Innovation Network
EASA AI Task Force

Daedalean AG

Report

Concepts of Design Assurance for Neural Networks (CoDANN) II

Public extract

May 19, 2021

Version 1.0

An agency of the European Union

Authors (in alphabetical order)

EASA

Giovanni Cima
Jean-Marc Cluzeau
Xavier Henriquel*
Hassan Semde
Guillaume Soudain
Stephane Vaubourg

ai@easa.europa.eu

Daedalean AG

Vincent van der Brugge
Dr. Luuk van Dijk
Lukasz Janyst
Dr. Mathias Körner
Dr. Corentin Perret-Gentil*
Ruben Polak
Ian Whittington

ipc-feedback@daedalean.ai

Citing this report

EASA and Daedalean, *Concepts of Design Assurance for Neural Networks (CoDANN) II*, May 2021.

```
@techreport{CoDANN2,
  author = {EASA and Daedalean},
  title = {Concepts of Design Assurance for Neural Networks (CoDANN) II},
  month = 5,
  year = 2021
}
```

Disclaimer

*: Principal authors.

Contents

Executive summary

This report is one of the principal outcomes of a project between the European Union Aviation Safety Agency (EASA) and Daedalean AG, which took place within the scope of an Innovation Partnership Contract (IPC) between July 2020 and May 2021.

This project is a follow-up to a first IPC between EASA and Daedalean, which resulted in the publication of a 104-page report in March 2020, titled *Concepts of Design Assurance for Neural Networks* [CoDANN20].

First Daedalean/EASA IPC (2020) The main goal of the first project was to investigate the possible use of systems employing machine learning/neural networks in safety-critical applications, looking in particular at potential challenges with respect to trustworthiness (see [EAS20, p.14]), such as the ability to provide performance guarantees, as well as the applicability of existing guidance such as [ED-79A/ARP4754A; ED-12C/DO-178C].

In addition to in-depth discussions around these aspects, an important outcome of [CoDANN20] was the identification of a W-shaped development process (Figure 1) adapting the classical V-shaped cycle to machine learning applications.

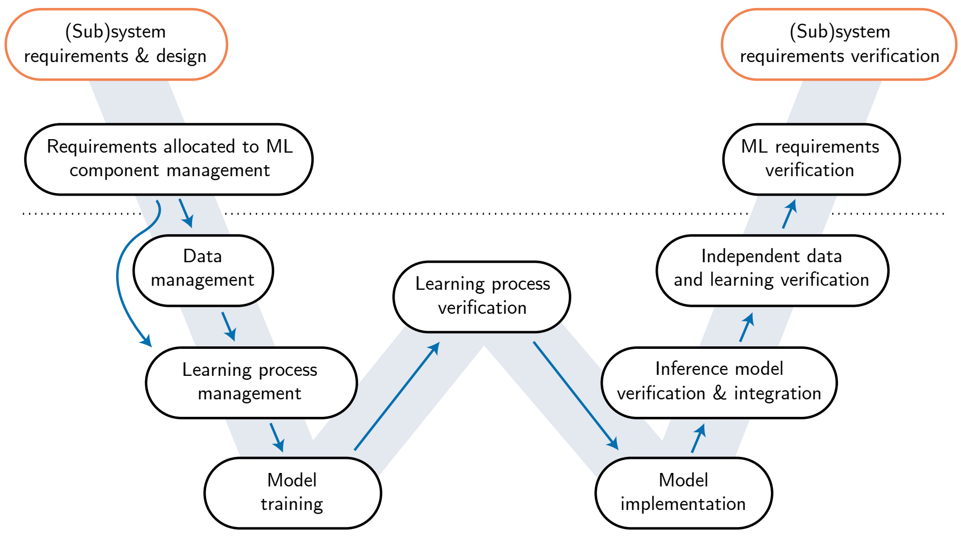

(Sub)system requirements & design
Requirements allocated to ML component management
Data management
Learning process management
Model training
Learning process verification
Model implementation
Inference model verification & integration
Independent data and learning verification
ML requirements verification
(Sub)system requirements verification

Figure 1: W-shaped development cycle for *Learning assurance* from [CoDANN20].

An agency of the European Union

CoDANN II (2021) The goal of this second project was threefold: investigate topics left out in [CoDANN20], mature the concept of *Learning assurance* and investigate remaining trustworthy AI building blocks from [EAS20]. A conclusion was reached on each of the following topics:

- *Implementation and inference* parts of the W-shaped process (hardware, software and system aspects), encompassing:
  - The *development* of machine learning models with the numerous challenges that can arise compared to classical software development.
  - The *deployment* of models on the operational platform, with the need for complex hardware to perform neural network inference.

