Constructing Search Spaces for Search-Based Software Testing Using Neural Networks

Leonid Joffe(✉) and David Clark

University College London, Gower Street, London WC1E 6BT, UK
leonid.joffe.14@ucl.ac.uk

Abstract. A central requirement for any Search-Based Software Testing (SBST) technique is a convenient and meaningful fitness landscape. Whether one follows a targeted or a diversification driven strategy, a search landscape needs to be large, continuous, easy to construct and representative of the underlying property of interest. Constructing such a landscape is not a trivial task often requiring a significant manual effort by an expert.

We present an approach for constructing meaningful and convenient fitness landscapes using neural networks (NN) – for targeted and diversification strategies alike. We suggest that output of an NN predictor can be interpreted as a fitness for a targeted 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 convenient search space which is strongly representative of a number of properties. We believe that such a search space can be readily used for driving a search towards specific properties of interest.

For a diversification strategy, we propose the use of an autoencoder; a mechanism for compacting data into an n-dimensional "latent" space. In it, datapoints are arranged according to the similarity of their salient features. We show that a latent space of execution traces possesses characteristics of a convenient search landscape: it is continuous, large and crucially, it defines a notion of similarity to arbitrary observations.

Keywords: Search-Based Software Testing · Software engineering · Fitness function · Machine learning · Neural networks

1 Introduction

Search Based Software Testing (SBST) [16,30] methods are widely used in software engineering. They rely on a feedback mechanism that evaluates candidate solutions and directs the search accordingly. The effectiveness of any feedback mechanism depends on the choice of representation and fitness function [15]. In the context of automated search driven testing, an additional choice is that of

© Springer Nature Switzerland AG 2019
S. Nejati and G. Gay (Eds.): SSBSE 2019, LNCS 11664, pp. 27–41, 2019.
https://doi.org/10.1007/978-3-030-27455-9_3

a search strategy. In this paper we focus on constructing a convenient fitness function for a search-based testing process.

According to Harman and Clark, the search space of a good fitness function ought to have a number of desirable characteristics [14]. It needs to be large and approximately continuous, the fitness function needs to have low computational complexity and 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, a good metric needs to be truly representative of the property it seeks to denote [38]. Second, a metric imposes an order relation over a set of elements by definition, and for a metric to be useful as a fitness function, the order needs to be meaningful. In this paper we present an approach for constructing fitness functions with desirable characteristics for two fundamental testing strategies – property targeting and diversification driven.

1.1 Property Targeting Search Landscape

A fitness function for an execution property targeting search strategy needs to indicate a "proximity" of a candidate solution to a property of interest (given that the property has not been yet observed). 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 [45]. 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?

A neural network trained on execution traces and crash/no crash labels can produce a "suspiciousness" score for each candidate solution. So rather than simply observing a "no crash" output, we query a neural network to say that some inputs exhibited a behaviour or "looks suspiciously like a crash". In this work we show how such a fitness function can be constructed, and that it possesses useful characteristics.

1.2 Diversity Driven Search Landscape

Diversity is widely accepted as beneficial for testing. Various representations have been proposed as targets for diversification, e.g. [2,5,6,10]. Perhaps the most common manifestation is code coverage, yet the effectiveness of coverage driven testing strategies has been disputed [11,17,20,27,40]. This suggests that diversifying over coverage – i.e. 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, we need a notion of similarity. 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 [38]. For instance, is "*cat*" < "*dog*"? Lexicographically – yes. By average weight of the animal – usually. By preference as a pet – debatable.

We propose defining an order relation and thus similarity using a neural network architecture called an autoencoder to process execution traces. An autoencoder is trained to reproduce input data on outputs. Its (n-dimensional) intermediate layer forms an encoding of the data known as a *latent* space. The autoencoder arranges the data based on the 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. Importantly, an autoencoder architecture can be applied to arbitrary data formats. This means that we are not restricted to any particular representation of execution traces. We believe that this notion of similarity can be useful for diversification strategies.

1.3 Contributions and Scope

In this paper we propose an approach to building search landscapes for SBST by using neural networks to process observations of executions. The approach relies on predictor and autoencoder neural networks for property targeting and diversification driven testing strategies respectively. We illustrate the approach with a corpus of small C programs and several real world applications.

Our 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.

This work is part of a larger effort in which we intend to integrate these landscapes for use in SBST. The scope of this paper is to present the search landscapes themselves, along with an analysis of their characteristics. Here we do *not* evaluate their effectiveness for discovering properties of interest.

