the test set, and 180 and 10,298 for the validation set.

Furthermore, four real world applications were used. The first one is *xmllint* from libxml (Veillard, 2019). It processes a string input to determine whether it is valid XML. The second is *Cjpeg* from libjpeg (Lane et al., 1998), used for compressing image files into jpeg format. The third program is *sparse* (Jones, 2019), a lightweight parser for C. Fourth, *cJSON* is a parser for the JSON format.

These programs were chosen because they are open source, they have been used in other academic works, they are sufficiently quick to fuzz, their inputs can be easily interpreted and their input formats are dissimilar. They also take complex inputs that are not readily orderable nor easy to generate. Test input corpora for each of these programs were generated by fuzzing. A larger corpus of real world programs may have been used for a more comprehensive evaluation of the approach, but at this proof of concept stage these four were deemed sufficient.

3.2.2 Properties of Interest

Execution properties of interest were produced by an instrumentation framework for dynamic analysis called Valgrind. Valgrind has been used extensively in academia and industry (Nethercote and Seward, 2007, 2017c). It tracks every instruction as a program executes in a simulated environment. The standard tools that come bundled with the framework include Memcheck, Cachegrind and Callgrind.

Memcheck reports errors related to memory management, such as illegal addresses of memory read and write, conditional statements with uninitialised values, mismatching allocations and frees, as well as leak summaries (Nethercote and Seward, 2017b). In this work, the various memory errors and leaks are used as examples of non-observable properties of interest. They are undesirable, rarely occurring behaviours that an SBST process may attempt to discover.

Specifically, Memcheck's output of illegal reads and writes, use of uninitialised values, definitely lost memory blocks and memory still reachable at the end of execution, are considered. Illegal reads and writes refer to an attempt by a program to read or write "at a place which Memcheck reckons it shouldn't". For instance, this can be an access to a freed memory location. Use of uninitialised values means the use of uninitialised, undefined values, e.g. `int x; printf("%d",x);`. "Definitely lost" blocks means that no pointer to a memory block can be found, which is typically a symptom of a lost pointer, and ought to be corrected. "Still reachable" is a memory block that has not been properly freed at exit. The latter two issues are not necessarily crucial problems but are still considered as examples of properties of interest. Further details of the above items are available on Memcheck's manual pages (Nethercote and Seward, 2017b).

Another tool in the Valgrind framework is Cachegrind (Nethercote and Seward, 2017a). It reports the number of reads, writes and misses on different levels of cache. With its default settings of a simulated cache architecture, the values are instruction cache reads (Ir), first and last level instruction cache read misses (I1mr, ILmr), data cache reads and writes (Dr, Dw), first and last level data cache read misses (D1mr, DLmr), and first and last level data cache write misses (D1mw, DLmw).

As any execution has a numeric value for a cache behaviour, their values are used here as examples of numeric properties which might be the target of

optimisation in SBST – not as rare, undesirable properties. The use case for discovering these properties is e.g. when a cache behaviour is difficult to measure and needs to be approximated from an easily observable trace. The values of cache behaviour properties are effectively unbounded, which makes them inconvenient for neural networks – training on such values is known to become unstable (Salimans and Kingma, 2016). In these experiments they are therefore log-normalised. Not only does this make the values amenable to training an NN, an order-of-magnitude estimate of these values, it is argued, is a reasonably interesting property.

The last tool, Callgrind, tracks all function calls made during an execution (Weidendorfer, 2017). These include both program's internal function calls as well as any library calls. Callgrind call traces are used in Chapter 4 as examples of observable traces used for representation of an SBST task. Here Callgrind is only mentioned as another tool belonging to Valgrind.

3.2.3 PIN Traces

One of the representations of program behaviour in this work is based on the PIN instrumentation framework (Luk et al., 2005). PIN captures machine code instructions from uninstrumented binaries which can yield enormous, but very granular traces. An example of a tiny snippet of a PIN execution trace is shown in Listing 3.1.

```
mov esi, r14d
mov r11d, 0x1
jmp 0x7f3d8c26455e
mov eax, dword ptr [rip+0x20c7c0]
test ah, 0x1
jz 0x7f3d8c264573
or dword ptr [rip+0x20c7dd], 0x4000
test ah, 0x80
jz 0x7f3d8c264582
```

```
or dword ptr [rip+0x20c7ce], 0x8000
cmp dword ptr [rip+0x20c78b], 0x6
jle 0x7f3d8c2645ac
mov eax, 0x7
xor ecx, ecx
```

Listing 3.1: Example snippet of a PIN Execution Trace. The traces are very granular and long, and thus unwieldy. To counter this, only the first instance of a block's execution is recorded. Furthermore, the traces contain literals of memory addresses and data values, which introduces the alpha renaming problem. This makes discovering patterns in these traces very complicated. The issue of alpha renaming is circumvented by only considering op-codes of operations, e.g. simply "mov" instead of the whole first line.

The raw execution traces are inconvenient for two reasons however. First, they are infeasibly large. A single execution of a simple program yields a trace file of size in the order of tens of gigabytes. Second, literal values of instruction arguments become problematic. For instance, the target address in the conditional jump `jle 0x7f3d8c2645ac` is assigned by the memory manager and is not consistent across program executions. It is also not meaningful over executions of different programs; an execution trace with the value `0x7f3d8c2645ac` in program A is not meaningful for program B. This is a major problem known as alpha renaming (Gordon and Melham, 1996).

The above problems are resolved as follows. First, PIN's built in ability to only instrument the first instance of a block execution is used. For instance, a loop body is only recorded the first time it is executed. This reduces the sizes of traces dramatically while maintaining information of the sequence of events. The problem of alpha renaming is ignored by discarding any literal data. Thus `jle 0x7f3d8c2645ac` is only recorded as `jle`. This certainly loses a lot of possibly pertinent information, but attempting to solve alpha

renaming is out of scope of this work. Furthermore, the sequence of op-codes is expected to be sufficiently rich for these experiments.

3.2.4 Ftrace Traces

Another profiling tool used to collect execution traces is ftrace (Rostedt, 2017). It is a native Linux internal system tracer for kernel events. Whenever a program is executed in user space, it interacts with the kernel at a low level. Kernel function call traces were therefore expected to be a useful representation for a variety of properties of program executions. The execution traces produced by ftrace are somewhat verbose but not to the extent of PIN. This makes them problematic for an analytical approach, but perfectly amenable for NN analysis.

Listing 3.2 shows a small excerpt of an instrumentation output produced by ftrace. The indentation in Listing 3.2 corresponds to the call hierarchy. To make the size of trace data manageable however, in these experiments the hierarchy is flattened.

```
... 1063014.264604: ...   mutex_unlock
... 1063014.264605: ...   __wake_up
... 1063014.264606: ...    _raw_spin_lock_irqsave
... 1063014.264606: ...    __wake_up_common
... 1063014.264606: ...    _raw_spin_unlock_irqrestore
... 1063014.264606: ...   __wake_up
... 1063014.264606: ...    _raw_spin_lock_irqsave
... 1063014.264606: ...    __wake_up_common
... 1063014.264606: ...     autoremove_wake_function
... 1063014.264606: ...      default_wake_function
```

Listing 3.2: Example of an ftrace instrumented trace. The left hand column is a time stamp. The right hand column is a hierarchical call trace of function calls in the kernel space, with the indent corresponding to the call depth. The traces are verbose, but not nearly to the extent of those produced by PIN.

3.3 Experimental Setup

The empirical portion of this work is composed of two sets of experiments. The first presents a method for constructing a search space for a property targeting search strategy. The second shows an approach for synthesising a search space for a diversification strategy.

3.3.1 Exp.1 – Search Landscape for a Property Targeting Strategy

The search landscape for a property targeting search strategy relies on a regression classifier neural network. During training, it takes a PIN trace as input and a ground truth property as the target. During inference, it outputs the likelihood or the estimated value of a ground truth property

given an execution trace, for categorical and numeric properties respectively. The setup is illustrated in Figure 3.2. The characteristics of the datasets for this experiment are summarised in Table 3.1.

The network is made up of convolutional and recurrent layers (described in Section 2.2). Sequence data is typically handled with recurrent cells such as the LSTM. Due to the vanishing gradient problem however, LSTMs can only handle sequences of up to several hundred elements. PIN traces are thousands of elements long and therefore need to be shortened. This is done with strided convolutional layers.

The network takes a PIN trace as input. The second layer is 64-dimensional embedding. This is followed by a stack of nine convolutional layers with a stride of two. The strides of the convolutional layers halve the sequence length, so the initial sequence length is shortened by a factor of 2^9. The next layer is composed of 500 LSTM cells. Each layer is followed by a dropout to reduce the risk of overfitting. This architecture was chosen by trial and error, using the *Codeflaws* dev dataset, *CF Dev*.

The output layer of the network is a single neuron. For categorical variables, it is sigmoid activated, and the network is trained with binary cross-entropy loss. For numeric values, the network is trained with a mean square error loss. The networks are trained using the Adam optimiser. The parameters were tuned manually by observing the performance on the validation dataset.

A single network is trained for each property of interest, using the *Codeflaws* training dataset *CF Train*. They were then tested using the *Codeflaws* test dataset *CF Test*, as well as each of the real world programs. That is, in this experiment, the models were not trained on real world programs – only tested on them. The purpose of this was to illustrate that a model trained on a corpus of (even small) program can generalise.

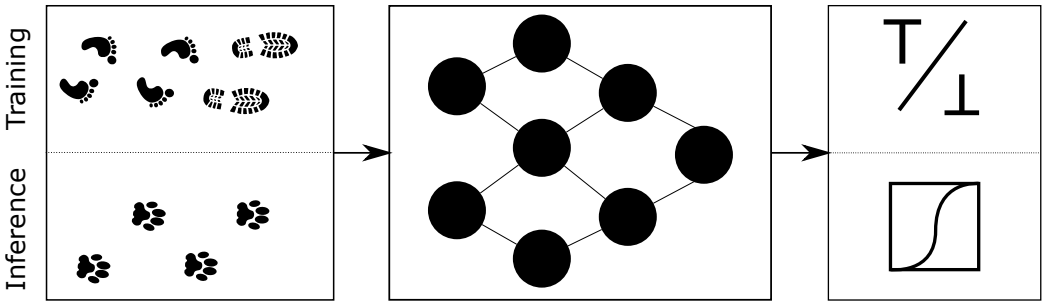


Figure 3.2: Experimental setup for Exp.1. A neural network is trained on execution traces of PIN instrumented programs as inputs, and properties of interest as prediction targets. During inference, it outputs an estimate of the property as a probability in $[0, 1]$ or a numeric value for categorical and numeric properties respectively.

	CF Train	*CF Test*	*CF Val*	*Cjpeg*	*Sparse*	*cJSON*
total traces	303,233	35,829	10,298	22,396	1,260	1,000
crashes	24,334	4,254	915	1,722	260	0
deflost_blocks	4,085	503	341	0	9	187
illegal_reads	40,103	5,420	828	3,781	49	0
illegal_writes	5,941	1,278	601	0	0	0
reachable_blocks	3,869	1,118	551	16,779	0	813
uninit_values	2,289	456	23	934	0	0

Table 3.1: Statistics of the programs and properties of interest in the dataset for Exp. 1. The *CF* datasets are taken from the *Codeflaws* repository, whilst *Cjpeg*, *Sparse* and *cJSON* are real world open source programs. Program inputs were generated by fuzzing each program with AFL, and properties of interest were collected using Valgrind. Each row (except for the first one) corresponds to a property of interest.

3.3.2 Exp.2 – Search Landscape for a Diversity Driven Strategy

A search landscape for a diversification strategy is constructed using a variational autoencoder (VAE, introduced in Subsection 2.2.5), composed of encoder and decoder portions. The VAE takes AFL's bitmap representation of an execution trace as input. The hidden layer is a ReLu activated densely connected layer of 2048 neurons. This is followed by a 3-dimensional encoding layer. The reason for using three dimensions is so that the latent space can be plotted. The decoder has a structure symmetrical to the encoder: the encoding layer is followed by a hidden layer of 2048 neurons, which feeds into the output layer of 65536 (size of AFL's bitmap) neurons on the output. The encoding layer is modelled on work by Kingma et al. (Kingma and Welling, 2013), with random noise and regularisation. This is intended to force the points close to zero and to provide a continuous landscape for interpolation.

An autoencoder is trained for each real world program in the dataset. The training data is produced by a modified version of AFL which dumps the bitmaps of all executions in its queue, and the bitmap of its current execution into a temporary file. When the temporary file is consumed, AFL dumps the bitmap of the current input again. This way our autoencoder always has training data: both the traces of AFL's queue and the traces of AFL's latest candidate solutions. During inference, all elements in AFL's queue are encoded into the latent space.

3.4 Results

This work has three central results, related to properties of the produced landscapes. First, the search landscapes are continuous and arbitrarily large. Second, they are correlated with various properties of interest, i.e. they meet

the representation condition. Third, the latent space produces a meaningful ordering on a set of seemingly non-orderable candidate solutions. Landscapes produced in this manner will hopefully be of potential use for both property targeting and diversification driven search strategies in future work.

3.4.1 Size and Continuity of Landscapes

Common landscape characterisation techniques like population information content and negative slope coefficient require a notion of a neighbourhood. The neighbourhood of a candidate solution is composed of other candidate solutions within a single *search step*. A step, and hence the neighbourhood, depends on the search operators of the SBST framework. The landscapes presented in this work are not defined with respect to search operators, but with respect to a neural network's interpretation of traces. These techniques are therefore inapplicable.

Instead, the result on continuity and size are based on the following facts and findings. First, neural networks are continuous by construction, as discussed in Subsection 2.1.2. This means that the number of possible fitness values is limited by the resolution of the representation. If two candidate solutions can be distinguished in the original representation, they can be mapped to distinct points in the fitness landscape. In this case, the limitation here is the number of possible traces and the precision of floats used by the NN; practically unbounded. Examples of property targeting and diversification strategy search landscapes are shown in Figures 3.3 and 3.4 respectively.

Second, in both sets of experiments, the ratio of fitness values to the number of unique traces was over 0.95. That is, most interesting (by AFL's definition) traces were mapped to a distinct point in the fitness landscape. There are two potential reasons for why the traces were not *all* mapped to distinct values. First, although AFL considered the executions interesting, their traces may not have been distinct in term of ftrace and PIN profiling. Second, the

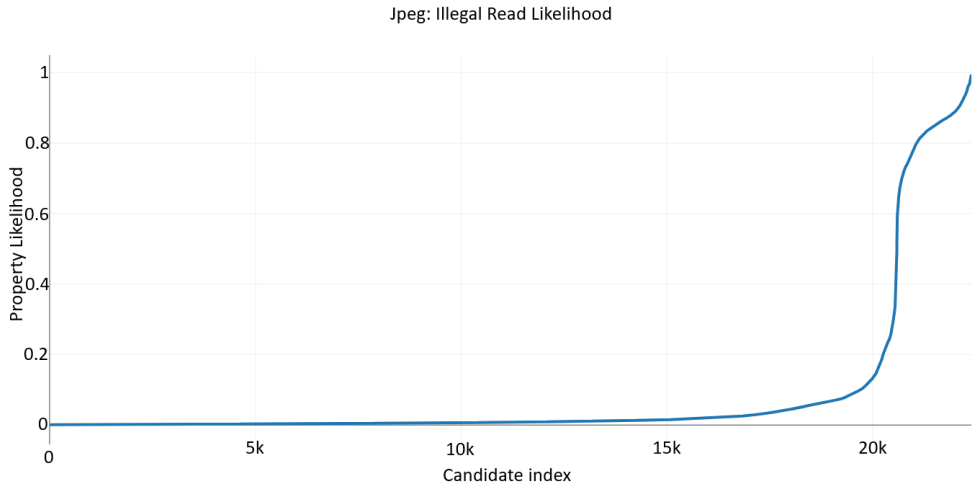


Figure 3.3: A plot of an NN classifier's estimate of the likelihood of an illegal write (y-axis) of a corpus of candidate solutions (input, PIN execution trace and label). The candidate solutions are sorted in increasing order and the x-axis is the index of a candidate solution. Although the classifier is trained on the *Codeflaws* train dataset, this plot is of a set of candidate solutions for the *Cjpeg* program, which suggests that the trace representation is applicable across programs. It is further suggested, that a fitness landscape such as this one can be used as a fitness landscape for a property targeting search strategy. The strategy would prioritise candidate solutions that the classifier considers to be more "suspicious", i.e. has higher values on the y-axis.

traces may have simply been indistinguishable with respect to the property of interest. For instance, multiple patterns of observations were considered "non-suspicious" by the NN, and assigned a fitness value of `0.0`.

3.4.2 Representation Condition

The neural network classifiers of Exp 1. have a strong predictive power for a range of properties of interest. This means that the landscapes they produce are strongly related to properties of interest, which in turn means that they meet the representation condition.

	CF Test	*Cjpeg*	*Sparse*	*cJSON*
crash	0.87	0.998	0.794	-
deflost_blocks	0.992	-	0.915	0.772
illegal_reads	0.966	0.885	0.344	-
illegal_writes	0.915	-	-	-
reachable_blocks	0.985	0.251	-	0.187
uninit_values	0.735	0.751	-	-

Table 3.2: Predictive ability of an NN for categorical properties of interest in Exp. 1 by ROC score. The performance is good on an independent test set of programs from the same dataset as the training data. The generalisability to real world applications is limited, but not non-existent. This is evident by the low ROC scores of some test sets. Blanks mean that there were no instances of executions with the property in the dataset.

	CF Test	*Cjpeg*	*Xmllint*	*Sparse*	*cJSON*
D1mr	0.151%	10.926%	7.462%	10.854%	1.583%
D1mw	0.817%	13.122%	11.195%	20.587%	0.926%
DLmr	0.747%	4.621%	8.549%	10.805%	0.594%
DLmw	0.008%	6.058%	13.755%	24.374%	2.783%
Dr	1.154%	2.413%	7.580%	16.656%	1.173%
Dw	0.695%	9.011%	3.388%	17.464%	2.326%
I1mr	0.699%	9.037%	22.508%	19.610%	1.607%
ILmr	0.265%	7.865%	16.132%	15.747%	1.586%
Ir	0.578%	13.382%	8.619%	8.808%	1.706%

Table 3.3: The predictive ability of an NN for numeric properties of interest in Exp. 1 by percentage error. The results indicate that these numeric properties can be predicted from PIN execution traces, and that the prediction meets the representation condition. The generalisation to arbitrary programs is not uniformly good however which can likely be improved with a larger training dataset.

This result is supported with the quantitative results of Exp. 1, summarised in Tables 3.2 and 3.3. Table 3.2 shows the Area Under Curve for the ROC – a plot of the false positive versus the true positive rate of a binary classifier.

High values in Table 3.2 are examples where the model, which was trained on an isolated training dataset of *Codeflaws*, predicts the property of interest well. In these cases, the NN has learnt to distinguish and generalise features of execution traces pertinent to properties of interest. Some values are low however. For instance, the presence of reachable blocks in the *Cjpeg* dataset has a low ROC score; the model's understanding of execution trace features indicative of this property did not generalise. That is, the patterns in execution traces of the training (*Codeflaws*) corpus were not always similar to those in the other programs, thus the model could not learn those patterns. A larger, more representative dataset could alleviate this issue.

Table 3.3 summarises the networks' predictive ability for cache behaviour values. These are numeric properties, and the results are given as percentage errors from the ground truth. These results give an insight into the fact that the performance of a neural network depends strongly on the training data: they have a strong predictive ability on the test set of *Codeflaws* programs but poorer performance on the real world programs. The *cJSON* test set is an exception in that the models predict its cache behaviour well. This is likely due to some inherent similarity of *cJSON* and the programs in *Codeflaws*. An in depth investigation of possible inherent similarities is an interesting direction of future work but out of scope for this work.

