Philosophy & Technology
https://doi.org/10.1007/s13347-023-00642-7

RESEARCH ARTICLE

From Coding To Curing. Functions, Implementations, and Correctness in Deep Learning

Nicola Angius[1] · **Alessio Plebe**[1]

Received: 23 January 2023 / Accepted: 4 May 2023

Abstract

This paper sheds light on the shift that is taking place from the practice of 'coding', namely developing programs as conventional in the software community, to the practice of 'curing', an activity that has emerged in the last few years in Deep Learning (DL) and that amounts to curing the data regime to which a DL model is exposed during training. Initially, the curing paradigm is illustrated by means of a study-case on autonomous vehicles. Subsequently, the shift from coding to curing is analysed taking into consideration the epistemological notions, central in the philosophy of computer science, of function, implementation, and correctness. First, it is illustrated how, in the curing paradigm, the functions performed by the trained model depend much more on dataset curation rather than on the model algorithms which, in contrast with the coding paradigm, do not comply with requested specifications. Second, it is highlighted how DL models cannot be considered implementations according to any of the available definitions of implementation that follow an intentional theory of functions. Finally, it is argued that DL models cannot be evaluated in terms of their correctness but rather in their experimental computational validity.

Keywords Philosophy of computer science · Deep learning · Specification · Implementation · Correctness · Autonomous vehicles

Abbreviations

DL	Deep Learning
LoA	Level of abstraction
AV	Autonomous vehicle
XAI	Explainable artificial intelligence

✉ Nicola Angius
nicola.angius@unime.it

Alessio Plebe
alessio.plebe@unime.it

[1] Department of Cognitive Science, University of Messina, Messina, Italy

Published online: 05 July 2023

1 Introduction

Let us loosely use the term 'coding' for the activity of developing and writing programs as conventional in the software community. As such, the practice of coding has an intimate relationship with the desired goals of the program and its expected functions. This paper contrasts coding with 'curing', a different activity that has emerged in the last few years as a result of the increasingly widespread use of Deep Learning (DL) in the computing world. More precisely, the object of curation is the data regime to which a DL model is exposed during training. The functions performed by the trained model depend much more on its training dataset than on the lines of code that implement the model algorithms themselves. The code of a neural model certainly depends on a large number of architectural parameters, as well as parameters on the training mode. However, these are often parameters that aim to optimize the model performance and do not significantly affect the final function of the model, to a degree comparable to the influence of the training dataset. The shift taking place from the practice of coding to the practice of curing deserves attention from the philosophy of computer science (Angius et al., 2021), which developed as an epistemological and methodological analysis of computing in its coding paradigm. This work aims to begin to fill this gap, by specifically focusing on the notions of function, implementation, and correctness in the curing paradigm.

In the traditional practice of programming there are well established philosophical analyses about the complex relations between the expected goals and functions of a software and its coding. Typically, functions of high-level language programs are expressed by *specifications* which articulate, often in formal terms, behavioural properties that the program to be encoded is required to realise. Specifications express *what* the requested properties are, without saying *how* they are to be fulfilled. The latter is the task of an *implementation*. Not all implementations are *correct* implementations, i.e. implementations that fully comply with the requested specifications. Implementations need to be *verified* against the advanced functional requirements in order to ascertain whether they are or are not correct. Specification, implementation, and correctness are three critical and central concepts in the epistemology of software development following the coding paradigm. This paper aims to show how curing activities require to revision these notions for DL systems.

DL is a direct descendant of artificial neural networks (Rumelhart & McClelland, 1986), an approach that late in the '80s attempted to loosely mimic the learning capabilities of the brain. The major difference of current DL is the use of more than three layers of neurons. It may seem like a minor technical detail, but the difficulties in training neural networks with more than three layers using backpropagation (de Villers & Barnard, 1992) have been a significant limitation to the potential of artificial neural networks for decades. The computation of the gradient of weights with respect to the errors requires using the chain rule with the need to multiply each layer weight and gradients together across all the layers, leading to vanishing gradients when there are more than three layers. The first solution to the vanishing gradients was achieved by Hinton & Salakhutdinov (2006), who were able to train models with four and five hidden layers combining pre-training with restricted Boltzmann Machines and

standard backpropagation. Subsequently, the evolution of standard backpropagation in terms of stochastic gradient descent and its various improvements (Hinton et al., 2011; Kingma & Ba, 2014) has made pre-training no longer necessary. The breakthrough of the three-layer barrier and the ease of training neural networks with many layers have made them very successful.

Nowadays DL technology underpins everyday products and services, such as digital assistants, as well as emerging technologies, for instance self-driving cars. Untrained DL models are intrinsically agnostic about the functions that they should perform and that will emerge only during the training phase. It is the kind of data fed during learning that fixes the function performed by the program. The current tendency in the use of DL relies on out-of-the-box architectures, and development is more and more shifted from coding to preparing the dataset. Moreover, the role of dataset curation has grown to the point of becoming a discipline in its own right. Progresses achieved just by dataset curation encompass applications as diverse as tuberculosis screening (Kim et al., 2020), self-driving cars (Sadat et al., 2021), and prediction of aqueous solubility of drugs (Meng et al., 2022).

It is possible to trace in coding and curing two long standing philosophical roots, roughly referable to rationalism and empiricism, both of which began with Alan Turing in the early days of computer science. The Turing (1936) machine establishes the most exemplary link between the coding of instructions and the behaviour of a program, offering support for a rationalist view. Turing's later *unorganized machine* (Turing, 1948) anticipated the empiricist approach in computing, where the final functions of a program do not depend much on its own code, but rather on a suitable organizing training regime.

Only now, many decades after Turing's unorganized machines, the breakthrough brought about by DL is prompting a radical shift in the practice of computer science. Functions are not expressed by specifications but are the outcome of the training process of a network given an opportunely curated dataset; DL models are not developed so as to comply with requested specifications, hence they are not, epistemologically speaking, implementations. Above all, without requested functions and implementing levels it is not possible to define correctness. At the same time, many DL systems are involved in the so-called *safety-critical* contexts, namely circumstances potentially dangerous for human lives; safety-critical DL systems include autonomous weapons, medical diagnosis systems, and autonomous vehicles.

This paper specifically focuses on the case of Autonomous Vehicles (AVs) to analyse the notions of function, implementation, and correctness in the shift from coding to curing. Section 2 provides an introductory analysis of the coding and curing paradigms, while Sect. 3 supplies a more in-depth examination of the curing approach from the perspective of AV by introducing a study-case. Section 4 compares the epistemological roles of specifications in traditional software development and learnt functions in DL by means of a toy example from formal development methods. Section 5 takes into consideration the main definitions of the notion of implementation available in the literature to show that none of them is satisfied by a DL model, program, or system. Finally, Sect. 6 argues that the notion of correctness does not apply to the case of DL models and identifies the concept of *experimental computational validity* from

Primiero (2020) as the one that better complies with the actual practice of evaluating the dependability of DL systems.

2 From coding to curing

Coding is here intended as the software development process defined by the different available development methods for computational systems: ideally, it is the practice of specifying, encoding, implementing, and verifying programs, usually involving many different Levels of Abstraction (LoAs) (Floridi, 2008), as in the following list from Primiero (2016, 2020)autoedited,[1]

- Intention
- Specification
- Algorithm
- High-level language program
- Assembly/machine code program
- Execution

The intention level specifies the general goals that the computational system to be developed is requested to achieve; the specification level articulates the upper level into a set of property specifications, that is, of sentences expressing functional properties that the system must comply with; the algorithm level establishes how those functions are realised by the system, in terms of procedures often expressed in some pseudocode language; algorithms are implemented into a high-level language program which, in its turn, is implemented by a compiler into a machine code program, finally executed by a physical computing machine (Primiero, 2016, 2020).

The listed LoAs constitute one main attempt in defining the ontology of software and induce important epistemological issues characterising the philosophy of computer science, including the relation between functions and specifications, the definition of implementation, and the notion of correctness examined, respectively, in Sects. 4, 5, and 6 below.

In the coding paradigm most of the development efforts are spent in programming activities, that is, in the act of realising functional specifications into high-level language programs and in revising those programs in case verification and testing processes show that they fail to satisfy their specification set, as exemplified in Fig. 1.

For curing there is not as precise a definition as for coding. The term curing can be loosely used when the object of curation are datasets that have the power to influence the functions performed by a neural network. Curing, construed in this sense, becomes a practice worthy of attention in that it implies refining or modifying the functions performed by a program without refining or modifying the program code.

[1] The LoAs here listed should not be taken to suggest a cascade software development model (Sommerville, 2021): from intention to execution, but rather to define a stratified ontology for computational systems. The different development methods, such as cascade, spiral, or agile, may go through some or all those levels. See on this Angius et al. (2021).

