Designing Safety Critical Software Systems to Manage Inherent Uncertainty

Alexandru Constantin Serban*†,
*Radboud University, Nijmegen, The Netherlands
†Software Improvement Group, Amsterdam, The Netherlands,
Email: *a.serban@cs.ru.nl

***Abstract*—Deploying machine learning algorithms in safety critical systems raises new challenges for system designers. The opaque nature of some algorithms together with the potentially large input space makes reasoning or formally proving safety difficult. In this paper, we argue that the inherent uncertainty that comes from using certain classes of machine learning algorithms can be mitigated through the development of software architecture design patterns. New or adapted patterns will allow faster roll out time for new technologies and decrease the negative impact machine learning components can have on safety critical systems. We outline the important safety challenges that machine learning algorithms raise and define three important directions for the development of new architectural patterns.**

***Keywords*-Software architecture, Software safety, Machine Learning**

I. Introduction

Recent developments in Machine Learning (ML) and, in particular, Deep Learning (DL) showed human level performance can be achieved (and even exceeded) in tasks where it is impossible to specify procedural rule sets. Some examples are object recognition, machine translation or speech recognition.

The increase in computational power and the rapid development of high level APIs have enabled DL algorithms to be explored in a variety of new scenarios and enabled thinking about new commercial applications - many of which are cyber-physical and safety-critical systems. For example, DL is explored in developing self driving cars [1], unmanned aerial vehicles (UAV) [2] and surgery robots [3].

However, the probabilistic nature of DL algorithms sometimes conflicts with the safety culture adopted when developing safety-critical systems.

Instead of guiding the algorithms to learn a task based on a selection of useful information (which can be carefully controlled by engineers), DL algorithms are usually trained by chaining non-linear and linear transformations on raw inputs. These transformations are tweaked by propagating the error backwards and adjusting their parameters. As long as the series of transformations is differentiable and large amounts of training data are available, DL algorithms are able to learn very complex tasks without any guidance.

This research was funded by NWO through the i-CAVE project.

This method of learning, called end-to-end learning (ETE), proves so efficient that it can completely replace large software stacks. For example, Bojarsky et al. [1] were able to train a DL algorithm able to perceive the environment and send commands straight to the actuators of a self-driving car in an ETE fashion. This means no other software was used except the actuator software interfaces and the DL algorithm.

Since no control is enforced over the features used in taking decisions, our understanding of the inner workings of DL algorithms or their explanatory capacity is limited. Therefore, from a system perspective, they are regarded as black box components - making it hard to reason about safety or give guarantees needed in order to certify a system to a safety standard (such as ISO26262 [4]).

Moreover, recent research showed that DL algorithms exhibit low robustness to input distribution shifts - when small perturbations are applied to inputs, the output and the confidence of a DL model drop considerably [5]. These properties of DL algorithms can severely impact the safety of systems using them [6, 7].

We call this phenomenon *inherent uncertainty* - the total uncertainty that comes from using machine learning components in a system and propose to address it from a system and software architecture point of view, rather than (only) at the level of DL components.

Moving the problem in the architectural space allows us to reason about it from a system-level perspective and in a trade-off analysis fashion, specific to software architecture [8]. Moreover, it includes more people in the analysis loop, since questions regarding ”how much uncertainty can a system support?” or ”how can uncertainty sources be contained in the system?” can be answered by a wider audience consisting of a variety of system engineers and not only ML experts.

We discuss the impact of deploying fully probabilistic and opaque components in safety critical systems and raise a number of challenges where software architecture design can play a mitigating role, allowing faster roll-out and integration of such technologies. Although some forms of inherent uncertainty have been investigated in safety critical systems, with a focus on autonomous driving [9, 10, 11, 12], our focus is on mitigating these issues through software architecture.

We propose the development of safety patterns for architecture design in order to restrict the negative impact DL algorithms have on systems safety, while allowing a system to harness their full power. We conclude laying down a roadmap for future research.

The rest of the paper is structured as follows. Section II briefly defines software safety and software architecture design for safety. Section III introduces the challenges raised by deploying DL algorithms in safety critical systems. Section IV proposes three directions for developing or adapting safety architectural patterns. Section V presents related work, followed by conclusions and future research in Section VI.

II. Background

Safety is the ability of a system to avoid unintended and harmful behavior during its operation. Whenever a system can lead to physical damages or loss of equipment, damage to the environment and even injury or death of human beings, safety is crucial.

