The Utility of Neural Network Test Coverage Measures

Rob Ashmore, Alec Banks
Defence Science and Technology Laboratory
{rdashmore,abanks}@dstl.gov.uk

Abstract

In this position paper, we are interested in what test coverage measures can, and cannot, tell us about neural networks. We begin with a review of the role of test coverage measures in traditional development approaches for safety-related software. We show how those coverage measures, in the neural network sense, cannot achieve the same aims as their equivalents in the traditional sense. We provide indications of approaches that can partially meet those aims. We also indicate the utility of current neural network coverage measures.

1 Introduction

Neural Networks (NNs) have demonstrated significant utility in a range of safety and security related applications (e.g. autonomous cars (Tian et al. 2018)). In such cases there is a need to provide a commensurate level of assurance that a particular NN implementation is correct. Approaches to NN assurance have drawn inspiration from those used for traditional safety-related software. The concept of test coverage is one such example.

NNs are different to traditional software. We contend that: these differences fundamentally change the meaning of several types of test coverage (e.g. requirements and structural measures); there are approaches that can partially achieve the intent of these traditional test coverage measures in an NN context; and currently-proposed measures of NN test coverage have utility in different ways.

The remainder of this paper is structured as follows: Section 2 provides an overview of the role of test coverage measures in the development of traditional safety-related software; Section 3 summarises a selection of NN test coverage measures proposed in the literature; Section 4 outlines NN-based approaches that can achieve some of the aims of traditional test coverage; Section 5 summarises the value that can be gained from current NN coverage measures; Section 6 concludes the paper.

2 Traditional Safety-Related Software Development

Our discussion of safety-related software development is based on DO-178C (RTCA 2011), which is commonly used in the aerospace domain. This approach is generally applicable, since the principles adopted in DO-178C are mirrored elsewhere (e.g. (Hawkins, Habli, and Kelly 2013)).

Before the discussion, we note that although a significant amount of practical experience provides confidence in DO-178C, and its predecessor, DO-178B, there is little, if any, explicit evidence to support the specific coverage criteria that are used (Holloway 2013). This limitation is not restricted to DO-178C: it applies across many safety-related software standards (Graydon and Holloway 2015). Nevertheless, we believe a traditional safety-related software development standard is an appropriate basis for this work. This is partly because of the excellent safety record of industries that use such standards and partly because many NN test coverage measures are (implicitly or explicitly) based on measures used in traditional software engineering.

The development of safety-related software begins with system-level requirements that have been allocated to software. These high-level requirements are hierarchically decomposed, in a traceable manner, from requirements that detail 'what' behaviour is needed, down to requirements that detail 'how' to achieve this. The software requirements are also independently used to produce test cases in a Requirements-Based Testing (RBT) approach. The tests are run and the coverage of the code structure is measured dynamically. RBT is an attractive approach to test design since it acknowledges that exhaustive testing is (generally speaking) infeasible and it focusses on the demonstration of the intended behaviour.

Assuming all tests pass, if structural coverage is incomplete then one of the following three conditions holds, where the notion of correctness directly relates to the intent of the high-level requirements:

- The software's behaviour is correct, but the *software-level requirements are incomplete*;
- The software-level requirements are correct, but the *software includes additional, unnecessary behaviour* (we include unreachable as well as redundant code here);
- The software behaviour and software-level requirements are correct, but the *test set is incomplete*. This includes, for example, cases where development tools introduce code for runtime efficiency.

Considering all three of these conditions provides con-

fidence that (once a suitable level of coverage has been achieved) the code sufficiently implements the requirements and, furthermore, it contains no undesirable additional behaviour. The link between requirements and behaviour is important, because requirements are the key mechanism for communicating expectations about software behavoiur with actors in the development, test and integration of the software.

Structural coverage is typically measured at the software unit level. Equivalently, at this level behaviour can be defined, and verified. Hierarchical decomposition of requirements, supported by architectural design, provides confidence that unit-level behaviour can be aggregated to provide the required system-level behaviour. Not all behaviour can be tested at this level, thus testing continues through the integration processes.