The results presented here are an instantiation of the proposed approach – they are conditional on the representation, the properties of interest and the training dataset. Given a larger, more representative dataset the approach ought to perform better. This is based on the fact that given a sufficient dataset and model size, neural networks are known to avoid local optima.

That is, if there is a pattern in the data, a neural network will find it. This is, of course, a "Deus ex machina" (or rather, "Deus ex data") argument: *given enough data*, a neural network turns into a silver bullet. Nonetheless, even with the limited dataset, the results clearly demonstrate the effect of the technique. Furthermore, as will be shown in Chapter 5, thanks to a tireless oracle – the program – endless training data can be generated.

3.4.3 Meaningful Ordering of Candidate Solutions

The techniques proposed in this work induce a meaningful ordering on an arbitrary representation. In the case of a property targeting search landscape (Exp. 1), the ordering is obvious – by a classifier's estimate of the property of interest.

When there is no explicit property of interest however, an ordering is not apparent. A latent space of an autoencoder produces an ordering based on features of observations. This latent space is meaningful, and potentially useful for a diversity driven search strategy.

Figure 3.4 is an example of a latent space of candidate solutions for *xmllint*. It is a three dimensional space[1] onto which elements of AFL's fuzzing queue are mapped. The axes themselves do not correspond to any specific feature, they are simply the internal states of the autoencoder. The locality in the latent space represents the similarity of *salient* features of execution traces. The colours correspond to the sequential id of a candidate solution. The earliest candidates are in dark purple, while more recent ones are yellow. This image is analogous to the RH one in Figure 2.5: the locality of the datapoints corresponds to the presence of features, and the colour is a true label mapped on the point *post* training.

Several observations of the nature of this landscape can be made. First,

[1]The dimensionality is arbitrary, three is chosen here so that the space can be plotted for qualitative analysis.

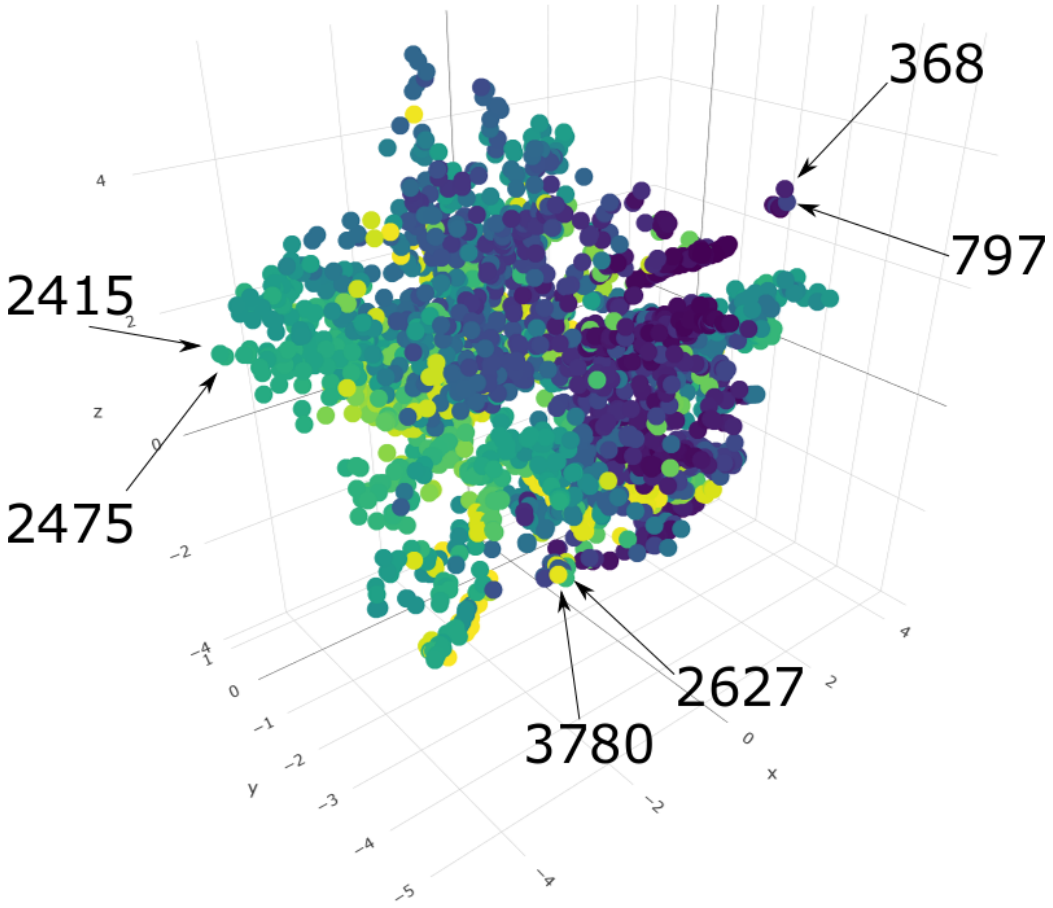


Figure 3.4: A 3-dimensional latent space encoding of the execution traces of the AFL queue for *xmllint*. The position of each point in the latent space is determined by features of execution traces that the autoencoder found most useful for distinguishing one trace from another. The points are coloured by the sequential index of the queue elements: darker points correspond to traces found by AFL earlier in the search. Although the candidate solutions are spread throughout the latent space, there are regions with denser clusters. A diversity driven search strategy could be directed to explore these less populated regions. The numbers are ids of example candidate solutions discussed below.

the locality in the latent space is correlated with the progression of AFL's search process. This is evident by points of similar colour being grouped into adjacent regions of the space. Early candidate solutions (purple) produced similar traces. As the search progressed, new behaviours (green clusters) were discovered. AFL then turned its focus to some earlier examples and used those as starting points to yield newer traces still (yellow). This is a general observation which may not be immediately useful in and of itself, but it does give insight towards developing an intuition about search processes, program behaviours and latent spaces. Specifically, it demonstrates how AFL's search yields ever more diverse behaviours as the process progresses.

Second, the arrangement of points in the space is not uniform. The autoencoder is regularised to attempt to arrange the points close to zero and to prevent points from being arranged too close to each other (Gaussian random noise). Despite this, there are clear concentrations of datapoints in some areas. This means that certain kinds of executions are relatively more explored and others less so.

Third, upon closer manual inspection of several candidate solutions, it becomes apparent that the latent space locality is related to program inputs. Consider the candidate solutions pointed to by arrows in Figure 3.4. The input strings that triggered them are the following.

```
 368: 0x1f 0x8b 0x94 0x80

 797: 0x1f 0x8b 0xff

2415: <S:L>><S:F>><S:R>><S:k>><S:FFFFdS:W>>5>M5>M<

2473: <S:L>><S:F>><S:R>><S:k>><S:FS:RSKFS>><FFFFFF:W>>5>Ma>M<

2627: 0xff 0xfe < 0x00 0xff --------C--ii------------ 0x00 0x80 -ii ---------
      L---------------- 0x00 0x80 -ii-------------------- 0x00 0x80 0x05
      0x80 0x10 0x05 0x80 0x10
```

```
3780: 0xff 0xfe< 0x00 0xef 0x0b@! 0x12 0xfb @! 0x12 0xff :R>kF@<S@! 0x13
      0x19 >5>M5>M 0x01 \0xff 0xff 0x05
```

Ids 368 and 797 are close to each other in the latent space. The strings are short and syntactically similar. 2415 and 2473 are likewise close to each other and their syntactic structure is also similar. They are rather different from the first pair however – both in their position in the latent space and their syntax. Finally, 2627 and 3780 are close in the latent space, and while they share some syntactic features, they are far from identical. The similarity of their traces and hence proximity in the latent space is likely due to their shared prefix. There happens to be a connection between input strings and latent space locality because the program is a linter whose purpose is to process strings – and exhibit corresponding behaviours. This notion of similarity is much more general however: it captures the innate similarity of features of data, without manual human involvement.

The above results suggest the following implications. First, a latent space representation gives a way of reasoning about similarity of behaviours given an arbitrary representation: something that was not naturally ordered can now be compared in a convenient, continuous n-dimensional space. In the context of a diversification strategy, one can utilise this notion of similarity to drive a search towards less explored behaviours, i.e. towards less densely populated regions of the latent space[2].

3.5 Conclusion

The effectiveness of any SBST process depends on a good fitness function. The landscape ought to be large, continuous and representative of the underlying property of interest. Constructing such a landscape is not trivial.

[2]Böhme *et al.* showed that enforcing diversity on AFL's search is beneficial(Böhme et al., 2017b). Future work can be an investigation into how the effectiveness using their notion of diversity compares with that proposed here.

This work proposes the use of neural networks for constructing search landscapes with convenient characteristics for both property targeting and diversity driven search strategies. It is suggested, that the output of a regressor can be interpreted as a fitness for a property targeting search strategy. The results of this chapter show that such a landscape is continuous, large and highly representative of various properties of interest.

For a diversity driven strategy, a search landscape can be constructed by training an autoencoder on execution traces. An autoencoder maps observations onto an n-dimensional space where the location is determined by the most distinguishing features of the data. This chapter shows how such a space can be created and illustrates that it possesses useful characteristics such as size, continuity and meaningful ordering.

The results and experiments of this work present the approach of constructing search landscapes, and their characteristics are discussed. The techniques described here form the fundamentals for NN-generated search spaces for targeted and diversity driven search strategies. Implementations where these landscapes are attached to search processes are presented in Chapters 4 and 5.

Chapter 4

Fitness Functions for Property Targeting Search

Searching to optimise coverage criteria and diversity has attracted significant research effort, though coverage has been observed to have limitations (Inozemtseva and Holmes, 2014; Kochhar et al., 2015; Gay et al., 2015; Staats et al., 2012; Heimdahl et al., 2004). Fitness functions targeting executions with particular properties, by contrast, is a relatively unexplored area and this is a gap that ought to be filled.

It is not immediately obvious how to go about creating an effective targeted fitness function for an arbitrary property. What information to observe? How to interpret the observed representation as a fitness?

It was suggested in the previous chapter, that an NN can identify features of an execution traces that are indicative of a property of interest. Its likelihood estimate of the presence of the property of interest can then be interpreted as a fitness – a candidate solution's proximity to the property. Prioritising candidate solutions that are "closer" to the property ought to lead to more efficient discovery of that property. This work implements the idea of synthe-

sising a targeted fitness function for SBST, as proposed in Chapter 3. An NN generated targeted fitness function is attached to an SBST tool. The method is showcased with an NN model trained on program executions, labelled with crashes and non-crashes.

The results suggest that guiding a testing process using the model's prediction of crash likelihood improves the efficiency of crash discovery. Although this is just a proof of concept, this work does illustrate that this definition of a targeted fitness function has a practical effect. It exemplifies how such fitness functions can be used to drive a search strategy to specific properties of interest.

4.1 Overview

There are two steps to this work. The first step is the construction of a fitness function from execution traces, much like in Chapter 3: a neural network is trained to predict the presence of a property of interest based on an execution trace. The NN is a regression classifier, so its output is a continuous value for an execution's crash likelihood: a smooth search landscape. The discrete space of execution traces is thus transformed into a continuous value. The implementation uses standard C library call traces as observations, and whether the execution resulted in a crash as the property of interest.

The second step of the work is the use of the above fitness function as an explicit property targeting fitness function, to direct an SBST process towards a crash. The NN is coupled with AFL by including the crash likelihood as an additional prioritisation heuristic (see Subsubsection 2.1.1.5). *Non-crashing* executions to which the classifier assigns a higher crash likelihood are given a higher priority during fuzzing. This alteration replaces AFL's diversification strategy with a directed search. The effect of the alternative search strategy is compared against baselines of the unmodified tool which is driven by

diversification.

Four central findings are presented. First, a strong correlation between C library call traces and a presence of a crash is shown. This is not a novel concept, since the representation condition of traces with respect to crashes was already shown in Chapter 3. It does however establish that the representation condition holds in this instance as well. Second, a clear guiding effect of a fitness function based on the crash likelihood on the search strategy *vs.* a baseline is presented. Third, the generalisability of the correlation for a presence of a crash to real world programs is shown. Finally, a mixed result for the generalisability of the fitness function driven by a crash likelihood is presented: the fitness landscape is rich for some programs and the targeted search works, but for other programs the representation of standard C library call traces is inadequate.

4.2 Research Goals

There are three research goals. The first investigates the validity of the chosen representation with respect to the execution property of interest. The second looks at the use of an NN constructed fitness function for a targeted SBST strategy. The third addresses the generalisability of the method to real world applications.

4.2.1 Representation

For a representation to be useful as the basis for a fitness function, it needs to be correlated with the execution property of interest. It is thus imperative to determine whether program executions represented by standard C library calls are correlated with crashes, when interpreted by a neural network. In the stand-alone (i.e. the conference paper) version of this work, this question is of a fundamental nature. Here, in the context of the thesis, it is incremen-

tal but practical, since the principle of the idea was already established in Chapter 3. The first research question of this work is **RQ1**:
"Are traces of standard C library calls correlated with crashes?"

4.2.2 Fitness Function

Once a representation is constructed and its correlation with the property of interest is established, it is to be used to construct a fitness function. The effectiveness of the fitness function then needs to be evaluated. This is done by comparing its crash discovery rate against a baseline – an SBST process guided by a non-targeted fitness function. The second research question is **RQ2**:
"Does the property targeting fitness function result in a higher crash discovery rate than a baseline?"

4.2.3 Generalisability to real world Programs

The effectiveness of machine learning techniques depends on the representativeness of the training data. This work is a proof of concept, so the generalisation of the trained model to arbitrary real world programs is expected to be limited. The following two research questions investigate the degree to which the model generalises.
RQ3.1: *"What is the crash predicting ability of a pre-trained model on real world applications?"*
RQ3.2: *"Is a fitness landscape produced by a pre-trained model applicable to real world applications?"*

4.3 Experimental Setup

The experimental setup consists of a large dataset of instrumented program executions, a neural network regression classifier and a modified AFL fuzzer.

The experimental frameworks for RQ1 and RQ2 are shown in Figure 4.1 and Figure 4.2 respectively.

4.3.1 Dataset

Like in Chapter 3, main corpus for this work was the augmented *Codeflaws* (Tan et al., 2017). Due to AFL's generation heuristics, some programs ended up with a disproportionately large number of inputs. In order to not skew the training data towards these programs, the number of inputs per program was capped at a uniformly sampled subset of a thousand executions. Overall, the augmented corpus is some 372,598 executions over 6089 programs. The dataset was then split 75%:25% into a train set and a test set, by the same principle as described in Subsection 3.2.1. The whole *Codeflaws* corpus was not used simply due to time limitations.

In addition, three real world programs were considered, for the purpose of investigating the generalisability of a model trained on *Codeflaws*. Adapting real world programs to the experimental harness and running the experiments requires a considerable engineering effort and time. The three programs were deemed sufficient to gauge the generalisability of the approach at this stage. The first one is the 2.0.3 version of the *VLC* media player. This version of *VLC* was chosen because it was known to have a crash causing fault in its subtitle module. The second one is *mpg321*, a library for the mp3 file format. The last one is the *Cjpeg* image library which had previously been used in Chapter 3. The latter two programs are also known to have crash producing faults, and have been used as benchmarks in recent work on fuzzing. The initial seed input for each program was a blank file.

4.3.2 Trace Instrumentation

The representation of an execution trace was the count of standard C library call transitions. This was collected for the whole dataset with Valgrind's

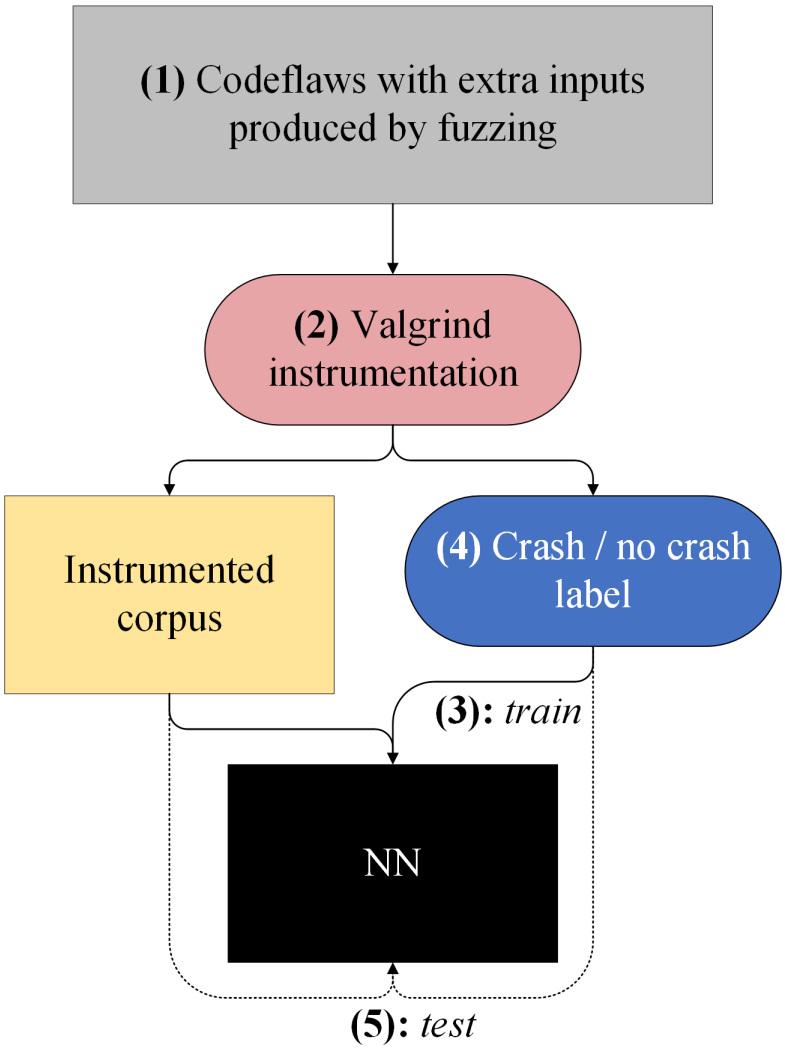


Figure 4.1: Experimental setup for RQ1. AFL is used to augment *Codeflaws* into a large corpus of crashing and non-crashing executions **(1)**. The corpus is then instrumented with Valgrind **(2)**. A neural network is trained on the instrumented executions **(3)** and crash / no crash labels **(4)**. The performance of the NN classifier is then evaluated on a held out test set **(5)**.

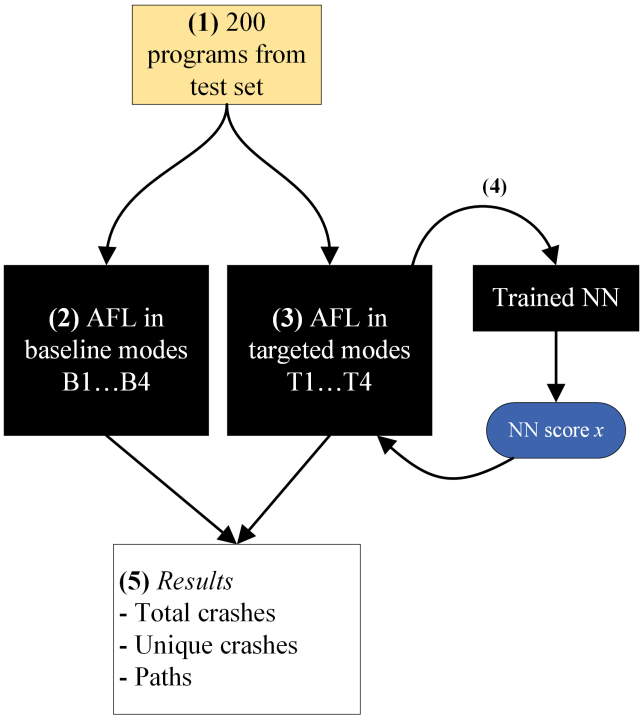


Figure 4.2: Experimental setup for RQ2. 200 programs are sampled from the test set **(1)**. The programs are fuzzed in baselines modes **(2)** and modified targeted modes **(3)**. Targeted modes query a trained NN for crash likelihood scores **(4)**. The number of total crashes, unique crashes and discovered paths are recorded **(5)**.