At the system level, safety focuses on identifying and avoiding *hazardous situations*. Whenever software is used in safety critical systems, designers must ensure it does not cause or contribute to a system reaching a hazardous state.

The ability of a system to withstand hazards and maintain safe operation is called *fault tolerance*. Generally, fault tolerance consists of detecting and recovering from small defects, before they trigger larger failures. It boils down to maintaining the system in a safe state despite any faults.

In some cases, when a fault is detected the safe state can be immediately reached - e.g. by stopping parts of the system from operation. In others, the system should continue to operate safely despite any failures, or use some time for graceful degradation - until a potential safe state is achieved. For example, the braking system of a vehicle can not stop operation immediately. However, it can slowly decay letting the driver know it has to drive to a safe place in a designated time interval.

Software architecture is the starting point for developing a safety strategy [13]. In software architecture, recurrent design problems are abstracted and managed through architectural *patterns*. Safety is a non-functional property of a software system that can heavily influence its design. Designing for safety means designing for minimum risk [14]. In order to support safety design, a number of architectural patterns have been proposed [15]. For example, the triple modular redundancy pattern is used to enhance safety and reliability of a system when there is no safe state.

In some cases, design patterns have multiple versions or are very granular. This makes reasoning about the safety properties achieved when using them difficult [16]. Whenever this is the case, the process is supported by external mechanisms such as formal analysis.

Until recent years, only *deterministic* software components or *probabilistic* components *with limited input and output spaces* were explored and used in safety critical systems. However, the advent of fully probabilistic components with large input and output spaces poses challenges (1) for finding appropriate safety design patterns, (2) for reasoning about the safety properties achieved by using them and (3) for selecting the most appropriate patterns for the task at hand. In the next section, we explore some of these challenges.

III. Safety Challenges

Recent developments in ML and DL allowed probabilistic systems with large input and output spaces to be explored in safety critical systems - making it harder to reason about safety and to design safe software systems. We explore some of the properties of these algorithms which raise safety challenges - where software architecture can help:

- *Model complexity and opacity.* Most successful ML and DL constructs, such as neural networks (NN), are very complex and difficult to understand or explain. Trying to interpret complex or uninterpretable models post hoc does not ensure safety (since the interpretation is not equivalent to the decision rule used in making predictions) [7].
- *Fully probabilistic output.* The output of DL systems represents a probability distribution over a (large) set of possible outcomes. For example, in object recognition the set of possible outcomes is approximately the set of objects in the real world. This makes it impossible to perform any range checks on the output or design thorough safeguards.
- *Sensitivity to distribution shifts.* DL algorithms assume training and testing data are drawn from the same distribution, making algorithms sensitive even to small distribution drifts. In safety critical systems where the input distribution can not be controlled (e.g. autonomous vehicles) this assumption can be easily violated.
- *Limited testing scenarios.* Due to the large input space, it is often impossible to test or approximate all possible inputs. Moreover, because training data is limited, during deployment the systems will face an even larger set of inputs, which can not be approximated during development or testing.
- *Formal verification is impossible or not scalable.* Applying classic formal reasoning for safety, such as model checking, is impossible due to the large input space, limited to small models or computationally intractable [6].
- *Limited fault detection, prevention and containment.* In systems that learn in an ETE fashion, fault detection is limited to checking if a system is online (similar to watchdog). Developing other fault detection mechanisms is limited by the lack of internal components and by the probabilistic nature of the output.
- *Limited reasoning about systematic faults.* Since the design space is minimal, it is harder to reason about errors coming from specifications or requirements.

- *Limited reasoning about code.* Most DL systems heavily rely on computational software libraries and involve very limited coding. Thus reasoning on source code or programming language constructs is limited.
- *Limited heterogeneous redundancy.* In most cases only some classes of algorithms achieve the accuracy needed for a task. For example, only NN achieve high performance in object recognition. Unfortunately, all algorithms that use the same constructs exhibit the same weaknesses [6], making heterogeneous redundancy impossible.

IV. Directions for Safety architectural design

Coming up with architectural design patterns that can successfully or partially mitigate the issues mentioned above will allow faster integration and roll-out of DL technologies in safety critical systems. As outlined in Section V, the impact of software architecture design in these scenarios was not yet explored. We propose and discuss three directions for future developments.

Delegation of Safety Responsibility: Instead of implementing hard safety mechanisms for DL algorithms, such as validating the output or implementing heterogeneous redundancy, one can try to *delegate* all safety responsibility to other components or *wrap* DL algorithms in *safety envelopes*. We can think of these patterns as the *thinker* and *doer* human traits; where thinkers may adopt unsafe ideas, but doers will restrain them to safe implementations.

