Traceability of Deep Neural Networks

Vincent Aravantinos fortiss GmbH
Guerickestr. 25
80805 Munich, Germany
Email: aravantinos@fortiss.org

arXiv:1812.06744v1 [cs.SE] 17 Dec 2018

Abstract—*[Context.]* **The success of deep learning makes its usage more and more tempting in safety-critical applications. However such applications have historical standards (e.g., DO178, ISO26262) which typically do not envision the usage of machine learning. We focus in particular on** *requirements traceability* **of software artifacts, i.e., code modules, functions, or statements (depending on the desired granularity).** *[Problem.]* **Both code and requirements are a problem when dealing with deep neural networks: code constituting the network is not comparable to classical code; furthermore, requirements for applications where neural networks are required are typically very hard to specify: even though high-level requirements can be defined, it is very hard to make such requirements concrete enough, that one can qualify them of low-level requirements. An additional problem is that deep learning is in practice very much based on trial-and-error, which makes the final result hard to explain without the previous iterations.** *[Proposed solution.]* **We investigate which artifacts could play a similar role to code or low-level requirements in neural network development and propose various traces which one could possibly consider as a replacement for classical notions. We also propose a form of traceability (and new artifacts) in order to deal with the particular trial-and-error development process for deep learning.**

Index Terms—**traceability; neural networks; deep learning; machine learning; safety**

I. Introduction

The success of deep learning (DL), in particular in computer vision, makes its usage more and more tempting in many applications, including safety-critical ones. However the development of such applications must follow standards (e.g., DO178 [1], ISO26262 [2]) which typically do not envision the usage of machine learning. At the moment, practitioners therefore cannot use machine learning for safety-critical functions (e.g., ASIL-D for ISO26262, or DAL-A for DO178).

There exist various attempts to address this issue whether in standardization committees (e.g., ISO/IEC JTC 1/SC 42 or DKE/DIN [3]) or in the academic community (various initiatives towards explainable AI, e.g., [4]), but they are all far from mature and definitely not usable as of today or do not really address the problem: most standardization approaches just try to map one-to-one classical software engineering processes like the V-model to deep learning. Furthermore, no academic solution, at the moment, provides a solution to the lack of understandability of deep neural networks (DNN).

In this paper, we try to find a pragmatic approach, which focuses on artifacts rather than on processes: we are not prescriptive regarding the activities which produced these artifacts. More precisely, we focus only on artifacts which are worth being identified during the development of DNNs

for the sake of traceability. Consequently, this paper does not provide a ready-made solution, which a practitioner could follow one-to-one. However, it provides concrete descriptions which should at least be sufficient to provide a first guidance.

We restrict the scope of this paper to the following:

- Deep neural networks for supervised learning (no reinforcement learning, no unsupervised learning).
- We focus only on software, not on system: traces from software requirements to system requirements are out of scope, as well as FMEAs or failure rates.
- We do not focus on binary code or deployment thereof on hardware platform.
- We assume a fixed, non-evolving, dataset: this does not comply with most real cases in, say, autonomous driving, where data is continuously collected. Even if not continuously collected, the dataset has so much influence on the training that one can hardly ignore these evolutions for proper traceability. Still, there are already sufficiently many questions to address without considering this evolution, which is why we leave this out of focus in this paper.
- We focus essentially on functional requirements.

Lifting these restrictions is left to future work.

The rest of the paper is organized as follows: Section II presents related work. Section III recalls the concept of traceability. Section IV provides a traceability-amenable presentation of deep learning. Section V contains the main contribution of this paper: it analyzes which DNN artifacts replace classical software artifacts and suggests new artifacts and traces to enable the traceability of DNNs. Section VI identifies various gaps of the present work for future research. Finally Section VII summarizes the paper.

II. Related work

In general, the safety of DNNs is commonly recognized as a huge challenge [5]. There are more and more attempts towards the certification, verification, or explainability of DNNs, of which we provide now a short overview. None of them however (and, as far as we know, no other work either) addresses the traceability of DNNs.

Back in 1996 a set of requirements for a standard certifying the use of neural networks in safety critical applications was gathered [6]. Traceability is (indirectly) mentioned as a problem to address, but no solution is provided.

In 1999, the classical waterfall model was adapted to the development of DNNs [7]. Even though the problematic of

traceability (in particular to the data) is indirectly addressed, the proposed process is very much oriented towards *activities* described in an informal manner, rather than on a concrete list of artifacts to provide. The resulting process is therefore worth taking inspiration from, it remains however very high-level: there is a large freedom of interpretation about what the artifacts shall actually contain. Reading the paper definitely does not provide enough information to trace DNNs.

There has been attempts to apply principles of software engineering (or even of engineering in general) to NNs [8]. It particularly attempts to address the lack of reproducibility in the development of NNs. Even though the terminology and techniques have changed a lot since 2004, the identified problems are still relevant. Still, the solutions of the paper do not answer the need for traceability and seem to hardly match nowadays' practice.

The discrepancy between the recommendations of the ISO 26262 and the methods actually used in practice was analyzed in [9]. This cannot be directly used for traceability but is indirectly a very useful source of information.

A safety argumentation for the usage of DNNs, in particular formalizing it partly using the Goal Structuring Notation (GSN), was developed [10]. The granularity level of this work is low, meaning in particular that we cannot derive directly from it any notion of trace. The same applies for more recent work also applying GSN in the context of NNs [11,12].

There exists of course lots of heuristics, lessons learnt, best practices, all of which are available on the internet, e.g., [13,14], or analyses regarding the technical debt of machine learning [15]. None of them provide of course a concrete notion of traceability but the corresponding analyses allow to learn which information it is worth to trace or not.

The questions of interpretability and explainability of DNNs are very hot topics, with few actual solutions at the moment. We refer to [4] which gathers various such solutions into a set of metrics to use in order to assess a network. Papers on distill like, e.g., [16], are also excellent sources to help understanding DNNs. None of these are entirely satisfying nor do they relate to traceability, however they all could provide technical insights in order to assess what should be traced or not.

There has been attempts to set grounds for a "rigorous science" of machine learning [17], again very useful, but not related to traceability. Finally, many safety-related problems have been identified for AI, especially for reinforcement learning [18]. The identified challenges are relevant and the paper proposes a few attempts of solutions. Most of them are a source of inspiration to identify sources of problem and to analyze whether those sources can be tackled with traceability (even though it turns out not to be really the case for solutions identified in the present paper).

III. PRELIMINARY: TRACEABILITY

It is very difficult (at least nowadays) to find engineers or researchers who know both safety-aware software engineering

and deep learning.[1] This paper really attempts to answer a problem which lies at the intersection of two communities and tries therefore to be self-contained for both. Consequently, we first recall the concepts and terminology related to traceability, as used in this paper. This should be a trivial reading for the safety-critical systems software engineer, but we do recommend reading it to ensure that the terminology is clear in the rest of the paper. Even though not a proper formal source, we still recommend Wikipedia [19] on this topic.

A. Artifacts

When developing classical software, the only product to deliver is executable code. One might provide also source code if the software is open source; the software itself might be part of a bigger system if it is embedded; but, all in all, from the perspective of the software engineer, one just needs to deliver some sort of executable. For safety critical systems, this is not enough: one needs to deliver not only the executable code itself, but also a *justification* that the executable code indeed does what it is supposed to do or that it is resilient to faults. Such a justification is provided in the form of documents, source code, executable, etc., which are not intended for the final consumer, but for the authority (whether it is an independent authority or a company-internal one) in charge of validating the safety of the product. We call these *(development) artifacts*.