Callgrind tool, which produces a trace of function calls. An example can be seen in Listing 4.1. The raw traces like the one shown in the listing, were parsed into call counts of the number of times each function called another. Program specific functions were filtered out, so each execution is only characterised by its trace of standard C library functions. This preprocessing produced a corpus of instrumented executions across a range of various programs, with crash / no crash labels.

```
< .../dl-runtime.c:_dl_fixup (4x) \\A
< .../dl-machine.h:_dl_relocate_object (80x) \\B
* .../dl-lookup.c:_dl_lookup_symbol_x \\C
> .../dl-lookup.c:do_lookup_x (84x) \\D
```

Listing 4.1: An example of a Valgrind instrumented trace. The lines beginning with symbols <, * and > correspond to "called by", "function itself" and "calls" respectively. The last number in brackets shows how many times this call occurred in a given execution. In this example, the function on row C is called by functions on rows A and B, 4 and 80 times respectively, while it calls the function on row D 84 times. The trace produced is {(A, C : 4), (B, C : 80), (C, D : 84)}.

4.3.3 Experiments

For RQ1, the train-test partitioning, training and testing procedure was repeated 20 times, and the results were averaged. For RQ2, where each search was run for several minutes per program, using the whole test set would not have been feasible. Therefore, 200 programs were sampled from the test set, uniformly at random. The only restriction in selecting the programs, was that they showed at least a single crash in the corpus generation stage. Each program was then fuzzed for up to two times the number of executions in the corpus generation stage, or up to a maximum time limit of 15 minutes. The total number of crashing executions, the number of unique crashes and the number of distinct paths was recorded.

For RQ3, the model was *not* retrained or modified. Due to time constraints, experiments were not carried out on all modes – only on B4 and T4 (modes are explained below, in Subsection 4.3.5). They are the most representative benchmark between a "blind" strategy and a guided one: these modes are not impacted by AFL's built-in heuristics, so the effect of different search strategies is clearest. Each mode was run for 12 hours, five times each, and results averaged.

4.3.4 Regression Classifier Neural Network

The experimental framework is composed of two parts. The first component is a regressor NN which takes instrumented traces of program executions as inputs, and a binary label of a crash as true labels. The NN's output is the likelihood of a crash, given an execution trace. The framework is pictured in Figure 4.1.

A binary classifier outputs values close to zero or one, indicating the value of the binary label. However the purpose of the model in this framework is to produce a smooth search landscape for an SBST search process, and a classifier which outputs values close to `0.0` and `1.0` would not fit the purpose. The model was therefore trained with noise to yield estimates across the whole `(0.0, 1.0)` range.

NNs are sometimes designed with a three way train-validate-test split of a corpus. This is important in cases where the model's hyperparameters are adjusted *during* training, thus potentially contaminating the evaluation. In this work, a model with a fixed architecture was used however. The structure was chosen by trial and error; by visual inspection of a distribution using a small sample of 100 executions. Thus a simpler train-test split was appropriate.

The network is composed of seven layers. The input layer takes the vector

of function call counts. There were 1733 distinct function call transitions in the corpus, which is therefore the size of the input layer. The second layer is batch normalisation. Layers three through six are densely connected layers with sizes 128, 64 and 32 neurons. Hyperbolic tangent activation is used for non-linearity. The last layer is a single neuron with a sigmoid activation which produces a floating point value in the range `(0.0, 1.0)`. The network was implemented using the Keras framework and trained using the RMSprop optimiser (Chollet et al., 2018).

4.3.5 Modifications to AFL

The second component of the framework is a modified version of AFL, whose original functionality is detailed in Subsection 2.1.1. Here, AFL's prioritisation heuristics are altered to consider the crash likelihood score provided by the NN. Every element in AFL's queue is assigned a fitness score based on a number of heuristics, including its execution time, path coverage and recency. This score, the fuzzing budget α, determines the number of times the queued input is modified and executed. The modification introduced in this work adds another prioritisation heuristic – the crash likelihood score from the trained neural network.

4.3.5.1 Calls to trained NN for new queue elements

AFL adds an input to its queue when the execution path is found to be interesting. In this work, every element that AFL determined as sufficiently interesting to be added to the queue, gets sent to a mechanism which executes it under Valgrind, instruments the trace and scores it with a trained NN model. Running the modified fuzzer with AFL's uniqueness heuristics disabled (i.e. by calling on the NN+Valgrind mechanism for each modified input) was prohibitively slow due to the instrumentation overhead. Furthermore, the signal from the model was too weak: the crash likelihood scores

would rarely vary. This means that given the search operations of AFL on the input strings, the execution paths would alter only slightly, and appear very similar in terms of the standard C library call representation. AFL's built-in definition of uniqueness was thus kept enabled, and the framework only asks for the model's estimate intermittently – at the start of a new cycle.

4.3.5.2 Fuzzing budget weighting

The crash likelihood scores are then used to calculate an alternative fuzzing budget β.

The first stage of the weighting is normalisation, since the values produced by the NN are not directly comparable across programs. For instance, executions of program A might have a mean suspiciousness of 0.1, while the suspiciousness mean of values for program B may have a mean of 0.6. That is, 0.5 is not guaranteed to be the cut-off point of "suspicious" *vs.* "not suspicious". The raw values are therefore normalised, as shown in Equation 4.1. This maps the crash likelihood score into a range of $[0.0, 2.0]$, where scores between x_{min} and μ get an adjusted score $w(x)$ in the range $[0.0, 1.0]$, and those above the mean in $(1.0, 2.0]$.

For the second stage of weighting, the normalised score $w(x)$ is used as a multiplier of the final fuzzing budget β. This is shown in Equation 4.2. Depending on the mode (see Subsubsection 4.3.5.4), the initial fuzzing budget α is either a fixed constant, or it is given by AFL's heuristics. This value is then weighted by the normalised score $w(x)$. In addition, to prevent discarding elements simply due to the NN's opinion, a minimal value is introduced. This allows adjusted values to get at least some airtime, and not be discarded simply because of being non-suspicious.

$$w(x) = \begin{cases} \frac{x - x_{min}}{\mu - x_{min}}, & \text{if } x \leq \mu \\ \\ 1 + \frac{\mu - x}{x_{max}}, & \text{otherwise} \end{cases} \tag{4.1}$$

$$\beta = \max(20,\ \alpha \times w(x)^2) \tag{4.2}$$

4.3.5.3 Queue sorting, ad-hoc budget extension & deterministic steps

The unmodified AFL uses a first in first out queue which is then scored according to prioritisation heuristics. In this work, a queue sorting at the start of every cycle by descending β was introduced. A fuzzing cycle therefore starts with the most suspicious seeds. This is important, because AFL's ad-hoc fuzzing budget extension heuristic dynamically increases the assigned budget, if the queue element keeps producing new paths. A sorted queue makes it more likely that queue elements with high suspiciousness scores will have their fuzzing budget extended by the ad-hoc mechanism. This makes the effect of the new strategy more apparent.

In fact, the ad-hoc budget extension ambiguates the effect of a fuzzing budget by allowing it to grow dynamically, regardless of the initial heuristics based value. So to further highlight the effect of a particular strategy, modes 3 and 4 (see below) had the ad-hoc extension disabled altogether. It is an artificial weakening of AFL's heuristics intended for experimental purposes.

One more feature of AFL, deterministic fuzzing operations, was switched off. The ability to bypass the deterministic steps is a built in option in AFL. By default, AFL begins fuzzing with deterministic steps such as common interesting arithmetic operations. These operations are likely to bias the search towards finding the same crash triggering inputs regardless of the prioritisa-

tion heuristics. To remove this bias and make the effect of different strategies more apparent, the deterministic fuzzing operations were disabled.

4.3.5.4 Modes of modified AFL

The experiments were run in 8 modes: four baselines (**B1-4**, with fuzzing budget α) and four targeted modes (**T1-4**, with fuzzing budget β). The purpose of several different modified and baseline modes is to investigate whether a targeted strategy noticeably changes the crash discovery rate. Modes other than **B1** and **T1** disable some of AFL's original heuristics in order to make the effect more apparent.

B1 The first baseline mode is the completely unmodified AFL. The fuzzing budget is α, which is calculated by AFL's original heuristics.

B2 The second baseline assigns a constant budget of $\alpha = 100$ to each queue element and uses ad-hoc extension.

B3, 4 The third and fourth baselines are the same as **B1** and **B2** but with the ad-hoc fuzzing budget extension disabled.

T1 The first modified mode considers both the original AFL weighting and the score given by the NN. The fuzzing budget β is calculated by weighing the original AFL budget α by the adjusted NN score.

T2 The second modified mode assigns the budget β purely from the NN, with a constant $\alpha = 100$.

T3, 4 Third and fourth modified modes are same as **T1** and **T2** but with the ad-hoc fuzzing budget extension disabled.

4.4 Results

This work has three main results. First, the regressor was found to discover a strong correlation between the chosen representation and the property of interest. The representation condition was thus established. Second, the fitness function based on the NN's crash likelihood estimate was found to be clearly more effective at crash discovery than a baseline. This suggests that constructing a fitness function in this manner is feasible. Third, the question of generalisability had a mixed result. The model that was trained on the *Codeflaws* corpus, did generalise to real world programs to some extent, but it performed worse than on the *Codeflaws* test set baseline.

4.4.1 RQ1 – Correlation of C Library Call Traces and Crashes

The first part of the experiments addressed RQ1 whose purpose was to investigate the model's ability to predict crashes based on C library call traces. An NN regression classifier was trained and its performance was evaluated with the AUC of ROC of a test set. Across the 20 random train-test splits, the mean and median area under curve of the ROC were 0.897 and 0.901 respectively. The strong correlation suggests that traces of standard C library calls are a useful representation for crash discovery.

As intended, the outputs produced by the network were not tightly grouped at the extremes of 0.0 and 1.0. This was observed visually across the score distributions of the test sets.

An example of a distribution of crash scores of a test set is shown in Figure 4.3 with the corresponding ROC curve shown in Figure 4.4. The two curves correspond to the distribution of crashing and non-crashing execution labels. The left hand peak shows that non-crashing executions are grouped close

to a likelihood score of 0. Similarly, the right hand peak shows crashing executions tending towards a crash likelihood score of 1. The distribution shows a considerable portion of crashing elements appearing between the two peaks. This space is a middle ground between "definitely crashing" and "definitely not crashing". This distribution of outputs is desirable for the purpose of using the network's output as a fitness function as it provides a smooth fitness landscape over which a search can be conducted.

A distribution with high peaks would have been a search landscape with plateaus – with many candidate solutions sharing the same fitness value, making search impossible. A better accuracy might have been achieved with a stronger classifier if noise had not been intentionally introduced. The result is therefore a compromise of the two aims: predictive ability and a degree of uncertainty which yields a navigable search space.

4.4.2 RQ2 – Fuzzing with a Targeted Fitness Function

To address RQ2, the second part of this work looked at how well the NN's estimate works as a fitness for a targeted search strategy. The results indicate that targeted modes have a higher crash discovery efficiency than the baselines. This suggests that the targeted fitness function constructed in this way is effective. The scope of this work is to show a proof of concept for future work to be built on – not to produce a practically applicable tool. There are recent major improvements on AFL (as described in Section 2.1) that the proposed approach does not attempt to outperform.

4.4.2.1 Performance measure

Three values were recorded for each execution: total number of crashes, number of unique (by AFL's definition of uniqueness) crashes and the number of paths discovered. Fuzzing is inherently random, so each program was

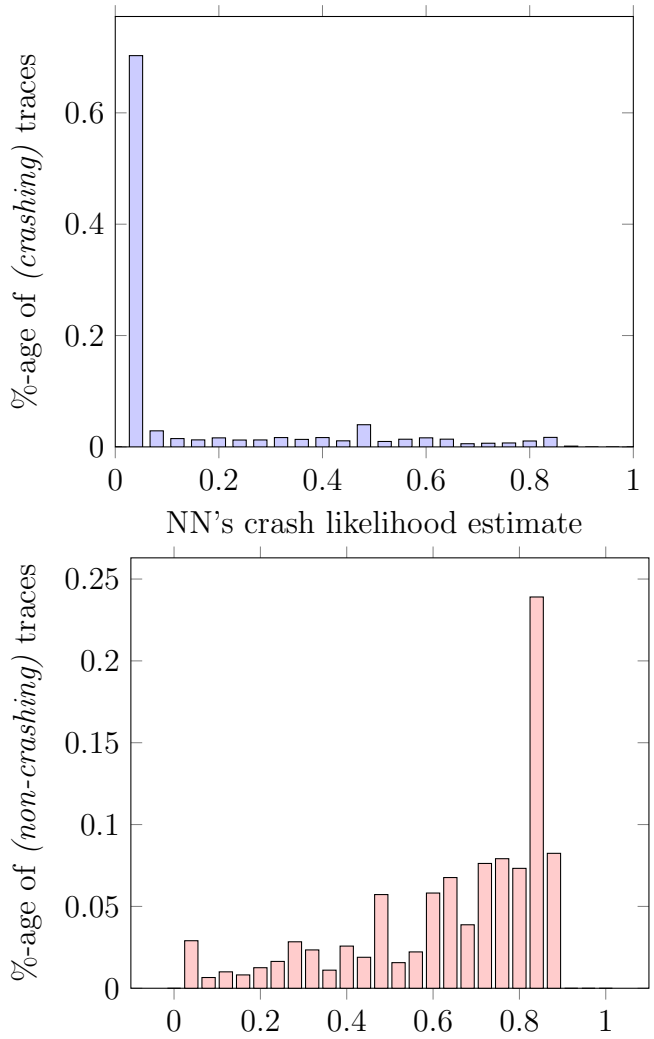


Figure 4.3: An example of a distribution of crash likelihoods from the *Codeflaws* test set. The two plots correspond to no-crash and crash labels – blue (top plot) and red (bottom plot) respectively. The x-axis label is the NN's crash likelihood estimate, y-axes are the proportions of executions in each 2-percent bucket of x. The two peaks on the left and right indicate the majority of test set elements falling into two distinct classes. The points in between the peaks are values where the NN was uncertain about the presence of a crash. Other experiments in the dataset had similar distributions.

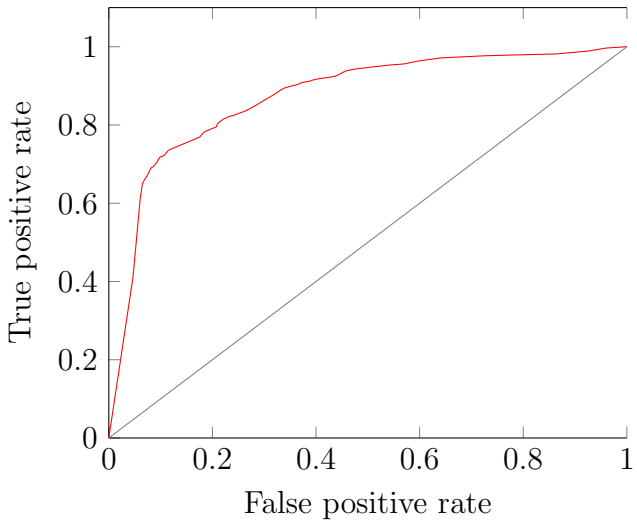


Figure 4.4: The Receiver Operator Characteristic curve for the distribution in Figure 4.3. The red curve represents the false positive *vs.* true positive rate of crashing execution classification. This corresponds to the red (RH peak) distribution in Figure 4.3. A plot of the blue (LH peak) is symmetrical, and is omitted here for clarity. The total AUC here is 0.887. The black diagonal line represents the indifference curve, i.e. a classifier that guesses at random.

fuzzed under each experimental mode (B1-4 and T1-4) three times, and the median values were taken.

The evaluation of performance for each mode rewards both the number of crashes (or paths) discovered, as well as the rate at which this happened. The measure is therefore the area under curve for each collected statistic. An example of this accumulation is shown in Figure 4.5: the result of fuzzing program 48-C-255529 from *Codeflaws*. The left hand plot shows all modes finding 4 or 5 unique crashes, which would suggest equal performance. The right hand plot is the corresponding area under curve of the top right plot. It distinguishes between the modes' performance as some of them found unique crashes quicker than others.

Another adjustment of performance was necessitated by the fact that absolute values are not comparable across different programs: e.g. some programs had one or two unique crashes, others had dozens. Therefore, rather than simply considering the numbers of crashes or paths, the different modes are scored based on their rank of the cumulative performance measure. The rank yields a score between 0 and 7, for worst and best respectively. The significance of differences between modes was measured by the p-values of pairwise Wilcoxon tests (Wilcoxon, 1992).

4.4.2.2 Filtering trivial programs

Before getting to answer any of the research questions, it was observed that in numerous cases, the performance of different modes was indistinguishable across programs of the test set. This was the case for programs with high crash rates or very few paths. The reasons are, most likely, the following. If a program crashes very frequently, it makes no difference how to look for crashes. Any strategy does approximately equally well. Likewise, when a program only has a handful of possible paths, and they all get discovered quickly, a search strategy has no effect.

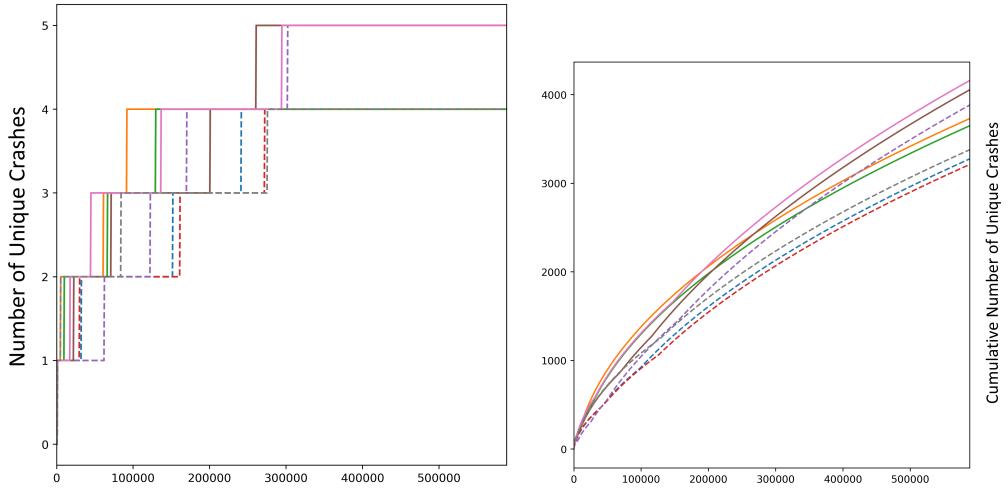


Figure 4.5: An example of performance of the different modes for fuzzing a SUT (48-C-255529 in *Codeflaws*), showing how the evaluation measures are calculated. On the LH plot, the x-axis is the number of executions, and the y-axis is the number of unique crashes after x number of executions. The RH plot shows the same values but accumulated over executions up to that point. A score between 0 and 7 is given to each mode based on the cumulative performance. The different colours correspond to different modes, with solid lines being the modified modes and the dashed lines being baselines.

In order to observe a difference in the modes, trivial programs were therefore discarded from the final results. These included programs with crash rates of over 2% of executions, and with fewer than 50 paths. This left a total of 52 programs for the results described next.

4.4.2.3 Performance of Targeted Search *vs.* Baseline

The central finding of this research question is that a crash targeting search strategy does indeed yield higher rates of crash discovery. This is evidenced by the fact that the modified modes of AFL produced consistently more crashes than the baseline modes.