For example, a computer vision based planning algorithm for autonomous vehicles might generate a series of unsafe manoeuvres or trajectories. However, safe executors will benefit from other sensors to delay execution of a maneouver or choose a different path in order to safely achieve the same goals.

We find similar paradigms in reference architectures for autonomous systems, where some components are responsible for decisions and others for execution [17, 18]. While no reflection is made about safety, any reference architecture or design pattern can be adapted to safely incorporate DL components by embedding some safety reasoning in executors.

In such cases, simple safety mechanisms such as watchdogs or homogeneous redundancy will suffice to ensure safe deployment of DL algorithms, on the cost of increased complexity for executors. Some advantages of delegation are the limited constraints imposed on the algorithms used and the increased level of abstraction of executors, which makes adapting or designing new patterns easier.

Partial Rejection of Safety Responsibility: The output of DL algorithms is a probability distribution over a set of possible outcomes together and a confidence score. Therefore, a system can decide to reject the predictions or to delay any decisions until it is confident enough. Similar cases occur when partial properties of a DL system can be formally verified.

For example, UAV operating at high altitudes can not encounter birds or similar objects. This property can easily be verified by checking the altitude (where altimeters corresponds to classic safety critical systems). If such a prediction is given by a sensing system, the system can decide to reject it or take further actions (such as to trigger a fail safe mode).

These systems allow partial verification of the output and small forms of heterogeneous redundancy, easing the requirements on testing scenarios. Therefore, such systems can hold partial safety responsibility.

Design patterns in this class can be similar to partial n-self checking programming patterns, sanity checks or partial fault detection [15]. One can imagine that simpler, formally verifiable, algorithms can help verifying partial properties of a more complex algorithm, thus leading to ensemble or cascading redundancy. The advantage of partial rejection is one can allow some uncertainty in a system, while imposing relatively low constraints on the DL algorithms used.

Fully Acceptance of Safety Responsibility: In some cases it will be impossible to allow any uncertainty in a system. One can imagine an intelligent planning algorithm for a spacecraft embarked in deep space exploration, where communication to Earth takes a very long time. Any non-verifiable properties of a component will most likely mark it unusable and lead to new system designs.

In such cases the discussion about safety will be balanced towards choosing a less powerful algorithm for which necessary safety properties can be verified. The development of some algorithms is driven by theoretical, mathematical, grounds and not by empirical evidence (as in the case of DL). Therefore, giving some guarantees for this family of algorithms is possible. Whenever safety properties can be verified and the algorithms fit functional requirements, the safety patterns developed will be more similar to deterministic software.

However, formal verification or strong proofs will not alleviate all the challenges raised in Section III. Software architecture will still plays an important role in *integrating* such algorithms in safety critical systems (and any other type of software system). For example, the limited design space of DL algorithms and the limited reasoning about systematic faults can still be managed through software architecture design. Moreover, fault detection and, particularly, fault containment will have to be solved through architecture.

Although fully acceptance of safety responsibility brings the design space closer to classic software components by imposing strong requirements on the probabilistic algorithms used (thus removing uncertainty), it is (at the moment) almost impossible to achieve for complicated tasks.

V. Related Work

There is little to no literature on the intersection between safety architectural design, architectures for intelligent systems and safe ML. Although several reference architectures for developing intelligent and autonomous systems, possibly including DL components, have been proposed in robotics - e.g. [17, 18, 19] - they do not take into consideration safety, possible failures, or misbehaviour of any of the components.

At the other end, the development of safety architectural patterns focused on deterministic systems and systems where fault detection and correction or formal verification is possible [20, 15].

The only related work we are aware of is the work of Salay and Czarnecki [21]. The authors assess the readiness of ISO 26262 safety standard for ML components. The standard section on software architecture design is reviewed, however, the authors consider possible to check the output of ML algorithms and detect possible data errors. This statement is in contrast with recent research which shows that detecting small perturbations applied to inputs is very hard [6].

VI. Conclusions and Future Research

We consider the problem of integrating DL components in safety critical software systems. Due to their opaque nature and to high model complexity, DL algorithms raise numerous challenges regarding system safety. We argue software architecture design can mitigate some of these challenges and enable faster development and deployment of new technologies. In this paper we sketch three directions for the development of architectural patterns that will help mitigate some of these issues. However, no pattern is yet presented.