One such essential document is the one describing *requirements*: requirements describe what the software is *supposed* to do, without providing implementation details. In many non-safety critical applications, requirements are expressed in a very unstructured manner, e.g., in some statement of work, in an issue tracker, or in slides communicated from the client. In safety critical applications however, it is essential to have these requirements in a way that they can be structured, referenced, or even categorized. For instance: *functional requirements* describe the expected function of a component, *timing requirements* describe the temporal constraints for a given function, *interface requirements* describe the input/output types of a component. Requirement documents found in safety-critical industry typically use dedicated software like IBM Rational DOORS.

Example 1 (Functional requirement [20]):

> The [system] shall be capable of predicting the paths of the subject vehicle as well as principal other vehicles in order to identify the vehicle(s) whose path(s) may intersect with the subject vehicles path.

Requirements are only one sort of document among many to be provided: source code, test results, development plans or any other sort of document which turns out to be necessary to

---

[1] This is especially a problem for the development of autonomous vehicles, or cyber-physical systems in general, where diverse domains have a strong influence on software (safety engineering, control engineering) or on safety (machine learning): this interdependency is deep enough that software and safety engineers cannot afford not to be *at least* initiated to the other domains. This poses of course a great problem because most engineers are trained in one domain only: classical engineering can be solved by simply *synchronizing* engineers of diverse disciplines. This is not sufficient for highly complex cyber-physical systems.

justify that the final software can be used in a safety-critical system. This list is non-exhaustive and typically defined by a standard, like ISO26262 [2] or DO178C [1].

B. Traces

The delivered artifacts generally have dependencies between each other: typically, the source code should fulfill the requirements, software requirements should refine system requirements, executable code derives from source code. Keeping these dependencies implicit increases a lot the risk that a dependency be wrong or forgotten. *This is the purpose of traces to make these dependencies explicit.*

Every pair of artifacts is in principle subject to being traced from/to each other. In this paper we consider especially traces from code to requirements.

Example 2: As an example, consider a requirement (defined in some document, e.g., a Word document or a DOORS database) being identified with an ID, say REQ_123 ; take then a piece of code defining many functions, one of them – say f_456 – implementing REQ_123. Then a trace is typically nothing more than a comment just before the function simply stating $[REQ\_123]$:

```
1    // f_456 takes as arguments:
2    // − x: ...
3    // − y: ...
4    // It returns ...
5    //
6    // [REQ_123]
7    int f_456(int x, float y) {
8        ...
9    }
```

The trace is the comment on line 6.

Another typical example is a trace between a test case and a requirement: it is important to ensure that the test cases indeed support the verification of requirements and that no requirement is forgotten. Even further, it is essential to also trace the *results* of the tests to the test cases themselves to ensure that the tests are indeed done and maintained.
Writing down a trace is in general a *manual* activity: engineers look up the code and the requirements and add manually the comment above.[2]

C. High- vs Low-level requirements

In many cases, requirements are not concrete or precise enough to be traced directly with the above level of granularity (see Example 1). Therefore, it is often recommended to first refine the requirements into more concrete requirements, which can be traced from the code. These artifacts can have different denominations. For instance, the standard for the development of software for civil avionics (DO178C [1]) names them high-level and low-level requirements (HLR/LLR) respectively (but the concepts is transferable to other standards and domains),

with the following definition for LLR: "Low-level requirements are software requirements from which Source Code can be directly implemented without further information." [1]. LLR should themselves be traced to HLR in order to have complete traceability.[3] Note that the definition of HLR and LLR is not absolutely clear: we encountered examples where some requirements were considered as high-level by a company and low-level by another.

In general, refining HLR into LLR goes hand in hand with architectural decisions: the requirements can be decomposed only once the function is decomposed into smaller functions, to which one can assign more concrete requirements. This is why the DO178C, for instance, refines the HLR into two artifacts: the LLRs on one hand, and the Software Architecture on the other hand. More concretely, the software architecture defines a set of components and connections between these components – or, more precisely, a set of *interfaces* (i.e., data types for inputs and outputs), since the software architecture does not cover the implementation of the components. Interfaces typically contain even more information like the physical units of the types (e.g., meters, centimeters, meter per second), or, if relevant, refreshing rates. The LLRs can then be mapped to each interface. Finally, the LLR and the software architecture are the only information necessary to write down the source code. Whether defined in a requirement or separately, there is always a definition of interfaces. In the following, we will generically refer to such a definition as an *interface requirement*.

Fig. 1 represents the artifacts mentioned above. Of course, every artifact refining a previous one shall be traced to the latter, this should be typically bi-directional: every piece of information found in a refined artifact shall be found in the corresponding refining artifact, and conversely, every piece of information – except design decisions – found in a refining artifact shall be found in the refined one. In the DO178, the software architecture is not traced back to the HLR because it is a design decision.

The figure also presents the test artifacts: test cases shall be traced as well to requirements (high- or low-level depending on the context), and test *results* shall be traced to test cases.

D. Rationale

Understanding the rationale behind traces 1. enables to understand why it is challenging to trace DNNs, and 2. gives hints to investigate relevant alternatives to classical traces.

A trace serves the purpose of ensuring that a piece of code is justified by a requirement. This is not a structured or formal justification, which are in practice seldom applicable, however it at least enforces that people *think* about this justification. In fact, traceability does enable to identify sources of error: when an engineer attempts but does not manage to trace a piece of code then they might indeed get aware that this code is not necessary or, even worse, that it introduces unwanted functionality. Conversely, if a requirement is not traced back

---

[2]Note that if there was a practical way to automatize it, then we would most probably not need traceability at all: we could just ask a tool to verify automatically that the implementation satisfies the requirements. That is the objective of research in formal verification, e.g., model checking [21].

[3]The reader familiar with contract-based development [22] might see here an analogy. Remember however that requirements are *informal*: they are not specifications like contracts.
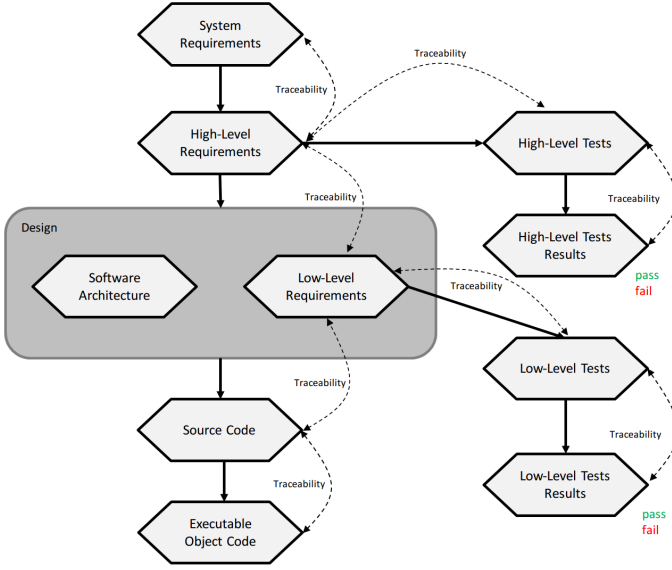
Fig. 1. Classical artifacts – presentation inspired from the DO333 [23]

by any code, then it is often a hint that the requirement has been forgotten. For the same reason, traceability is a relevant tool for assessors in order to detect potential pitfalls during development. This is what is illustrated in Fig. 1 by the bi-directional arrows for traceability: having traces syntactically on each side is easy; it is however harder to ensure *coverage* of traceability on both sides, e.g., *all* HLR are traced to some LLR and *all* LLR are traced back to some HLR (the latter typically does not happen since some LLRs depend on design decisions).