The type of structural coverage that is required depends on the criticality of the software. In less-critical cases, statement coverage suffices; more demanding cases require branch coverage; and the most critical cases require Modified Condition / Decision Coverage (MCDC) (RTCA 2011). A useful tutorial on these types of structural coverage can be found at (Hayhurts et al. 2001). It is apparent that all of these coverage levels are based on code structure. This reflects an implicit assumption that code structure, rather than data, is the main influence on software behaviour. Cases where data significantly affects behaviour would be expected to be defined in a requirement and addressed by a test case.

The combination of RBT and coverage measurement is important. Even though behaviour is, typically, not strongly data-dependent, there is still value in using suitably realistic test values. Simply optimising a test set to achieve a given level of coverage, with little or no consideration of the requirements, is less effective than RBT (Gay et al. 2015). The reverse approach, of testing against requirements without measuring coverage, cannot provide confidence that the requirements suitably encapsulate the software's behaviour. It can say that the software satisfies the requirements, but provides no information on what else the software may do. This information is an important part of assuring safety-related software.

In summary, for traditional software: hierarchical decomposition of requirements means that software behaviour can be understood at the unit level; independent interpretation of requirements provides confidence that requirements have been implemented correctly; and the combination of RBT and structural test coverage provide confidence that the requirements suitably describe the software's behaviour, both what it does and what it does not do.

3 Neural Network Coverage Measures

Our main interest is in Artificial Intelligence (AI), particularly AI implemented using Machine Learning (ML) techniques. Due to their prevalence, we focus on NNs, but much of what follows is applicable to other forms of ML.

Historically, NN testing was based on measures like precision and recall, calculated using a set of data that was held back from training. The inadequacy of these measures is demonstrated by adversarial inputs (Szegedy et al. 2014). Networks that, based on these measures, perform extremely well against verification data may display significant undesirable behaviour when exposed to previously unseen, but valid, inputs. This does not necessarily mean concepts like precision and recall should be abandoned: it means that, in isolation, they do not provide sufficient confidence for the use of NNs in safety-related applications.

More recently, a variety of coverage measures, based on internal properties of an NN, have been proposed. It should be noted that, to the best of our knowledge, use of these measures has thus far been demonstrated using von Neumann architecture hardware. For some, different, architectures it may not be efficient or even possible to collect the information on which these measures are based.

Some notable example coverage measures are summarised below.

DeepXplore (Pei et al. 2017) aims to systematically test a collection of similar NNs for erroneous corner case behaviours. It uses a gradient-guided local search that starts from a seed input and solves a joint optimisation problem, seeking to find new inputs that cause different neuron activation patterns and lead the NNs to behave differently on the same task. In the same paper, the notion of neuron coverage is developed based on the fraction of neurons that are activated for at least one test input, with a neuron considered active if its output is above a threshold value (e.g. 0). Equivalently, 100% coverage is achieved if each neuron is activated at least once by the test set.

DeepGauge (Ma et al. 2018) considers two levels of coverage: neuron-level and layer-level.

- Neuron-level coverage splits the output range of each neuron (established during the training phase) into k equal sections. The k-multisection coverage of that neuron is the fraction of sections that the neuron's output falls into across all inputs in the test set. The k-multisection coverage of the NN is the average of these neuron coverages.
- Layer-level coverage is based on the l neurons, and combinations thereof, in each layer that have the greatest output value. Top-l neuron coverage is the fraction of neurons that are in the top l neurons in their layer for at least one of the test inputs.

DeepCT (Ma et al. 2019) is inspired by combinatorial testing (see (NIST 2010)). Two, layer-level coverage measures are defined:

- Sparse coverage considers the fraction of m-way subsets in which all neurons are activated for at least one test input. For example, a layer of four neurons will have six 2-way subsets.
- Dense coverage considers the fraction of m-way activation patterns that are activated for at least one test input. For example, a 2-way subset has four activation patterns.

DeepConcolic (Sun et al. 2019) is based on a variety of measures, loosely inspired by MCDC:

- Sign-sign coverage measures whether the change in output sign (i.e. moving from zero to non-zero) of a neuron in layer n independently affects the output sign of a specific neuron in layer $n + 1$ (i.e. the subsequent layer).