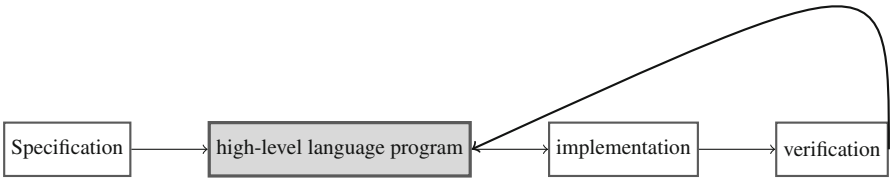

Fig. 1 The coding paradigm: the coloured rectangle is the area where most of the development takes place

Probably the first suggestion about a transition from coding to curing was provided precisely by the one who founded the science of programming: Alan Turing (1948). He envisioned a machine based on distributed interconnected elements, called *B-type unorganized machine*. The elements were simple NAND gates with two inputs, randomly interconnected; each NAND input could be connected or disconnected, and thus a learning method could 'organize' the machine by modifying the connections.

This idea, taken to its extreme, configures a computer program devoid of any rules or steps that have reference to the specifications of the function to be performed. The main task of the program consists in interpreting the educational instructions provided from the outside, and translating them into operations that organize the machine towards the goal contained in the education. The function to be realized would then be derived only from the external "education" regime to which the machine has been subjected. Of course, the account of education and interference in Turing's report remains entirely suggestive, and far from the specification in terms of training set, achieved after decades of painful research for an efficient equivalent of human learning for machines. Nevertheless, if an unorganized machine can theoretically reach an organized form for any purpose through learning, it is implicit that the effort shifts from programming the initial machine to arranging the specific educational regime for the desired task. Turing's idea of a machine capable of coming up with any function, simply by learning, was grounded in a strong commitment to empiricism concerning the human mind. In his own words: "We believe then that there are large parts of the brain, chiefly in the cortex, whose function is largely indeterminate. [...] All of this suggests that the cortex of the infant is an unorganised machine, which can be organised by suitable interfering training." (Turing, 1948, 16).

Curing deserves attention today because the current spread of AI is due to a direct descendant of Turing's unorganized machine, artificial neural networks, in their most advanced form better known as DL. Of course, Turing had not gone into the question of how to actually interfere on the connections of a distributed network so as to fix what the system was to learn from the examples, and accomplishing this proved tremendously difficult. A workable solution was found forty years after Turing, known as backpropagation (Rumelhart et al., 1986) since it takes its cue from the error of the network output and propagate corrections back towards the inner layers. Initially the artificial neural model for which backpropagation applies were feedforward network, made of units organized into distinct layers, with unidirectional connections between each layer and the next one, but it gradually extended to all variants of neural networks.

Generally, being $\vec{w}$ the vector of all learnable parameters in the network, and $\mathcal{L}(\vec{x}, \vec{w})$ a measure of the error of the network with parameters $\vec{w}$ when applied to

the sample $\overrightarrow{x}$, the backpropagation updates the parameters iteratively, according to the following formula:

$$\overrightarrow{w}_{t+1} = \overrightarrow{w}_t - \eta \nabla_w \mathcal{L}\left(\overrightarrow{x}_t, \overrightarrow{w}_t\right) \tag{1}$$

where t spans all available samples $\overrightarrow{x}_t$, and η is the learning rate. Backpropagation was a decisive breakthrough in the '80 s, but neural networks powered by backpropagation are nowhere near the ideal unorganized machine that learns any function without coding. There are families of different neural architectures that better fit diverse applications, and there are architectural choices to be made in order to best adapt a network to its function, beside curing a training set. Still, a significant part of the code development in artificial neural networks was targeted for generic learning machines, more than for specific applications.

At the beginning of the century, neural networks appeared to have exhausted their potential, a source of limitation was that backpropagation worked well for feedforward neural networks with three layers, but became critical when using four layer or more (de Villers & Barnard, 1992).

The recent shift from "shallow" to "deep" neural networks derives from several strategies for training networks with a large number of layers (Hinton & Salakhutdinov, 2006; Bengio et al., 2007; Hinton et al., 2011; Krizhevsky et al., 2012). At present, almost all learning strategies for deep neural models are related to the principle of stochastic gradient descent; its basic formulation is the following:

$$\overrightarrow{w}_{t+1} = \overrightarrow{w}_t - \eta \nabla_w \frac{1}{M} \sum_{i}^{M} \mathcal{L}\left(\overrightarrow{x}_i, \overrightarrow{w}_t\right) \tag{2}$$

and is not much different from standard backpropagation. Instead of computing the gradients over a single sample t, in equation (2) a stochastic estimation is made over a random subset of size M of the entire dataset, and at each iteration step t a different subset, with the same size, is sampled. Although the refined backpropagation, and all its recent variants (Duchi et al., 2011; Kingma & Ba, 2014), are not radical innovations from a mathematical point of view, they have had tremendous and unforeseen success (Plebe & Perconti, 2022).

This success, just as it had happened in the '80 s, triggered an impetus for the coding of progressively more sophisticated models. This effort has since resulted in the construction of a number of DL frameworks, such as TensorFlow (Abadi et al., 2015) and PyTorch (Ketkar, 2017). Gradually, the endeavour of developing DL based applications is shifting from coding new software to curing datasets for training existing, consolidated, software.

3 A study case on autonomous vehicles

The field of Autonomous Vehicles (AVs) is one in which DL is achieving an increasing success. The idea of an autonomous vehicle is way ahead of DL: General Motors

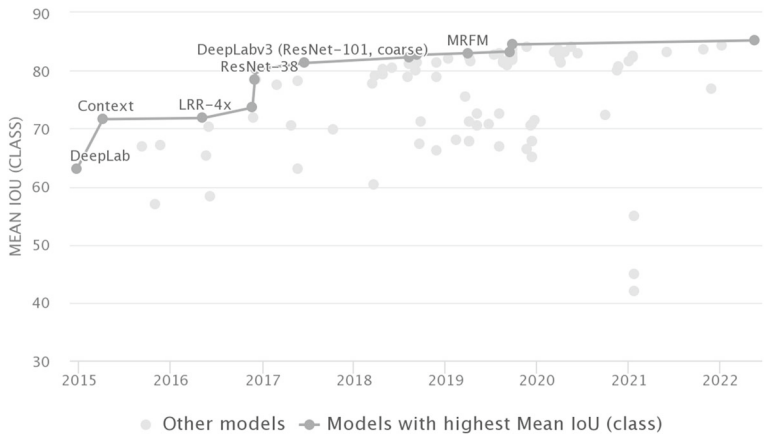

Fig. 2 Performance results when using the Cityscapes dataset. The effort in coding is reflected in ever more marginal improvements in performance. Source: https://paperswithcode.com/sota/semantic-segmentation-on-cityscapes

released its first prototype self-driving vehicle in 1958 (Ackerman, 2016). Since then, there have been competitions between quasi-autonomous vehicles built by research teams related to the automotive industry around the world, but soon the goal proved to be far more arduous than initially anticipated. Before DL, the perception of the road environment was the most serious hindrance to the development of AVs. Perception is the task enjoying the greatest leap forward because of DL, making the goal of autonomous driving realistic (Kuutti et al., 2019; Janai et al., 2020), with actual examples of full autonomy, even if not yet available for the market.

Collecting data for training and testing AV models is quite an easy task: data is usually collected by means of cameras, radars, ultrasonic sensors, and other devices including LiDAR (laser imaging detection and ranging), maps, and GPS.[2] Some vehicles, such as Tesla electric cars, allow the owner to send her recorded data to the parent company.

However, raw data are useless for AV models until they are labelled: the involved DL models need to know that, for instance, a certain object is a vehicle or a pedestrian in order to be trained to plan safe motion. At the same time, labelling everything is a practically unachievable task: typical datasets for training, validating, and testing AV models include on average 140/150 h of recorded videos wherein each frame may contain up to 50 objects. Accordingly, a problem arises as to what to label, given the goal of selecting the most realistic drive scenarios. This is a common data curation task.

If manually labelling what is deemed to be interesting is time consuming and highly expensive, recent approaches, such as *active learning* (Settles, 2012), allow a learning

[2] Andrej Karpathy, former director of AI at Tesla, recently revealed that much of the self-driving dataset for Tesla cars is being collected using cameras only (https://www.youtube.com/watch?v=g6bOwQdCJrc).

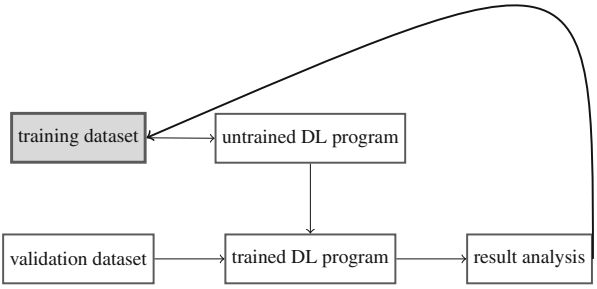

Fig. 3 The curing paradigm: the coloured rectangle is the area where most of the development takes place

model to select which data are significant for the task of interest and to ask an oracle (usually human) to provide labels for those data. Active learning has been mainly applied to image classification tasks (Joshi et al., 2009; Li & Guo, 2013) but is hardly applicable to AVs with equal success: in active learning different datasets are selected for diverse tasks performed by distinct models. By contrast, in AVs various tasks have to be performed by the same model, including world perception, anticipation of behaviours of traffic participants, or safe motion planning.

Current trends in DL modelling for AVs rather aim at using the very same model but curing differently the dataset, allowing the model to perform the various desired tasks. Indeed, by looking at results obtained through the years by using, for instance, the Cityscape dataset[3] (Cordts et al., 2016), as they are depicted in Fig. 2, one notices that in the last couple of years the effort in encoding different DL models (bullets) does not correspond in significant performance improvements (values in the ordinate axis). Major efforts are being spent in curing differently the dataset instead.