The section following this introduction presents the tools and datasets used in our experiments. Section 3 describes the experiments we carried out. Section 4 reports our findings. Finally, Sect. 5 summarises and concludes the paper.

2 Tools and Datasets

2.1 AFL

We use the American Fuzzy Lop (AFL) [46] fuzzer [33] for two purposes. First, to augment a training corpus of programs with additional inputs. Second, to produce a representation of execution traces to train autoencoders. AFL's representation is the following. Before fuzzing, AFL instruments a program at every decision point. During fuzzing, transitions between these points form a hashmap ("bitmap") of edges and their hit counts. For performance purposes, hit counts are assigned into eight buckets: 0, 1, 2, 3, 4–7, 8–15, 16–31, 32–127, 128+. The bitmap also 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.

AFL's representation is suitable for our second experiment (described below) for three reasons. First, the bucketisation of the bitmap and the fixed size make it convenient for processing by a neural network. 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. Finally, AFL has a built in notion of "interestingness", defined over the hit counts of a bitmap. All inputs it deems interesting are kept in a persistent queue for further fuzzing.

2.2 Pin

We use the *Pin* instrumentation framework [29] to collect execution traces as sequences of instruction. Raw instruction sequence data is inconvenient for two reasons however. First, the traces 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 0x1132` 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 `0x1132` in program A is not meaningful for program B. This is a major problem known as alpha renaming [13].

We bypass the above problems as follows. First, we use *Pin's* built in ability to only instrument the first instance of a block execution. For instance, a loop body is only recorded the first time it is executed. This reduces the sizes of traces dramatically while maintaining information on the sequence of events. The problem of alpha renaming is ignored by discarding any literal data. Thus `jle 0x1132` 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 paper. Furthermore, the sequence of op-codes is expected to provide enough information for our purposes.

2.3 Valgrind

Valgrind is a powerful instrumentation framework which tracks every instruction as it executes a program in a simulated environment [34,43]. We use two of its tools, Memcheck and Cachegrind, to record properties of interest (properties that a search aims to discover) for our datasets.

Memcheck reports properties relating to memory management. We record 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.

The first three are self explanatory. "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. Neither of these issues are necessarily crucial problems and we include them in our experiments as a proof of concept: that a proximity to a rare, as yet unobserved property – "a needle in a haystack" – can be characterised by features of an execution trace as interpreted by a neural network.

Cachegrind 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).

These values are used as an example of a numeric property which might be the target of optimisation in SBST. As any execution has a numeric value of a cache behaviour (i.e. it is not a rare binary property), the use case here is *not* to build a search space representing the proximity to a rare behaviour. Instead, it may be the case that 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 is known to become unstable [37]. We therefore log-normalise them. Not only does this make the values amenable to training a neural network, we believe that an order-of-magnitude estimate of these values is an interesting property.

2.4 Dataset

Our dataset is based on a large repository of simple C programs called Codeflaws [42], and five real world applications.

Codeflaws. Codeflaws is a program repository of thousands of small C programs, along with test cases and automatic fix scripts. Although the intended purpose of Codeflaws is to allow for a "comprehensive investigation of the set of repairable defect classes by various program repair tools", we chose to use it because it provides a vast number of varied programs conveniently arranged.

The neural networks of our approach require large training datasets, so the test cases of the repository were not sufficient. Additional inputs were therefore generated by fuzzing. Each program was fuzzed with AFL to produce a grand total of 365,393 executions across 4714 unique programs. This dataset was then split into training, testing and validation datasets. The number of unique programs and inputs were 3978 and 303,233 for the test set, 587 and 52,092 for the test set, and 149 and 10,068 for the validation set.

Real World Applications. We use five real world programs in our experiments. The first one is *lintxml* from libxml [44]. It processes a string input to determine whether it is valid XML. The second is *cjpeg* from libjpeg [28]. It is used for compressing image files into jpeg format. The third program is *sed-4.5* [31], the Unix stream editor for filtering and transforming text [4]. The fourth program is *sparse* [21], a lightweight parser for C. Finally, *cjson* is a parser for the JSON format. These programs were chosen because they are open source, sufficiently quick to fuzz, and their inputs can be easily interpreted. Furthermore, as we aim to investigate the order relation of a latent space, programs that take string inputs are of interest.

3 Experimental Setup

We conducted 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.

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 Fig. 1. The characteristics of the datasets for this experiment are summarised in Table 1.

The network is made up of convolutional and recurrent layers. Sequence data is typically handled with recurrent cells such as the LSTM [18]. 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 [9,22].