E. Process vs Artifacts

Many standards like, e.g., DO178C, do not impose an order on how artifacts shall be developed. For instance, even though code shall be traced to requirements, it does not mean that one is forced to follow a waterfall model: one might just as well work iteratively, define requirements, then code, then go back to requirements, etc. The main point of traceability is that, no matter how one reached the final state of development (e.g., iteratively or waterfall), it should be possible to justify that this final state is coherent. Consequently, one might very well develop all the artifacts without traceability, and only at the end develop the traces.[4] This is why we emphasized in introduction that this paper is not process- but artifact-oriented: we do not impose *how* engineers should work but only *what* they should deliver.

IV. DEEP LEARNING ARTIFACTS

This section presents the concepts and terminology related to deep learning, in a way which makes it amenable to comparison with the artifacts of the previous section.

To implement a required function using a DNN, one collects a lot of data matching as an input with their corresponding

[4]This is of course not recommended though: this might delay detection of development errors. In addition, defining and maintaining traces early during the process makes change impact analysis easier.

expected outputs (the outputs are not collected but typically manually annotated). This data is then used by a given DL framework to train the network. Examples of such frameworks are TensorFlow [24] or PyTorch [25]. A typical example of a function where one would want to use a DNN is the following: "given an image represented by a matrix of pixels, return a list of 4-tuples (x, y, w, h) representing rectangles which contain pedestrians" (such rectangles are typically called *bounding boxes*). One might require to identify various classes of objects (e.g., pedestrian, car, bikes) and to associate every bounding box with a label indicating to which class the object belongs, see Fig. 2.

Fig. 2. Bounding boxes – *image taken from [26]*

To teach a DNN, one needs the following:

- A *dataset* containing both the input, e.g., images, and the output, e.g., annotations denoting bounding boxes for pedestrians' positions. In the following, we will consider these two aspects separately: the *raw dataset*, i.e., only the input, and the *labels*, i.e., only the corresponding expected output.
- A deep neural network *architecture*. Prior to learning time, one cannot really consider that there is an actual neural network, but rather a skeleton thereof: the learning process will fill in this skeleton (more specifically, the weights), and doing so will generate the neural network used for the required function (in practice, the skeleton is actually randomly pre-filled and the random weights are progressively changed during learning). Such a skeleton is however more than just a box: the deep learning engineer decides on the shape of this skeleton, which does influence the learning process. A DNN architecture is typically designed as a layer-based architecture where the input, represented as a (potentially huge) vector or matrix (e.g. $640 \times 480 \times 3$ for an image of width 640, height 480 and with three color components R, G and B), flows through various massively parallel operations transforming it until the output has the expected form (e.g., a vector containing 3 real numbers between 0 and 1 indicating each the confidence of the image containing a pedestrian, a car or nothing). The engineering then amounts to designing this architecture, meaning defining these operations: their types and the dimensions they transform their input into. See Fig. 3 for an example.
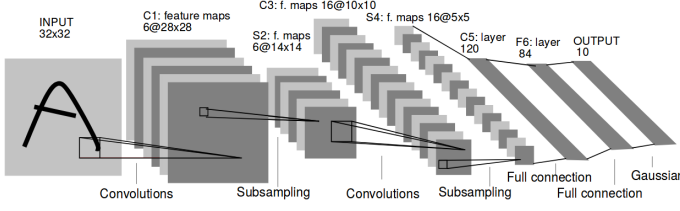
Fig. 3. DNN architecture – *image from [27]*

- A *loss function*. To train a DNN, one must have a way to tell the machine learning framework when the DNN is wrong, in order to correct it. In theory, this is easy: if the network provides a wrong answer to an input for which we know the correct answer, then we just tell the framework what the right answer was. However, in practice, the functions addressed with DNN typically output a *confidence* rather than a perfect answer. One should therefore be more subtle than just telling "right" or "wrong". Instead one can provide a positive real number, basically a grade, telling *how wrong* the system is. If the number is null, then there is no error; otherwise, the higher the number, the more important the error.[5] Consequently, a *loss function* takes the expected and actually obtained results of the DNN as inputs, and returns a real number stating how bad the difference between both is. Mathematically, *distances* make good candidates for losses.

  Example 3: Example of a loss function:

  $$\mathcal{L}(\theta) = \|y_{pos} - \hat{y}_{pos}\|_2 - \sum_{c \in C} log(y_c).\hat{y}_c$$

  where:
  - θ denotes the set of all parameters of the DNN (i.e., its weights),[6]
  - y_{pos} (resp. $\hat{y}_{pos}$) denote the position of an inferred bounding box (resp. the actual, labelled, position of the bounding box – *ground truth*),
  - $\|.\|_2$ denotes the L2 norm,
  - C is the set of classes considered in the problem at hand, e.g., $\{pedestrian, car, cyclist\}$,
  - y_c (resp. $\hat{y}_c$) denotes the class assigned to the inferred bounding box (resp. the actual, labelled, class of the bounding box), via a so-called one-shot encoding, i.e., a vector of the size the number of classes, where each element contains a real number between 0 and 1 assessing the confidence of belonging to the corresponding class.

  We leave it to the reader to observe the variation of the function depending on the error of the network (or lack thereof).

In practice the loss function is expressed using *code*: this code does not go in the final product but controls the learning of the DNN within the DL framework.

As we will see, it will be essential for the rest of the paper not just to understand the artifacts themselves, but how they are developed. Typically, the sequence of decisions are as follows:

1) Collect data and, possibly, preprocess it: re-shape the information, fix missing values, extract features, or achieve much more advanced tasks like matching label ground truth boxes to so-called prior boxes [28] (we do not focus on this activity in this paper).
   Delivered artifacts: raw dataset, preprocessing functions.
2) Annotate the raw data.
   Delivered artifact: labelled dataset.
3) Split the dataset in *training*, *validation* and *testing* sets.
   Delivered artifacts: labelled training-, validation- and testing-datasets. The difference between the validation and testing datasets is that, after evaluating the DNN on the validation dataset, the engineer will take the result as a feedback into account to improve their design. When done and no more correction is planned, then the engineer will assess the quality of their DNN with the testing dataset. This should *not* entail further iterations of the design (see step 12).[7]
4) Design the DNN architecture.
   Delivered artifact: DNN architecture (typically as python code making use of the selected framework).
5) Define the "learning configuration" this includes picking a loss, picking learning parameters (e.g., dropout [29], learning rate, maximum learning steps), or search strategies for these hyper-parameters (e.g., grid or random search), or even strategies involving the exploration of the dataset itself (e.g., curriculum learning). This learning configuration is a placeholder artifact for all aspects which potentially influence the learning process, e.g., the used version of the various software dependencies or the used random seeds. We do not make this list exhaustive since this is not the focus of this paper. Overall, the configuration shall be understood as the minimal piece of information such that the tuple [training set, architecture, learning configuration] characterizes uniquely the learned DNN. This requirement aims at ensuring the reproducibility of the learning.[8]
   Delivered artifact: Typically not "one" artifact but rather various pieces scattered across different artifacts: e.g., fine-tuning parameters stored in code, loss having its own source file, etc. Ideally, this could be gathered in some configuration files, as provided by some DL management platforms [30].
6) Train the DNN architecture with the loss function on the training dataset using a selected deep learning frame-

---

[5]In many cases, the objective is only to minimize the loss, not necessarily to nullify it.