Andrey Karpathy from Tesla revealed, at the 2021 Workshop on Autonomous Driving CVPR, that in the development cycle given by the process *data collection - labelling - training - deploy,* DL models remain almost unchanged in the training phase, while most of the work at each cycle is devoted to cure data at the labelling stage.[4] Such a shift in the development efforts from coding to curing activities is depicted in Fig. 3.

While in the traditional software development processes unsatisfactory verification results lead to program revision, and most of the development activities are eminently coding activities, in current approaches to DL modelling a result analysis calling for revisal brings to dataset curation. Accordingly, functions performed by the AV model depend much more on the dataset curation than on the actual program implementing the model.

Sadat et al. (2021) propose a dataset curation method for AVs DL modelling based on what they call *complexity measures*. Complexity measures are not thought for a specific dataset or a particular DL model; they are a collection of metrics aimed at

[3] The Cityscape dataset contains video recordings from 50 cities with 25.000 labelled frames, concerning street scenes in distinct seasons. See https://www.cityscapes-dataset.com/. The graph in Fig. 2 is taken from https://paperswithcode.com/sota/semantic-segmentation-on-cityscapes.

[4] Karpathy's keynote talk at the 2021 IEEE / CVF Computer Vision and Pattern Recognition Conference (CVPR) is available at https://karpathy.ai/; unfortunately, no published paper or proceedings is associated with the talk.

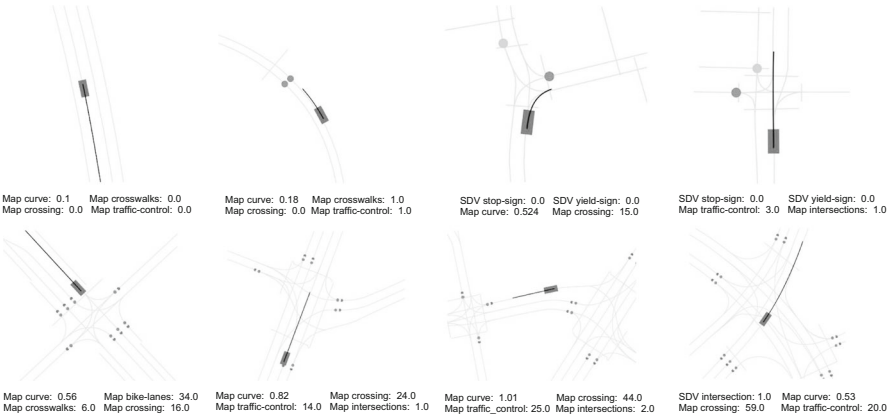

Fig. 4 Infrastructure-related complexity measures from Sadat et al. (2021), showing variations in the curvature and intersections of paths

differentiating meaningful driving scenarios in a given dataset allowing the model to perform new desired functions, thereby improving the performance of the model. This is accomplished by selecting which frames from a video recording are more interesting to label. The experiments conducted by Sadat et al. (2021) achieved, on average, a 6, 5% improvement in perception tasks and 6% improvement in motion forecast with respect to active learning and by using different DL models.

One first important choice is that of labelling a set of frames instead of single frames; this gives the opportunity of considering also *temporal properties*, that is, properties describing how the environment evolves over time, such as predicting a lane-change of the vehicle in front of the ego-vehicle. In particular, the proposed data curation method focuses on snippets of 25 s, containing as many as 250 frames, each snippet being called a scenario. The authors propose a procedure to solve the optimization problem of selecting the set of most interesting scenarios which be as complete and diverse as possible given a set of desired tasks. The general tasks here considered are perception, motion planning, and prediction of the motion of other objects.[5]

Complexity measures are of three kinds: infrastructure-related, traffic participants, and ego-car manoeuvrers. Infrastructure-related complexity measures include: topology of the driving paths, intersections and traffic lights, bike-lanes and cross-walks, and the height variations of the drivable area. As exemplified in Fig. 4, each snippet is provided with measures for each complexity aspect relevant for that scenario. Traffic participants measures involve a quantification of crowdedness, the diversity of actors, the diversity of paths and velocity. Finally, metrics are provided to measure the AV manoeuvres, in particular, path and speed of the ego-car, its interactions with other traffic participants, and specific moves such as passing another vehicle, dodging an object on the driving lane, or making u-turns.

[5] The algorithm proceeds by first ranking snippets given a set of complexity measures and a task, then by selecting the most interesting snippets (with high ranking scores), and finally by ensuring diversity of the snippet set, quantifying diversity as a difference in their complexity measures.

The experiments conducted by Sadat et al. (2021) were carried out using a 140 h recording dataset, divided in training, validation, and test sets; the dataset was cured differently, namely using complexity measures, active learning, and random sampling. Experiments were divided for detection tasks and motion prediction tasks; different AV DL modes were used for each of the two major tasks. DL models outperformed when using the dataset cured using complexity measures both in detection and prediction tasks, independently of the chosen model. In particular, models outperformed in the detection of traffic participants, mainly pedestrians and vehicles, in the prediction of the motion of those actors, as well as in the prediction of the collisions of their routes.

Given a general task, like detection or motion-forecast, it is the labelling of the dataset that ensures specific functions to be performed by the model. It is the choice of specific complexity measures that allows new functions to be performed, in this way outperforming the same model fed with a differently curated dataset. Measuring, in the dataset, the presence of bikes, the geometry of the bike lanes, and the route of traffic participants, may allow a motion-forecast model to satisfy the new function 'dodge a bicycle which suddenly moves out from the bicycle lane into the drive lane'. It is claimed here that such a function is new in that the model is not able to satisfy it when the training dataset is not labelled and cured in the proper way. Other new functions for AV models may be 'predict a vehicle speed decrease as a consequence of a drivable-area height variation', satisfied when the dataset is cured considering the drivable area height variations; or 'detect a right-side object as a vehicle that pulled-over instead than as a building', which may potentially move back in the drivable area, when crowdedness, diversity of actors, and diversity of paths and velocity are appropriately measured.

The fact that new functions emerge from the DL model as a consequence of dataset curation rather then being specified and correctly implemented by a program, as it is the case for the coding paradigm of Fig. 1, calls for an epistemological examination of the notions of function and specification, that the next section now turns to analyse.

4 Functions and specifications

AVs are a paradigmatic example of safety critical systems, that is, of systems which may endanger human or ecosystemic safety. Other common examples of safety critical systems are the control software of nuclear power plants, rockets, and traffic lights, software involved in robotic surgery, or in autonomous weapons. Formal development methods are usually preferred for the design, implementation, and verification of software operating in safety critical situations, in that formally specifying and verifying programs is assumed to assure a higher level of correctness than agile development methods and software testing (Baier & Katoen, 2008). Another purported reason is that many safety critical systems cannot be empirically tested, testing being either too expensive, as it would be for a rocket launch, or unsecure for safety reasons, as in the case of robotic surgery. Formal proofs of correctness, the argument goes, should be rather preferred.

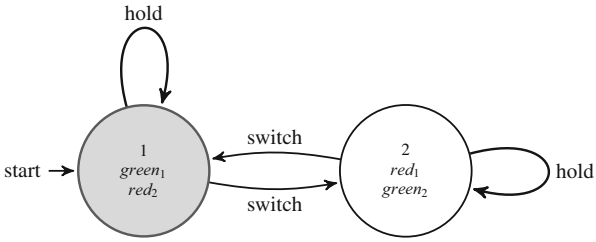

Fig. 5 A transition system for a traffic light system. State 1 is both a starting state (indicated by an incoming arrow) and final state (depicted as a coloured state)

For the sake of clarity, let us examine a driving-related toy example of a safety critical system that is developed according to the coding paradigm. Consider a traffic light system composed by a couple of traffic lights at a crossroad; each traffic light can assume either state red or state $green$. The system can be modelled as the transition system of Fig. 5.[6]

One safety property, call it S, one would like this system to satisfy may be "the two traffic lights will never be in state $green$ at the same time". This is to be considered a safety property in that it is intended to avoid a situation which may be dangerous for human security. More in general, a safety property is a property that states that something deemed "bad" will never occur (Alpern & Schneider, 1987). By contrast, requiring that something "good" will happen infinitely often is tantamount to formulate a *liveness* property (Alpern & Schneider, 1985). A liveness property, call it L, for the traffic light system may be "each traffic light must enter the $green$ state infinitely often". This will ensure the desirable situation in which vehicles in either direction will be enabled to pass through the crossroad infinitely often.

