2.3.3 – Verification of the learned model

Practical guidance – cross-domain and automotive

Authors: Dr Heidy Khlaaf and Dr Philippa Ryan (Adelard LLP)

This guidance considers the use of static analysis and formal verification techniques for Machine Learning (ML) models that may be deployed in autonomous vehicles.

Formal verification of ML models

Formal verification is the process of establishing whether a system satisfies some requirements (properties), using formal methods of mathematics. Traditionally, formal methods are based on the premise that we can determine the functional properties of a system by the way we design it and implement it. While this holds for traditional systems, it does not hold for ML systems as a whole, as their design determines how they learn, but not what they will learn [1]. Furthermore, formal verification methods are based on the premise that we can infer functional properties of a software product from an analysis of its source code. Again, while this holds for traditional systems, it does not hold for ML systems, whose behaviour is also determined by their learning theory. Indeed, it is the case that traditional formal methods techniques cannot be applied as they are. Novel verification techniques and specifications thus must be devised to address ML algorithms, specifically, for Neural Networks (NNs).

Research to create formal methods that can verify ML models is in its infancy. The difficulty arises given that the output behaviour of an ML algorithm may not always be clear or expected relative to the inputs. To consider the novel complexities of ML systems, traditional formal properties must be reconceived and redeveloped for ML models.

Formally verifying pointwise robustness

The majority of existing research has attempted to find ways of specifying types of ML robustness (i.e. pointwise robustness) that would be amenable to formal verification of an ML model's robustness. Pointwise robustness aims to identify how sensitive a classifier function is against small perturbations in the input distribution. A number of techniques have been developed, including Pulina et al. [2], Huang et al. [3], and most notably, the tool Reluplex [4]. No matter the technique, these verification methods suffer from the same set of limitations:

- It is difficult to define meaningful regions and manipulations
- The neighbourhoods surrounding a point that are used currently are arbitrary and conservative
- We cannot enumerate all points near which the classifier should be approximately constant (i.e. we cannot predict all future inputs)

Some preliminary work extending [3] has been introduced, proposing that instead of relying on an exhaustive search of a discretised region, one can compute the upper and lower bound case confidence values of a point [5], which may alleviate some of these limitations.

Generally speaking however, pointwise robustness, although an interesting property, is not expressive enough nor conducive to producing confidence for assuring an ML model, given that one cannot predict all future inputs. It is thus unclear what assurances, if any, a pointwise robustness analysis would provide, or to see the claim for which pointwise robustness can provide support or evidence.

Formally verifying well-specified systems

Reluplex [4] can also be used to verify more general behaviours regarding ML algorithms. The Reluplex technique requires functional specifications, written as constraints, to be fed into a specialised linear programming solver to be verified against a piecewise linear constraint model of the ML algorithm. The Reluplex technique provides a promising solution for well-specified traditional systems requiring new ML implementations.

However, like similar techniques the generalisation of this to deep learning is challenging as it requires well-defined, mathematically specifiable system specifications as input. These techniques are thus only applicable to well-specified deterministic or tractable systems that can be implemented using traditional methods (e.g. programming in C) or via ML models. As a consequence, these techniques cannot be straightforwardly applied to arbitrary contexts, and domain-specific effort is currently required even to specify properties of interest, let alone verify them. Consider for example perception systems (e.g. camera, LIDARs, sensor fusion) crucial to the operation of autonomous vehicles: their system specifications are not bounded, and are not amenable to formalisation of constraints that can be proved.

Static analysis of supporting systems

Static analysis is the term used for source and object code analysis that examines the code without execution, and may encompass some formal verification techniques. These analyses do semantic and syntactic checks on the source and object code. Static analysis can be performed on software that supports ML functionality to look for potential issues and vulnerabilities that could impact the performance or functionality of the code. ML software has been shown to be vulnerable to traditional threats. Overflow errors within supporting software have been seen to propagate and affect the functionality of an ML model (such as where a Not a Number (NaN) code error caused uncontrolled acceleration) [6].