- Value-value coverage measures whether a change in output value (e.g. similar to the k-sections used by DeepGauge) of a neuron in layer n independently affects the output value of a specific neuron in layer $n + 1$.
- The notions of sign-value and value-sign coverage naturally follow.

These measures show a progression, from simple aspects of individual neuron behaviour (e.g. activation or non-activation), through more complex aspects of individual neuron behaviour, to the joint behaviour of combinations of neurons, either in the same layer or in neighbouring layers.

4 Towards Traditional Coverage for Neural Networks

Constraints

For the purposes of this paper, we are primarily concerned with structural coverage (although we remain interested in requirements throughout). For completeness, we note that other attributes will be important in an overall assurance argument supporting the use of an NN. A key area is the NN development process. This is covered in detail in (Ashmore, Calinescu, and Paterson 2019) and SCSC-153A (SASWG 2020).

Recall, traditional software test structural coverage measures, when used in conjunction with RBT, provide a level of confidence that software satisfies requirements and requirements cover software behaviour. When viewed in that light it is apparent that the types of NN coverage measures discussed above cannot achieve the same aims as traditional software coverage measures. As illustrated in the following paragraphs, none of the requisite building blocks, which allow traditional coverage to work in this way, are present in the NN context.

We do not have a complete set of requirements. There is often a good understanding of the purpose of an NN, for example: 'recognise hand-written digits' or 'determine sentiment from social media messages'. However, in our experience, there is rarely, if ever, a set of requirements that is accurate, complete, unambiguous and verifiable.

Not all NN requirements can be hierarchically decomposed. Some level of methodical decomposition is often possible; for example, 'recognise hand-written digits' could be decomposed to explicitly cater for 'triangular 4s', 'open 4s' and 'crossed 7s'. But, requirements like these cannot be decomposed to a level that can be directly coded against. In particular, in a safety-related environment, if we could decompose to a directly-codeable level then there would be no need to use an NN and traditional software would be the preferred approach (Salay and Czarnecki 2018).

We do not have a meaningful software unit level at which software behaviour can be described. As indicated previously, current NN coverage measures focus on neuron behaviour, either individually or in patterns. Although this behaviour controls the network's output, it does not (and cannot) describe software behaviour in a way that is meaningful to a user. Some approaches to explainability may help, for example: identifying pixels that positively weight towards a particular class (Ribeiro, Singh, and Guestrin 2016) or visualising feature maps from a final convolutional layer (Chattopadhay et al. 2018). Whilst useful, these do not provide the requisite understanding. Furthermore, they do not generalise across all types of NNs or all NN applications.

Software Satisfies Requirements

One of the strengths of NNs is their ability to generalise from incomplete specifications. It may seem that asking for a demonstration that an NN satisfies a set of requirements negates this strength.

From our perspective, the distinction is in the level at which requirements are expressed. Placing, and verifying, requirements on the network's internals does not provide traceability between requirements and behaviour. That traceability might be achieved by placing (an appropriate set of) requirements on the NN's $Input \rightarrow Output$ behaviour.

Adversarial examples (Szegedy et al. 2014) are one aspect of this behaviour. 'Robustness' to these examples has often been suggested, sometimes implicitly, as a requirement that an NN should meet. For feed forward networks, this can be demonstrated, or a counter-example found, using techniques based on Satisfiability Modulo Theories (SMT) (Huang et al. 2017).

DeepPoly (Singh et al. 2019) couples an abstract domain (specifically, a combination of floating-point polyhedra with intervals) with abstract transformers for common neural network functions. This allows guarantees to be made, for example, that all samples within an L_∞ ball will be classified correctly, or that all image rotations (up to a given angle) will be classified correctly. Note that, in these cases, 'classified correctly' means all samples in the region will be given the same class. It is *assumed* that this behaviour is correct.

Marabou (Katz et al. 2019) is an SMT-based tool that can be used to check whether a particular NN satisfies a specific property. If the property is not satisfied then a concrete input for which the property is violated (i.e. a counter-example) is provided. Amongst other things, the tool has been used to prove properties about an Airborne Collision Avoidance System (ACAS) for unmanned aircraft (Julian et al. 2016).