As stated in Sect. 1, a computational system is developed, according to the coding paradigm, by firstly advancing a set of property specifications, that is, of properties specifying the desired functions the system to be developed must realise. In formal development methods, specifications are formalised using many different formalisms, including Typed Predicate Logic (TPL) (Turner, 2009), Z (Spivey, 1988), VDM (Jones, 1990), or temporal logics (Kröger & Merz, 2008). As in the case of AVs, also in the case of the traffic light system it is essential being able to specify temporal properties, that is, properties that hold true during given time intervals. Temporal properties are usually formalised with *temporal logics*. Safety property S can be formalised as $\mathbf{G}(green_1 \veebar green_2)$, specifying that globally ($\mathbf{G}$), i.e. for every state of the system, it holds that either traffic light 1 is in state $green$ or traffic light 2 is in state $green$. And liveness property L can be formalised as $\mathbf{FG}(green_1) \wedge \mathbf{FG}(green_2)$, requiring that the system

[6] Recall that a finite transition system $TS = (S, A, T, I, F, AP, L)$ is a set-theoretic structure defined by a finite set of states $S = \{s_0, \ldots, s_n\}$, a finite set of labels for transitions A, a transition relation $T \subseteq S \times A \times S$, each transition having form $s_i \xrightarrow{\alpha} s_j$, a set of initial states $I \subseteq S$, a set of final states $F \subseteq S$, a finite set of labels for states AP, and a state labelling function $L : S \rightarrow 2^{AP}$. The example in Fig. 5 is inspired by a similar one in Baier and Katoen (2008, 100-102).

will finally (**F**) enter a state from which every reachable state (**G**) is a state wherein traffic light 1 is green; the same holds for traffic light 2.[7]

Given a set of property specifications defining the functions of the simple traffic light system, including properties S and L, a traffic light system is developed so as to comply with the desired specification set. One question addressed by the philosophy of computer science concerns the difference holding between specifications and computational models. In the end, the transition system in Fig. 5 conveys the same information provided, among others, by formulas S and L. Turner (2011) argues that the difference between the two lies in their *intentional stance*: computational models are aimed at *describing* some target computational system, its functions and behaviours; by contrast, specifications are *normative*, i.e. they are aimed at *prescribing* those functions and behaviours. So, even though specifications and models may not differ as what concerns their *informational content*, they differ about their "*direction of governance*" (Turner, 2020).

This is to say that in case a model turns out not to be a faithful description of a computational system, for instance when the latter does not display functions and behaviours represented in the model, it is the model that is refined. Here the "direction of governance" is from the system to the model. In case the system does not implement the functions prescribed by a specification, or realises function not requested again by the specification, the system is said to *malfunction* and it needs to be fixed,[8] "Direction of governance" is in this case from the specifications to the system.

The relation between models and functions highlights one first fundamental difference between the coding and the curing paradigms. According to the former, desired functions for a computational system to be developed are put forward and formally expressed by means of a set of specifications. Once a piece of software has been encoded and implemented, it is evaluated whether it complies with the desired specification set. Here it comes the need of representational models, which are used both in formal verification methods, such as in *model checking* (Baier & Katoen, 2008), and in empirical *testing* (Ammann & Offutt, 2016). Models are used to reason about the represented computational systems by identifying which computations in the model satisfy or violate a given specification (Angius, 2013). For instance, it is easy to see that the model in Fig. 5 satisfies safety property S and liveness property L.

In the curing paradigm, functions are not fixed by any specification and computational systems are not built so as to comply with those specifications. In other words, *the is no direction of governance*. This is true in either directions: there are no specifications prescribing functions for AV systems, and the involved DL models are not taken to describe a target system and its functions. A DL model before training does not satisfy any function; again, functions will be learnt in the training phase. Not only, as the AV study case exemplifies, in case a system does not behave satisfactorily, the

[7] Clearly, that both traffic lights enter the *green* state infinitely often together is avoided by requiring safety property S.

[8] Distinct cases should be made for the two different forms of malfunction: *dysfunction* when the system does not display required behaviours, and *mysfunction*, when the system displays behaviours not requested by its specifications (Floridi et al., 2015).

model is not fixed, as it would require a coding approach, but it remains the same, it is rather the data set that is cured differently.

What fixes the function performed by an AV system is the kind of data fed to the DL model during learning and the kind of data curation the dataset has been subject to. It is a chosen complexity measure, for instance the high variation of the driving lane, that allows the AV model to predict a speed decrease of the front vehicle. Therefore, functions performed by a DL system depend on both the data set and the model but neither of them is normative, nor descriptive. The epistemological relation between functions and systems remains thus to be clarified for the curing paradigm; this is tantamount to analysing the notion of implementation for DL models.

5 Functions and implementation

Different attempts have been made in defining what characterises an implementation, providing definitions that reflect the actual software development practice. An implementation can be considered to be a physical machine that realises a high-level language program, or a high-level language program itself instantiating an algorithm, or a machine language program that implements a given high-level language program.

Assuming a LoA ontology of computational systems developed following the coding paradigm, an implementation is considered to be any LoA realising any other LoA standing upper in the abstraction hierarchy. So, for instance, a physical execution process can also be considered to be an implementation of a specification or of a machine code program; by the same token, a high-level language program is an implementation of a set of specifications; and an algorithm is an implementation of both intentions and specifications.

According to Angius et al. (2021), one can distinguish three definitions of implementation complying with the software development practice in the coding paradigm. One first definition sees implementation as a *semantic interpretation in a medium of implementation* (Rapaport, 1999, 2005): an implementation LoA supplies one with a semantic interpretation of an upper LoA, called the abstract or syntactic level. A high-level language program is said an implementation of an algorithm, *qua* syntactic level, in that the program gives meaning to the algorithm in terms of an interpretation in a high-level programming language. The programming language is what Rapaport (1999) calls the medium of implementation. It easy to see that such an account works for any LoA in the abstraction hierarchy defining a computational system. In particular, the medium of implementation can be both abstract and concrete; accordingly, the physical structure of a computational machinery is the concrete medium of implementation allowing a machine to provide a semantic interpretation of an abstract domain, such as a specification set or a high-level language program.

Turner (2018) stresses that a definition of implementation should be able to distinguish correct from incorrect implementations. This can be achieved understanding implementation as the relation *specification-artefact*. An artefact is an (abstract or concrete) artificial entity that has been specifically engineered in order to satisfy one or more given functions (Hilpinen, 1992; Kroes, 2012). Artefacts are known for their dual ontology defined by functional and structural properties (Kroes & Meijers, 2006).

Functional properties define *what* functions the artefact fulfils; structural properties express *how* those functions are fulfilled, in terms of the physical or abstract properties of the structural level. So, for instance, a bridge is defined by functional properties, such as crossing an obstacle, supporting the weight of vehicles etc.; structural properties are the mechanical properties of the bridge and of its components, including toughness and ductility, that allow those functions to be realised.

Turner (2018) argues that computational systems are artefacts as well, in that they are defined by functional and structural properties. Each LoA defining a computational artefact is a structural level realising the functions required by its functional level, that is, its immediate upper LoA. In this proper sense, each LoA is an artefact implementing the upper level as its functional level[9]

Finally, Primiero (2020) liberalises Turner (2018)'s notion of implementation so as to range over any two LoAs in the abstraction hierarchy, and not just two subsequent ones. Specifically, Primiero (2020) understands implementation as an *instantiation relation* between any LoA couple. This avoids, from an epistemological viewpoint, reinterpreting each level as a structural level or a specification level every time one has to consider different implementations.[10]

What all these views share is the idea that implementation is defined over an abstract (Rapaport, 1999) or functional (Turner, 2018; Primiero, 2020) level that the implementation respectively gives meaning to or fulfils. It has been shown how in the curing paradigm there is not any such level since there is no specification providing an abstract domain or a functional requirement, nor can other levels be identified to have a normative stance in that there is no direction of governance. Again, functions emerge in the model during training and after the data set has been opportunely curated.

A different, more recent, approach on implementation starts exactly by criticising the idea that implementation is a *medium independent* relation between an abstract domain and a structural one. According to the *implementation as resemblance* theory (Curtis-Trudel, 2021), a physical system implements a computation in case it resembles a computational architecture. By computational architecture Curtis-Trudel (2021) means a "blueprint" of a computational physical systems which specifies physical, syntactic, and semantic features that the implementing system must realise. The formal definition of a Turing machine, for instance, which includes the specification of a tape, a reading/writing head, a finite alphabet, a finite set of internal states, and a finite set of instructions, provides a blueprint for any physical Turing machine. A physical system is a physical Turing machine juts in case it resembles the computational architecture of a Turing machine. By resemblance Curtis-Trudel (2021) intends that the physical system has all and only the features specified by the computational architecture; so a physical Turing machine is such in that it comprises a tape, a reading/writing head, a finite alphabet and so forth.

[9] It follows from this analysis that any LoA can also be considered a specification level for its lower LoAs, in that it corresponds to the functional level of an artefact prescribing functions for its implementations.

[10] That is, if the focus is on a high-level language program as an implementation of an algorithm, the program is understood as a structural level; supposing then focus is made on the computational machine as an implementation of the high-level language program, the latter has to be reinterpreted as a functional level.

If the implementation as resemblance relation avoids the so-called triviality argument in favour of pancomputationalism (Chalmers, 1996), and if it does not rely on an abstract level focusing on functions to be performed, computational architectures are still taken to have a normative stance to their implementations and a direction of governance is given from architectures to physical systems.

It follows that DL models cannot be understood as implementations. Nor can DL programs: they cannot be considered implementations in that it is only after their execution that functions will emerge. It can be said that a DL program implements a DL model, in that it instantiates the learning algorithms, such as the backpropagation algorithm (see equation 1) or its updated variants (see equation 2), that play the epistemic role of specification level for the program-artefact, and which are necessary to put the DL model at work. Indeed, from a computational point of view, the development of a program implementing a DL model follows a coding, rather than a curing, paradigm.