The network takes a *Pin* trace as input. The second layer is 64-dimensional embedding [32]. 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 [39]. 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 [25]. The parameters were tuned manually by observing the performance on the validation dataset.

Exp. 2: Search Landscape for a Diversity Driven Strategy. We construct a search landscape for a diversification strategy using a variational autoencoder [8,26]. It composed of an encoder and a decoder. The encoder takes AFL's

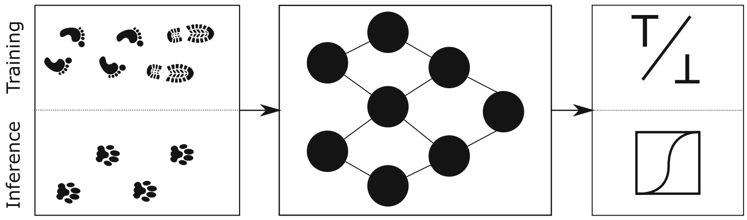

Fig. 1. Illustration of the setup for Exp. 1. A neural network is trained on execution traces of *Pin* instrumented programs as inputs, and properties of interest as prediction targets. During inference, it outputs an estimate of the property as a probability in [0, 1] or a numeric value for categorical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

Table 1. Statistics of the programs and properties of interest in our dataset for Exp. 1.

	CF Train	*CF Test*	*Cjpeg*	*Sparse*	*Cjson*
Total traces	43685	43685	22396	1260	1000
crashes	4458	4458	1722	260	0
deflost_blocks	163	163	0	9	187
illegal_reads	9149	9149	3781	49	0
illegal_writes	626	626	0	0	0
reachable_blocks	1141	1141	16779	0	813
uninit_values	195	195	934	0	0

bitmap representation of an execution trace as input. The hidden layer is a ReLu [23] activated densely connected layer of 2048 neurons. This is followed by a 3-dimensional encoding layer. The decoder has a symmetrical structure 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. [26], 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. The modified AFL 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: the traces of AFL's queue and traces of AFL's latest candidate solutions. During inference, we encode all elements in AFL's queue into the latent space.

4 Evaluation

We present three results. First, we show that the search landscapes are continuous and arbitrarily large. Second, we demonstrate that they are correlated with various properties of interest. Third, we suggest that the latent space produces a meaningful ordering on a set of seemingly non-orderable candidate solutions. We believe these search landscapes to be of potential use for both property targeting and diversification driven search strategies.

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 [1]. 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. Our landscapes 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, we argue our claims of continuity and size with the following facts and findings. First, neural networks are continuous by construction [12]. This suggests 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. Second, we observe that in both sets of experiments, the ratio of fitness values

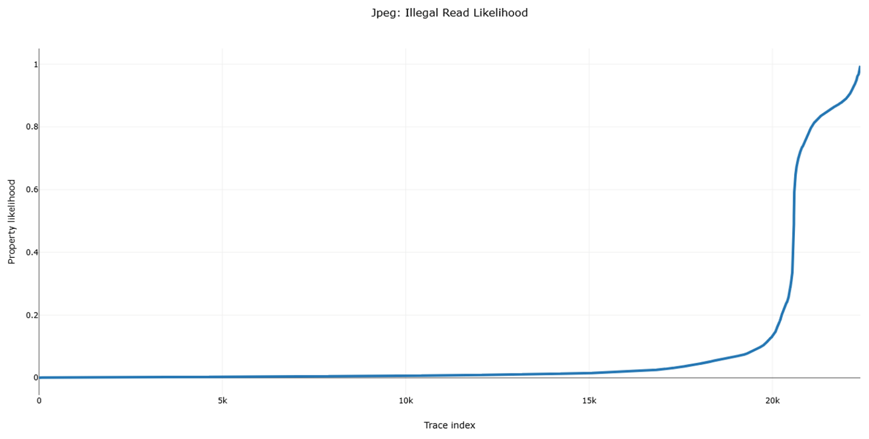

Fig. 2. A plot of the output of a neural network classifier showing its likelihood estimate of whether a trace included an illegal write, for the Jpeg testing dataset. The classifier is trained on the Codeflaws train dataset, with *Pin* execution traces as inputs and an illegal write error as the prediction label output. We suggest that this likelihood can be used as a fitness for a property targeting search strategy. Such a strategy would prioritise candidate solutions that the classifier considers to be more "suspicious".

to the number of unique traces was over 0.95. That is, most distinct traces were mapped to a distinct point in the fitness landscape. Figures 2 and 3 are examples of a property targeting and diversification driven landscape respectively.