Deep Learning with Differentiable Logic (DL2) (Fischer et al. 2018) can support training and querying of NNs. In the training application, logical constraints are translated into non-negative loss functions, which are incorporated into the overall optimisation the training is attempting to complete. The querying application allows constraints over properties not directly computed by the network (e.g. constraints can consider the likelihood of an input being in one of a set of classes). These types of constraint can readily express system-level requirements.

Even if the (suitably-measured) performance of an NN is adequate, this does not necessarily mean that an NN is making decisions in the same way as a human would. There is evidence that NNs use highly-predictive, but non-intuitive (to a human) features to support classification (Ilyas et al. 2019). If the distinction between 'good performance' and 'good performance, deciding like a human' is important then it should be captured as a requirement and explicitly tested.

Requirements Cover Software Behaviour

As noted earlier, a key question is whether there are aspects of software behaviour that are not captured by the requirements. That is, if a user fully understands the requirements then will they ever be surprised by the software's behaviour? This latter formulation is helpful as it clarifies what we mean by 'behaviour'. In particular, we are interested in behaviour that is externally observable. From the perspective of an NN, we are primarily interested in behaviour in the sense of $Input \rightarrow Output$ mappings.

In the NN context, one way of categorising different contributors to behaviour is by considering different types of input. Four related spaces are defined in (Ashmore, Calinescu, and Paterson 2019): the *input domain* space, which are inputs that the NN can accept; the *operational domain* space, which are inputs that the NN may receive when used operationally, the *failure domain* space, which are inputs the NN may receive if there are failures elsewhere in the system; and the *adversarial domain* space, which are inputs the NN may receive if it is being attacked by an adversary.

Using this structure there are a number of techniques that can help increase coverage of potential NN behaviour: space-filling designs for computer experiments (Santner, Williams, and Notz 2018) are potentially relevant for the input domain; the notion of situation coverage (Alexander, Hawkins, and Rae 2015) is potentially relevant for the operational domain; Failure Modes and Effects Analysis (FMEA) should be useful for the failure domain; and 'red teaming' (Kardos and Dexter 2017) should inform the adversarial domain. Whilst they are useful, none of these techniques define a precise boundary of testing sufficiency. Greater experience of the practical use of NNs is likely to be required before such boundaries can be set.

These approaches provide a 'forward-looking' way of understanding behaviour; they rely on choosing inputs to invoke behaviour. This differs from the approach to traditional software requirements and structural coverage, where behaviour is invoked, in a sense, from within the behavioural regimes defined in the low-level requirements structure.

Generative Adversarial Networks (GANs) could help fill this gap. For example, they could be used to find plausible operational domain inputs that exhibit different behaviours to those observed in the training data. These could be inputs that are similar to training samples but result in different outputs; for classification networks, this is the same as finding adversarial inputs. Alternatively, they could be used to find inputs that are sufficiently different from any sample in the training data.

Another approach involves looking for specific undesirable behaviours in an NN. Detection and mitigation of backdoor attacks (Wang et al. 2019) is one example.

In some cases, specifically for feed-forward networks, it may be possible to automatically infer formal properties (Gopinath et al. 2019). This is a helpful way of understanding aspects of the NN's behaviour. However, there is no guarantee that the inferred properties will be meaningful, in the sense of system-level requirements.

Summary

The way that NN are constructed means test coverage measures cannot perform the same function as they do for traditional software. There are a number of approaches that can be used to provide some confidence that the NN satisfies requirements. There are also approaches that can provide some confidence that the NN's behaviour is understood.

5 The Utility of NN Coverage Measures

The previous discussion highlights a distinction between the implicit motivation for NN coverage measures and their utility. We propose four ways in which NN coverage measures, like those discussed in Section 3, could have utility.

Firstly, following the analogy that training data represent the low-level behavioural requirements, the measures could be used *to optimise training data*, for example, by identifying whether a larger, or more diverse, training set was exercising a larger portion of internal network behaviour. If it is then this is an argument for using an alternative, possibly larger, data set, despite the additional overheads and increased risk of over fitting.