A summary of the scores of the 52 non-trivial programs in the test set is shown in Figure 4.6. The figure shows the proportion of ranks averaged across baseline modes and targeted modes. For instance, the two rightmost columns in the top chart show that a targeted mode was the top performer with a score of 7 for total crashes found in 81% of cases. The effect of a targeted strategy is most apparent in the total crashes found, as evidenced by the top plot. In terms of unique crashes and paths discovered, a targeted mode comes out on top in 63.5% and 61.5% of cases respectively. These findings indicate that an estimate of an NN model trained on standard C library calls of crashing and non-crashing executions has a clear guiding effect.

4.4.2.4 Performance by Mode

A more granular analysis of performance by mode, likewise indicates that the targeted strategies outperform the baselines. A mode-wise comparison of scores is shown in Table 4.1; it breaks down the summarisation of Figure 4.6 into numbers.

The first column shows that the targeted modes produce consistently more crashing executions than any of the baselines. The p-scores for the modified versus their baseline counterparts are low, which indicates a considerable

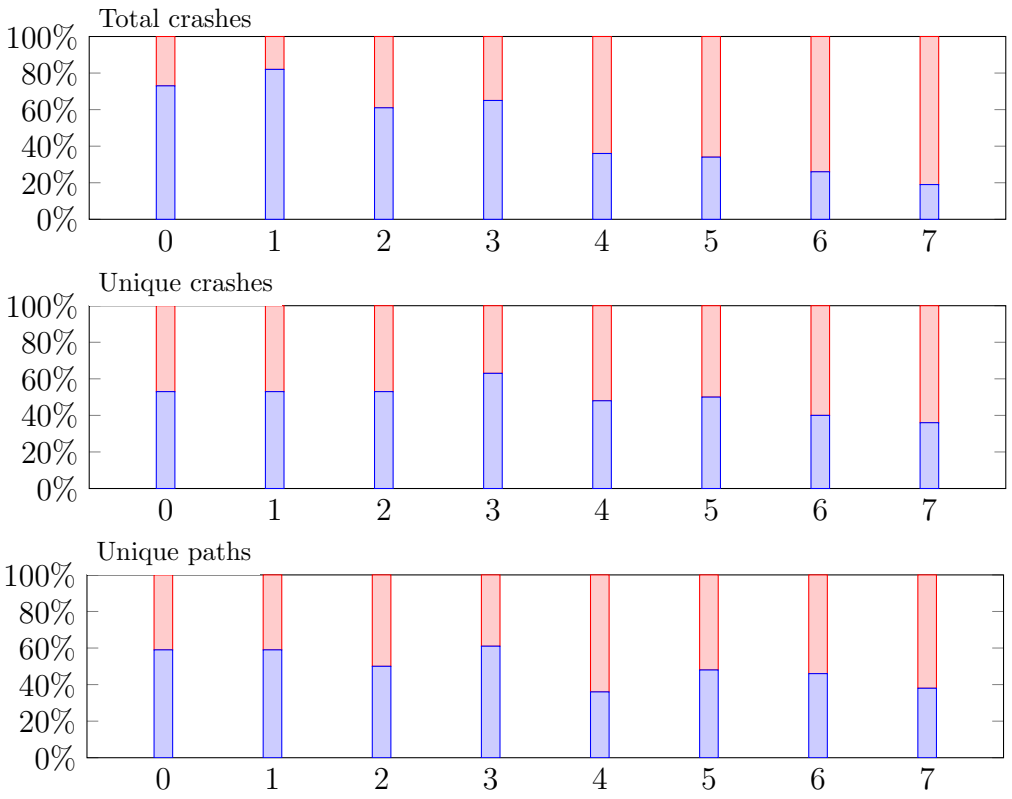


Figure 4.6: Distribution of mean scores of baseline modes (blue, bottom of column) and targeted modes (red, top of column). The plots are of total crashes found, unique crashes found and paths found from top to bottom. The scores show that targeted modes scored much higher in terms of total crashes found – top plot; somewhat higher in unique crashes found – middle plot; and approximately the same (perhaps slightly higher) in terms of unique paths discovered – bottom plot.

difference. As expected, the difference is more apparent in the bottom two modes where the ad-hoc budget extension was disabled, i.e. when the fuzzing budget is fixed by a fitness function and not allowed to increase dynamically. Here it is important to reiterate that AFL does *not* use crashing executions as seeds for future fuzzing. Thus the difference in performance can be attributed to the targeted strategy, rather than continued fuzzing of crashing seeds.

Although the modified modes did not specifically target uniqueness, they achieved higher scores with respect to unique crashes nonetheless. This is evidenced by the higher scores in the middle column of the table. The significance of this difference is much smaller than for total crashes however, as indicated by the higher p-values. As expected, the difference is more significant (lower p-scores) with respect to the constant budget modes (B2/T2, B4/T4) than in those with built-in AFL prioritisation (B1/T1, B3/T3). In other words, the targeted search strategies are noticeably more effective at finding unique crashes than constant budget baselines, but fare only somewhat better when used to augment existing AFL heuristics.

Finally, the modified modes also perform somewhat better than baselines in terms of paths found. This is surprising, as targeting crash discovery was expected to have diminished exploration performance. The differences are not very significant however, as suggested by the high p-values (except for B2/T2 comparison which may be an outlier). A possible explanation for this marginally improved performance is the modified strategies' penalising effect of non-suspicious executions: what appears as benign to the NN, happens to be a trivial execution overall, and should not be used for fuzzing. A more in-depth analysis of this effect is outside the scope of this work.

Mode	Total crashes	Unique crashes	Total paths
B1 / T1	2.250 / **4.269** < 0.0001	3.231 / **3.654** 0.3897	2.981 / **3.385** 0.4534
B2 / T2	3.212 / **4.462** 0.0107	3.500 / **3.981** 0.2472	2.539 / **3.692** 0.0261
B3 / T3	2.654 / **4.539** < 0.0001	3.192 / **3.654** 0.3727	3.808 / **4.481** 0.1786
B4 / T4	2.058 / **4.558** < 0.0001	3.00 / **3.789** 0.1032	3.442 / **3.673** 0.6084

Table 4.1: This table shows the mean scores of each experimental mode, from 0 to 7 for worst to best respectively. The RH values of each number pair are scores of the targeted strategy modes, and the LH values are the scores of their corresponding baseline modes. They show that the targeted modes consistently outperform the baseline modes in terms of total crashes, unique crashes and even paths discovered. That is, the targeted modes discover the property of interest more efficiently given the same number of executions. The significance of the difference in performance varies however, as indicated by the varying p-values, shown underneath each pair of numbers.

4.4.3 RQ3 – Generalisability of Representation

The third set of experiments aimed to gauge whether the fitness function for NN models trained on *Codeflaws* would generalise to real world programs. The correlation between C library calls and crashes that the *Codeflaws*-trained model discovered, appears to transfer to real world programs. As for the usefulness of the produced fitness landscape – not so much. The landscapes are not very convenient, which suggests that in this instance they are not useful for testing the real world programs.

4.4.3.1 RQ3.1 – Generalisation of Representation Condition

The results show that the correlation between the traces and crashes generalises well to previously unseen programs. The AUCs are 0.907, 0.880 and 0.612 for *VLC*, *Cjpeg* and *mpg321* respectively. The value of 0.612 for *mpg321* indicates a weakness of the model, and ought to be improved – either by training on a more diverse corpus or by considering additional observations.

An analysis of the values produced by the neural network revealed an issue with the representation in *VLC* and *mpg321*. The landscape had many plateaus: in the whole dataset of "interesting" inputs generated by AFL, there were only 183 and 92 unique crash likelihood values for *VLC* and *mpg321* respectively – *vs* 14853 for *Cjpeg*. In other words, the whole input datasets of the two former programs mapped to very few points in the search landscape. This is likely due to the fact that these programs use standard C libraries to a lesser extent. Whatever the reason, it makes any attempts at using these search landscapes somewhat moot. This negative result does not invalidate the technique however. It only shows that in this instance, this representation, trained on these simple programs, was not applicable to these real world programs.

4.4.3.2 RQ3.2 – Generalisation of Targeted Search Strategy

As said, a search across landscapes with as many plateaus as were seen for *VLC* and *mpg321* would have been meaningless, so targeted search experiments were only conducted on *Cjpeg*. The strategy found a median of 1267, 61 and 13681 unique crashes, paths and total crashes, versus 1313, 63 and 8542 found by a baseline. The number of paths found and unique crashes found are similar in both modes, but the targeted strategy reached far more total crashes. This is indicative of the effect of a targeted strategy on fuzzing given a sufficiently rich representation. These numbers are in line with those of the *Codeflaws* experiments.

An interesting observation about the rate of crash discovery was made. For the first ca. 10% of the fuzzing process, the baseline outperforms the targeted strategy. In 10% to 50% of fuzzing, the targeted strategy gained considerably over the baseline. Finally, for the second half, the baseline picked up again. An example of this effect is shown in Figure 4.7. This pattern was consistently repeated across all fuzzing runs, both for the targeted mode and the baseline mode.

This behaviour can be attributed to the fact that while there are very few seeds (the process was started with a blank file), heavily prioritising one over others limits exploration. Once more paths are found, focusing on the most suspicious ones yields the most benefit. Finally, when discovery stalls, exploration becomes once again more viable. These observations are not central findings of this study, but their consistency is surprising and interesting. The causes of the effect, including the above hypothesis, ought to be investigated further in future work.

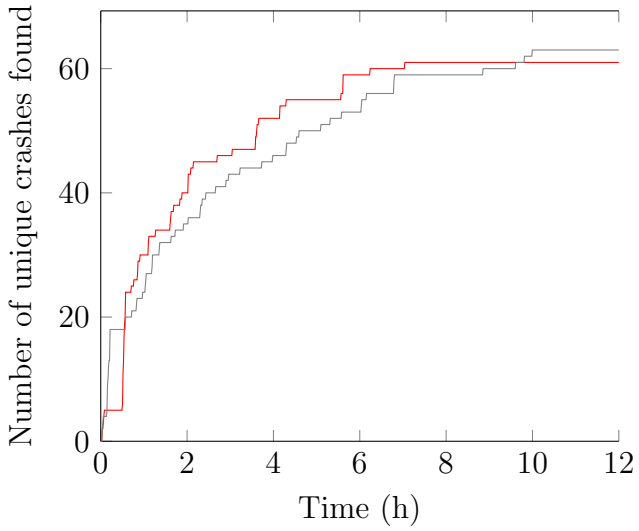


Figure 4.7: The number of unique crashes discovered by the targeted mode **T4** (red) and the baseline mode **B4** (black) follow an interesting pattern. To start, the exploration-heavy baseline mode **B4** does better. As the search progresses, the modified mode **T4** gains. Finally, the effectiveness levels out. These effects ought to be studied further in future work.

4.5 Future Work

Most importantly, this work is meant to enable future research, and there are numerous possible directions. First, as the work in Chapter 3 shows, a number of other properties of interest can be predicted, so targeting properties other than crashes should be investigated in a similar way.

Second, as also indicated by the results of the work in Chapter 3, various other representations can be used. That is, the traces do not need to be specific to C or any other language but rather, they depend on the profiling tool used. Overall then, an overarching aim of future work is to investigate the viability of the technique given other representations and properties of interest.

Third, from a more technical point of view, rather than simply weighting a fuzzing budget with a crash likelihood score, a fitness function may be constructed differently. For instance, a fitness function need not be based on a single value like the crash likelihood given a trace of C library calls. Instead, it may be a combination of multiple targets based on a representation of several traces.

Fourth, the underlying SBST tool can be something other than AFL. Any other SBST input generator that uses a fitness function to evaluate intermediate results could be used.

Finally, the pattern of crash discovery shown in Figure 4.7 should be studied further. Although it is not immediately related to the mechanics of this work, it is of potential interest to SBST in general. That is, if the observed effect really is the consequence of exploration *vs.* exploitation, it may be of use in designing hybrid search strategies – combinations of targeting and diversity, depending on the stage of a search.

4.6 Conclusion

This work implements an approach for constructing a property targeting fitness function using neural networks. This is done by first training a classifier on observations of a corpus of crashing and non-crashing library call execution traces, across a range of various C programs. The classifier provides crash likelihood estimates to a fuzzer which prioritises *non-crashing* candidate solutions based on this value. Results show that a fitness function constructed in this way has a clear effect on the rate of crash discovery. This suggests, that constructing a fitness function for a property targeting search strategy in this way, is feasible in practice.

The results were obtained with two sets of experiments. The first one shows that the execution traces represented by standard C library calls are strongly correlated with the presence of a crash. The mean AUC of the ROC measure is 0.901 for a held out test dataset. This is evidence of the traces fulfilling the representation condition, and thus being potentially useful for constructing a fitness function specifically targeting crashes. This result was similar in nature to those presented in Chapter 3, but here the instantiation was different and required confirmation for the further research questions.

The second set of experiments shows that the crash likelihood estimate can, in fact, be effectively used as a fitness function in practice. On a test set of programs, a crash targeting fuzzing strategy based on the crash likelihood, outperformed a baseline in 80.1%, 63.5% and 61.5% of cases with respect to total crashes, unique crashes and total paths discovered respectively.

However, in these experiments, where the model was trained on a corpus of small C programs, the generalisability to real world programs was limited. This is indicated by the result of RQ3. The search landscapes produced by this model were not convenient for two of the three real world programs in the experiments, as they had multiple plateaus. This suggests that either

the representation does not suit those programs, or that the training corpus needs to be larger and more diverse. In the former case, an alternative representation can be used – something the approach is perfectly capable of handling, as shown in other chapters of this thesis. In the latter case, finding a large, diverse and representative corpus is a challenging task, and the engineering effort for constructing per-program execution harnesses over thousands of programs is considerable (unlike the situation with *Codeflaws* where all programs had a similar interface). This would diminish the benefit of using an automatic approach.

Another drawback of the current implementation is the instrumentation overhead. The training corpus generation, model design and training would have made it infeasible in practice. However, the setting here is experimental, and the tool is not intended to be practical.

Indeed this work is intended as a proof of concept of the use of a targeted fuzzing strategy. It is not meant to be a direct improvement of AFL. The evaluation was designed to highlight the effect of the alternative search strategy based on an NN-generated landscape.

Chapter 5

Deconvolutional Generative Adversarial Fuzzer

This chapter presents the Deconvolutional Generative Adversarial Fuzzer (DCGAF) automated testing framework. The DCGAF uses two NNs to generate program inputs and process execution traces. It implements a diversity driven search first presented in Chapter 3. Furthermore, it is a first of its kind example of a generative model which produces its own training data. This work shows that the architecture is feasible: it can, in fact, be trained, it produces sensible program inputs, it explores diverse behaviours and discovers crashes.

5.1 Introduction

This thesis aims to address a number of challenges of SBST using neural networks. Although these have been discussed earlier in the thesis, it is worth recapping them here for clarity and context.

First, fitness landscapes may have plateaus, or contain local optima which

may stall the search or lead it to a sub-optimal solution (McMinn, 2011; Aleti et al., 2017). *Second*, the choice of a representation is a complicated task, requiring domain knowledge and expert involvement (Shepperd, 1995). *Third*, it is not clear whether a property targeting or a diversity-driven search strategy ought to be applied (Gay et al., 2015; Inozemtseva and Holmes, 2014). *Fourth*, it is not apparent how to assign an ordering onto the search landscape, likewise requiring an involvement of an expert (Shepperd, 1995; Harman and Clark, 2004). *Finally*, the generation of new candidate solutions is a big can of worms with various approaches and solutions (McMinn, 2004; Anand et al., 2013; Ali et al., 2010; Alshraideh and Bottaci, 2006; Fraser and Arcuri, 2013, 2011; Fraser and Zeller, 2012; Pacheco and Ernst, 2005; Korel, 1992). What search operators to use? How much prior knowledge is required and available? How to produce the next candidate solution?

Inherent properties of NNs make them ideal for tackling these issues. *First*, NNs are trained by a process of backpropagation (Rumelhart et al., 1986) which means that their elements must be differentiable by construction (Glasmachers, 2017). Hence, their outputs must be be continuous. Furthermore, it has been shown that given sufficient size, NNs avoid local optima (Kawaguchi, 2016; Swirszcz et al., 2016; Nguyen and Hein, 2017, 2018). *Second*, the above property also implies that if a representation contains a useful signal, an NN will discover it. This means that NNs can discern useful information from representations that are difficult to interpret manually. The ability to discern useful signals from poorly interpretable data comes with a risk of susceptibility to noise, if the data is noisy or redundant. These problems however can be addressed with feature selection (Verikas and Bacauskiene, 2002; Leray and Gallinari, 1999; Wang et al., 2014) and modern, deep architectures suffer less from this problem anyway (Li et al., 2018a). The properties of continuity and lack of local optima make NNs' intermediate representations a natural candidate for use as SBST search spaces. *Third*, NNs can be used to guide both property targeting, as well as diversity-driven search strategies, as suggested

in previous chapters. *Fourth*, due to their differentiable nature and training by backpropagation, NNs' intermediate representations "take on properties that make the [classification] task easier" (Goodfellow et al., 2016). For a classification task to be easier, the embeddings impose an order relation onto the data (e.g. (Kingma and Welling, 2013)), which represents the similarity of features. They may thus help us reason about similarity and diversity of program behaviour in a principled, continuous way. *Lastly*, NNs can be used for generating new data without analytical human effort (Goodfellow et al., 2016; Kawthekar et al., 2017; Chollet, 2017a; Graves, 2013; OpenAI, 2016; Radford et al., 2015; Van Den Oord et al., 2016).

The previous chapters have shown how NNs can be applied to SBST, to alleviate some of these issues, and the work presented in this chapter has all the same benefits of NNs that made them applicable for use cases of previous chapters. In addition, the third, fourth and fifth of the above challenges are specifically addressed here: using NNs to define and implement a diversification strategy, exploiting the fact that they impose an ordering on the underlying data, and using them to generate new program inputs.

However, neural networks do have limitations of their own. The big one is that they are data hungry; they need large representative training datasets (Beleites et al., 2013). This appears like a disqualifying issue in the context of SBST. If one wishes to train an NN to be used as a fuzzer, and has sufficient data to train the NN, this data could itself be simply used to test the program. This defeats the purpose of building and training an NN-based fuzzer.

The architecture proposed in this work can address all these problems – SBST and NN related alike. It is a generative model that produces its own training data. This might seem like a bizarre proposition as there is no apparent way to evaluate the quality of the generated data: you can produce all the random data you like, but how do you know if it is any good? The suggestion here

is that the principle of diversification can be adapted from SBST to evaluate the generated data; the generative process is *driven by diversity*.

On a high level, DCGAF's functionality is as follows. It starts off by throwing randomly generated inputs at a program. Most of these will be rejected by the program. Some, however, will trigger an unusual execution trace. Those are prioritised and kept in the training dataset. As the process continues, DCGAF generates program inputs that trigger new behaviours and uses them in its training dataset. Since the mechanism is NN based, it has continuity in its generation and configurable, probability distribution-based sampling parameters. It also defines a quantified notion of similarity of program executions. These properties of DCGAF address all of the issues outlined above, as will be shown below.

The work presented in this chapter implements DCGAF and presents a number of findings. First and foremost, it is shown that such a system can in fact be trained and that it produces diverse program inputs. Furthermore, the inputs are sensible with respect to the syntax of the program under test. In addition, the syntactic similarity of generated strings can be controlled. Finally, rudimentary as the current state of DCGAF is, it does actually discover crashes. Currently it is a proof of concept accompanied by several outstanding questions. As such, it cannot be readily compared with fully fledged fuzzers like AFL.