Run-time issues such as overflow/underflow and access to data out of bounds can directly impact on the performance of an ML element such as a neural network, as it may perform a series of matrix multiplications on edges and nodes. That is, for ML it may be more difficult to identify which parts of the software are affected by the error, and hence what the impact might be. Consider a loop that can potentially access data outside an array. For a traditionally developed system, the final purpose of that array and the loop are likely to be relatively transparent (even when found in a generic library function). However, for ML, the use of undefined data in one array manipulation may or may not have an impact, depending on the sensitivity to that data point. Additionally, overflow/underflow in a neural network could lead to edges or nodes having a multiplication factor of zero for some operations. Again, it is difficult to predict the impact this might have on a task such as classification except in a very broad sense.

The aforementioned errors are only applicable to statically-typed languages such as C and C++, not to a dynamically typed language such as Python (a popular language utilised in the

implementation of numerous ML libraries and frameworks) since the semantics and implementation of the language itself prevent overflow/underflow errors.

Python has an increasing presence in the safety-critical due to its use within popular ML frameworks (e.g. TensorFlow, PyTorch, etc). These frameworks can be deployed in safety-critical contexts, such as autonomous vehicles, yet static analysis and formal verification techniques for the Python language are almost non-existent. The lack of existing techniques is due to a few factors:

- Dynamically and weakly typed languages are often discouraged from use in safety-critical domains given the common type faults they may produce
- Python's lack of use in critical domains has never incentivised the creation of novel formal analysis techniques addressing dynamically typed languages
- Dynamically typed languages are significantly harder to statically analyse, or verify, given the difficulty of constructing control or data flow graphs, which are easier to derive from languages such as C/C++

In a dynamically typed language, every variable name is bound only to an object. Names are bound to objects at execution time by means of assignment statements, and it is possible to bind a name to objects of different types during the execution of the program. This behaviour is not easily captured, particularly in a program logic setting. A semantic analysis currently cannot alleviate this issue, as the semantics of Python are intricate and complex, and yet to be fully defined. No static analysis or formal verification methods exist to allow for the analysis of Python code, beyond simple linear analyses. Indeed, this is a significant gap within the formal methods field given the deployment of autonomous vehicles utilising Python.

Further details on this guidance can be found in [9].

Example of application of guidance

This section provides an example of preliminary static analysis experiments on the You Only Look Once (YOLO) [16] CNN source code using Polyspace Bug Finder [7]. YOLO provides object recognition support, as illustrated in Figure 1. Polyspace is an industry standard tool, used by many developers of high integrity software, to look for potential bugs and to assure compliance with the MISRA coding guidelines [8].

Figure 1: Example object recognition support by YOLO on public road

The analysis identified potential modifications that could be made to the off-the-shelf (OTS) software, improving robustness without impacting on the deployed functionality. This could support a case for high integrity use, along with additional evidence of the appropriateness of the training. Note that security concerns of using a public source image training set should also be considered, as this could provide an attack vector. However, our experiments were limited to examining the underlying source code only.

We analysed the core C and C++ software YOLO software library, including:

- The main classification application, which loads a weights and configuration file (in this case trained using COCO) and applies it to a specified image file
- YOLO libraries including training code (which would not typically be used during operation but would impact the robustness/reliability of the output), image manipulations and matrix calculations
- Third-party GPU source code from NVIDIA as the library supports multi-threaded training using a GPU
- Third-party code for accessing a webcam for “live streaming” of input/output

The latter two items were included so that the potential impact of concurrency problems could be considered.