Secondly, the measures could be used *to compare training data, used during development, with verification data, used by an independent team*. Suppose, for example, that the full set of training and verification data achieves greater coverage than just the training data. This outcome would demonstrate the independent verification activity is exercising additional types of internal NN behaviour to those observed during the development phase. This could be evidence that a suitably independent verification activity has been conducted.

Thirdly, many of the approaches used to measure NN coverage can also be used *to generate additional inputs that would extend coverage*. As such, they provide an indication of ways in which training data could be meaningfully extended. Obviously, for situations where the NN is being developed using supervised learning, the appropriate output needs to be produced for each of these new inputs.

Fourthly, the measures can be used *to choose between two different NNs* that otherwise offer similar levels of performance. In such situations, the NN for which a fixed set of training data achieves greater coverage might be preferred. In general, this would be expected to be the NN with the simpler structure.

6 Conclusions

NNs have demonstrated significant utility. Their use in safety-related systems is predicated on confidence in their behaviour, both what they do and what they do not do. For traditional safety-related software much of this confidence comes from test coverage measures in an RBT context. The approaches used to measure test coverage for NNs cannot provide an equivalent confidence.

There are approaches that can provide some aspects of this confidence. Appropriate consideration of different input spaces can help, as can GAN-based methods for finding 'new' inputs. NN test coverage measures can provide value in other ways. They can, for example, provide a principled,

structured way of choosing between different training data sets, or between different trained models.

In conclusion, NN test coverage measures can have significant utility. They represent different types of confidence than is found in their traditional software testing forebears. However, more work is required before a holistic and complete understanding is achieved in the relationship between the coverage measures and confidence in NN behaviour.

References

Alexander, R.; Hawkins, H. R.; and Rae, A. J. 2015. Situation coverage–a coverage criterion for testing autonomous robots. *University of York* .

Ashmore, R.; Calinescu, R.; and Paterson, C. 2019. Assuring the machine learning lifecycle: Desiderata, methods, and challenges. *arXiv preprint arXiv:1905.04223* .

Chattopadhay, A.; Sarkar, A.; Howlader, P.; and Balasubramanian, V. N. 2018. Grad-cam++: Generalized gradient-based visual explanations for deep convolutional networks. In *2018 IEEE Winter Conference on Applications of Computer Vision (WACV)*, 839–847. IEEE.

Fischer, M.; Balunovic, M.; Drachsler-Cohen, D.; Gehr, T.; Zhang, C.; and Vechev, M. 2018. DL2: Training and querying neural networks with logic. *Internet* .

Gay, G.; Staats, M.; Whalen, M.; and Heimdahl, M. 2015. The Risks of Coverage-Directed Test Case Generation. *Software Engineering, IEEE Transactions on* 41(8): 803–819. ISSN 0098-5589.

Gopinath, D.; Converse, H.; Pasareanu, C. S.; and Taly, A. 2019. Property Inference for Deep Neural Networks. *arXiv* 1904.13215v2.

Graydon, P. J.; and Holloway, C. M. 2015. Planning the Unplanned Experiment: Assessing the Efficacy of Standards for Safety Critical Software. Technical Report NASA/TM-2015-218804, NASA.

Hawkins, R.; Habli, I.; and Kelly, T. 2013. The principles of software safety assurance. *31st International System Safety Conference, Boston, Massachusetts USA* .

Hayhurts, K.; Veerhusen, D.; Chilenski, J.; and Rierson, L. 2001. A Practical Tutorial on Modified Condition/Decision Coverage. Technical report, NASA.

Holloway, C. M. 2013. Making the Implicit Explicit: Towards An Assurance Case for DO-178C. *NASA Langley Research Center* .

Huang, X.; Kwiatkowska, M.; Wang, S.; and Wu, M. 2017. Safety Verification of Deep Neural Networks. In *Proceedings of the 29th International Conference on Computer Aided Verification*, 3–29.

Ilyas, A.; Santurkar, S.; Tsipras, D.; Engstrom, L.; Tran, B.; and Madry, A. 2019. Adversarial examples are not bugs, they are features. In *Advances in Neural Information Processing Systems*, 125–136.