Even though it is a prototype, DCGAF presents a number of novel ideas. It is a generative NN-based tool that does not require a training dataset – it produces its own. It is a new example of a deconvolutional generative model for string generation. It uses a novel quantified notion of similarity for program executions. It presents a prioritisation method for program executions based on a greedy algorithm called Farthest-First Traversal (FFT). This sequencing does not appear to have been formerly used in SBST. Finally, it uses "unusualness" and diversity as an explicit training target and although

this idea is common in testing, it is novel under this instantiation.

5.2 Overview of the Approach

The proposed framework implements a diversification driven strategy which aims to explore the behaviour space of a Software Under Test (SUT). The system does this by generating inputs for the SUT, observing the executions under those inputs, and adjusting further generation of inputs towards the most unusual behaviours. It is essentially an evolutionary fuzzer, albeit with convoluted[1] generative and feedback mechanisms.

The tool is based on two neural networks, an execution trace profiler, and a ranking and prioritisation mechanism for executions. The structure is shown in Figure 5.1 and its algorithm is presented in Algorithm 1. A single epoch of the algorithm corresponds to two passes through the framework shown in Figure 5.1: the first is a generative pass where the networks' weights are not updated, and a training pass – where they are. The algorithm and framework are described next, with the numbers in brackets corresponding to the lines in Figure 5.1 and Algorithm 1.

The process is initialised by feeding Gaussian noise to an untrained Generative Neural Network (GNN, see Subsection 2.2.8) (*2-6*). It produces a batch of program inputs X. Most of these will be nonsensical; strings of random characters. The inputs are then executed by the SUT and the execution traces T are collected (*12*). As most inputs are just noise, most execution traces will yield the "invalid input" execution trace. Some inputs however may contain features that are valid, which will be reflected in their execution traces.

The execution traces are then encoded by a Variational Autoencoder (VAE, see Subsection 2.2.5) (*14*). It casts the discrete, unorderable execution trace

[1] Pun intended.

into an n-dimensional latent space embedding E. The latent space is a quantifiable representation of the features of execution traces, and their similarity can be assessed by Euclidean distance. The encoded executions are then ranked by the "unusualness" of their encoded traces using the Farthest-First Traversal (FFT, see Subsection 5.4.3) algorithm (*18*). Redundant datapoints are discarded and the most interesting ones are kept in a persistent training dataset (*20*). The dataset is composed of the execution traces T, their encodings E, and the program inputs that triggered them X.

The dataset is then used to train both the VAE and the GNN (*24, 26*). As the VAE learns to encode the execution traces of the training dataset, new, unusual ones stand out from the bunch. The GNN, in turn, learns to produce program inputs that trigger a variety of traces, i.e. program behaviours. Noise is also added to perturb the dataset towards exploration so that more novel behaviours are found (*29*). This framework is a novel approach to using neural networks for diversity driven testing of programs.

5.3 Research Questions

While many of the individual mechanisms of DCGAF are inspired by other work, the overall structure is novel. It is a loop that generates training data for itself by sampling the probability distribution on the output of a GNN, and evaluates samples with an external diversity-driven oracle. The novelty brings about a huge number of design and configuration choices, all of which have an effect on the research questions outlined below.

First, there was no guarantee that the training of such a system would converge at all. It is also not clear what optimisers, layer sizes, numbers of hidden layers etc. to use. Non-convergence is essentially underfitting – the mechanism does not learn to approximate the data. Whether DCGAF's training converges is the focus of the first research question.

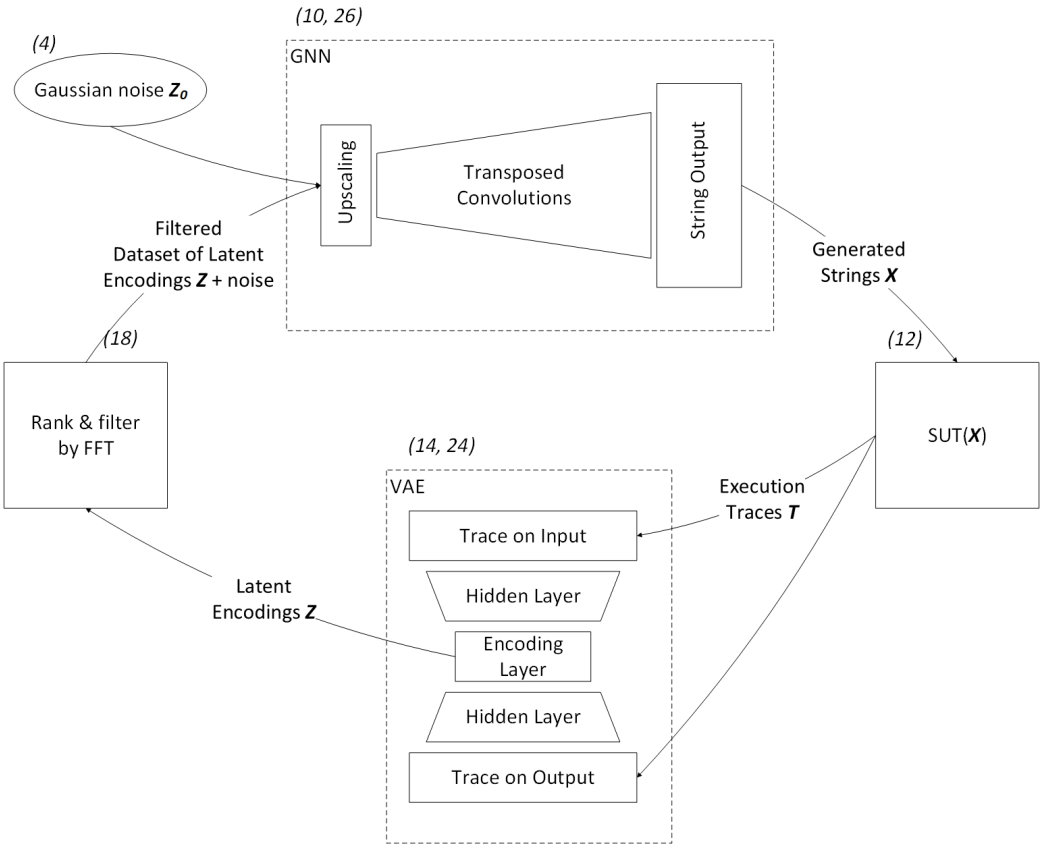


Figure 5.1: The DCGAF Framework – a self-feeding generative neural network architecture. The GNN portion of the framework generates input strings that are fed to a program. The execution traces of those inputs are collected and embedded into a latent space by a VAE. The candidate solutions are then ranked and filtered with the Farthest First traversal, to prioritise the least redundant candidate solutions. The embeddings are fed as inputs to the GNN which produces a new generation of candidate inputs. A detailed explanation of the training and generative processes is given in Section 5.2. The algorithm corresponding to this image is shown in Algorithm 1, with numbers in brackets corresponding to line numbers.

```
// Initialise epoch counter
e ← 0;
// Initialise inputs to GNN from Gaussian noise
{Z_0} ~ N(μ, σ^2);
// Initialise dataset of program inputs, traces and GNN
    inputs
{Res_0} ← {< X = ∅, T = ∅, Z = Z_0} ;
while ⊤ do
    begin Generative pass
        // Generate a batch of program inputs
        {X_e} ← GNN(Z_e);
        // Execute SUT, collect traces
        {T_e} ← SUT(X_e);
        // Get encodings of traces
        {Z_e} ← VAE_{train=⊥}(T_e);
        // Append new inputs, traces and strings to the
            dataset
        {Res_{e-1}} ∪ {< X_e, T_e, Z_e >}
        // Rank traces with FFT
        R_e ← FFT(T_e);
        // Discard redundant datapoints, limit of k=5000
            due to hardware resource limitation
        {Res|r_i < k};
    end
    begin Training pass
        // Train VAE on representative traces
        VAE(T_e)_{train=⊤};
        // Train GNN on representative noises and inputs
        GNN(< X_e, Z_e >)_{train=⊤};
    end
    // Update the inputs with encodings of traces
    {Z_e} ← VAE(T_e)_{train=⊥} + ε ~ N(μ, σ^2);
    // Increment epoch
    e ← e + 1;
end
```

Algorithm 1: The algorithm for training and generating inputs with DCGAF. The algorithm corresponds to the image in Figure 5.1 and is described in Section 5.2.

RQ1: *"Does DCGAF framework's training converge?"*

Second, it is insufficient for DCGAF's training to simply converge. It also needs to *not* converge too far, so as to continue generating new datapoints. New datapoints are essential both for exercising varied behaviours of the SUT, as well as for building up a diverse dataset for training. Much like non-convergence in RQ1 means underfitting, converging too far corresponds to overfitting – the system learns to produce only a few datapoints, and since those are kept in the training dataset, their effect becomes ever stronger. The second research question looks at diversity during training.

RQ2: *"Does DCGAF maintain diversity throughout training?"*

Third, if the mechanism does train in an acceptable way, the quality of the produced program inputs needs to be evaluated – whether they are sensible with respect to the SUTs. Granted, a program may crash under a completely unexpected random input, but a fundamental principle of fuzzing (and indeed any other testing) is that inputs ought to be consumable by the SUT, beyond an "invalid input" check. This means that they ought to be *somewhat* well-formed, or at the very least have some relevant syntactic features.

AFL generates strings that can scarcely be called well-formed, as its instrumentation is only a crude representation of a program's behaviour (see Subsection 2.1.1). Since the program instrumentation of DCGAF is lifted from AFL, the strings generated by DCGAF were expected to contain some syntactic features (qualitatively similar to those made by AFL), but there was no expectation of them being properly well-formed. The aim of the third research question is to see whether the generated strings are completely random or comparable to those made by AFL.

RQ3: *"Do the generated program inputs have syntactic features similar to those generated by an AFL baseline?"*

Fourth, one of the intended features of DCGAF is the ability to control the similarity of syntactic features of the produced inputs, by adjusting the input.

Once DCGAF is trained, the latent space that was used as input to the GNN can be replaced with n-dimensional normal noise. The GNN thus becomes a standalone generator which takes a vector of reals as input, and generates a string on the output. Input values close to each other are ought to produce similar strings, while distant inputs should produce dissimilar ones. Whether this is the case, is investigated in the fourth research question.
RQ4: *"Do strings generated from nearby points in the latent space share syntactic features vs. those far apart?"*

Fifth, by the same design as syntactic similarity of generated program inputs, DCGAF ought to be able to generate strings that would trigger specific behaviours. After all, one of the components of the loss function is the reconstruction of input (explained below in Subsection 5.4.4). The fifth research question looks at this aspect.
RQ5: *"Can DCGAF generate input strings that trigger specific behaviours of a SUT?"*

Finally, although DCGAF is only a proof of concept with a lot of additional work to be done, its preliminary feasibility as an automated testing tool ought to be investigated: whether the framework "has legs" as a design for a fuzzer. The ultimate aim of a fuzzer is to find crashes, and the last research question is simple but poignant.
RQ6: *"Does DCGAF discover crashes in* sparse[2] *?"*

5.4 Experimental Setup

This section presents the implementation details of each component of DCGAF, as well as the experiments that were conducted. The proposed framework is novel in many aspects so a large part of the experiments was exploratory. Various designs and configurations were considered and tested,

[2]The only program used in these experiments, in which AFL found crashes.

albeit not exhaustively due to resource limitations. Alternative designs are summarised in Table 5.1, with final choices boldfaced. The final choices are discussed in Section 5.5. The NN mechanisms were implemented with the Tensorflow framework (Abadi et al., 2015).

5.4.1 Trace Profiling

The program instrumentation (and hence representation) and the execution harness were taken from AFL. These mechanisms were not modified or parametrised, so there are no options to be considered, save for, of course, replacing the instrumentation with an alternative tool altogether. To be clear, *none* of AFL's generative or prioritisation mechanisms were used.

The first element borrowed from AFL is the execution trace representation. It is a count of decision point transitions, as explained in Subsection 2.1.1. An execution trace is a 64K long, typically very sparse vector of 9 distinct values. Although DCGAF does not take advantage of the efficiency of this representation, it is nonetheless appropriate for two reasons. First, it is simple to implement, since AFL is open source and the code can be modified to fit the framework's needs. Second, the effectiveness of AFL is evidence of the trace representation being useful. As there are many new aspects to DCGAF, using an alternative, potentially ineffective trace representation would add unnecessary uncertainty to the results and findings.

The second mechanism taken from AFL is its execution harness, referred to as the "fork server". The basic idea of the fork server is to run SUT until the `main()` function is entered, and a snapshot of that state of the program is stored. Whenever the SUT is executed under a new input, this initial state is copied from this snapshot and execution proceeds from there. Thanks to this simple mechanism, the time required for initialisation is avoided and fuzzing becomes much faster overall.

VAE	
Hidden layers	→ # of layers: [0,**1**...4] → sizes: [8... **512**...8192] → activations: linear, ReLu, **LReLu**
Encoding layers	→ sizes: [2... **4**...1024] → noise σ: [0.01... **0.1** ...2.0]
Losses	→ MSE, **Categorical cross-entropy**
Optimisers	→ **RMSProp**, Adam, LR=**0.00025**...0.001
GNN	
Generator Type	→ RNN: single, bi-directional, stacked, various sizes → DCNN: regular/**residual** deconv
GNN layers	→ # of layers: [4... **42**...64] → # of filters: [4... **256, 512, 1024**...4096] → kernel sizes: [2... **3**...10] → activation functions: ReLu, **LReLu**, Elu → batch normalisation: **yes**/no, before or **after** activation
Input	→ Gaussian noise / **VAE latent space** → size of upscaling layer [0...**2048**] → upscaling: **None**, size of deconv layer, up to 3 layers
Optimisers	→ **RMSProp**, Adam, LR=**0.00025**...0.001
Loss	→ Categorical cross-entropy of generated strings **and** / or MSE of input

Table 5.1: Various configurations were considered in design of DCGAF. The framework is novel and NN parameters are notoriously difficult to set in advance, so the options needed to be chosen by trial and error. The final, boldfaced parameters are discussed in Section 5.5.

5.4.2 Variational Autoencoder

The VAE is modelled on the architecture of Kingma's work (Kingma and Welling, 2013). The first layer is the input of size 64K. This is followed by one or more densely connected hidden layers and an encoding layer, which follows a design described in Subsection 2.2.5. The output of the z layer is fed to hidden layer(s) of the decoder. The output of the decoder is a categorical softmax layer of size $(64K, 9)$. This is specific to this trace representation, where counts of state transitions are bucketed into 9 values. This allows the VAE to be trained using categorical cross-entropy, so as not to enforce an ordering. In retrospect, a numeric loss such as MSE could have been used, since the numeric values of edge counts are naturally ordered: more is more, and less is less. This may be implemented in future work.

The configurable parameters of the VAE are the number, size and activation functions of hidden layers, the size and noise rate of the encoding layer as well as the losses and optimisers. These parameters affect the nature of the VAE.

From first principles of NNs, one can assume that the latent space must not be underfitted nor overfitted. Avoiding underfitting means that the VAE actually manages to encode the features of the data. The purpose of this is obvious: if the network does not encode the data, its latent space has no meaning whatsoever. In this case, under visual inspection, the latent space appears like a spherical cloud of points around zero. Quantitatively, underfitting can be easily identified by whether the model's loss value descends, i.e. whether the network trains.

A VAE can likewise overfit. An overfitted latent space looks like a manifold, e.g. a line. In this case the network learns to simply memorise the datapoints and maps them to distinct points, rather than into a space. In the case of overfitting, the locality in the latent space is likewise meaningless.

5.4.3 Ranking with Farthest First Traversal

Once the latent embedding space to represent program behaviours has been constructed, the points within it need to be evaluated for unusualness and redundancy. It ought to be reiterated, that a VAE arranges datapoints by the presence of salient features. Elements furthest from the centre are not necessarily the most *unusual* ones with respect to the whole dataset, but they are those with the strongest features. That is, the datapoints in the middle of the latent space are not those that have features most common in the dataset, but rather the features are least distinct. Correspondingly, datapoints close to the centre of the latent space are those that have an ambiguous mixture of features.

Consider the embedding shown on the LH image of Figure 2.5. Although that latent space represents the features of hand-written digits, the principle of encoding features into an embedding is similarly applicable to encodings of execution traces in DCGAF. The LH image shows that the feature of a diagonal line lies in the bottom left of the latent space, while the feature corresponding to a circle sits in the top right. Various other features are placed in the intermediate regions of the latent space. Any prioritisation algorithm should *not* simply take the outermost points (i.e. ones with the most distinct features) but a collection that is most representative of the whole dataset.

This work proposes doing this using an algorithm known as the Farthest-First Traversal (FFT) or the greedy permutation (Gonzalez, 1985)[3]. FFT arranges a set of points into a sequence such that the minimal distance to each following point is maximised. In other words, the prefix of the sequence is always maximally representative of the dataset as a whole.

[3]The algorithm was developed independently from scratch by the author of this thesis, to later discover it had been invented before.

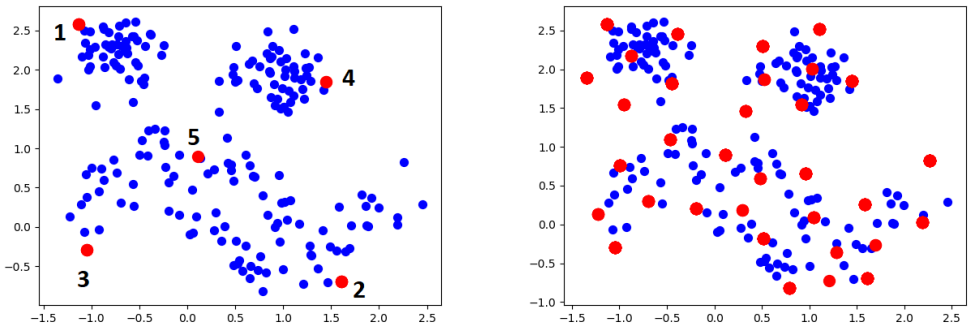


Figure 5.2: The Farthest First Traversal (FFT) algorithm sequences points in an n-dimensional space (two, in this example) by maximising the minimal distance from the prefix. The LH image shows the order in which blue points are picked (turned red). Eventually, the sequence ends up sampling the non-uniform space in the most representative, uniform manner, as illustrated on the RH image.

The algorithm is shown in Algorithm 2 and described below, with line numbers shown in brackets. Prior to the actual sequencing, the pairwise distance of each datapoint is calculated (**2**). Then the result is initialised with two maximally distant elements (**3**). The next element to append is chosen such that it is farthest from ones already in the result sequence, i.e. the minimal distance is maximised (**5**). Each following element in the sequence is thus maximally different from ones already in the sequence. An example of the sequencing in two dimensions is shown in Figure 5.2

Once the dataset is sequenced, its tail (the relatively redundant portion) is discarded. In the experimental setting of this work, the number of datapoints to be kept was set at 5000 due to hardware limitations; the complexity of calculating pairwise distances grows exponentially. Pruning the dataset according to this notion of similarity ought to keep it diverse and maximally representative.

Data: List X of n-dimensional coordinates of elements x
Result: Result Y of tuples y of (indices, distances) (i, d)
begin
  Calculate the pairwise distance matrix D;
  Initialise Y with y_0, y_1 of d_{max};
  while $|Y| < |X|$ **do**
    Append x s.t. **min** distance to $\forall y \in Y$ is **max**;
  end
  return $(i, d) \in Y$
end

Algorithm 2: The greedy permutation or Farthest-First Traversal (FFT) sequences elements such that each successive element maximises the minimal distance to elements in the result.

5.4.4 Generative Neural Network

The heart of DCGAF is a GNN that is trained to produce program inputs using the culled, representative dataset. As the dataset expands, the generator gets new datapoints to drive its training towards previously unseen behaviours and program inputs. A very early version of DCGAF was implemented with an RNN (modelled on works by Bowman *et al.* (Bowman et al., 2015) and Sutskever *et al.* (Sutskever et al., 2011)), but it was much too slow to train and to generate new inputs, so that design was abandoned. The generator is therefore based on an alternative design, inspired by the Deconvolutional Generative Adversarial Network (DCGAN) (Radford et al., 2015)[4].