[6] Contrarily to one would mathematically expect, θ *is* present on the right-hand side, but only implicitly: rigorously, one should write y as the result of applying the function represented by the DNN, as parametrized by θ. We follow however the conventions used in the classical DL literature.

[7]Note that the terms *testing dataset* and *validation datasets* are sometimes exchanged in the literature.

[8]We are on purpose quite vague on this matter because reproducibility is way harder to reach than one might think: not only random seeds influence the learning process, but also potentially the operating system, the library versions and even the hardware, which might, e.g., swap instructions differently non-deterministically.

work.

Delivered artifact: (trained) weight values. Note that the artifact is not the code, which, *per se*, is not different before and after training: the learning process alters the values of the variables used by the code, not the code itself. Consequently, the artifact is actually the resulting information stored in those variables.

7) Post-process the trained DNN (if necessary): many learning strategies require a change between learning and inference phase (e.g., drop out is applied only during learning).

Delivered artifact: inference architecture. In that case, it is the opposite to the previous step: the code changes but the data does not. Note however that in most cases, the switch from the learning architecture to the inference one is so standard and systematic that there is no need for any separate artifact: typically, a DL framework will simply provide an optional argument which one shall set to true if in learning mode or to false in inference mode.

8) Test the resulting DNN on the validation dataset.

Delivered artifact: test results (e.g., a metric like accuracy in the form of a number between 0 and 1), typically stored in a variable of the python runtime, or in a log file, or in a CI/CD system, if any.

9) Change the architecture or the learn configuration (4–5) based on the results and repeat steps 6–9 until the targeted objectives are reached.

10) Assess the quality of the inference DNN with the test set

Delivered artifact: final validation results.

11) Depending on the used framework, serialize/export the network in order to use it in production, e.g., to be linked from a C++ source file, and compile it.

Delivered artifact: executable code usable in production,

Quite similarly to code development, the process yielding the finally delivered DNN is a typical trial-and-error process. There is a major difference though: code resulting from a trial-and-error process can still be understood. This is typically not the case of DNNs: often, the only way to understand why a given architecture is finally obtained, is by looking back at the changes which led to it. This has of course a big impact on justifiability of a DNN and therefore on the traceability. We will get back to that point in Section V-B.

Note that steps 1 and 9 are not duals: the former is a pre-processing of the *data*, which must therefore also happen at runtime; while the latter is a post-processing of the *DNN* itself, which therefore happens once and for all at design time and is not repeated at runtime.

Fig. 4 summarizes the DL artifacts in a similar way to Fig. 1. Note again that this does not denote a process, but really a set of delivered artifacts: no sequence is imposed on the order in which the artifacts are developed. In particular, it is strongly to be expected that, once a developer decides to implement a function using a DNN, additional requirements (called "derived" in the DO178) might have to be added a posteriori: the choice of using DL as a technology might indeed entail new considerations at the requirement level.

Fig. 4. Adaptation of Fig. 1 to DNN artifacts

V. Tracing DL artifacts

How can we map the DL artifacts presented in Section IV to the classical ones presented in Section III? First notice that both sections are not exactly targeting the same level of granularity: Section IV did not mention requirements, but there are of course requirements when developing DNNs for safety critical systems. Contrarily to software however, we believe that requirements implemented with DNNs generally cannot be refined into a software architecture and an LLR. This is not particularly a property of DNNs per se, but rather of the *functions* for which it makes sense to use DNNs: most applications for which DNNs are used successfully compared to classical methods, are applications where humans have difficulty decomposing the problem in hierarchical simpler sub-problems. One can even interpret the success of DNNs precisely under this angle: the learning activity does not just learn a solution to the problem, but also learns its own decomposition of the problem. With respect to requirements, this supports the claim that applications where DNNs are useful are precisely those where it is very hard to come up with a decomposition of HLR into LLR: refining HLR into LLR is intrinsically difficult – otherwise one could most probably use a classical (i.e., non-DNN) method. Consequently the only artifacts between the HLR and the source code are all the inputs to the DL framework: architecture, learning configuration and, of course, training dataset. High-level tests are now replaced by the testing/validation set: the name differs but the role is the same.

Let us analyze how artifacts from Fig. 1 map to the ones of Fig. 4 in order to highlight similarities and differences:

- System requirements, HLR, tests cases, test results and executable code are found in both cases.
- As hinted by Fig. 4, the source code is still present but it is split between the architecture part and the weights part.
- As mentioned above, the LLR and software architecture cannot really be mapped to the DL artifacts, unless one maps them to the complete design block, which does not bring anything.

When it comes to traceability, traces between preserved artifacts are maintained. Traces between source code and object code can also be considered as preserved since these traces basically amount to trace the code generated by the compiler back to the source code: this is not different for DNNs and for classical software. However traces between HLR and Design, and Design and Source code shall be adapted. The next sections are dedicated precisely to these traces.

More precisely, we need to consider traces between:

1) HLR and training dataset,
2) HLR and learning configuration,
3) HLR and architecture,
4) training dataset and source code,
5) learning configuration and source code,
6) architecture and source code.

For the source code, one can differentiate inference architecture and learnt weights. Inference architecture simply can be traced trivially to the design architecture (when it is not the same artifact anyway as mentioned earlier) and to no other design artifact. The next section deals with traces between HLR and training dataset, the following section deals with all other traces.

A. Traceability between HLR and training dataset

Traces between HLR and dataset may seem simple: one just needs to trace every element of the dataset to the HLR. Some aspects are easy to think of tracing: the type of the raw data can be traced to the input definition in the interface requirement, or the type of the labels can be traced to the output definition. This sort of traceability can be targeted but we believe that it is too trivial to support the identification of any relevant problems: type mismatches between dataset and interface are not real sources of problem in practice. In addition, any such problem typically breaks anyways during integration of the DNN components with the rest of the system, so that there is no real possibility of encountering such an error when delivering a safety-critical system. We still go into more details about it in Appendix A in case the reader finds the problem relevant to their particular use case.

Let us focus rather on the traceability of every piece of data to HLR, e.g., "The function shall recognize obstacles in urban context", "The function shall recognize obstacles by nice weather". In principle, it is simple to trace the dataset to such requirements: e.g., pictures in the dataset taken by nice weather shall be traced to the corresponding requirement, pictures in urban context as well, etc. However, the sort of information usually found in an HLR often applies uniformly to all elements of a dataset: e.g., if the function shall work only in urban context then *all* images of the dataset will be urban. This would entail tracing the entire dataset to the HLR, which would be so general that it would not really support the rationale of tracing: tracing the entire dataset to the HLR does not really provide a justification of this particular dataset. Instead, one expects every datum to be justified individually and therefore to be traced potentially differently from another datum.