The absence of a relation of implementation between a specification level and an algorithm or model level is inherited all the way down in the LoA hierarchy that defines the computational system. A DL program is not required to satisfy any requested function for the computational system to be developed, the same holds for the assembly/machine-code and, last, for the execution level. The machine running a DL program is not implementing, in any of the three meanings of implementation provided above, functions required for such system. Rather, it is at the execution LoA, when the learning program is launched and executed, that the functions will emerge. Those functions will be acknowledged only during the validation and testing phase.

DL systems, as any other computational artefact, are defined by functional levels and structural levels. What distinguishes computational artefacts developed according to a coding paradigm and ones developed following a curing approach is the relation between functions and structure.

In the coding paradigm, the relation between functions and structure is defined by an *intentional theory of function* (Turner, 2018) which sees functions as fixed into the technical artefact by one or more agents pursuing declared intentions or goals (McLaughlin, 2001; Searle, 1995). A bridge fulfils the functions and intentions of the engineers who designed and realised the artefact. Typically, functions satisfied by computational artefacts are put forward by developers, clients, and the other stakeholders involved in a software development project. As the AV study case has shown, functions satisfied by a trained network are not declared beforehand by developers and then fixed in the system. At a first glance, it looks like the relation between functions and structure for DL systems is not captured by the intentional theory of function and one might be tempted to apply a *causal theory of function*.

Indeed, the intentional theory of function is often opposed to the causal theory of function (Cummins, 1975), according to which the physical capacities of the structural level of a system cause the functions implemented by the system. The causal theory of function applies well to natural systems realising functions, mainly biological ones, where the evolved structure of the system allowed some biological function that, by increasing the adaptive fitness, was inherited by descendants. In this proper sense it can be said that the structural properties of, say, the stomach, with its acids, cause the function of food digestion.

Functions performed by a DL model depend from the inner organizations of neurons in the model (how many neuron hidden layers, how many neurons per layer etc.) and their weights, that is, from the training the model underwent and, consequently, from how the dataset was curated. Seemingly, functions are caused by the inner structure of a model and a dataset in conformity with the causal theory of functions. And a system can be said to implement a function when its structural properties cause those functions.

However, there is a significant flaw associated with the causal theory of function when applied to artefacts: it is known to be unable to distinguish between intended functions and side-effects and, even worse, between correct and incorrect implementations (Turner, 2018). Consider a computational system performing some given computable functions, like a desk calculator. Following a causal theory of function, the structural properties of the special purpose micro-processor in the calculator allow the machine to perform the desired arithmetic operations. Suppose now that the calculator starts heating; since heating is certainly caused by the physical properties of the calculator, it has to be considered a new function of the calculator, rather then a side-effect.[11]

Most importantly, consider the case in which the calculator, when given number n as input and asked to calculate the square root, gives an $error$ message for $n > 0$ and $\sqrt{n}$ when $n < 0$. According to the causal theory of function, this is the square root function of the calculator, in that it is caused by the physical capacities of the machine; one cannot argue here that this is not the *intended* square root function. In other words, to discriminate between correct and incorrect behaviours of a computational system, agents are needed fixing functions for the artefacts to be developed. The intentional theory of function is able to distinguish between requested functions and side-effects, and between correct and incorrect behaviours, exactly because correct requested behaviours are formulated by developers into property specifications.

Even though there are not specifications expressing correct behaviours for DL systems, the latter are nonetheless often said to malfunction, or to be flawed (Thung et al., 2012). Horner & Symons (2019) argue that any software system S contains errors if and only if it fails to satisfy some specification H. A question consequently arises as how errors in DL modelling are defined if not with respect to a specification set.

One preliminary consideration is that DL systems contain two coarse families of errors. One family deals with the implementation of a DL model in to a high-level language program which, as already stated, follows a coding paradigm. The full-fledged taxonomy of software errors advanced by Primiero (2014), and the empirical analysis of the most encountered software errors of Horner & Symons (2019), still apply to incorrect implementations of a neural network. In addition, one has to consider a second family of flaws which deals with errors in the modelling of a network.

Horner & Symons (2019) argue that any theory of software errors should adress two distinct questions, namely why errors take place in software development and what an error is. In relation to the first question, Humbatova et al. (2020) have recently

[11] Note that this is not a problem for biological systems in evolution theory, new functions caused by the same structure being known as exaptations (Gould & Vrba, 1982).

provided a taxonomy of errors that may be found in DL models based on an empirical study on developers' reports and interviews. The authors identified as many as five types of error sources, namely:

- *Model*: flaws related to the choice of a particular model (such as a recurrent network in place of a convolutional network), number of layers, properties of a single layer (for instance the number of neurons per layer), choice of the more appropriate activation function.
- *Tensors and inputs*: errors in the choice of tensors and input datatypes or shapes.
- *Training*: errors related to the training of the network; these may be due to the tuning of the hyperparameters, the choice of the loss or optimization functions, the pre-processing and curation of the training dataset.
- *GPU* and *API*: errors emerging at the execution step and related to a wrong GPU and API utilisation.[12]

In relation to the second question put forward by Horner & Symons (2019), namely what an error is, Humbatova et al. (2020) explicitly state that an error is a behaviour of a DL system which is deemed by developers not adequate for the general task the system is trained for. As extensively examined, in the curing paradigm functions are not fixed by specifications, they rather emerge during training. Consequently, errors are not, strictly speaking, violations of specifications as defined by Horner & Symons (2019); errors are learnt functions not meeting the general requests, or intentions, put forward by stakeholders. In the case of AV, an error of this kind might be a function, say 'crash into the side guardrail to avoid a collision with a bicycle that suddenly moves out from the bicycle lane into the drive lane', that is deemed inadequate with respect to the general request of safe driving.

This, in conclusion, brings us back to the intentional theory of function: in the curing paradigm, functions are established indirectly by an agent curing the training dataset, observing the performed functions, and evaluating whether the emerged functions comply with the general intentions the system is developed for.

6 Functions and correctness

In the coding paradigm, specifications are said to provide *correctness jurisdiction* over computational artefacts (Turner, 2011): the latter are said correct when they satisfy their specifications or, in other words, when the displayed behaviours of the system match with behaviours prescribed by the specifications. The traffic light of Fig. 5 is correct with respect to specifications S and L: the two traffic lights do not enter the $green$ state at the same time and each of them is able to enter the $green$ state infinitely often. Specifications also provide criteria for malfunctioning: a traffic light

[12] It would be interesting at this point analysing such a taxonomy in the light of the conceptual distinctions made, for instance, by Fresco & Primiero (2013); Primiero (2014); Floridi et al. (2015) and comparing it to the empirical classification of traditional software errors carried out by Horner & Symons (2019). This, however, would go far beyond the scope of this paper.

system allowing both traffic lights to enter the *green* state simultaneously would be considered incorrect, or malfunctioning, with respect to S.

Turner (2018) underlines how a satisfactory notion of implementation for computational systems should include the concept of correctness: a good implementation is a correct implementation. And *vice versa*, a computational system is correct if and only if it is defined by correct implementations.

The absence, for DL systems, of specification levels and of a notion of implementation fully based on an intentional theory of function prevents from providing correctness criteria as defined in the coding paradigm. In DL, models are evaluated in terms of their *generalization*: a DL model generalises in case it provides successful predictions when applied to the test dataset; a related property is that of *optimization*, concerning a model providing successful predictions when applied to the training dataset. Unfortunately, whereas optimization increases along with learning (the more the model learns, the more successful prediction it provides over the training dataset), generalization does not increase along with optimization. At a certain point of the learning process, optimization keeps increasing but generalization decreases and the model is said to *overfit*. What happens is that the learning processes start being related too much to the patterns found in the training dataset and the model is unable to predict successfully with new data never seen before. In order to avoid overfitting, datasets are rather divided into a training set, a *validation* set, and finally a test set. In this way the model is trained over the training set but evaluated over the validation set. Once generalization of the model is maximised, the model is tested over the test set (Chollet, 2021).[13]

At the same time, one cannot do without a notion of correctness for DL models involved in safety-critical systems, wherein, as in the case of AVs, criteria are needed for evaluating their dependability and security. One notion that may serve this role is that of experimental computational validity from Primiero (2020). The author distinguishes among *formal computational validity*, *physical computational validity*, and *experimental computational validity*, depending on the chosen LaA at which validity is defined.

Formal computational validity is determined at the high-level language program LoA as a conformity between models of the program on one hand and models of its specifications on the other. Let us spell this out. According to formal computational validity, a program validates a specification when it can be formally proven that the program is correct with respect to it. Proving program correctness in formal verification amounts to representing formally both the program and the specification and demonstrating that the (formalization of the) specification is a logical consequence of (the model of) the program (Leeuwen, 1990). Formal verification methods of this sort include *theorem proving*, wherein programs are represented by axiomatic systems

[13] Notice that in case the validation process requires to fix the model may times, modifying the so called hyper-parameters (such as the number of layers or the size of the layers), the model may also overfit to the validation data set. In this case it said that an *information leaks* from the validation set to the trained model takes place. For more details see Chollet (2021, pp. 97-100).