There are two significant problems with directly applying DCGAN to generation of program inputs. The first problem is that unlike pixel values in images, discrete variables such as characters in strings are not readily orderable, as mentioned in Subsection 2.2.4. The other problem, inherent to neural networks, is the need for a lot of training data (Beleites et al., 2013).

[4]Obviously, the name of the presented tool is a reference to DCGAN.

DCGAN requires a corpus of real datapoints for training the discriminator. This in itself is a disqualifying requirement for a fuzzer, as it would require a large, representative dataset of valid program inputs. This defeats the purpose of a fuzzer: if you have a truly representative dataset of inputs, it could just be used to test the program. Furthermore, using only valid inputs would train the generator to produce only valid inputs. This is not ideal for fuzzing, as the program ought to be tested on both valid and invalid inputs.

DCGAF's architecture is fundamentally different. First, it has a continuous space both on its inputs and outputs. The inputs, thanks to the VAE, are ordered by features of execution traces. The outputs are probability distributions of possible characters – once again continuous. This structure makes "cat" >> "bat" comparisons possible, with respect to an underlying ordering. Second, there is unlimited training data thanks to DCGAF producing its own. These design features thus address two major problems of generative models in string generation.

In the machine learning arena, the individual networks are not novel. Recent advances in GNNs have brought about numerous novel architectures, many of them for image generation, e.g. (He et al., 2016b; Karras et al., 2019; Zhang et al., 2017), and even for text generation (Bowman et al., 2015; Wang et al., 2018). DCGAF does have novelty with respect to work in ML, in that it produces its own training data.

The general structure of the GNN is the following. The input to the GNN is the encoding of a trace produced by the VAE, with the addition of a small Gaussian noise. The purpose of this noise is to perturb the data so that the framework explores novel behaviours. Much like in DCGAN, the input is then upscaled with a densely connected layer(s), and reshaped to match the size of the output of the GNN (using deconvolutions with stride > 1). The main portion of the GNN is a stack of transposed convolutions (commonly called "deconvolutions" – hence DCGAN) with strides of one or two. This

stack of deconvolutions may also include residual blocks (He et al., 2016a). These are intended to strengthen the signal through the network, thereby alleviating the vanishing gradient problem (Hochreiter, 1998).

The output layer is of shape $(str\_len_{max}, dict\_size)$. In the current implementation, the maximum string length str_len_{max} was fixed at 512 and the number of possible characters $dict\_size$ was 129: 128 ASCII characters and 0 for padding.

The output layer is trained with a loss function that combines softmax cross-entropy of one-hot encoded strings, and the reproduction error of the encoded trace (Equation 5.1). The first term $-\sum_{c=1}^{M} y_{o,c} \log(p_{o,c})$ trains the GNN to approximate the mapping of trace encodings to strings. It is a standard loss for training categorical classifiers (Goodfellow et al., 2016). The second term $|\hat{z} - z|$ aims to approximate the SUT's semantics within the GNN. This term is minimised when the value of the encoded trace on the input z to the GNN matches the *true* trace $\hat{z}$ encoding that the SUT produces, when executed under the generated input. The overall loss function maps traces to strings *and* tries to make those strings produce a specific behaviour.

$$loss_{GNN} = -\sum_{c=1}^{M} y_{o,c} \log(p_{o,c}) + |\hat{z} - z| \tag{5.1}$$

During generation, rather than simply choosing the most likely character for each index of the output string ("argmax", as is typically done in string generation with GNNs, see Subsection 2.2.8), the generator samples from the output probability distribution, $x = \langle x_i \sim p_i; i < str\_len_{max} \rangle$. For instance, given an output $< 0.6, 0.3, 0.1 >$ (which corresponds to characters $<' a', 'b', 'c' >$), the generator draws samples from the distribution, rather than just choosing the most likely character $'a'$. This sampling technique is meant as a way to select plausible characters, given a learned probability

distribution. This is a novel approach suggested here, not modelled after any work in the literature.

There are numerous possible parameter configurations for the GNN. Common options include the size and the number of upsampling and deconvolutional layers, activation functions and optimisers. In addition, fundamental architectural options had to be considered. Namely, whether to use string reconstruction cross-entropy, trace reconstruction MSE or their combination as a loss, whether to use trace encodings or Gaussian noise as the input, and whether to sample or use argmax on the output.

5.4.5 Experiments Conducted

Experiments were conducted on three SUTs that take strings with complex grammars as input. The first one is an XML linter called *xmllint* from libxml (Veillard, 2019). The second is *sparse* (Jones, 2019), a lightweight parser for C (Jones, 2019). The last program is *cJSON*, a parser for the JSON format (Gamble, 2019). Generating inputs for these programs out of thin air is not a trivial matter. Also, since these programs validate their inputs by design, the validation ought to affect execution traces. Execution traces that depend on syntax validation would give DCGAF and AFL a good representation to work with.

For the comparative analysis AFL was first run against each program for 48 hours. This produced 3240, 11286 and 2071 queue inputs for *xmllint*, *sparse* and *cJSON* respectively. Of the three SUTs, AFL only found crashes in *sparse*.

For the DCGAF implementation, dozens (if not hundreds) of parameter configurations were evaluated. There was not a fixed time budget, nor a strict performance measure for each configuration. Instead it was a trial and error exploration of the effects of various configurations, with respect to the

research questions. Once the most promising structure and configuration was found, DCGAF was trained against each program for 48 hours. These models were then used for RQs three through six.

5.5 Results

The research questions explore the nature of the proposed architecture. The findings of the investigation show that DCGAF does indeed train while maintaining diversity, that it produces sensible program inputs, and that proximity in the latent space corresponds to similarity of generated strings. However, the result to RQ5 was negative: DCGAF did not generate program inputs that would trigger specific program behaviours. Finally, DCGAF does find crashes in *sparse* – the SUT in the corpus where AFL also found crashes.

The various options of architecture configurations are summarised above in Table 5.1, with the final configuration boldfaced. The reasons for arriving at the final configuration are discussed next.

5.5.1 RQ1 – Training Convergence

DCGAF trained successfully under some configurations, however it failed under others. Unlike standard NNs, a successful training is not meant to converge arbitrarily close to zero because DCGAF ought to continually produce new data. Instead, the convergence ought to be simply *noticeable*. That is, the training loss must decrease, but not reach zero.

The primary definition of failure here is failure to converge, i.e. underfitting. A secondary notion of failure is instability – a case when the training failed with the loss reaching `inf`. These two effects correspond to the vanishing and exploding gradient problems respectively (Hochreiter, 1998; Pascanu et al., 2012).

The VAE's loss failed to converge when the noise in the encoding layer was too high and when the hidden layers were very small, e.g. 8 units. Lowering the amount of noise to one tenth of original and using 512 or more neurons in the hidden layers allowed the VAE to converge.

GNN's training did not converge when the structure was too weak; there were too many layers or too few filters. When it came to depth, more than a dozen layers did not appear to converge in a reasonable time. Residual blocks resolved this issue however and structures of up to 64 layers converged. With fewer than 32 filters the network also failed to converge.

Initial configurations often resulted in instability, indeed much more often than typically with NNs. There are three plausible reasons for the instability. The first is aggressive optimisers, particularly Adam (Kingma and Ba, 2014). Adam is susceptible to instability when close to convergence (Wilson et al., 2017). The second reason is Gaussian noise in the latent layer of the VAE and on the input to the GNN. Lowering the noise σ to 0.1 resolved this instance of instability.

The last potential cause, is the constantly shifting dataset. This is the most unusual reason, specific to DCGAF, but not to other NNs. Modern optimisers like RMSProp (Tieleman and Hinton, 2012) and Adam both keep track of past gradients to calculate the next optimisation steps. Adam also uses momentum. It may be, that injecting new data throws these optimisers off: taking a step towards what previously seemed an optimal point, and finding a completely unexpected datapoint there, causes the gradients to misbehave and the loss to diverge. Updating the dataset like DCGAF does is unprecedented in the literature, so this suggestion is speculative and warrants further research. In the end, using RMSProp and reducing the learning rate to 0.0001 (one tenth from default) solved the instability issues. This, of course, slowed the rate of convergence.

The training loss followed an unusual trajectory: first quickly down, then up,

then slowly down again. Typically, if an NN trains, the loss tends to decrease monotonically, albeit with fluctuations. This is not a major finding per se, but an observation of expected behaviour, given that the mechanism keeps producing new data. While there are few datapoints at the start of training, the model quickly learns how to reproduce them and the loss goes down. As new datapoints are found, the loss increases, and then slowly descends again, as new features are encountered and learnt.

The central finding of this RQ is that DCGAF trains. In other words, a self-feeding generative neural network architecture is fundamentally possible. The positive outcome to this RQ validates the principle of DCGAF and makes subsequent questions worthwhile.

5.5.2 RQ2 – Diversity During Training

The second property of a successful training process is that it maintains diversity. Since DCGAF produces its own training data, it is critical for the framework not to collapse into generating the same outputs over and over again. Overfitting was seen both in the VAE and the GNN. The model was considered as failing to maintain diversity, if no new traces were produced over 20 epochs of generation and training.

When the VAE was too powerful, the loss tended to be close to zero and under visual inspection, the latent space looked like the datapoints are placed on a manifold. This occurred when there was no noise, or the hidden layers were too powerful (too many cells or layers). In this scenario, the locality in the latent space is meaningless which rids the VAE of its purpose. That is to say, an overfitting VAE is a useless structure with respect to ordering the execution traces.

Making the GNN too strong would, in turn, yield a reduction in diversity of generated strings. Too many filters in the deconvolutional layers, too little

noise on the input, or when the dropout rate was set too low, all contributed to loss of diversity. There is no apparent explanation for this, but it may be due to the GNN effectively learning to ignore the input layer and just encoding the dataset into its structure, before it manages to become sufficiently diverse and representative – an artefact of the recursive, self-feeding nature of DCGAF.

The fundamental architectural choice of sampling on the output, rather than taking the maximum, had a very strong effect on diversity. In an untrained model, changes in the input propagated to the output very weakly. That is, when feeding inputs drawn from a normal distribution to the GNN, and then taking argmax at the output, the generated strings altered very slightly, if at all. Sampling on the output however, readily produced numerous varied outputs. This provides ample training data for DCGAF, while also following the learnt output distribution.

The best loss function with respect to diversity was the combination of cross-entropy on the outputs and the MSE of encoding. Using only cross-entropy, the model did produce diverse outputs initially, but then lost diversity and stabilised into preferred outputs. Using only the encoding reproduction, the framework did not appear to converge, and new inputs were found very slowly, perhaps even coincidentally. In combination however, the model kept producing new outputs seemingly indefinitely. With the combined loss, the model was run for over 96 hours and it kept generating new inputs.

In addition, this combination of components for the loss function had a clear effect on crash discovery, as described below in Subsection 5.5.6. It is not clear why it is the *combination* of these loss components that has this property and this is a central direction for future work.

5.5.3 RQ3 – Syntactic Features

One of the desired characteristics of DCGAF is that it ends up generating sensible program inputs. "Sensibleness" of the generated corpora was evaluated by inspecting the most common n-grams and comparing them with the corpora produced by AFL. The analysis is comparative, because the current implementation of DCGAF uses AFL's instrumentation. Thus it could not produce better formed strings than those made by AFL.

N-grams are sequences of characters of lengths [3...10] that occur most frequently in the corpora. Typical features found in the corpora produced by AFL and DCGAF are discussed below and sample snippets are presented in Listings 5.1 and 5.2.

There were similarities in n-grams across corpora generated by both AFL and DCGAF. For *xmllint*, the most prevailing features are the triangular braces `<` and `>`. Usually these are not properly matched however. That is, neither tool finds that opening and closing braces ought to come in pairs. Colon `:` also appeared often. In XML, this special character is used to denote a name prefix to resolve name conflicts. Though `:` comes up often, it is not in the correct location, i.e. within a tag name. Another control sequence that kept appearing is the processing instruction `<?` which sends the command within a tag to third party software (w3schools, 2019).

Sparse is a very simple parser for C code, and it only identifies basic syntactic errors. It is therefore unsurprising that the generated strings did not appear very much like actual C code. Two features stood out however: braces `(` and `)`, and semicolons `;`.

Common character sequences for *cJSON* included curly and square braces `{`, `}` and `[`, `]`, colons `:` and line breaks `\n`. These are all special characters associated with the correct syntax of JSON. Again though, neither AFL nor DCGAF generated anything resembling well-formed JSON. These observa-

tions were in line with the expectation of some syntactic features, but not proper well-formedness.

There were also differences in the corpora generated by AFL and DCGAF. The first difference is the length of generated strings. While AFL's heuristics keep the maximum length unrestricted, under the current implementation, the maximum string length of DCGAF is capped at 512 characters. Furthermore, AFL occasionally prunes the corpus by deleting slices of the strings and observing whether this changes the execution traces. DCGAF, on the other hand, has no preference for shorter strings, so most of the strings in its corpus are ca. 500-510 characters in length. Strings of length shorter than 512 are due to removed padding bytes in generated strings.

Second, strings of AFL's corpus contain very long sequences of repeated characters and character sequences. For instance, there are input strings where the letter `K` is repeated thousands of times. Although DCGAF sometimes repeats sequences as well, not nearly to such a degree.

Third, strings produced by DCGAF appear to be more exploratory: where AFL finds an interesting n-gram and keeps reusing it, DCGAF does not.

Finally, while neither tool is restricted to only printable characters, AFL uses them more frequently than DCGAF does. AFL does not have heuristics that prioritise printable characters, so there is no immediate explanation for this effect.

It is obvious that generated strings are a far cry from well-formed, whether generated by AFL or DCGAF. Nonetheless, they are not random sequences of characters either; some syntactic features are clearly identifiable.

The investigation thus concludes that DCGAF generates strings that are comparable in nature to those made by AFL. This means that the feedback mechanism works, and that DCGAF leverages it effectively to learn significant features of the SUTs' input syntax.

```
// xmllint
<YY>vYY&&&aZaaaa
<b:><b:\\><
<Y:YvYYb></Y:YvY
<?xml.ver:V?VFxx
<YvHP>><?PM>
<IQQQQQQQQQQQQQQQQQQQQQQQY><YvY<YvQQQQQQY=pnQ
<\xd3\xbe>\xde[]:f\xde
(((((((((()
// Sparse
[KKQKKKKKKKKmKKKKKKKK*KKKKKKKKKKKKKKKmK
K?;K(C){K
**/****\x00\x00\x00
//////////
// cJSON
"\n{":3 \x1c\x01}
\n{":3 \x1c\x01}\x13
+\u9'999\u+\u9A99
```

Listing 5.1: Sample snippets of input strings generated by AFL.

```
// xmllint
<:p><n:/>\n
HQ%\x1f>\n#jS,u\x03\x1f<\x05UV\x05\n1
+\n&<?\n7>"<?
n=#+\n_<G:7>
[WH\x13}\x19Bc<]
<(eCQ\x05\x1bkL{GX\x01=O%Fck\x05Q)

// Sparse
i;();;gT;/*[n
```

```
11      if () [[*(P**nc*H.F^//)
12      //\x044U\x15COrm
13      /*Q9\x15C1\x01q
14      cU\x01UUv {s\x1c\x15r8\x05)!0C;\r
15      #if ({;+;?;*;U;1>Om;
16
17      // cJSON
18      {*,\x04}
19      {\x80\x1aK:5\x0843"\x08y\x0b^\x04\x19"#\x02}
20      {"\x1d\x7f\x10
21      <@?\x086\x1f\x1d\x08\x1d\x02\x1b{=\x08=\x08}
```

Listing 5.2: Sample snippets of input strings generated by DCGAF.

5.5.4 RQ4 – Similarity of Elements in Latent space

By design, DCGAF intends to map encodings of program executions to program inputs that triggered those executions. The latent space encodings define a notion of similarity and this ought to be reflected in the strings generated by the GNN: strings generated from points close to each other in the latent space should be similar, and strings from distant points dissimilar.

This property was evaluated in the following way. Ten thousand vectors from a normal distribution were drawn, and ordered by two algorithms: the greedy FFT algorithm from most to least representative, as well as an opposite "closest first traversal" (CFT). CFT orders the datapoints such that the pairwise distance of each following element is minimised with respect to the existing sequence. The first hundred elements of the FFT and CFT sequences were then fed to a trained GNN and the generated strings compared. The expectation was that the strings from the FFT sequence ought to be less syntactically similar than those generated from CFT.

SUT	FFT	CFT
xmllint	0.199767	0.453304
sparse	0.01205251	0.05063868
cJSON	0.090320	0.337037

Table 5.2: Mean Jaccard similarity of the n-grams of the first 100 elements in FFT and CFT sequences. Strings generated from nearby points in the input space have a higher overlap of n-grams, as shown by the higher values of the CFT column. This means that the syntactic similarity of elements generated by DCGAF can be controlled.

Syntactic similarity was assessed by the Jaccard index of the overlap of n-grams of lengths [1...10] of each string *vs.* the n-grams of the whole dataset of 100 elements. That is $\frac{|A \cap B|}{|A \cup B|}$, where A is the set of n-grams in each individual string and B is the set of n-grams in all 100 strings.

The mean Jaccard indices are shown in Table 5.2. Since the inputs are randomly generated, the process was repeated ten times and results averaged. The values are not to be compared across programs, but rather between the two sequences. They show that strings generated from adjacent input values are consistently more similar in terms of n-grams *vs.* those generated from distant points. This means that DCGAF can generate new program inputs with a notion of relative similarity. Such an ability is a step towards ultimately defining search strategies in a continuous Euclidean space – one of the future prospects this work aims to enable.

5.5.5 RQ5 – Targeting Specific Behaviours

A further intended property of DCGAF is the ability to generate program inputs that trigger a specific behaviour. In other words, given an input z to the GNN, it generates a program input x, which produces a trace t when executed by the SUT. When t is encoded into z', the encoding ought to be close to z.

The evaluation here was straightforward. 100 GNN inputs z were sampled, passed through the framework and cosine distances $D_{cos}(Z, Z')$ were looked at. A cosine distance of 0.0 is perfect correlation, 2.0 inverse correlation and 1.0 is a lack of correlation. Results to this RQ were negative: consistently within 0.1 of 1.0, the actual encoding values of traces z' were random with respect to the inputs z. This means that DCGAF cannot at this point generate a string that would trigger an exact, specific behaviour.

There is no clear explanation for this negative result, save for this speculative line of reasoning. The result to RQ4 shows that strings generated from nearby points in the latent space are syntactically more similar than those generated from distant points. This suggests that the generative portion of the framework works and that the locality in the latent space is meaningful. Otherwise, the latent space would have no bearing on the generated strings. The reason for failure must thus be elsewhere.

The first potential explanation is that the SUT's behaviour is simply much too complex. Although DCGAF can control the syntactic similarity of strings generated from specific points in the latent space, syntactic similarity of input strings may not translate to similarity of behaviours. That is, although the framework learns how to produce syntactically viable strings, it may be unaware of cases where a slightest change in syntax yields a vastly different behaviour. That said, preliminary results in Chapter 3 have shown that parser-like programs (ones that validate a string input) tend to exhibit similar behaviours given similar input strings. This suggests that perhaps the issue is related to the implementation of this prototype. Given the scope of the potential debugging effort however, and the novelty already presented, further investigation of this negative result is left for future work.

5.5.6 RQ6 – Finding Crashes

This RQ is very important for a fuzzer in the long run. At this stage however, DCGAF is a proof of concept with some limitations and numerous ideas for future work (discussed below in Section 5.6). The question of whether DCGAF finds crashes is therefore merely an indication of its potential – not an evaluation of its current ability as a fuzzer. The only SUT where AFL discovered crashes is *sparse* and DCGAF found crashes in *sparse* as well. This is not evidence of DCGAF being a better fuzzer than AFL, but that the framework has the potential of being made into one, down the line.