At that stage, we recommend developing the interface requirement *much more* than it usually is, in addition to the types and units of the inputs/outputs, it should describe in a detailed manner the output and – especially – input domain, with the purpose of defining what is an acceptable coverage of the domain. This can be done either as a requirement among the HLR or as a separate artifact, which we call "domain coverage model". Getting back to the example above, "urban" is not enough: one should actually detail which different forms of environment are encountered in an urban environment, e.g., "one-way street", "roundabout", etc. (of course, in that case, the input domain coverage model connects strongly to the Operational Design Domain – ODD – but it needs not be the case if the function to be performed by the DNN does not directly work with data coming from the sensors). These should be themselves traced towards higher-level requirements, e.g., system-level requirements: this might even be a useful tool to identify misunderstandings regarding the environment, e.g., imagine a portion of highway which is within the limits of a city: is it urban or not?

If working in a very structured context, e.g., where model-based requirements engineering is used (see, e.g., [31]), the domain coverage model could really be formalized to some extent, via coverage criteria on the domain coverage model. In such cases, this activity comes in close connection to model-based testing [32], the main difference with these classical approaches being merely the size of the model, which is typically huge in DL, much bigger than for classical approaches. Similar approaches have been carried out in machine learning in the literature, see e.g., [33], to a much smaller and less systematic extent. Note finally that, from a control engineering perspective, this is a bit similar to modelling the plant of a controller. Contrarily to a controller, the resulting NN is not analyzable. The domain coverage model plays thus an even more important role, which therefore justifies that it becomes a first-class citizen w.r.t. traceability.

Note that it is typically very hard for a requirement engineer to know *beforehand* which level of granularity to put in such an input domain coverage model. Actually the level of granularity probably depends on the dataset itself, and can thus be identified only once the dataset is already (at least partially) present: this is counter-intuitive regarding the usual notion of requirement (even though it matches the practice thereof: requirements are never perfect from the beginning, they always need iterations). However remember that we do not focus on the order in which artifacts are delivered but only on ensuring their mutual consistency. In this respect, it is acceptable to generate or modify *a posteriori* such a requirement.[9]

To find out the proper level of granularity, one shall keep in mind that such a domain coverage model shall serve as a *tool* to analyze the dataset by justifying why a particular datum is in there, and identifying cases where some situation might not be covered. Consequently, if too many pieces of data are tracing to the same environment requirement, then this environment requirement probably does not serve its purpose. Conversely, if

---

[9] There is actually a good reason why HLR generally are not written with that level of detail: it is typically impossible to know *ahead* to which granularity level one should describe the environment.

very few pieces of data trace to one environment requirement only, then either this requirement is too specific or the dataset needs to be completed. Defining "too many" or "very few" is beyond the scope of this paper, but should be of course defined in a rigorous manner depending on the context.

If the domain coverage model is defined with a very low-level of granularity, then we have the above situation that traceability becomes useless because applying equally to the entire dataset. On the other hand, if the domain coverage model is defined with a very high-level of granularity, then its coverage is probably not reachable as displayed in Fig. 4: the traceability arrow between HLR and dataset is not bidirectional.

Note finally that, even though the discussion above targets especially the raw dataset, the same applies to the labels if their domain is complex enough: for instance, if the DNN shall provide the position on a pedestrian, then it is important to ensure that the domain of positions is adequately covered

Fig. 5 updates Fig. 4 to reflect the new artifact and the corresponding traceability.
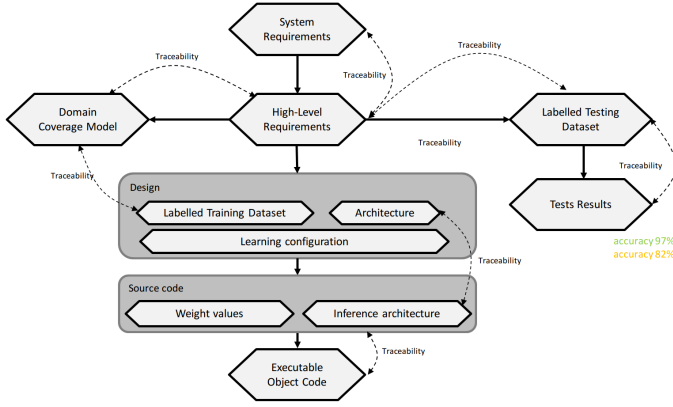
Fig. 5. Extension of Fig. 4 to integrate the domain coverage model

B. Trial-and-error traces

The following traces remain:
1) HLR and learning configuration,
2) HLR and design architecture,
3) training dataset and learnt weights,
4) design architecture and learnt weights,
5) learning configuration and learnt weights.

Even if simple to implement, a first essential trace is the one between the training dataset version and the learnt weights: indeed, it is easy in practice to lose track of which version of a dataset was used to train a given network. This trace requires no more than a unique identifier for a given version of the training dataset and a reference to this identifier in the trained DNN.

For more meaningful traces, one can trace these artifacts just the same way as one does for classical software engineering: trace code to requirements. Since code has a very specific structure for DNNs, we can be a bit more precise: one can try tracing *neurons* to requirements. For instance, we could impose on the DL framework to keep a trace of which input

datum impacted more which neuron. This is precisely the approach shortly mentioned in [5].

Even though doable in theory, this approach brings nothing in practice, it is acknowledged as impossible – at least as of today – to interpret, understand or explain the role of one particular neuron. In addition, the size of the DNN and of the dataset are so huge that one cannot expect to understand a-posteriori any useful piece of information out of it (though this might change in the future if explainable AI becomes successful). Consequently, this sort of traces will not fulfil the traceability rationale: if a reviewer inspects the involved artifacts in their state at the end of the project, they will not understand them nor their connection to previous artifacts.

Remark. Note that the problem is new, but also has a part of well-known aspects to it: DNNs are, in essence, generated; therefore, like every generated code, they are much harder to understand and to trace than manually written code and thus cannot be trusted without further argumentation – which is why standards like DO330 exist [34]. Classically generated code can however usually be understood, which is not the case of DNNs, adding tremendously to the "classical" difficulty.

Instead of waiting for explainable AI to provide solutions[10], we suggest in this paper to trace the engineers' decisions instead of the artifacts themselves: if artifacts are not understandable, engineers' decisions shall be. How do engineers come up with architectures or learning configurations? They essentially *try* them, test them, and try again until they cannot improve the results anymore. In other words, these decisions are intrinsically based trial-and-error: see Fig. 6 for an illustration. Trial and error is usually not considered at the level
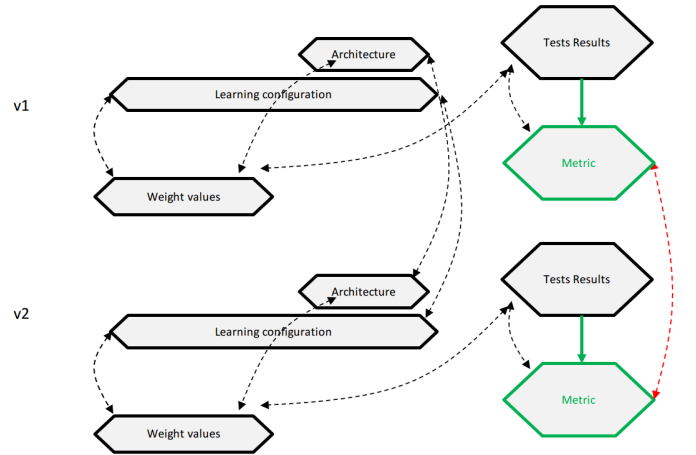

Fig. 6. Trial-and-error

of traceability: as mentioned earlier, it is rather the opposite, one expects from traceability that we can ensure the coherence of the artifacts *in their final state*, i.e., independently of how they were obtained, by trial-and-error or not. However, DNN development relies so much and so intrinsically on trial-and-error, that we feel necessary to embrace this kind of activity even for traceability. Future developments might provide more

[10]Which might even never happen, see, e.g., [35] for an informal discussion on the matter.

predictable and reproducible approaches to the development of DNNs, in which case the approach of the present section will become obsolete. At the moment, instead of simply avoiding this reality and hoping for techniques which might never come, we make an attempt for a pragmatic approach usable today.