(Hoare, 1969), and *model checking*, which represents programs as state transition systems (Clarke et al., 1999).

The absence of a specification LoA in DL modelling and development denies in principle the chance of pursuing formal computational validity for DL systems. This would indeed require to formally prove that a DL model satisfies a formula expressing a specification. The need to develop dependable DL models employed in many safety-critical systems, including AVs or autonomous weapons, is stimulating a new research area in DL which tries to apply verification techniques to DL. Liu et al. (2021) present an overview of current approaches to algorithmic verification of properties of DL models, properties intended as functional relations between inputs and outputs. As in traditional formal verification, such properties are violated in case a counterexample is found, namely an output which does not correspond to the output mapped by the functional relation for a given input. Algorithmic verification methods successfully applied to DL networks include reachability analysis, performing a layer-by-layer analysis of the model to define the set of reachable outputs given a set of inputs; optimization, a counter-example guided verification technique trying to make the network compute a violation of the functional property of interest; and search methods, which look for counterexamples by searching through potential activation patterns in the network (see Liu et al. (2021) for details).[14]

One should be careful to notice here that while algorithmic methods for DL networks are certainly *verification* methods, what one is proving here is not *correctness*, which is an *epistemological* relations holding between required functions and implementations, as examined in Sects. 4 and 5. In the coding paradigm, verification methods prove correctness in that they formally demonstrate that a formula expressing a behavioural property that *must* hold true of a system, and which is formalised before the system is developed, does indeed hold true of a model of the system. In the verification methods sketched by Liu et al. (2021), functional properties formally derived from a formal representation of a network are known only after the network has been trained: they are the functions *actually satisfied by the network*, not functions that must be satisfied; in other words, they do not provide "correctness jurisdiction" over the network.

Distinct cases should be made for networks applied to image analysis, diagnosis analysis, or any other context wherein the function to be performed is clear since the very beginning and other applications such as AVs, autonomous weapons, or other robotic applications, where specific functions emerge from the interaction with an environment and depend more on the training data. In the former case a model is trained to, say, recognise faces or predict cancer; even though not properly formalised, desired functions possess the epistemological status of specifications: they specify requirements for the network to be trained. In the case of AV, functions emerge during

[14] It should be noted that, as it is for common formal verification, applying those methods to real cases often requires using abstractions and approximations to allow computational tractability. However, abstractions and approximations lead to the incompleteness of the involved verification method: the algorithm may either terminate with an 'unknown' answer, or terminate with a false negative, that is, a counterexample showing a violation of the verified functional property to which, though, does not correspond any actual model computation.

training, depend on how the data set has been curated, the model may well have realised different satisfactory functions, and it cannot be said to be correct with respect to the actually satisfied functions.

Let us now turn to examine what Primiero (2020) calls physical computational validity. The latter considers all of the LoAs, not only algorithms and high-level language programs as in formal computational validity. In particular, physical computational validity is concerned with evaluating the correctness of physical implementations of programs. A computational artefact validates a specification if all the LoAs defining the artefact, from the algorithm to the execution level, are correct with respect to the specification. Validity is still defined functionally over specifications and cannot be applied to the case of DL. Clearly, physical computational validity cannot be proven formally, in that formal verification only applies to abstract mathematical levels (algorithms and programs). In this case correctness is evaluated empirically through *testing* (Ammann & Offutt, 2016), by executing the program and observing whether the manifest behaviours comply with those required by the specifications.

Recent research lines in the validation of DL models involved in safety-critical systems are devoted to extend the practice of software testing to the case of neural networks. Ma et al. (2018); Pei et al. (2017); Odena et al. (2019) are some cases in point. This is felt as particularly urgent in the AV and avionics contexts (Gerasimou et al., 2020). However, even though DL systems are tested, they are not tested *against* some specification. This fact is recognized as one main limitation in the application of software testing to learning networks (Salay et al., 2017).[15]

In the coding paradigm, testing a system is a difficult task as well: non-trivial software is characterised by potentially infinite inputs and a problem arises as which inputs to use to evaluate whether the involved program complies with the desired specifications. *Coverage criteria* are methods used to select with which inputs to launch and observe the program and are defined over the specifications of interest. In particular, models of the program, such as data flow graphs, are used to identify the behaviours which, if executed, would violate the given specification, and the inputs that may induce those behaviours (Ammann & Offutt, 2016).

In absence of a specification, proposed coverage criteria for testing DL networks aim at selecting those data from the testset that will induce the network to perform as diverse behaviours as possible. For instance, the testing method *DeepXplore* (Pei et al., 2017) proposes *neuron coverage* criteria aimed at selecting which neurons to activate to obtain diversity of behaviours (Ma et al., 2018; Pei et al., 2017). Diversity of behaviours is requested in order to evaluate the extent to which the model generalises. As the name 'DeepXplore' suggests, what these methods pursue is *exploratory testing* (Itkonen & Rautiainen, 2005), that is, a form of testing wherein systems are not tested against *requested* behaviours but are observed in their operating environment.

As it was the case for formal computational validity, DL systems are tested but the aim of testing is not that of evaluating physical correctness. Accordingly, physical computational validity does not apply either. Both in case of formal and of physical

[15] Another restraint is given by the difficulty of controlling every parameter involved in a specific function, given the complex layered structure of contemporary networks (Gerasimou et al., 2020).

computational validity, the unfeasibility of verifying the correctness of a DL system is to be taken as an *in principle* limitation. One should also consider practical limitations that impede formal and empirical verification to be achieved *in practice* and that put DL systems on a par with traditional software systems. The verification problem is known to be insolvable in practice due to the infinitary time resources that would be required (Symons & Horner, 2020).

Finally, experimental computational validity deals with the validity of experimental computational processes, mainly simulative methods. In simulative sciences, physical systems are typically represented by a set of differential equations and simulations consist in building a mathematical model approximating those equations, implementing the mathematical model by a computational model, usually a computer program, and finally executing the program to calculate solutions to the approximated equations.

First, Primiero (2020) defines an experimental computational process as a correct implementation of a computational model; a valid experimental computational process is then such that the mathematical model is *fit for purpose*, the computational model is isomorphic or partial isomorphic to the mathematical one, and the computational system implementing the computational model is *usable*. A mathematical model is fit for purpose when it is able to represent some of the requested variables and inferential properties of the represented system[16] and a computational implemented system is considered usable when it allows to accomplish the desired experimental analysis and to construct a fit for purpose mathematical model.

Let us examine whether the notion of experimental computational validity may apply to DL systems. One first consideration is that the execution of a DL system can be acknowledged as an experimental computational process in the sense of Primiero (2020): a DL system is a correct implementation of a computational model, namely the program implementing the DL model. Recall that whereas, in the curing paradigm, the notions of implementations and correctness do not apply to DL models, here the mathematical model, they do apply to the program instantiating the model and to the system executing the program. A DL system is required to be a correct implementation in the sense that it is requested to execute correctly the learning algorithms in the DL model.

The validity of the executions of a DL system can then be considered valid in case first, the mathematical model, namely the DL model, is fit for purpose. A DL model is fit for purpose when it includes some of the variables of the simulated system. For instance, an AV DL model is taken to learn and act as a human driver and it should include, in order to be considered fit for purpose, some of the relevant variables characterising a human driving system. Those variables are the magnitudes included, or labelled, in the dataset. In the AVs study case, those variables were, for instance, the topology of the driving paths, the height variation of the drivable area, or crowdedness. And a DL system should also be usable, that is, it should permit satisfactory predictions. In other words, a usable model is a model that generalises. Accordingly, a usable system of this sort allows one to evaluate when a model is fit for purpose.

[16] A mathematical model is called by Primiero (2020) *robust*; in case it is able to include *all* variables of interest and *all* the inferential properties of the simulated empirical system.

One should be careful to notice here that evaluating when a model is fit for purpose is hampered, or fostered, respectively by the *opacity* or *transparency* of the involved simulative models. Indeed, the *epistemic opacity* of common simulative models is known to be one main obstacle in evaluating what Primiero (2020) calls experimental computational validity. Humphreys (2004) defines a simulative process as epistemically opaque in case an agent does not know all the epistemic relevant elements of the process. Durán & Formanek (2018) explore Humphreys' definition by understanding as opaque a process that is not surveyable and accessible step-by-step.

DL models are known to be highly opaque in that it appears impossible to come to know which patterns are identified by the network, how they are identified, and what patterns are used by the system to output the displayed result (Lipton, 2018). The computational processes of a deep neural network are epistemically opaque precisely in the meaning of Durán & Formanek (2018), since an agent cannot survey how all the network parameters are updated at each backpropagation iteration. So even though the training dataset contains some of the variables of interest of the simulated system, one does not know which of them are identified and used by the network. In other words, one does not know the extent to which a DL model is fit for purpose.

Experimental computational validity is therefore jeopardised by epistemic opacity. This may not constitute a barrier to the usage of simulative methods when they are considered as technologies merely aimed at enhancing human capacities (Humphreys, 2004). Alvarado (2022a) in particular defines AI systems, including those involving DL modelling, as *epistemic technologies* directed at enhancing our epistemic capacities, namely data analysis, prediction, or inferential processes. As such computer simulations involving AI systems are better understood as *artefacts* turned to epistemic enhancement rather than instances of formal or empirical reasoning (Alvarado, 2022b).