Crash discovery was affected by the loss function of the GNN surprisingly strongly. It was observed above in Subsection 5.5.2 that the MSE component of GNN's loss is useful for maintaining diversity. Furthermore however, without this component, DCGAF did not discover crashes in *sparse*. To confirm this effect, DCGAF was retrained against *sparse* ten times: 5 with the MSE loss, and 5 without. In each case, it did not discover crashes without the MSE loss. This is a somewhat baffling finding with two consequences. First, the MSE loss is indeed important for exploration as discussed in Subsection 5.5.2. Second, exploration is, in this case, important for crash discovery. At this point it is unclear why the MSE component helps exploration and thus crash discovery, and the effect can be investigated thoroughly in future work.

5.6 Future Work

The current implementation of DCGAF is a proof of concept prototype. As such, it has a number of limitations that can be investigated in future work.

First, there are just very many possible configuration options and given the novelty, there are hardly any reference systems on which to model the pa-

rameters. A larger, more systematic ablation study should be conducted in the future.

Second, DCGAF uses a generator based on transposed convolutions where the maximum string length is fixed. Methods for allowing variable and indeed unlimited string lengths should be investigated.

Third, the alphabet used for generating strings is the whole ASCII character set. Introducing some prior knowledge of the target syntax, e.g. using sequences of characters or a restricted subset may improve the performance of DCGAF.

Fourth, the trade-off between training convergence and diversity was investigated using fixed learning rates, learning schedules and noise levels, which might not have been optimal. That is, a better trade-off of learning and exploration could be achieved by dynamically adjusting the learning schedules and noises.

Fifth, the expressiveness of the trace representation from AFL is limited; it is a somewhat crude abstraction of program behaviour. In AFL itself, this representation is used for efficiency. This efficiency gain is not beneficial for DCGAF. Alternative execution traces profilers ought to be studied.

Sixth, currently the GNN of DCGAF is itself a fuzzer, and AFL's search mechanisms are not used. The generator could instead be used in conjunction with AFL: the GNN would produce strings, and they would then be fed to AFL for further concurrent fuzzing. This design would be analogous to using the GNN for seeds for AFL, like in works described in Subsection 2.3.2.

5.7 Conclusion

This work presents the Deconvolutional Generative Adversarial Fuzzer (DCGAF) framework for automated program testing. It is intended to address

multiple problems of SBST. These include problems with the selection, design, interpretation and implementation of representations, fitness functions and input generation methods. Furthermore, it bypasses a fundamental issue of scarcity of representative and useful training data for neural networks.

DCGAF generates program inputs with a generative NN, collects the resulting execution traces, and maps them onto an n-dimensional space with an autoencoder. The encoding imposes an ordering on executions so one may reason about similarity of executions quantitatively. DCGAF uses this encoded representation to prune redundant datapoints in order to keep the dataset maximally representative of the range of observed program behaviours. The process is continued ad infinitum with the framework continually exploring the SUT's behaviours while learning to produce ever more interesting inputs. DCGAF is a first of its kind generative neural network, in that it generates its own training data using an external oracle to assess the quality of the produced data.

The behaviour of the proposed framework was explored in a series of research questions. The exploration concluded that such an architecture can be trained, and that it keeps producing new, diverse and sensible program inputs. It also showed that DCGAF's notion of execution trace similarity translates to syntactic similarity of generated strings. This means that new program inputs can be produced with a continuous control of their syntactic similarity. Although it was intended, DCGAF cannot currently generate program inputs such that they would trigger an exact target behaviour.

The work is conceptually novel for SBST, NNs and their combination. The analysis was not a complete, systematic exploration and ablation of all the possible configurations and options. Currently DCGAF is a proof of concept which requires further work. On the other hand, it also opens numerous directions for future research and will be hopefully met with interest.

Chapter 6

Conclusion

Software reliability is paramount in the modern world. There are various techniques to ensure reliability, each with its strengths and weaknesses. One such technique is testing: the execution of a program for the purpose of discovering faults. Although the stated purpose of testing is the discovery of faults, the techniques considered in this thesis are generalisable to various properties – not just faults. The purpose of testing is therefore restated as the discovery of properties of interest.

There are various subcategories of testing, but the one considered in this thesis is Search-Based Software Testing (SBST). Specifically, feedback-driven automated test generation. The basic workflow of such an approach is the following.

1) Execute the program under a batch of inputs, observe the executions.

2) Evaluate the observed executions with a fitness function.

3) Generate the next batch of inputs, given the fitness of the current batch.

The high level process may seem simple, but the devil is in the details: there are many decisions and possible problems when it comes to the design of the

feedback mechanism. These choices determine the effectiveness of the testing process. First, the choice of what to observe about an execution; the choice of representation. Second, the choice of how to evaluate the observations; the choice of a fitness function. Finally, how to direct the search further and generate new candidate solutions; the choice of a search strategy and the generative mechanism.

6.1 Problems of SBST Addressed

A wrong choice for any of the aspects of an automated testing framework can hamper a search process. A poor representation does not capture sufficient and relevant information about the execution of the program, or its collection and processing is infeasible. This, in turn, produces a fitness function which yields inappropriate fitness values, i.e. a poor search landscape.

An inconvenient landscape has several undesirable characteristic. One, it has plateaus – regions where multiple candidate solutions have the same fitness value and are thus indistinguishable. Two, it has local optima where the search process may get stuck and produce a sub-optimal solution. Three, the search landscape does not impose a meaningful ordering. That is, inspecting the output of the fitness function will not inform the tester as to which candidate solution is more (dis)similar than another.

The problems caused by an inappropriate or insufficient combination of a representation and fitness function have a knock-on effect on the choices available for a search strategy. Two high level strategies are considered in this thesis: diversification and property targeting. Both of these will face issues due to poor representations and fitness functions. One, if the fitness function does not yield an estimate of proximity to the property of interest, a search strategy which targets that property cannot be used. Two, if the representation was looking at the wrong observations, the notion of diversity

will not be meaningful and thus a diversity-driven strategy will be weak.

This thesis presents methods for addressing the above issues of SBST with the aid of neural networks. It is argued, that NNs are particularly well suited for tackling these problems. One, they can process arbitrarily complex observations with minimal involvement from the tester. Two, they can discover relevant signals even in very verbose data, which may not be viable for a human. These properties mean that representations that were not formerly viable for SBST, can now be considered. Three, they are continuous by nature. This allows them to produce search landscapes that are smooth and large; search landscapes with desirable properties. Four, NNs naturally impose a quantifiable, numeric ordering on the underlying data. This property is useful for defining more refined search strategies and notions of similarity. Finally, NNs can be used for generation of new data, namely program inputs. This use case is not an explicit challenge of SBST addressed in this thesis, but as the use of GNNs in software testing is underexplored, the mechanism proposed in Chapter 5 is a contribution in its own right.

6.2 Contributions and Summary

This thesis explores the uses of neural networks for SBST in three main parts, presented in Chapters 3, 4 and 5 respectively. The proposed approaches are novel, both in the application of NNs to problems of SBST, as well as in the architectures of the NNs themselves. The empirical studies are largely exploratory, and the implementations are proofs of concept.

The first work presents the principle for the approach for constructing fitness landscapes, both for targeted and diversification strategies. It suggests that a fitness function for a property targeting search can be defined using a regressor NN trained on a corpus of execution traces and properties of interest. The NN's estimate of the presence of a property of interest can then be

interpreted as a fitness. The work also proposes a method for generating a search landscape for a diversity-driven search strategy. The approach relies on training an NN to identify the most salient features of data, in this case execution traces. This builds a smooth, continuous, n-dimensional encoding space over which, it is proposed, a diversification strategy ought to be defined. Although this work does not evaluate whether the landscapes are beneficial for discovering properties of interest, it does demonstrate that they possess beneficial properties, such as size and continuity. Furthermore, the landscape intended for a targeted strategy is strongly representative of various properties of interest, and the one intended for a diversification strategy imposes a meaningful ordering on candidate solutions. The work does not however evaluate the performance of these search landscapes in the context of search.

The second part of the thesis evaluates the performance of an NN-generated fitness function, which is attached to an SBST tool to drive a property targeting search strategy. First, a regressor is trained on a corpus of programs, their execution traces and crash/no-crash labels, much like proposed in the first work. Then the trained NN's output is interpreted as a fitness for use with an SBST tool, in this case the AFL fuzzer. The results indicate a clear, statistically significant positive effect on discovery of crashes. This suggests that the principle of constructing a property targeting fitness landscape for an SBST process is feasible.

Finally, the third part of the thesis implements a diversity driven search based on an NN-generated search landscape. The proposed framework, termed the Deconvolutional Generative Adversarial Fuzzer (DCGAF), is based on two NNs: a generative one for producing new string inputs for a program, and an autoencoder to evaluate the novelty of observed behaviour when the program is executed under the generated inputs. The work presents a number of findings. It shows that such a framework can be trained in principle, that it

can produce sensible program inputs, that the similarity of the inputs can be controlled, and that it can, in fact, discover crashes. In addition, it showcases a generative neural network that does not require training data, but instead uses an external oracle – the program under test.

The overarching contribution is the application of NNs in novel ways to SBST problems. This may not be of immediate use in and of itself, but it does illustrate the fact that NNs can be adapted to software testing problems.

6.3 Future Work

This thesis presents a number of new uses of NNs in automated software testing, and opens numerous directions for future work.

The following suggestions for future work are applicable to all approaches described in this thesis. First, additional experiments ought to be conducted using larger training datasets and additional real world programs. Although the results presented here are encouraging, they cannot be taken at face value without further investigation of applicability and generalisability. Second, various other representations can be considered. These may range from low level traces to, for instance, screenshots of a program's GUI as it is tested.

In addition, some future work ideas are specifically relevant to property targeting search. For instance, additional properties of interest ought to be studied. Without such a study, it is hard to speculate as to which properties can be searched for, given arbitrary observations. Other SBST tools than AFL ought to be considered. Finally, the fitness function that uses an NN's output prediction can be defined differently (see Subsection 4.3.5).

Last but not least, there are also several ways in which the work on diversity-driven search landscapes and DCGAF may be extended. It was shown that

the embedding of execution traces into a latent space has beneficial properties including continuity and ordering, but whether this notion of diversity is useful for testing remains to be seen. Initially, a comparison with other notions of diversity can be conducted. Then, property discovery rates of different notions of diversity can be compared. As for DCGAF, various directions for future work, including experiments with alternate architectures and use of prior knowledge were proposed in Section 5.6. In short, while promising, the DCGAF is a prototype that requires (and enables) additional work.

Bibliography

M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL `https://www.tensorflow.org/`. Software available from tensorflow.org.

N. Albunian, G. Fraser, and D. Sudholt. Causes and effects of fitness landscapes in unit test generation. In *GECCO 2020: Proceedings of the Genetic and Evolutionary Computation Conference*. ACM Digital Library, 2020.

A. Aleti, I. Moser, and L. Grunske. Analysing the fitness landscape of search-based software testing problems. *Automated Software Engineering*, 24(3): 603–621, 2017.

S. Ali, L. C. Briand, H. Hemmati, and R. K. Panesar-Walawege. A systematic review of the application and empirical investigation of search-based test case generation. *IEEE Transactions on Software Engineering*, 36(6):742–762, 2010.

N. Alshahwan and M. Harman. Coverage and fault detection of the output-

uniqueness test selection criteria. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pages 181–192. ACM, 2014.

M. Alshraideh and L. Bottaci. Search-based software test data generation for string data using program-specific search operators. *Software Testing, Verification and Reliability*, 16(3):175–203, 2006.

P. Ammann and J. Offutt. *Introduction to software testing*. Cambridge University Press, 2016.

S. Anand, E. K. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, P. Mcminn, A. Bertolino, et al. An orchestrated survey of methodologies for automated software test case generation. *Journal of Systems and Software*, 86(8):1978–2001, 2013.

C. Anderson, A. Von Mayrhauser, and R. Mraz. On the use of neural networks to guide software testing activities. In *Test Conference, 1995. Proceedings., International*, pages 720–729. IEEE, 1995.

A. Arcuri and L. Briand. Adaptive random testing: An illusion of effectiveness? In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, pages 265–275. ACM, 2011.

A. Arcuri, M. Z. Iqbal, and L. Briand. Formal analysis of the effectiveness and predictability of random testing. In *Proceedings of the 19th international symposium on Software testing and analysis*, pages 219–230. ACM, 2010.

B. Arkin, S. Stender, and G. McGraw. Software penetration testing. *IEEE Security & Privacy*, 3(1):84–87, 2005.

T. Avgerinos, S. K. Cha, A. Rebert, E. J. Schwartz, M. Woo, and D. Brumley. Automatic exploit generation. *Communications of the ACM*, 57(2):74–84, 2014.

M. Backes, M. Dürmuth, S. Gerling, M. Pinkal, and C. Sporleder. Acoustic side-channel attacks on printers. In *USENIX Security symposium*, pages 307–322, 2010.

S. Bai, J. Z. Kolter, and V. Koltun. An empirical evaluation of generic convolutional and recurrent networks for sequence modeling. *arXiv preprint arXiv:1803.01271*, 2018.

R. Baldoni, E. Coppa, D. C. D'elia, C. Demetrescu, and I. Finocchi. A survey of symbolic execution techniques. *ACM Computing Surveys (CSUR)*, 51 (3):50, 2018.

W. Banzhaf, P. Nordin, R. E. Keller, and F. D. Francone. *Genetic programming: an introduction*, volume 1. Morgan Kaufmann San Francisco, 1998.

C. Beleites, U. Neugebauer, T. Bocklitz, C. Krafft, and J. Popp. Sample size planning for classification models. *Analytica chimica acta*, 760:25–33, 2013.

R. Ben Abdessalem, S. Nejati, L. C. Briand, and T. Stifter. Testing advanced driver assistance systems using multi-objective search and neural networks. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pages 63–74. ACM, 2016.

M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2329–2344. ACM, 2017a.

M. Böhme, V.-T. Pham, and A. Roychoudhury. Coverage-based greybox fuzzing as markov chain. *IEEE Transactions on Software Engineering*, 2017b.

L. Bottaci. Instrumenting programs with flag variables for test data search by genetic algorithm. In *Proceedings of the 4th Annual Conference on Genetic and Evolutionary Computation*, pages 1337–1342. Morgan Kaufmann Publishers Inc., 2002.

L. Bottaci. Predicate expression cost functions to guide evolutionary search for test data. In *Genetic and Evolutionary Computation Conference*, pages 2455–2464. Springer, 2003.

S. R. Bowman, L. Vilnis, O. Vinyals, A. M. Dai, R. Jozefowicz, and S. Bengio. Generating sentences from a continuous space. *arXiv preprint arXiv:1511.06349*, 2015.

J. F. Bowring, J. M. Rehg, and M. J. Harrold. Active learning for automatic classification of software behavior. In *ACM SIGSOFT Software Engineering Notes*, volume 29, pages 195–205. ACM, 2004.

R. S. Boyer, B. Elspas, and K. N. Levitt. Select—a formal system for testing and debugging programs by symbolic execution. *ACM SigPlan Notices*, 10 (6):234–245, 1975.

L. C. Briand, Y. Labiche, and X. Liu. Using machine learning to support debugging with tarantula. In *Software Reliability, 2007. ISSRE'07. The 18th IEEE International Symposium on*, pages 137–146. IEEE, 2007.

L. C. Briand, Y. Labiche, and Z. Bawar. Using machine learning to refine black-box test specifications and test suites. In *Quality Software, 2008. QSIC'08. The Eighth International Conference on*, pages 135–144. IEEE, 2008.

A. E. Brownlee, J. R. Woodward, and J. Swan. Metaheuristic design pattern: surrogate fitness functions. In *Proceedings of the Companion Publication of the 2015 Annual Conference on Genetic and Evolutionary Computation*, pages 1261–1264, 2015.

B. R. Bruce, J. Petke, and M. Harman. Reducing energy consumption using genetic improvement. In *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*, pages 1327–1334. ACM, 2015.

R. Callan, F. Behrang, A. Zajic, M. Prvulovic, and A. Orso. Zero-overhead profiling via em emanations. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 401–412. ACM, 2016.

V. Chandola, A. Banerjee, and V. Kumar. Anomaly detection: A survey. *ACM computing surveys (CSUR)*, 41(3):15, 2009.

P. Chen and H. Chen. Angora: Efficient fuzzing by principled search. *arXiv preprint arXiv:1803.01307*, 2018.

T. Y. Chen, H. Leung, and I. Mak. Adaptive random testing. In *Annual Asian Computing Science Conference*, pages 320–329. Springer, 2004.

F. Chollet. Lstm text generation, 2017a. `https://github.com/fchollet/keras/blob/master/examples/lstm_text_generation.py`.

F. Chollet. Building autoencoders in keras, 2017b. `https://blog.keras.io/building-autoencoders-in-keras.html`.

F. Chollet et al. Keras: The python deep learning library, 2018. `https://keras.io/`.

J. Chung, C. Gulcehre, K. Cho, and Y. Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint arXiv:1412.3555*, 2014.

I. Ciupa, A. Leitner, M. Oriol, and B. Meyer. Artoo: adaptive random testing for object-oriented software. In *Proceedings of the 30th international conference on Software engineering*, pages 71–80. ACM, 2008.

C. Cortes and M. Mohri. Auc optimization vs. error rate minimization. In *Advances in neural information processing systems*, pages 313–320, 2004.

B. C. Csáji. Approximation with artificial neural networks. *Faculty of Sciences, Etvs Lornd University, Hungary*, 24:48, 2001.

A. Dertat. Applied deep learning - part 3: Autoencoders, 2017. `https://towardsdatascience.com/applied-deep-learning-part-3-autoencoders-1c083af4d798`.

J. Donahue, L. Anne Hendricks, S. Guadarrama, M. Rohrbach, S. Venugopalan, K. Saenko, and T. Darrell. Long-term recurrent convolutional networks for visual recognition and description. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2625–2634, 2015.

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

R. Feldt and S. Poulding. Finding test data with specific properties via metaheuristic search. In *Software Reliability Engineering (ISSRE), 2013 IEEE 24th International Symposium on*, pages 350–359. IEEE, 2013.

R. Feldt, S. Poulding, D. Clark, and S. Yoo. Test set diameter: Quantifying the diversity of sets of test cases. In *Software Testing, Verification and Validation (ICST), 2016 IEEE International Conference on*, pages 223–233. IEEE, 2016.

S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff. A sense of self for unix processes. In *Security and Privacy, 1996. Proceedings., 1996 IEEE Symposium on*, pages 120–128. IEEE, 1996.

P. G. Frankl and S. N. Weiss. An experimental comparison of the effectiveness of the all-uses and all-edges adequacy criteria. In *Proceedings of the symposium on Testing, analysis, and verification*, pages 154–164. ACM, 1991.

G. Fraser and A. Arcuri. 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*, pages 416–419. ACM, 2011.

G. Fraser and A. Arcuri. Whole test suite generation. *IEEE Transactions on Software Engineering*, 39(2):276–291, 2013.

G. Fraser and A. Zeller. Mutation-driven generation of unit tests and oracles. *IEEE Transactions on Software Engineering*, 38(2):278–292, 2012.

D. Gamble. Ultralightweight json parser in ansi c, 2019. `https://github.com/DaveGamble/cJSON`.

S. Gan, C. Zhang, X. Qin, X. Tu, K. Li, Z. Pei, and Z. Chen. Collafl: Path sensitive fuzzing. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 679–696. IEEE, 2018.

G. Gay, M. Staats, M. Whalen, and M. P. Heimdahl. The risks of coverage-directed test case generation. *IEEE Transactions on Software Engineering*, 41(8):803–819, 2015.