In case of trial-and-error, the only justification that one can provide is that a given artifact is better than its previous version. Consequently, we propose to trace every new artifact obtained by trial-and-error to its previous version. The objective of the trace being to demonstrates that the new version *improves* upon the previous version.

This requires storing not only the final artifact but also all the previous versions of it – or at least all those which are necessary to understand the artifact obtained at the end. It might sound like an overkill but note that it is actually standard to store previous versions of artifacts in the development of safety critical systems (where it is often encountered under the term "configuration management") or, of course, for normal software with version control (even though it is usually restricted to source code: not for binary artifacts). Pairing these classical techniques with traceability forces however the engineer to do more than just tagging a new version in their version control system: they must also think about the justification of the new increment.

Therefore, we suggest imposing the developers to define a *metric* (or *KPI*) to measure the quality of the inference DNN – which they anyways normally do, maybe however not always formally. Such a metric should not be the loss but be defined according to the actual goals that one plans to achieve with the function (e.g., a car can be mistaken for a pedestrian, but not the other way around). The metric can range from simple cases like accuracy and/or recall to complex combinations of functions [4]. As a new artifact, one must then explicitly store the values of this metric for a given DNN. Of course this value shall be traced to the weight values and inference architecture with which it was obtained. *The essential addition is then to require that every version of the network which is obtained by increment of a previous one shall be traced to the metric value obtained with this previous version: one can then easily check if the new value indeed is an improvement.* This metric should be the same in order to measure the quality of all the evolutions of the DNN. If it changes during the course of the project or is defined only a posteriori, then one needs to re-check the *entire trial-and-error chain* leading to the final version of the DNN. We summarize the change of artifacts in Fig. 7.

This whole process might sound like a big hindrance for the practitioner, but note that: 1. the problem of not providing a real argumentation for a so-called improvement is actually recognized as a problem, even by the machine learning community itself (see e.g. "Explanation vs Speculation" in [36]), and 2. it is still much easier to apply than any approach currently taken in the field of explainable AI.

Our recommendation in its current state can easily be "tricked": nothing forces a developer to deliver the previous versions of their DNN; they can just claim that the version they delivered was the first version that they developed, which, by
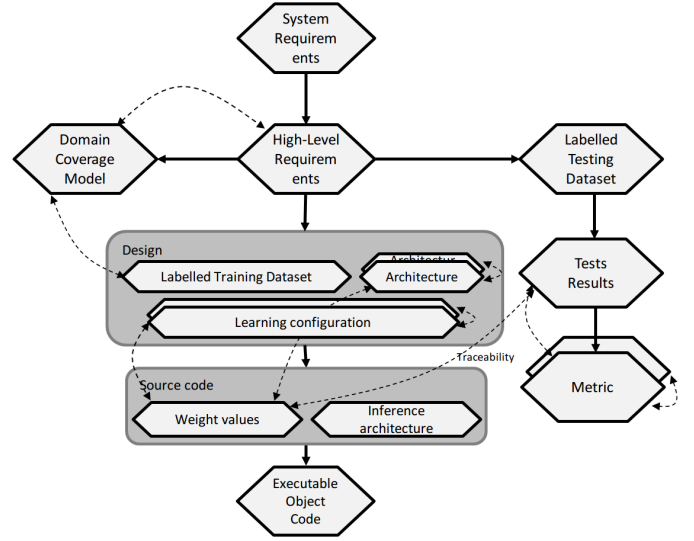

Fig. 7. Extension of Fig. 5 to integrate trial-and-error

chance, was extremely good. A way to circumvent this, is to impose some restrictions on the first delivered version, e.g., requesting that the first version shall belong to a catalogue of authorized "primitive" DNNs. A developer cannot then just deliver immediately a complex DNN without tracing it to a previous primitive one. Primitive DNNs can be defined in various ways and the definition impacts differently various artifacts: this goes beyond the present paper but shall be investigated.

Imposing a primitive catalogue is still not enough: imagine that an engineer developed a specific DNN classically (i.e., without following our recommendation of tracing the trial-and-error activities). Then, instead of going through the tedious work of analyzing the chain of increments which led to their final DNN until they reach their original "simple" DNN, they can just hide all the versions between the first and the last. In such a case the last version displays as their first improvement, which allows them to claim, that, by chance, their "first" improvement was the good one. Of course, this goes completely against the intent of our approach. To circumvent this, one should also restrict possible increments, or at least the justification for one increment. A naïve solution could be to have increments like adding only one layer at a time, having a default size for layers, etc. This might however be too restrictive in practice: some DNNs only show their benefits after having added a certain number of layers, but all the smaller versions with less layers are all equally bad. Investigating such restrictions in detail goes beyond the present paper.

VI. FUTURE WORK

This paper is, to our knowledge, the first to provide a precise list of traces which could potentially be written down for DNN. However, it does not address various development practices, which are encountered in real developments.

Gap between trained and inference DNN. The process highlighted in Section IV assumes more or less implicitly that

the interface of the trained DNN is the same as the one of the inference DNN. This assumption is often met (e.g., when using dropout the output type of the trained and inference DNN is the same) but not always: e.g., one might, even in a supervised context, train a sub-part of the final network in an unsupervised manner, for instance to learn valuable features (e.g., latent space of an auto-encoder [37]).

One might also train a DNN on a separate dataset or take a DNN already trained on another dataset (e.g., ImageNET for object detection [38]) then remove the latest layers (the most task-specific ones) to only adapt the DNN to the targeted functionality. In such cases, lots of intermediate steps are actually not immediately connected to the final task and therefore not traceable in the sense considered so far. We do not consider this sort of cases in this paper but insist on how essential they are: they reflect a reality of DL engineers which cannot be ignored.

Dataset. Another important aspect that has been ignored in this paper, is the evolution of the dataset: we assumed that the dataset (or more precisely, the datasets: training, validation, testing) is fixed. As mentioned, this is common practice when considering traceability: we are normally only interested in the final artifacts (except in our case, exceptionally, for trial-and-error activities). However, in reality, many systems actually gather new data along their lifetime. Therefore, one may not ignore the fact that data evolves permanently all along the life cycle of the autonomous system. In such cases, one should consider a form of incremental traceability, i.e., how to trace new data as it comes along. One should especially probably trace differently training data from testing data. In particular, one might need to argue why adding a new datum indeed provides additional valuable information. To do so, a possibility is to develop dataset coverage models. Depending on the context, one might need to trace the dataset itself to the sources used to generate it since they influence the dataset a lot and therefore the training: sensors calibration setup, sensor driver versions, etc.

Explainable AI. As mentioned from the beginning, we try in this paper to be independent of current approaches in the domain of explainable AI. We try in particular to be more pragmatic than academic. However, it is probably valuable to look more precisely into various approaches of explainable AI (see, e.g., [4] for a review) to discover new opportunities for relevant fine-granular traces.