Julian, K. D.; Lopez, J.; Brush, J. S.; Owen, M. P.; and Kochenderfer, M. J. 2016. Policy compression for aircraft collision avoidance systems. In *2016 IEEE/AIAA 35th Digital Avionics Systems Conference (DASC)*, 1–10. IEEE.

Kardos, M.; and Dexter, P. 2017. A Simple Handbook for Non-Traditional Red Teaming. Technical report, Defence Science and Technology Group Edinburgh, SA.

Katz, G.; Huang, D. A.; Ibeling, D.; Julian, K.; Lazarus, C.; Lim, R.; Shah, P.; Thakoor, S.; Wu, H.; Zeljić, A.; et al. 2019. The marabou framework for verification and analysis of deep neural networks. In *International Conference on Computer Aided Verification*, 443–452. Springer.

Ma, L.; Juefei-Xu, F.; Sun, J.; Chen, C.; Su, T.; Zhang, F.; Xue, M.; Li, B.; Li, L.; Liu, Y.; Zhao, J.; and Wang, Y. 2018. DeepGauge: Comprehensive and Multi-Granularity Testing Criteria for Gauging the Robustness of Deep Learning Systems. *arXiv* 1803.07519. O.

Ma, L.; Juefei-Xu, F.; Xue, M.; Li, B.; Li, L.; Liu, Y.; and Zhao, J. 2019. DeepCT: Tomographic Combinatorial Testing for Deep Learning Systems. In *Proceedings of the 26th IEEE International Conference on Software Analysis, Evolution and Reengineering*, 614–618. IEEE.

NIST. 2010. Practical Combinatorial Testing.

Pei, K.; Cao, Y.; Yang, J.; and Jana, S. 2017. DeepXplore: Automated Whitebox Testing of Deep Learning Systems. *arXiv* .

Ribeiro, M. T.; Singh, S.; and Guestrin, C. 2016. Why Should I Trust You?: Explaining the Predictions of Any Classifier. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 1135–1144.

RTCA. 2011. DO-178C: Software Considerations in Airborne Systems and Equipment Certification.

Salay, R.; and Czarnecki, K. 2018. Using Machine Learning Safely in Automotive Software: An Assessment and Adaption of Software Process Requirements in ISO 26262. *arXiv* 1808.01614.

Santner, T. J.; Williams, B. J.; and Notz, W. I. 2018. Space-filling designs for computer experiments. In *The Design and Analysis of Computer Experiments*, 145–200. Springer.

SASWG. 2020. SCSC-153A: Safety Assurance Objectives for Autonomous Systems.

Singh, G.; Gehr, T.; Püschel, M.; and Vechev, M. 2019. An abstract domain for certifying neural networks. *Proceedings of the ACM on Programming Languages* 3(POPL): 1–30.

Sun, Y.; Huang, X.; Kroening, D.; Sharp, J.; Hill, M.; and Ashmore, R. 2019. Structural Test Coverage Criteria for Deep Neural Networks. *ACM Transactions on Embedded Computing Systems (TECS)* 18(5s): 94.

Szegedy, C.; Zaremba, W.; Sutskever, I.; Bruna, J.; Erhan, D.; Goodfellow, I.; and Fergus, R. 2014. Intriguing Properties of Neural Networks. In *Proceedings of the 2nd International Conference on Learning Representations*, 1–10.

Tian, Y.; Pei, K.; Jana, S.; and Ray, B. 2018. DeepTest: Automated Testing of Deep-Neural-Network-driven Autonomous Cars. In *ICSE '18: 40th International*

Conference on Software Engineering, May 27-June 3, 2018, Gothenburg, Sweden. ACM, New York, NY, USA. arXiv:1708.08559v2.

Wang, B.; Yao, Y.; Shan, S.; Li, H.; Viswanath, B.; Zheng, H.; and Zhao, B. Y. 2019. Neural cleanse: Identifying and mitigating backdoor attacks in neural networks. In *2019 IEEE Symposium on Security and Privacy (SP)*, 707–723. IEEE.

Disclaimer