T. Glasmachers. Limits of end-to-end learning. *arXiv preprint arXiv:1704.08305*, 2017.

X. Glorot, A. Bordes, and Y. Bengio. Deep sparse rectifier neural networks. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, pages 315–323, 2011.

P. Godefroid, N. Klarlund, and K. Sen. Dart: directed automated random testing. In *ACM Sigplan Notices*, volume 40, pages 213–223. ACM, 2005.

P. Godefroid, H. Peleg, and R. Singh. Learn&fuzz: Machine learning for input fuzzing. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, pages 50–59. IEEE Press, 2017.

K. Gödel. Über formal unentscheidbare sätze der principia mathematica und verwandter systeme i. *Monatshefte für mathematik und physik*, 38(1): 173–198, 1931.

T. F. Gonzalez. Clustering to minimize the maximum intercluster distance. *Theoretical Computer Science*, 38:293–306, 1985.

I. Goodfellow. Generative adversarial networks for text, 2016. `https://www.reddit.com/r/MachineLearning/comments/40ldq6/generative_adversarial_networks_for_text/`.

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

Google. Framing: Key ml terminology, 2018. `https://developers.google.com/machine-learning/crash-course/framing/ml-terminology`.

A. D. Gordon and T. Melham. Five axioms of alpha-conversion. In *International Conference on Theorem Proving in Higher Order Logics*, pages 173–190. Springer, 1996.

A. Graves. Generating sequences with recurrent neural networks. *arXiv preprint arXiv:1308.0850*, 2013.

A. Graves, A.-r. Mohamed, and G. Hinton. Speech recognition with deep recurrent neural networks. In *Acoustics, speech and signal processing (icassp), 2013 ieee international conference on*, pages 6645–6649. IEEE, 2013.

T. Guardian. This article is more than 1 year old macos high sierra bug: blank password let anyone take control of a mac, 2017. https://www.theguardian.com/technology/2017/nov/29/macos-high-sierra-bug-apple-mac-unlock-blank-password-security-flaw.

M. Harman. The current state and future of search based software engineering. In *2007 Future of Software Engineering*, pages 342–357. IEEE Computer Society, 2007.

M. Harman and J. Clark. Metrics are fitness functions too. In *10th International Symposium on Software Metrics, 2004. Proceedings.*, pages 58–69. Ieee, 2004.

M. Harman and B. F. Jones. Search-based software engineering. *Information and software Technology*, 43(14):833–839, 2001.

M. Harman, R. Hierons, and M. Proctor. A new representation and crossover operator for search-based optimization of software modularization. In *Proceedings of the 4th Annual Conference on Genetic and Evolutionary Computation*, pages 1351–1358. Morgan Kaufmann Publishers Inc., 2002a.

M. Harman, L. Hu, R. Hierons, A. Baresel, and H. Sthamer. Improving evolutionary testing by flag removal. In *Proceedings of the 4th Annual Conference on Genetic and Evolutionary Computation*, pages 1359–1366. Morgan Kaufmann Publishers Inc., 2002b.

M. Harman, L. Hu, R. M. Hierons, J. Wegener, H. Sthamer, A. Baresel, and M. Roper. Testability transformation. Institute of Electrical and Electronics Engineers, 2004.

M. Harman, S. A. Mansouri, and Y. Zhang. Search-based software engineering: Trends, techniques and applications. *ACM Computing Surveys (CSUR)*, 45(1):11, 2012.

M. Harman, Y. Jia, and Y. Zhang. Achievements, open problems and challenges for search based software testing. In *Software Testing, Verification and Validation (ICST), 2015 IEEE 8th International Conference on*, pages 1–12. IEEE, 2015.

S. Haykin. *Neural networks: a comprehensive foundation*. Prentice Hall PTR, 1994.

K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016a.

K. He, X. Zhang, S. Ren, and J. Sun. Identity mappings in deep residual networks. In *European conference on computer vision*, pages 630–645. Springer, 2016b.

M. P. Heimdahl and D. George. Test-suite reduction for model based tests: Effects on test quality and implications for testing. In *Proceedings of the 19th IEEE international conference on Automated software engineering*, pages 176–185. IEEE Computer Society, 2004.

M. P. E. Heimdahl, D. George, and R. Weber. Specification test coverage adequacy criteria= specification test generation inadequacy criteria. In *High Assurance Systems Engineering, 2004. Proceedings. Eighth IEEE International Symposium on*, pages 178–186. IEEE, 2004.

S. Hochreiter. The vanishing gradient problem during learning recurrent neural nets and problem solutions. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 6(02):107–116, 1998.

S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

J. H. Holland. Adaptation in natural and artificial systems. an introductory analysis with application to biology, control, and artificial intelligence. *Ann Arbor, MI: University of Michigan Press*, pages 439–444, 1975.

J. Huang and C. X. Ling. Using auc and accuracy in evaluating learning algorithms. *IEEE Transactions on knowledge and Data Engineering*, 17 (3):299–310, 2005.

M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments of the effectiveness of dataflow-and controlflow-based test adequacy criteria. In *Proceedings of the 16th international conference on Software engineering*, pages 191–200. IEEE Computer Society Press, 1994.

L. Inozemtseva and R. Holmes. Coverage is not strongly correlated with test suite effectiveness. In *Proceedings of the 36th International Conference on Software Engineering*, pages 435–445. ACM, 2014.

S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

D. Jones. Sparse - a semantic parser for c, 2019. `https://sparse.wiki.kernel.org/index.php/Main_Page`.

N. Kalchbrenner, E. Grefenstette, and P. Blunsom. A convolutional neural network for modelling sentences. *arXiv preprint arXiv:1404.2188*, 2014.

A. Karpathy. Convolutional neural networks (cnns / convnets), 2016. `http://cs231n.github.io/convolutional-networks/`.

A. Karpathy, G. Toderici, S. Shetty, T. Leung, R. Sukthankar, and L. Fei-Fei. Large-scale video classification with convolutional neural networks. In *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*, pages 1725–1732, 2014.

T. Karras, S. Laine, and T. Aila. A style-based generator architecture for generative adversarial networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4401–4410, 2019.

K. Kawaguchi. Deep learning without poor local minima. In *Advances in Neural Information Processing Systems*, pages 586–594, 2016.

P. Kawthekar, R. Rewari, and S. Bhooshan. Evaluating generative models for text generation, 2017.

Y. Kim. Convolutional neural networks for sentence classification. *arXiv preprint arXiv:1408.5882*, 2014.

J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

D. P. Kingma and M. Welling. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*, 2013.

P. C. Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In *Annual International Cryptology Conference*, pages 104–113. Springer, 1996.

P. S. Kochhar, F. Thung, and D. Lo. Code coverage and test suite effectiveness: Empirical study with real bugs in large systems. In *Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd International Conference on*, pages 560–564. IEEE, 2015.

B. Korel. Automated software test data generation. *IEEE Transactions on software engineering*, 16(8):870–879, 1990.

B. Korel. Dynamic method for software test data generation. *Software Testing, Verification and Reliability*, 2(4):203–213, 1992.

A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

R. Lämmel. Grammar testing. In *International Conference on Fundamental Approaches to Software Engineering*, pages 201–216. Springer, 2001.

T. Lane, P. Gladstone, J. Boucher, L. Crocker, J. Minguillon, L. Ortiz, G. Phillips, D. Rossi, G. Vollbeding, and G. Weijers. libjpeg 6b, 1998. `http://libjpeg.sourceforge.net/`.

Y. LeCun and C. Cortes. The mnist database, 2017. `http://yann.lecun.com/exdb/mnist/`.

P. Leray and P. Gallinari. Feature selection with neural networks. *Behaviormetrika*, 26(1):145–166, 1999.

V. I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady*, volume 10, pages 707–710, 1966.

J. Li, K. Cheng, S. Wang, F. Morstatter, R. P. Trevino, J. Tang, and H. Liu. Feature selection: A data perspective. *ACM Computing Surveys (CSUR)*, 50(6):94, 2018a.

J. Li, B. Zhao, and C. Zhang. Fuzzing: a survey. *Cybersecurity*, 1(1):6, 2018b.

D. Lo, H. Cheng, J. Han, S.-C. Khoo, and C. Sun. Classification of software behaviors for failure detection: a discriminative pattern mining approach. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 557–566. ACM, 2009.

C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Acm sigplan notices*, volume 40, pages 190–200. ACM, 2005.

C. Lv, S. Ji, Y. Li, J. Zhou, J. Chen, P. Zhou, and J. Chen. Smartseed: Smart seed generation for efficient fuzzing. *arXiv preprint arXiv:1807.02606*, 2018.

Y. Mao, F. Boqin, Z. Li, and L. Yao. Neural networks based automated test oracle for software testing. In *International Conference on Neural Information Processing*, pages 498–507. Springer, 2006.

D. Maynor. *Metasploit toolkit for penetration testing, exploit development, and vulnerability research*. Elsevier, 2011.

P. McMinn. Search-based software test data generation: a survey. *Software testing, Verification and reliability*, 14(2):105–156, 2004.

P. McMinn. Search-based software testing: Past, present and future. In *Software testing, verification and validation workshops (icstw), 2011 ieee fourth international conference on*, pages 153–163. IEEE, 2011.

P. McMinn, D. Binkley, and M. Harman. Empirical evaluation of a nesting testability transformation for evolutionary testing. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 18(3):11, 2009.

G. Melis, C. Dyer, and P. Blunsom. On the state of the art of evaluation in neural language models. *arXiv preprint arXiv:1707.05589*, 2017.

N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, and E. Teller. Equation of state calculations by fast computing machines. *The journal of chemical physics*, 21(6):1087–1092, 1953.

T. Mikolov, K. Chen, G. Corrado, and J. Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

B. P. Miller, L. Fredriksen, and B. So. An empirical study of the reliability of unix utilities. *Communications of the ACM*, 33(12):32–44, 1990.

B. S. Mitchell and S. Mancoridis. Using heuristic search techniques to extract design abstractions from source code. In *Proceedings of the 4th Annual Conference on Genetic and Evolutionary Computation*, pages 1375–1382. Morgan Kaufmann Publishers Inc., 2002.

G. J. Myers, C. Sandler, and T. Badgett. *The art of software testing*. John Wiley & Sons, 2011.

A. S. Namin and J. H. Andrews. The influence of size and coverage on test suite effectiveness. In *Proceedings of the eighteenth international symposium on Software testing and analysis*, pages 57–68. ACM, 2009.

N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *ACM Sigplan notices*, volume 42, pages 89–100. ACM, 2007.

N. Nethercote and J. Seward. Cachegrind: a cache and branch-prediction profiler, 2017a. `http://valgrind.org/docs/manual/cg-manual.html`.

N. Nethercote and J. Seward. Memcheck: a memory error detector, 2017b. `http://valgrind.org/docs/manual/mc-manual.html`.

N. Nethercote and J. Seward, 2017c. `http://valgrind.org/gallery/users.html`.

Q. Nguyen and M. Hein. The loss surface of deep and wide neural networks. *arXiv preprint arXiv:1704.08045*, 2017.

Q. Nguyen and M. Hein. Optimization landscape and expressivity of deep cnns. In *International Conference on Machine Learning*, pages 3727–3736, 2018.

C. Olah. Understanding lstm networks, 2015. `http://colah.github.io/posts/2015-08-Understanding-LSTMs/`.

OpenAI. Generative models, 2016. `https://blog.openai.com/generative-models/`.

C. Pacheco and M. D. Ernst. Eclat: Automatic generation and classification of test inputs. In *European Conference on Object-Oriented Programming*, pages 504–527. Springer, 2005.

R. Pascanu, T. Mikolov, and Y. Bengio. Understanding the exploding gradient problem. *CoRR, abs/1211.5063*, 2, 2012.

G. Perarnau. Fantastic gans and where to find them, 2017. `https://guimperarnau.com/blog/2017/03/Fantastic-GANs-and-where-to-find-them`.

J. Petke, S. Haraldsson, M. Harman, D. White, J. Woodward, et al. Genetic improvement of software: a comprehensive survey. *IEEE Transactions on Evolutionary Computation*, 2017.

T. Petsios, J. Zhao, A. D. Keromytis, and S. Jana. Slowfuzz: Automated domain-independent detection of algorithmic complexity vulnerabilities. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2155–2168. ACM, 2017.

A. Radford, L. Metz, and S. Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks. *arXiv preprint arXiv:1511.06434*, 2015.

M. Rajpal, W. Blum, and R. Singh. Not all bytes are equal: Neural byte sieve for fuzzing. *arXiv preprint arXiv:1711.04596*, 2017.

S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos. Vuzzer: Application-aware evolutionary fuzzing. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2017.

R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

S. Rostedt. ftrace - function tracer, 2017. `https://www.kernel.org/doc/Documentation/trace/ftrace.txt`.

D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533, 1986.

S. Sahoo. Residual blocks — building blocks of resnet, 2018. `https://towardsdatascience.com/residual-blocks-building-blocks-of-resnet-fd90ca15d6ec`.

T. Salimans and D. P. Kingma. Weight normalization: A simple reparameterization to accelerate training of deep neural networks. In *Advances in Neural Information Processing Systems*, pages 901–909, 2016.

S. Santurkar, D. Tsipras, A. Ilyas, and A. Madry. How does batch normalization help optimization? In *Advances in Neural Information Processing Systems*, pages 2483–2493, 2018.

R. L. Sauder. A general test data generator for cobol. In *Proceedings of the May 1-3, 1962, spring joint computer conference*, pages 317–323. ACM, 1962.

K. Sen. Concolic testing. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 571–572. ACM, 2007.

D. She, K. Pei, D. Epstein, J. Yang, B. Ray, and S. Jana. Neuzz: Efficient fuzzing with neural program learning. *arXiv preprint arXiv:1807.05620*, 2018.

M. Shepperd. *Fundamentals of software measurement*. Prentice-Hall, 1995.

R. Spreitzer, V. Moonsamy, T. Korak, and S. Mangard. Systematic classification of side-channel attacks: a case study for mobile devices. 2018.

N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

M. Staats, G. Gay, M. Whalen, and M. Heimdahl. On the danger of coverage directed test case generation. In *Fundamental Approaches to Software Engineering*, pages 409–424. Springer, 2012.

N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *NDSS*, volume 16, pages 1–16, 2016.

M. Sundermeyer, R. Schlüter, and H. Ney. Lstm neural networks for language modeling. In *Thirteenth annual conference of the international speech communication association*, 2012.

I. Sutskever, J. Martens, and G. E. Hinton. Generating text with recurrent neural networks. In *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, pages 1017–1024, 2011.

M. Sutton, A. Greene, and P. Amini. *Fuzzing: brute force vulnerability discovery*. Pearson Education, 2007.

G. Swirszcz, W. M. Czarnecki, and R. Pascanu. Local minima in training of neural networks. *arXiv preprint arXiv:1611.06310*, 2016.

C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. Goodfellow, and R. Fergus. Intriguing properties of neural networks. *arXiv preprint arXiv:1312.6199*, 2013.

S. H. Tan, J. Yi, S. Mechtaev, A. Roychoudhury, et al. Codeflaws: a programming competition benchmark for evaluating automated program repair tools. In *Proceedings of the 39th International Conference on Software Engineering Companion*, pages 180–182. IEEE Press, 2017.

L. Tiao. Implementing variational autoencoders in keras: Beyond the quickstart tutorial, 2017. `http://tiao.io/posts/implementing-variational-autoencoders-in-keras-beyond-the-quickstart-tutorial/`.

T. Tieleman and G. Hinton. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural networks for machine learning*, 4(2):26–31, 2012.

A. Van Den Oord, S. Dieleman, H. Zen, K. Simonyan, O. Vinyals, A. Graves, N. Kalchbrenner, A. Senior, and K. Kavukcuoglu. Wavenet: A generative model for raw audio. *arXiv preprint arXiv:1609.03499*, 2016.

M. Vanmali, M. Last, and A. Kandel. Using a neural network in the software testing process. *International Journal of Intelligent Systems*, 17(1):45–62, 2002.

L. Vanneschi, M. Tomassini, P. Collard, and S. Vérel. Negative slope coefficient: A measure to characterize genetic programming fitness landscapes. In *European Conference on Genetic Programming*, pages 178–189. Springer, 2006.

V. K. Vassilev, T. C. Fogarty, and J. F. Miller. Information characteristics and the structure of landscapes. *Evolutionary computation*, 8(1):31–60, 2000.

D. Veillard. The xml c parser and toolkit of gnome, 2019. `http://xmlsoft.org/`.

A. Verikas and M. Bacauskiene. Feature selection with neural networks. *Pattern Recognition Letters*, 23(11):1323–1335, 2002.

w3schools. Xml syntax rules, 2019. `https://www.w3schools.com/xml/xml_syntax.asp`.

H. Wang, Z. Qin, and T. Wan. Text generation based on generative adversarial nets with latent variables. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining*, pages 92–103. Springer, 2018.

J. Wang, B. Chen, L. Wei, and Y. Liu. Skyfire: Data-driven seed generation for fuzzing. In *Security and Privacy (SP), 2017 IEEE Symposium on*, pages 579–594. IEEE, 2017.

Q. Wang, J. Zhang, S. Song, and Z. Zhang. Attentional neural network: Feature selection using cognitive feedback. In *Advances in Neural Information Processing Systems*, pages 2033–2041, 2014.

T. Wang, D. J. Wu, A. Coates, and A. Y. Ng. End-to-end text recognition with convolutional neural networks. In *Pattern Recognition (ICPR), 2012 21st International Conference on*, pages 3304–3308. IEEE, 2012.

J. Wegener and M. Grochtmann. Verifying timing constraints of real-time systems by means of evolutionary testing. *Real-Time Systems*, 15(3):275–298, 1998.

J. Wegener, A. Baresel, and H. Sthamer. Evolutionary test environment for automatic structural testing. *Information and Software Technology*, 43 (14):841–854, 2001.

J. Weidendorfer. Callgrind: a call-graph generating cache and branch prediction profiler, 2017. `http://valgrind.org/docs/manual/cl-manual.html`.

E. Weinberger. Correlated and uncorrelated fitness landscapes and how to tell the difference. *Biological cybernetics*, 63(5):325–336, 1990.

E. D. Weinberger. Local properties of kauffman's n-k model: A tunably rugged energy landscape. *Physical Review A*, 44(10):6399, 1991.

M. Whalen, G. Gay, D. You, M. P. Heimdahl, and M. Staats. Observable modified condition/decision coverage. In *Software Engineering (ICSE), 2013 35th International Conference on*, pages 102–111. IEEE, 2013.

D. Whitley. A genetic algorithm tutorial. *Statistics and computing*, 4(2): 65–85, 1994.

F. Wilcoxon. Individual comparisons by ranking methods. In *Breakthroughs in statistics*, pages 196–202. Springer, 1992.

A. C. Wilson, R. Roelofs, M. Stern, N. Srebro, and B. Recht. The marginal value of adaptive gradient methods in machine learning. In *Advances in Neural Information Processing Systems*, pages 4148–4158, 2017.

F. Wu, W. Weimer, M. Harman, Y. Jia, and J. Krinke. Deep parameter optimisation. In *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*, pages 1375–1382. ACM, 2015.

M. Zalewski. American fuzzy lop, 2007. `http://lcamtuf.coredump.cx/afl/`.

M. Zalewski. Binary fuzzing strategies: What works and what doesn't, 2014. https://lcamtuf.blogspot.co.uk/2014/08/binary-fuzzing-strategies-what-works.html.

H. Zhang, T. Xu, H. Li, S. Zhang, X. Wang, X. Huang, and D. N. Metaxas. Stackgan: Text to photo-realistic image synthesis with stacked generative adversarial networks. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 5907–5915, 2017.

Y. Zhou and D. Feng. Side-channel attacks: Ten years after its publication and the impacts on cryptographic module security testing. *IACR Cryptology ePrint Archive*, 2005:388, 2005.