Classical AI. Various approaches attempt to mix deep learning with expert knowledge, e.g., by transferring existing expert knowledge to a neural network (e.g., transfer learning [39]) where the expert knowledge can be expressed through rules or other forms; or by intertwining machine learning with probabilistic modelling [40]. All these approaches are valuable from the point of view of AI research, but they are also very promising for safety-critical systems because they allow to control the machine learning process to some extent and therefore to argue better that the final behavior is indeed satisfying. In some sense, one can interpret this as a form of explainability-by-design. It would therefore be very valuable to consider how to trace these methods, in particular the newly induced artifacts (e.g., generative model in the case of probabilistic programming).

Intellectual property. In domains like automotive or avionics, the development of the system is extremely distributed among various stakeholders: OEMs, tier 1, tier 2, or even tier 3 suppliers. In such cases, it is essential to deliver sufficient artifacts to guarantee safety, but it is also essential that every stakeholder keep their own intellectual property. This can be problematic for our approach to trial-and-error activities which forces practitioners to provide artifact evolutions which might reveal their production secrets. Similar problems exist for virtual validation and can been solved with approaches like the FMI standard [41]. It should in any case be investigated for the approach presented in this paper.

VII. Conclusion

In this paper, we addressed the traceability of neural network in a pragmatic manner: we first explicitly identified the challenge of tracing DNNs, then analyzed the parallels and differences between DNNs and classical software development, and proposed accordingly adaptations of the notion of trace for DNNs. Instead of blindly mapping classical software activities to DL activities, which would lead to mismatches with the actual practice of DL, we tried to embrace some of the specificities of "real-life" DL, in particular trial-and-error. We provided a solution (or the beginning thereof), which we believe supports both the rationale of traceability, while still being applicable for practitioners. The applicability might be controlled depending on the targeted safety level, as is classically done in safety-related software standards: for instance, one could require different coverage percentages for the domain coverage model whether the function is ASIL A, B, C, or D.

Appendix

A. Traceability of the dataset types

As mentioned in Section V-A, we go in this section more in detail about the traceability of dataset to interface requirements: the dataset being basically a set of examples, it should match the types of the inputs/outputs and therefore be traced to the interface requirement. Concretely, this means tracing 1. the raw dataset, and 2. the labels. Both should be traced to the interface requirements: the raw dataset to the input part of it, the labels to its output part.

For instance, if the interface requirement states that the input shall be images of dimension 640×480, then the raw dataset shall contain only such images, and shall therefore be traced to this input requirement. In case pre-processing is required, then there might not be a direct match, in which case the pre-processing function shall be mapped to the interface

requirement. Various approaches might then be employed: the dataset itself might be traced to the post-processing function directly, or one might introduce new requirements (called *derived* in the DO178-C) defining the interface of the post-processing function and then trace the dataset to this requirement. Or one might simply consider that the interface is a design decision, not to be traced (in DO178-C terminology: the interface definition would be part of the software architecture).

In a dual manner, suppose the interface requirement specifies that the *output* type is "list of 4-tuples" – representing bounding boxes. Then every label is a list of bounding boxes. Like previously, the dataset can therefore be traced to this type definition. However, if the structure of the output type is more complex (typically, if it contains sum types, i.e., enumerations), then traces can be defined *per datum* instead. Suppose for instance, that the interface requirement (say "REQ_123") specifies the following output instead:

1) output shall be a list of pairs,
2) where the first element is a 4-tuple like previously,
3) but the second element is a record containing the following fields:
   a) "pedestrian",
   b) "bike",
   c) "vehicle",
4) and where each of the fields contain a real between 0 and 1,
5) such that the sum of all field numbers is 1.

In such cases, the dataset can be traced as a whole to REQ_123-1 and REQ_123-2 since those parts of the type apply to every datum uniformly (more or less like before). On the other hand, for a given image, each label can be traced to REQ_123-3a, REQ_123-3b or REQ_123-3c: for instance, if an image is labeled as containing one pedestrian, and the label "pedestrian" shall be traced to REQ_123-3a. In such cases, we can trace every datum independently.[11]

Conversely, if one element of the dataset also identifies "trucks", then this label is not traceable to the requirement, which denotes a potential addition of unintended functionality. Note that there might be reasons why wanting to have data with labels not supporting the requirements: e.g., reuse of some data used in another context, use of the same data for another function, or desire to label *more* "just in case". Depending on the developed system, such cases shall probably not be forbidden, but their presence might give a hint about potential unintended functionality, which should then probably be addressed. For instance, depending on the case, the dataset should be preprocessed: the unwanted label should be erased or merged into another label, or maybe even gives hint that the requirement itself is not complete. Our main point is that the lack of traceability provides a hint about potential design decisions.

[11] The same actually applies for the raw dataset, in case the input datatype is a sum type. It just seems less common to have such types for inputs than for labels. But of course if such a case was to happen, then every datum should also be traced to the relevant sub-type of the input.

REFERENCES

[1] DO-178C, "RTCA DO-178C: Software Considerations in Airborne Systems and Equipment Certification," Tech. Rep., 2011.
[2] ISO, "Road vehicles – Functional safety," 2011.
[3] R. Welge, H. Putzer, J. Laurila-Drsch, and S. Heusinger, "Entwurfsmethodiken fr systeme mit komponenten der knstlichen intelligenz," DIN Mitteilung, to appear.
[4] C.-H. Cheng, G. Nhrenberg, C.-H. Huang, H. Ruess, and H. Yasuoka, "Towards dependability metrics for neural networks," arXiv preprint arXiv:1806.02338, 2018.
[5] C.-H. Cheng, F. Diehl, Y. Hamza, G. Hinz, G. Nhrenberg, M. Rickert, H. Ruess, and M. Troung-Le, "Neural networks for safety-critical applications - challenges, experiments and perspectives," arXiv:1709.00911, 2017.
[6] B. Morgan, D. F. Bedford, G. Morgan, and J. Austin, "Requirements for a standard certifying the use of artificial neural networks in safety critical applications," in *Proceedings of the International Conference on Artificial Neural Networks*, 1996.
[7] D. M. Rodvold, "A software development process model for artificial neural networks in critical applications," in *IJCNN'99. International Joint Conference on Neural Networks. Proceedings (Cat. No.99CH36339)*, vol. 5, July 1999, pp. 3317–3322 vol.5.
[8] A. Senyard, E. Kazmierczak, and L. Sterling, "Software engineering methods for neural networks," in *Tenth Asia-Pacific Software Engineering Conference, 2003.*, Dec 2003, pp. 468–477.
[9] R. Salay, R. Queiroz, and K. Czarnecki, "An analysis of ISO 26262: Using machine learning safely in automotive software," *CoRR*, vol. abs/1709.02435, 2017. [Online]. Available: http://arxiv.org/abs/1709.02435
[10] Z. Kurd and T. P. Kelly, "Using safety critical artificial neural networks in gas turbine aero-engine control," in *Computer Safety, Reliability, and Security, 24th International Conference, SAFECOMP 2005, Fredrikstad, Norway, September 28-30, 2005, Proceedings*, ser. Lecture Notes in Computer Science, R. Winther, B. A. Gran, and G. Dahll, Eds., vol. 3688. Springer, 2005, pp. 136–150. [Online]. Available: https://doi.org/10.1007/11563228_11
[11] S. Burton, L. Gauerhof, and C. Heinzemann, "Making the case for safety of machine learning in highly automated driving," in *Computer Safety, Reliability, and Security*, S. Tonetta, E. Schoitsch, and F. Bitsch, Eds. Cham: Springer International Publishing, 2017, pp. 5–16.
[12] L. Gauerhof, P. Munk, and S. Burton, "Structuring validation targets of a machine learning function applied to automated driving," in *Computer Safety, Reliability, and Security*, B. Gallina, E. Schoitsch, A. Skavhaug, and F. Bitsch, Eds. Springer International Publishing, 2018, to appear.
[13] M. Zinkevich, "Rules of ml," https://developers.google.com/machine-learning/rules-of-ml/, accessed: 2018-09-26.
[14] S. Ivanov, "37 reasons why your neural network is not working," https://blog.slavv.com/37-reasons-why-your-neural-network-is-not-working-4020854bd607, accessed: 2018-09-26.
[15] D. Sculley, G. Holt, D. Golovin, E. Davydov, T. Phillips, D. Ebner, V. Chaudhary, M. Young, J. Crespo, and D. Dennison, "Hidden technical debt in machine learning systems," in *Advances in Neural Information Processing Systems 28: Annual Conference on Neural Information Processing Systems 2015, December 7-12, 2015, Montreal, Quebec, Canada*, C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, Eds., 2015, pp. 2503–2511. [Online]. Available: http://papers.nips.cc/paper/5656-hidden-technical-debt-in-machine-learning-systems
[16] C. Olah, A. Satyanarayan, I. Johnson, S. Carter, L. Schubert, K. Ye, and A. Mordvintsev, "The building blocks of interpretability," *Distill*, 2018, https://distill.pub/2018/building-blocks.
[17] F. Doshi-Velez and B. Kim, "Towards a rigorous science of interpretable machine learning," 2017.
[18] D. Amodei, C. Olah, J. Steinhardt, P. Christiano, J. Schulman, and D. Man, "Concrete problems in ai safety," *arXiv preprint arXiv:1606.06565*, 2016. [Online]. Available: https://arxiv.org/pdf/1606.06565.pdf
[19] Wikipedia, "Requirements traceability — Wikipedia, the free encyclopedia," http://en.wikipedia.org/w/index.php?title=Requirements%20traceability&oldid=861938242, 2018, [Online; accessed 15-November-2018].
[20] "Functional requirements for integrated vehicle-based safety systems (ivbss) - light vehicle platform," 2008, prepared by The University of Michigan Transportation Research Institute, Visteon Corporation,