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 suggests that they meet the representation condition.

We support this argument with the numeric results of Exp. 1, summarised in Tables 2 and 3. Table 2 shows the Area Under Curve for the Receiver Operator Characteristic (ROC). 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 label class size independence [7,19], which makes it a more honest measure of a model's performance.

High values in Table 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, it 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 Jpeg dataset has a low ROC score; the model's understanding of execution trace features indicative of this property is insufficiently general.

Table 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 others. The Cjson test set is an exception in that the models

Table 2. The predictive ability of a neural network for categorical properties 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 our dataset.

	CF test	*Jpeg*	*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. The predictive ability of a neural network for numeric properties 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.

	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%

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 these inherent similarities is an interesting direction of future work but out of scope for this paper.

The results presented here are an instantiation of our proposed approach – they are conditional on the representation, the properties of interest and the training dataset. We expect that given a larger, more representative dataset our 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 [24,35,36,41]. That is, if there is a pattern in the data, a neural network will find it. We recognise the "Deus ex machina" (or rather, "Deus ex data") nature of this argument: *given enough data*, a neural network turns into a silver bullet. Nonetheless, even with the limited dataset, our results demonstrate a clear effectiveness of the technique.

4.3 Meaningful Ordering of Candidate Solutions

The techniques proposed in this work can induce a meaningful ordering given 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. We suggest that a latent space of an autoencoder has a ordering that is meaningful with respect to *features* of observations.

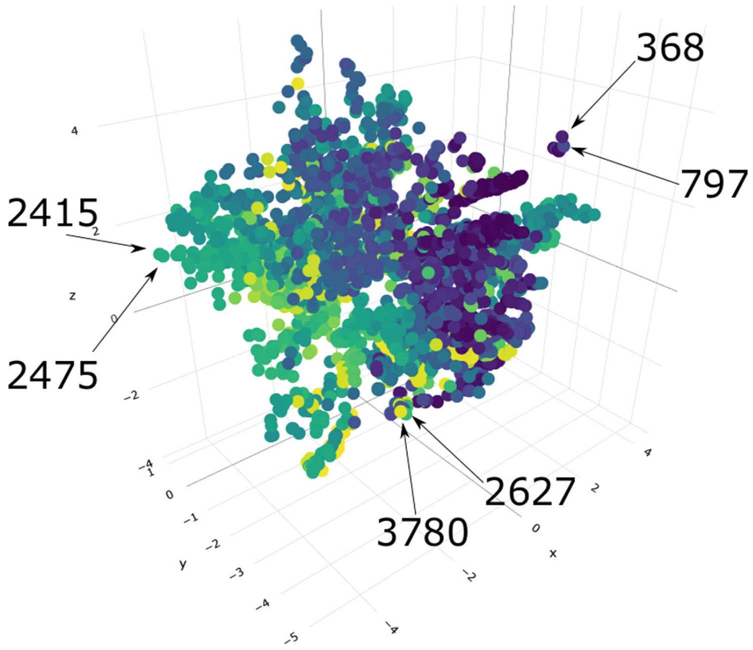

Fig. 3. 3-dimensional latent space encoding of the execution traces of the AFL queue for *xmllint*. The position of each point in the latent space is determined by characteristics of execution traces that the autoencoder found most useful for distinguishing one trace from another. The points are coloured by the sequential index of the queue elements, which allows AFL's search process to be visualised. Whilst the candidate solutions are spread throughout the latent space, there are regions with denser clusters and a diversity driven search strategy could be directed to explore the less populated regions. The numbers are ids of example candidate solutions discussed below. (Color figure online)

Figure 3 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 state 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.

We present several findings of the nature of this landscape. First, 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 search progressed, novel behaviours (green clusters) were discovered. AFL then turned

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

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 on its own, it nonetheless allows us to visualise and conceptualise a search process.

Second, we note that the arrangement of points in the space is not uniform. The autoencoder is strongly regularised to attempt to arrange the points close to zero (L2 regularisation) 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 suggests that some kinds of executions are relatively more explored.

Third, upon closer manual inspection of several candidate solutions, we note that the locality is related to program inputs. Consider the candidate solutions pointed to by arrows in Fig. 3. The inputs 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) may be 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 arbitrary data. Furthermore, its definition requires no manual effort.

We suggest the following implications based on the above observations. First, a latent space representation gives us 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, we 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].