For future work we propose to:

- Assess and adapt available architectural safety patterns able to mitigate any of the issues raised earlier (or new ones).
- Develop new patterns whenever adaptation is not possible.
- Build a system of safety architectural patterns for safety critical systems using DL.
- Develop a framework to help reasoning about the safety properties achieved when using these patterns and ease the selection process (similar to architectural tactics).

References

[1] M. Bojarski, D. Del Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. D. Jackel, M. Monfort, U. Muller, J. Zhang, *et al.*, "End to end learning for self-driving cars," *arXiv preprint arXiv:1604.07316*, 2016.

[2] A. Rodriguez-Ramos, C. Sampedro, H. Bavle, P. de la Puente, and P. Campoy, "A deep reinforcement learning strategy for uav autonomous landing on a moving platform," *Journal of Intelligent & Robotic Systems*, pp. 1–16, 2018.

[3] D. Sarikaya, J. J. Corso, and K. A. Guru, "Detection and localization of robotic tools in robot-assisted surgery videos using deep neural networks for region proposal and detection," *IEEE transactions on medical imaging*, vol. 36, no. 7, pp. 1542–1549, 2017.

[4] International Organization for Standardization (ISO), "ISO standard 26262:2011 Road vehicles - Functional safety," 2011.

[5] 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.

[6] A. C. Serban and E. Poll, "Adversarial examples - a complete characterisation of the phenomenon," *arXiv preprint arXiv:1810.01185*, 2018.

[7] K. R. Varshney and H. Alemzadeh, "On the safety of machine learning: Cyber-physical systems, decision sciences, and data products," *Big data*, vol. 5, no. 3, pp. 246–255, 2017.

[8] R. Kazman, M. Klein, M. Barbacci, T. Longstaff, H. Lipson, and J. Carriere, "The architecture tradeoff analysis method," in *IEEE ICECCS'98*, pp. 68–78, IEEE, 1998.

[9] S. Shafaei, S. Kugele, M. H. Osman, and A. Knoll, "Uncertainty in machine learning: A safety perspective on autonomous driving," in *International Conference on Computer Safety, Reliability, and Security*, pp. 458–464, Springer, 2018.

[10] A. C. Serban, E. Poll, and J. Visser, "Tactical safety reasoning. a case for autonomous vehicles.," in *IEEE 87th Vehicular Technology Conference (VTC Spring)*, pp. 1–5, IEEE, 2018.

[11] L. Gauerhof, P. Munk, and S. Burton, "Structuring validation targets of a machine learning function applied to automated driving," in *International Conference on Computer Safety, Reliability, and Security*, pp. 45–58, Springer, 2018.

[12] A. K. Saberi, E. Barbier, F. Benders, and M. van den Brand, "On functional safety methods: A system of systems approach," in *Systems Conference (SysCon), 2018 Annual IEEE International*, pp. 1–6, IEEE, 2018.

[13] International Electrotechnical Commission, "IEC 615038 - Functional Safety of Electrical/Electronic/Programmable Electronic Safety- Related Systems," 1998.

[14] B. O'Connor, "NASA Software Safety Guidebook," *NASA Technical Standard NASA-GB-8719.13*, 2004.

[15] C. Preschern, N. Kajtazovic, and C. Kreiner, "Building a safety architecture pattern system," in *European Conference on Pattern Languages of Program*, p. 17, ACM, 2015.

[16] W. Wu and T. Kelly, "Safety tactics for software architecture design," in *COMPSAC*, pp. 368–375, IEEE, 2004.

[17] J. S. Albus, "The NIST real-time control system (RCS): an approach to intelligent systems research," *Journal of Experimental & Theoretical Artificial Intelligence*, vol. 9, no. 2-3, pp. 157–174, 1997.

[18] E. Gat and R. P. Bonnasso, "On three-layer architectures," *Artificial intelligence and mobile robots*, vol. 195, p. 210, 1998.

[19] R. Volpe, I. Nesnas, T. Estlin, D. Mutz, R. Petras, and H. Das, "The CLARAty architecture for robotic autonomy," in *IEEE Aerospace Conference*, vol. 1, pp. 121–132, IEEE, 2001.

[20] A. Armoush, *Design patterns for safety-critical embedded systems.* PhD thesis, RWTH Aachen University, 2010.

[21] R. Salay and K. Czarnecki, "Using machine learning safely in automotive software: An assessment and adaption of software process requirements in ISO 26262," *arXiv preprint arXiv:1808.01614*, 2018.