and Cognex Corporation for U.S. Department of Transportation – Cooperative Agreement DTNH22-05-H-01232.

[21] E. M. Clarke, T. A. Henzinger, and H. Veith, "Introduction to model checking," in *Handbook of Model Checking.*, E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem, Eds. Springer, 2018, pp. 1–26. [Online]. Available: https://doi.org/10.1007/978-3-319-10575-8_1

[22] A. Benveniste, B. Caillaud, A. Ferrari, L. Mangeruca, R. Passerone, and C. Sofronis, "Multiple viewpoint contract-based specification and design," in *Formal Methods for Components and Objects, 6th International Symposium, FMCO 2007, Amsterdam, The Netherlands, October 24-26, 2007, Revised Lectures*, ser. Lecture Notes in Computer Science, F. S. de Boer, M. M. Bonsangue, S. Graf, and W. P. de Roever, Eds., vol. 5382. Springer, 2007, pp. 200–225. [Online]. Available: https://doi.org/10.1007/978-3-540-92188-2_9

[23] DO-333, "RTCA DO-333: Formal Methods Supplement to DO-178C and DO-278A," Tech. Rep., 2011.

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

[25] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, "Automatic differentiation in pytorch," in *NIPS-W*, 2017.

[26] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," *CoRR*, vol. abs/1704.04861, 2017. [Online]. Available: https://arxiv.org/abs/1704.04861

[27] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," in *Proceedings of the IEEE*, 1998, pp. 2278–2324.

[28] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. E. Reed, C. Fu, and A. C. Berg, "SSD: single shot multibox detector," *CoRR*, vol. abs/1512.02325, 2015. [Online]. Available: http://arxiv.org/abs/1512.02325

[29] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," *J. Mach. Learn. Res.*, vol. 15, no. 1, pp. 1929–1958, Jan. 2014. [Online]. Available: http://dl.acm.org/citation.cfm?id=2627435.2670313

[30] Klaus Greff, Aaron Klein, Martin Chovanec, Frank Hutter, and Jürgen Schmidhuber, "The Sacred Infrastructure for Computational Research," in *Proceedings of the 16th Python in Science Conference*, Katy Huff, David Lippa, Dillon Niederhut, and M. Pacer, Eds., 2017, pp. 49 – 56.

[31] Y. Mordecai and D. Dori, "Model-based requirements engineering: Architecting for system requirements with stakeholders in mind," in *2017 IEEE International Systems Engineering Symposium (ISSE)*, Oct 2017, pp. 1–8.

[32] Wikipedia, "Model-based testing — Wikipedia, the free encyclopedia," http://en.wikipedia.org/w/index.php?title=Model-based%20testing&oldid=866247302, 2018, [Online; accessed 23-November-2018].

[33] S. Nakajima and H. N. Bui, "Dataset coverage for testing machine learning computer programs," in *2016 23rd Asia-Pacific Software Engineering Conference (APSEC)*, Dec 2016, pp. 297–304.

[34] DO-330, "RTCA DO-330: Software Tool Qualification and Considerations," Tech. Rep., 2011.

[35] C. Kozyrkov, "Explainable ai won't deliver. here's why." https://hackernoon.com/explainable-ai-wont-deliver-here-s-why-6738f54216be, 2018, [Online; accessed 5-December-2018].

[36] Z. C. Lipton and J. Steinhardt, "Troubling trends in machine learning scholarship," 2018.

[37] P. Baldi, "Autoencoders, unsupervised learning and deep architectures," in *Proceedings of the 2011 International Conference on Unsupervised and Transfer Learning Workshop - Volume 27*, ser. UTLW'11. JMLR.org, 2011, pp. 37–50. [Online]. Available: http://dl.acm.org/citation.cfm?id=3045796.3045801

[38] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "ImageNet: A Large-Scale Hierarchical Image Database," in *CVPR09*, 2009.

[39] S. J. Pan and Q. Yang, "A survey on transfer learning," *IEEE Trans. on Knowl. and Data Eng.*, vol. 22, no. 10, pp. 1345–1359, Oct. 2010. [Online]. Available: http://dx.doi.org/10.1109/TKDE.2009.191

[40] N. D. Goodman, V. K. Mansinghka, D. M. Roy, K. Bonawitz, and J. B. Tenenbaum, "Church: a language for generative models," in *UAI 2008, Proceedings of the 24th Conference in Uncertainty in Artificial Intelligence, Helsinki, Finland, July 9-12, 2008*, D. A. McAllester and P. Myllymäki, Eds. AUAI Press, 2008, pp. 220–229. [Online]. Available: https://dslpitt.org/uai/displayArticleDetails.jsp?mmnu=1&smnu=2&article_id=1346&proceeding_id=24

[41] T. Blochwitz, M. Otter, M. Arnold, C. Bausch, H. Elmqvist, A. Junghanns, J. Mauss, M. Monteiro, T. Neidhold, D. Neumerkel, H. Olsson, J. V. Peetz, S. Wolf, and C. Clau, "The functional mockup interface for tool independent exchange of simulation models," in *Proceedings of the 8th International Modelica Conference*, 2011, pp. 105–114. [Online]. Available: http://dx.doi.org/10.3384/ecp11063105