[2] Böhme et al. showed that enforcing diversity on AFL's search is beneficial [3]. In future work we intend to investigate how the effectiveness using their notion of diversity compares with that proposed here.

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.

We propose the use of neural networks for constructing search landscapes with convenient characteristics for both property targeting and diversity driven search strategies. We suggest that a property targeting search strategy can use a landscape produced by a classifier neural network, and we illustrate this by experiment. Our results show that the landscape is continuous, arbitrarily large and representative of various properties of interest. For a diversity driven strategy, we propose constructing a search landscape using autoencoders. An autoencoder maps arbitrary observations onto an n-dimensional space where the location is determined by the most distinguishing features of the data. We show how such a space can be created and illustrate that it possesses useful characteristics such as size, continuity and meaningful ordering.

The results and experiments of this paper present the approach of constructing search landscapes, and comment on their characteristics. To the best of our knowledge, our approach is conceptually novel and we believe it to open new directions in SBST. This work is part of ongoing research in which the next steps include evaluating the application of these landscapes for discovery of properties of interest.

References

1. Aleti, A., Moser, I., Grunske, L.: Analysing the fitness landscape of search-based software testing problems. Autom. Softw. Eng. **24**(3), 603–621 (2017)
2. Alshahwan, N., Harman, M.: Coverage and fault detection of the output-uniqueness test selection criteria. In: Proceedings of the 2014 International Symposium on Software Testing and Analysis, pp. 181–192. ACM (2014)
3. Böhme, M., Pham, V.T., Roychoudhury, A.: Coverage-based greybox fuzzing as Markov chain. IEEE Trans. Softw. Eng. **45**, 489–506 (2017)
4. Bonzini, P.: sed(1) - Linux man page (2019). https://linux.die.net/man/1/sed
5. Chen, T.Y., Leung, H., Mak, I.K.: Adaptive random testing. In: Maher, M.J. (ed.) ASIAN 2004. LNCS, vol. 3321, pp. 320–329. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-30502-6_23
6. Ciupa, I., Leitner, A., Oriol, M., Meyer, B.: ARTOO: adaptive random testing for object-oriented software. In: Proceedings of the 30th International Conference on Software Engineering, pp. 71–80. ACM (2008)
7. Cortes, C., Mohri, M.: AUC optimization vs. error rate minimization. In: Advances in Neural Information Processing Systems, pp. 313–320 (2004)
8. Doersch, C.: Tutorial on variational autoencoders. arXiv preprint arXiv:1606.05908 (2016)
9. Dumoulin, V., Visin, F.: A guide to convolution arithmetic for deep learning. arXiv preprint arXiv:1603.07285 (2016)
10. Feldt, R., Poulding, S., Clark, D., Yoo, S.: Test set diameter: quantifying the diversity of sets of test cases. In: 2016 IEEE International Conference on Software Testing, Verification and Validation (ICST), pp. 223–233. IEEE (2016)