One way to avoid such a form of scientific instrumentalism is to recognise that a simulative model can be sanctioned (Winsberg, 1999) by relying on what Durán & Formanek (2018) call *computational reliabilism*, namely by relying on some process external to the computer simulation itself and that grounds trust in the simulation outcomes. For the case of traditional, equation-based, computer simulations, Durán & Formanek (2018) identify such external reliable processes with verification and validation methods, robustness analysis, reasoning based on previous successful implementations, and expert knowledge.

In the case of DL models, a whole research area has arisen going under the name of *Explainable Artificial Intelligence* (XAI) (Samek et al., 2019). XAI consists in a set of algorithms and methods that allows experts to understand and retrace how a DL model comes to a result. XAI is used to describe a network, its potential errors, and to characterize model transparency and interpretability. Transparency is understood as the identification of the parameters involved in a given learning process. While interpretability is often associated with the chance of making a decision process understandable by human agents. As such, XAI should fall under the expert knowledge reliabilism.

Leaving to a further investigation the question concerning to what extent DL modelling can be considered, from an epistemological point of view, a simulative method,[17] and how DL simulative models are to be sanctioned, it can be concluded that the actual practice of making DL models generalise given their operating environment is akin to pursuing experimental computational validity, namely an empirical validity that does not consider correctness (as physical computational validity) but that evaluates DL models in terms of their successful predictions.

7 Conclusions

The increasing, and in part unexpected (Plebe & Grasso, 2019), success of DL in many machine learning applications calls for an equally increasing interest from the philosophy of computer science. The ontology, epistemology, and methodology of computational systems developed so far has considered computational systems as they are developed in traditional software development methods, here named the coding paradigm. In particular, computational systems have been acknowledged, from an ontological point of view, as artefacts defined by functional and structural properties; functions are expressed by specifications and an implementation is defined as any structural level realising those functions. The dichotomy function/structure, which articulates through all the LoA hierarchy defining a computational artefact, distinguished the epistemology and methodology of those artefacts as well. The issues of explanation (Angius & Tamburrini, 2017; Piccinini, 2007), abstraction (Floridi, 2008; Colburn & Shute, 2007), modelling (Angius & Tamburrini, 2011), and correctness (Turner, 2018; Shapiro, 1997; Fetzer, 1988), to mention some, all leverage on the distinction between required functions and implementing structures. As for what concerns the notion of correctness, it has been defined as a correspondence between the outputs requested by a specification for given inputs and the outputs displayed by an implemented program when launched with those inputs. In other words, correctness is the functional agreement between specifications and implementations.

The dataset curation activities characterising DL modelling have been shown in this paper to unhinge the coding paradigm. The absence of a specification level formulating required functions deconstructs the function/structure dichotomy identifying traditional computational artefacts together with their epistemology. In what has been named curing paradigm, functions emerge in a model during training and depend much more on how the training dataset has been selected, labelled, and curated, rather than on the architecture of the DL model and the code of the DL program. Functions are not requested by specifications but learnt; by consequence, models and programs are not implementations of specifications, and they are not implementations at all, since all the available definitions of implementation for artefacts make reference either to a functional level or to a direction of governance. Finally, by not being implemen-

[17] There are indeed scientific contexts in which DL models used to predict the evolution of some empirical system also bear structural similarities with the target system. The convolutional DL model used by Monk (2018) to simulate parton shower, the neural model by Choudhary et al. (2020) for simulating the Henon-Heiles potential, and the MetNet-2 (Meteorological Neural Network 2) (Espeholt et al., 2021) DL model are some examples.

tations, DL systems cannot be said to be correct or incorrect, in that correctness is always defined against some specification or functional level. This paper argued that the dependability of DL systems, especially those working in safety-critical contexts, can be evaluated in terms of their experimental computational validity, that is, in terms of their ability of providing fit for purpose models and successful predictions.

The curing paradigm in DL should not be confined to provide insights into the restricted area of the philosophy of computer science but also in crucial related fields such as, to give a couple of examples, *computer ethics* (Johnson, 2009) and the *epistemology of computer simulations* (Winsberg, 2010). One issue in computer ethics affected by the present analysis is that of the moral responsibility of programmers. Even though achieving 100% of correctness is known to be unfeasible for computational systems developed in the coding paradigm, under the curing approach DL systems are to be evaluated in terms of their experimental computational validity. New criteria for assessing the responsibility of programmers should be conceived, taking into consideration that functions, once learnt, have to be evaluated with respect to the general intentions the system was trained and implemented for.

A second important issue mentioned in the previous section is whether DL applications can be considered simulative methods; indeed, defining a computer simulation is still an open problem (Durán, 2018; Winsberg, 2022). As a computer simulation, DL may trigger the analysis of many critical points in the epistemology of computer simulations, including the ontology of simulative models, the relation between validation and verification, or the epistemic opacity of simulative models. In conclusion, the curing paradigm in DL is likely to draw more and more attention from the philosophers of computing.

Author Contributions All authors equally contributed to the study conception and design of this paper. Material preparation and analysis were performed by both NA e AP. All authors read and approved the final manuscript.

Funding Nicola Angius was partially funded by the Research Project ANR-17-CE38-0003-01 (ANR - Agence Nationale de la Recherche) titled 'What is a (computer) program: Historical and Philosophical Perspectives'.

Declarations

Conflict of interest Nicola Angius and Alessio Plebe declare they have no financial interests.

References

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., et al. (2015). TensorFlow: Large-scale machine learning on heterogeneous systems. Technical report, Google Brain Team.

Ackerman, E. (2016). Self-driving cars were just around the corner — in 1960. *IEEE Spectrum*, 31 Aug.

Alpern, B., & Schneider, F. B. (1985). Defining liveness. *Information processing letters, 21*(4), 181–185.

Alpern, B., & Schneider, F. B. (1987). Recognizing safety and liveness. *Distributed Computing, 2*(3), 117–126.

Alvarado, R. (2022a). Ai as an epistemic technology. http://philsci-archive.pitt.edu/21243/

Alvarado, R. (2022). Computer simulations as scientific instruments. *Foundations of Science, 27*, 1–23.

Ammann, P., & Offutt, J. (2016). *Introduction to software testing*. Cambridge University Press.

Angius, N. (2013). Model-based abductive reasoning in automated software testing. *Logic Journal of IGPL, 21*(6), 931–942.
Angius, N., Primiero, G., and Turner, R. (2021). The Philosophy of Computer Science. In E. N. Zalta (Eds.), *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, Spring 2021 edition.
Angius, N., & Tamburrini, G. (2011). Scientific theories of computational systems in model checking. *Minds and Machines, 21*(2), 323–336.
Angius, N., & Tamburrini, G. (2017). Explaining engineered computing systems' behaviour: the role of abstraction and idealization. *Philosophy & Technology, 30*(2), 239–258.
Baier, C. & Katoen, J.-P. (2008). *Principles of model checking*. MIT press.
Bengio, Y., Lamblin, P., Popovici, D., & Larochelle, H. (2007). Greedy layer-wise training of deep networks. In *Advances in Neural Information Processing Systems* (pp. 153–160).
Chalmers, D. J. (1996). Does a rock implement every finite-state automaton? *Synthese, 108*, 309–333.
Chollet, F. (2021). *Deep learning with Python*. Simon and Schuster.
Choudhary, A., Lindner, J. F., Holliday, E. G., Miller, S. T., Sinha, S., & Ditto, W. L. (2020). Physics-enhanced neural networks learn order and chaos. *Physical Review E, 27*, 217–236.
Clarke, E. M., Grumberg, O., & Peled, D. A. (1999). *Model checking*. MIT Press.
Colburn, T., & Shute, G. (2007). Abstraction in computer science. *Minds and Machines, 17*(2), 169–184.
Cordts, M., Omran, M., Ramos, S., Rehfeld, T., Enzweiler, M., Benenson, R., Franke, U., Roth, S., & Schiele, B. (2016). The cityscapes dataset for semantic urban scene understanding. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 3213–3223).
Cummins, R. (1975). Functional analysis. *Journal of Philosophy, 72*, 741–764.
Curtis-Trudel, A. (2021). Implementation as resemblance. *Philosophy of Science, 88*(5), 1021–1032.
de Villers, J., & Barnard, E. (1992). Backpropagation neural nets with one and two hidden layers. *IEEE Transactions on Neural Networks, 4*, 136–141.
Duchi, J., Hazan, E., & Singer, Y. (2011). Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research, 12*, 2121–2159.
Durán, J. M. (2018). *Computer simulations in science and engineering: Concepts-Practices-Perspectives*. Springer.
Durán, J. M., & Formanek, N. (2018). Grounds for trust: Essential epistemic opacity and computational reliabilism. *Minds and Machines, 28*, 645–666.
Espeholt, L., Agrawal, S., Słonderby, C., Kumar, M., Heek, J., Bromberg, C., Gazen, C., Hickey, J., Bell, A., & Kalchbrenner, N. (2021). Skillful twelve hour precipitation forecasts using large context neural networks. abs/2111.07472.
Fetzer, J. H. (1988). Program verification: The very idea. *Communications of the ACM, 31*(9), 1048–1063.
Floridi, L. (2008). The method of levels of abstraction. *Minds and machines, 18*(3), 303–329.
Floridi, L., Fresco, N., & Primiero, G. (2015). On malfunctioning software. *Synthese, 192*(4), 1199–1220.
Fresco, N., & Primiero, G. (2013). Miscomputation. *Philosophy & Technology, 26*, 253–272.
Gerasimou, S., Eniser, H. F., Sen, A., & Cakan, A. (2020). Importance-driven deep learning system testing. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)* (pp. 702–713). IEEE.
Gould, S. J., & Vrba, E. S. (1982). Exaptation–a missing term in the science of form. *Paleobiology, 8*(1), 4–15.
Hilpinen, R. (1992). On artifacts and works of art 1. *Theoria, 58*(1), 58–82.
Hinton, G. E., Krizhevsky, A., & Wang, S. D. (2011). Transforming auto-encoders. In *International Conference on Artificial Neural Networks* (pp. 44–51). Springer-Verlag.
Hinton, G. E., & Salakhutdinov, R. R. (2006). Reducing the dimensionality of data with neural networks. *Science, 28*, 504–507.
Hoare, C. A. R. (1969). An axiomatic basis for computer programming. *Communications of the ACM, 12*(10), 576–580.
Horner, J. K., & Symons, J. (2019). Understanding error rates in software engineering: Conceptual, empirical, and experimental approaches. *Philosophy & Technology, 32*, 363–378.
Humbatova, N., Jahangirova, G., Bavota, G., Riccio, V., Stocco, A., & Tonella, P. (2020). Taxonomy of real faults in deep learning systems. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (pp. 1110–1121).
Humphreys, P. (2004). *Extending ourselves: Computational science, empiricism, and scientific method*. Oxford University Press.