Run time errors results

A number of different run-time errors were identified. Issues that could be of concern included:

- Known security vulnerabilities such as file I/O, which could be hijacked for system files, and buffer/data vulnerabilities allowing memory to be overwritten
- A number of memory leaks, such as files opened and not closed, and temporarily allocated data not freed, which could lead to unpredictable behaviour, crashes, and corrupted data

- A large number of calls to free() where the validity of its use is not checked, which could lead to incorrect (but potentially plausible) weights being loaded to the network
- Potential “divide by zeros” in the training code, including cost calculation, which could lead to crashes during online training, if the system were to be used in such a way

MISRA 2012 analysis

Compliance to the mandatory MISRA 2012 guidelines was assessed. Violation of these rules often means that bugs are present in the code. A summary is presented below.

- Rule 21.17 – violations that could lead to undefined behaviour if the input configuration files were corrupted or had invalid data. The impact of this might be undefined behaviour, either causing YOLO to crash or not be started.
- Rule 21.18 – violations that could lead to memory leaks. Again, this is a potential reliability issue.

Mitigations include having systems using YOLO detecting its liveness and having a safe state response if possible.

Concurrency analysis results

Polyspace identified a potentially conflicting access to a buffered image array during concurrent display/processing of a live webcam stream. The code is in a demo of functionality but for the purposes of exploration we have assumed it could be used, for example, to display to a car operator the output of the classifier. This could have two issues:

1. If the display is being used by an operator and the data appears in a corrupted form it could reduce confidence in the image classification and the operator would take control (if this option was available)
2. If the data being used during image classification was corrupted then items could be missed, mis-classified, or detected and classified when they did not exist. This is potentially more serious, but may depend on repeated identical failures input into a decision-making element. The decision-making element would need awareness of this type of failure mode and to have an appropriate mitigating action.

Running the YOLO source code through Polyspace indicated a number of potential issues that could impact the reliability and also the safety of the code if it were to be implemented as part of a safety critical system. The case study has demonstrated that traditional approaches to static analysis are still applicable to the supporting software, and could help make the code more trustworthy in operation and improve an assurance case for its use. However, further analysis is still needed, for example to consider the impact of mathematical bugs in training routines which might have impacted the CNN functionality derived.

References

- [1] Schumann, J., Gupta, P., Nelson, S. On verification & validation of neural network based controllers. NASA, 2003.

- [2] Pulina, L., & Tacchella, A. An abstraction-refinement approach to verification of artificial neural networks. In International Conference on Computer Aided Verification (pp. 243-257). Springer Berlin Heidelberg, July 2010.
- [3] Huang, X., Kwiatkowska, M., Wang, S., & Wu, M. Safety Verification of Deep Neural Networks. arXiv preprint arXiv:1610.06940, 2016.
- [4] Katz, G., Barrett, C., Dill, D., Julian, K., & Kochenderfer, M. Reluplex: An Efficient SMT Solver for Verifying Deep Neural Networks. arXiv preprint arXiv:1702.01135, 2017.
- [5] Ruan, W., Huang, X., & Kwiatkowska, M.Z. Reachability Analysis of Deep Neural Networks with Provable Guarantees. IJCAI, 2018.
- [6] Can robot navigation bugs be found in simulation? An exploratory study, T Sotiropouls et al. 2017 IEEE International Conference on Software Quality, Reliability and Security (QRS).
- [7] Polyspace Bug Finder, https://www.mathworks.com/products/polyspace-bug-finder.html last accessed March 2019.
- [8] Guidelines for the Use of the C Language in Critical Systems, ISBN 978-1-906400-10-1 (paperback), ISBN 978-1-906400-11-8 (PDF), March 2013. https://www.misra.org.uk/Publications/tabid/57/Default.aspx#label-comp
- [9] Bloomfield, R., Fletcher, G., Khlaaf, H., Ryan, P., Kinoshita, S., Kinoshit, Y., Takeyama, M., Matsubara, Y., Popov, P., Imai, K. and Tsutake, Y., 2020. Towards Identifying and closing Gaps in Assurance of autonomous Road vehicleS - A collection of Technical Notes Part 1. arXiv preprint arXiv:2003.00789.