11. Gay, G., Staats, M., Whalen, M., Heimdahl, M.P.: The risks of coverage-directed test case generation. IEEE Trans. Softw. Eng. **41**(8), 803–819 (2015)
12. Glasmachers, T.: Limits of end-to-end learning. arXiv preprint arXiv:1704.08305 (2017)
13. Gordon, A.D., Melham, T.: Five axioms of alpha-conversion. In: Goos, G., Hartmanis, J., van Leeuwen, J., von Wright, J., Grundy, J., Harrison, J. (eds.) TPHOLs 1996. LNCS, vol. 1125, pp. 173–190. Springer, Heidelberg (1996). https://doi.org/10.1007/BFb0105404
14. Harman, M., Clark, J.: Metrics are fitness functions too. In: Proceedings of 10th International Symposium on Software Metrics, pp. 58–69. IEEE (2004)
15. Harman, M., Jones, B.F.: Search-based software engineering. Inf. Softw. Technol. **43**(14), 833–839 (2001)
16. Harman, M., Mansouri, S.A., Zhang, Y.: Search-based software engineering: trends, techniques and applications. ACM Comput. Surv. (CSUR) **45**(1), 11 (2012)
17. Heimdahl, M.P.E., George, D., Weber, R.: Specification test coverage adequacy criteria = specification test generation inadequacy criteria. In: Proceedings of Eighth IEEE International Symposium on High Assurance Systems Engineering, pp. 178–186. IEEE (2004)
18. Hochreiter, S., Schmidhuber, J.: Long short-term memory. Neural Comput. **9**(8), 1735–1780 (1997)
19. Huang, J., Ling, C.X.: Using auc and accuracy in evaluating learning algorithms. IEEE Trans. Knowl. Data Eng. **17**(3), 299–310 (2005)
20. Inozemtseva, L., Holmes, R.: Coverage is not strongly correlated with test suite effectiveness. In: Proceedings of the 36th International Conference on Software Engineering, pp. 435–445. ACM (2014)
21. Jones, D.: Sparse - a semantic parser for C (2019). https://sparse.wiki.kernel.org/index.php/Main_Page
22. Kalchbrenner, N., Grefenstette, E., Blunsom, P.: A convolutional neural network for modelling sentences. arXiv preprint arXiv:1404.2188 (2014)
23. Karpathy, A.: Cs231n convolutional neural networks for visual recognition (2016). http://cs231n.github.io/neural-networks-1/
24. Kawaguchi, K.: Deep learning without poor local minima. In: Advances in Neural Information Processing Systems, pp. 586–594 (2016)
25. Kingma, D.P., Ba, J.: Adam: a method for stochastic optimization. arXiv preprint arXiv:1412.6980 (2014)
26. Kingma, D.P., Welling, M.: Auto-encoding variational bayes. arXiv preprint arXiv:1312.6114 (2013)
27. Kochhar, P.S., Thung, F., Lo, D.: Code coverage and test suite effectiveness: empirical study with real bugs in large systems. In: 2015 IEEE 22nd International Conference on Software Analysis, Evolution and Reengineering (SANER), pp. 560–564. IEEE (2015)
28. Lane, T., et al.: libjpeg 6b (1998). http://libjpeg.sourceforge.net/
29. Luk, C.K., et al.: Pin: building customized program analysis tools with dynamic instrumentation. In: ACM SIGPLAN Notices, vol. 40, pp. 190–200. ACM (2005)
30. McMinn, P.: Search-based software test data generation: a survey. Softw. Test. Verif. Reliab. **14**(2), 105–156 (2004)
31. Meyering, J., Gordon, A.: GNU sed (2019). https://www.gnu.org/software/sed/
32. Mikolov, T., Chen, K., Corrado, G., Dean, J.: Efficient estimation of word representations in vector space. arXiv preprint arXiv:1301.3781 (2013)
33. Miller, B.P., Fredriksen, L., So, B.: An empirical study of the reliability of unix utilities. Commun. ACM **33**(12), 32–44 (1990)

34. Nethercote, N., Seward, J.: Valgrind: a framework for heavyweight dynamic binary instrumentation. In: ACM SIGPLAN Notices, vol. 42, pp. 89–100. ACM (2007)
35. Nguyen, Q., Hein, M.: The loss surface of deep and wide neural networks. arXiv preprint arXiv:1704.08045 (2017)
36. Nguyen, Q., Hein, M.: Optimization landscape and expressivity of deep CNNs. In: International Conference on Machine Learning, pp. 3727–3736 (2018)
37. Salimans, T., Kingma, D.P.: Weight normalization: a simple reparameterization to accelerate training of deep neural networks. In: Advances in Neural Information Processing Systems, pp. 901–909 (2016)
38. Shepperd, M.: Fundamentals of Software Measurement. Prentice-Hall, Upper Saddle River (1995)
39. Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., Salakhutdinov, R.: Dropout: a simple way to prevent neural networks from overfitting. J. Mach. Learn. Res. **15**(1), 1929–1958 (2014)
40. Staats, M., Gay, G., Whalen, M., Heimdahl, M.: On the danger of coverage directed test case generation. In: de Lara, J., Zisman, A. (eds.) FASE 2012. LNCS, vol. 7212, pp. 409–424. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-28872-2_28
41. Swirszcz, G., Czarnecki, W.M., Pascanu, R.: Local minima in training of neural networks. arXiv preprint arXiv:1611.06310 (2016)
42. Tan, S.H., Yi, J., Mechtaev, S., Roychoudhury, A., et al.: Codeflaws: a programming competition benchmark for evaluating automated program repair tools. In: Proceedings of the 39th International Conference on Software Engineering Companion, pp. 180–182. IEEE Press (2017)
43. VP Users (2017). http://valgrind.org/gallery/users.html
44. Veillard, D.: The XML C parser and toolkit of Gnome (2019). http://xmlsoft.org/
45. Wegener, J., Baresel, A., Sthamer, H.: Evolutionary test environment for automatic structural testing. Inf. Softw. Technol. **43**(14), 841–854 (2001)
46. Zalewski, M.: American fuzzy lop (2007). http://lcamtuf.coredump.cx/afl/