Itkonen, J. & Rautiainen, K. (2005). Exploratory testing: a multiple case study. In *2005 International Symposium on Empirical Software Engineering, 2005.* (pp. 10–pp). IEEE.
Janai, J., Güney, F., Behl, A., & Geiger, A. (2020). Computer vision for autonomous vehicles, problems, datasets and state of the art. *Foundations and Trends in Computer Graphics and Vision, 12*, 1–308.
Johnson, D. G. (2009). *Computer ethics*. London: Pearson.
Jones, C. B. (1990). Systematic software development using vdm. In *Prentice Hall International Series in Computer Science*.
Joshi, A. J., Porikli, F., & Papanikolopoulos, N. (2009). Multi-class active learning for image classification. In *2009 IEEE conference on computer vision and pattern recognition* (pp. 2372–2379). IEEE.
Ketkar, N. (2017). *Introduction to PyTorch* (pp. 195–208). Berkeley (CA): Apress.
Kim, T. K., Yi, P. H., Hager, G. D., & Lin, C. T. (2020). Refining dataset curation methods for deep learning-based automated tuberculosis screening. *Journal of Thoracic Disease, 12*, 5078–5085.
Kingma, D. P. & Ba, J. (2014). Adam: A method for stochastic optimization. In *Proceedings of International Conference on Learning Representations*.
Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). ImageNet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*, pp. 1090–1098.
Kroes, P. (2012). *Technical artefacts: Creations of mind and matter: A philosophy of engineering design* (vol. 6). Springer Science & Business Media.
Kroes, P., & Meijers, A. (2006). The dual nature of technical artefacts. *Studies in History and Philosophy of Science, 37*(1), 1–4.
Kröger, F., & Merz, S. (2008). *Temporal Logic and State Systems*. Springer.
Kuutti, S., Fallah, S., Bowden, R., & Barber, P. (2019). Deep learning for autonomous vehicle control - algorithms, state-of-the-art, and future prospects. *Synthesis Lectures on Advances in Automotive Technology, 3*, 1–80.
Leeuwen, J. V. (1990). *Handbook of Theoretical Computer Science, Volume B: Formal models and semantics*. MIT Press.
Li, X. & Guo, Y. (2013). Adaptive active learning for image classification. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 859–866).
Lipton, Z. C. (2018). The mythos of model interpretability: In machine learning, the concept of interpretability is both important and slippery. *Queue, 16*(3), 31–57.
Liu, C., Arnon, T., Lazarus, C., Strong, C., Barrett, C., Kochenderfer, M. J., et al. (2021). Algorithms for verifying deep neural networks. *Foundations and Trends® in Optimization*, *4*(3-4), 244–404.
Ma, L., Juefei-Xu, F., Zhang, F., Sun, J., Xue, M., Li, B., Chen, C., Su, T., Li, L., Liu, Y., et al. (2018). Deepgauge: Multi-granularity testing criteria for deep learning systems. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering* (pp. 120–131).
McLaughlin, P. (2001). *What functions explain: Functional explanation and self-reproducing systems*. Cambridge University Press.
Meng, J., Chen, P., Wahib, M., Yang, M., Zheng, L., Wei, Y., Feng, S., & Liu, W. (2022). Boosting the predictive performance with aqueous solubility dataset curation. *Scientific Data, 9*, 71.
Monk, J. W. (2018). Deep learning as a parton shower. *Journal of High Energy Physics, 2018*, 21.
Odena, A., Olsson, C., Andersen, D., & Goodfellow, I. (2019). Tensorfuzz: Debugging neural networks with coverage-guided fuzzing. In *International Conference on Machine Learning* (pp. 4901–4911). PMLR.
Pei, K., Cao, Y., Yang, J., & Jana, S. (2017). Deepxplore: Automated whitebox testing of deep learning systems. In *Proceedings of the 26th Symposium on Operating Systems Principles* (pp. 1–18).
Piccinini, G. (2007). Computing mechanisms. *Philosophy of Science, 74*(4), 501–526.
Plebe, A., & Grasso, G. (2019). The unbearable shallow understanding of deep learning. *Minds and Machines, 29*(4), 515–553.
Plebe, A., & Perconti, P. (2022). *The Future of the Artificial Mind*. Boca Raton: CRC Press.
Primiero, G. (2014). A taxonomy of errors for information systems. *Minds and Machines, 24*, 249–273.
Primiero, G. (2016). Information in the philosophy of computer science. In *The Routledge handbook of philosophy of information* (pp. 106–122). Routledge.
Primiero, G. (2020). *On the foundations of computing*. Oxford University Press.
Rapaport, W. J. (1999). Implementation is semantic interpretation. *The Monist, 82*(1), 109–130.
Rapaport, W. J. (2005). Implementation is semantic interpretation: further thoughts. *Journal of Experimental & Theoretical Artificial Intelligence, 17*(4), 385–417.

Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986). Learning representations by back-propagating errors. *Nature, 323*, 533–536.
Rumelhart, D. E. & McClelland, J. L. (Eds.) (1986). *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*.
Sadat, A., Segal, S., Casas, S., Tu, J., Yang, B., Urtasun, R., & Yumer, E. (2021). Diverse complexity measures for dataset curation in self-driving. In *2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)* (pp. 8609–8616). IEEE.
Salay, R., Queiroz, R., & Czarnecki, K. (2017). An analysis of iso 26262: Using machine learning safely in automotive software. arXiv preprint arXiv:1709.02435.
Samek, W., Montavon, G., Vedaldi, A., Hansen, L. K., & Müller, K.-R. (2019). *Explainable AI: interpreting, explaining and visualizing deep learning* (vol. 11700). Springer Nature.
Searle, J. R. (1995). *The construction of social reality*. Free Press.
Settles, B. (2012). Active learning. *Synthesis Lectures on Artificial Intelligence and Machine Learning, 6*(1), 1–114.
Shapiro, S. (1997). Splitting the difference: the historical necessity of synthesis in software engineering. *IEEE Annals of the History of Computing, 19*(1), 20–54.
Sommerville, I. (2021). *Software Engineering*. London: Pearson.
Spivey, J. M. (1988). *Understanding Z: a specification language and its formal semantics* (vol. 3). Cambridge University Press.
Symons, J., & Horner, J. K. (2020). Why there is no general solution to the problem of software verification. *Foundations of Science, 25*, 541–557.
Thung, F., Wang, S., Lo, D., & Jiang, L. (2012). An empirical study of bugs in machine learning systems. In *2012 IEEE 23rd International Symposium on Software Reliability Engineering* (pp. 271–280). IEEE.
Turing, A. (1936). On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society, 42*, 230–265.
Turing, A. (1948). Intelligent machinery. Technical report, National Physical Laboratory, London. Reprinted in Ince, D. C. (ed.) *Collected Works of A. M. Turing: Mechanical Intelligence*, Elsevier Science Publishers, 1992.
Turner, R. (2009). *Computable models* (vol. 1193). Springer.
Turner, R. (2011). Specification. *Minds & Machines, 21*(2), 135–152.
Turner, R. (2018). *Computational artifacts*: : *towards a philosophy of computer science*. Springer.
Turner, R. (2020). Computational intention. *Studies in Logic, Grammar and Rhetoric, 63*(1), 19–30.
Winsberg, E. (1999). Sanctioning models: The epistemology of simulation. *Science in Context, 12*(2), 275–292.
Winsberg, E. (2010). *Science in the age of computer simulation*. University of Chicago Press.
Winsberg, E. (2022). Computer Simulations in Science. In E. N. Zalta, & U, Nodelman (Eds.), *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, Winter 2022 edition.