  In both topics, a fundamental requirement is to ensure that performance guarantees are not lost in the transition from the development environment to the operational environment. This advances the discussion from theoretical considerations on learning assurance in [CoDANN20] to practical ones.
- Definition and role of *explainability* within the scope of both Learning assurance and human-machine interaction. Techniques have been identified as well as their contributions in Learning assurance and Human-Machine Interaction.
- Details on the *system safety assessment process*: out-of-distribution detection, runtime monitoring, uncertainty estimation, and integration with filtering/tracking. This concludes discussions on the integration of neural networks into complex systems and their evaluation in safety assessments.

As in the first project, the majority of the considerations are generic to machine learning/neural networks. They are exemplified throughout with the use case of a neural network-based visual traffic detection system. This is a different use case from [CoDANN20] in order to explore further the new type of applications that will soon reach the aviation market thanks to AI/ML technology.

Thanks to the first Daedalean/EASA IPC [CoDANN20] and the current project, all steps of the W-shaped process have now been investigated. Points of interest for future research activities, standards development and certification exercises have been identified, and they will contribute to stir EASA efforts toward the introduction of AI/ML technology in aviation.

◇ ◇ ◇

The European Union Aviation Safety Agency (EASA) is the centerpiece of the European Union's strategy for aviation safety. Its mission is to promote the highest common standards of safety and environmental protection in civil aviation. The Agency develops common safety and environmental rules at the European level. It monitors the implementation of standards through inspections in the Member States and provides the necessary technical expertise, training and research. The Agency works hand in hand with the national authorities which continue to carry out many operational tasks, such as certification of individual aircraft or licensing of pilots.

Daedalean AG is building autonomous flight control software for civil aircraft of today and advanced aerial mobility of tomorrow. The Switzerland-based company has brought together expertise from the fields of machine learning, robotics, computer vision, path planning as well as aviation-grade software engineering and certification. Daedalean has partnered with incumbent avionics manufacturers including Honeywell Aerospace and Avidyne to bring to market the first-ever machine-learning based avionics. The company has developed an onboard visual awareness system demonstrating crucial early capabilities on a path to certification for airworthiness.

Chapter 1

Introduction

Machine Learning (ML), a method to develop Artificial Intelligence (AI) systems from data, is currently revolutionizing several fields of computer science [Sej18], and presents major opportunities for the aviation industry, both for human assistance and towards autonomous operations.

In February 2020, the European Union Aviation Safety Agency (EASA) published a first AI Roadmap [EAS20] aimed at creating a risk-based "AI trustworthiness" framework to enable future AI/ML applications and support European research and leadership in AI. This roadmap is in particular guided by the European Commission's "Ethics and Guidelines on Trustworthy AI" 2019 report [EGTA].

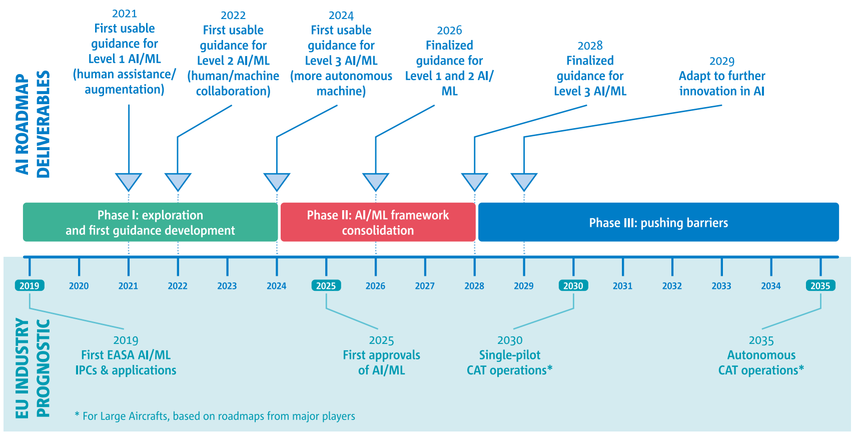

AI ROADMAP DELIVERABLES
2021 First usable guidance for Level 1 AI/ML (human assistance/ augmentation)
2022 First usable guidance for Level 2 AI/ML (human/machine collaboration)
2024 First usable guidance for Level 3 AI/ML (more autonomous machine)
2026 Finalized guidance for Level 1 and 2 AI/ ML
2028 Finalized guidance for Level 3 AI/ML
2029 Adapt to further innovation in AI
Phase I: exploration and first guidance development
Phase II: AI/ML framework consolidation
Phase III: pushing barriers
2019
2020
2021
2022
2023
2024
2025
2026
2027
2028
2029
2030
2031
2032
2033
2034
2035
EU INDUSTRY PROGNOSTIC
2019 First EASA AI/ML IPCs & applications
2025 First approvals of AI/ML
2030 Single-pilot CAT operations*
2035 Autonomous CAT operations*
* For Large Aircrafts, based on roadmaps from major players

Figure 1.1: Illustration of EASA's AI Roadmap, [EAS20, p.13].

EASA's roadmap distinguishes several levels of AI (from human assistance, to aid to decision, collaboration and autonomous systems), with a phased regulatory approach aligned with industry. The initial phase targets the development of a first set of guidelines, while the second one would use these to develop regulations, acceptable means of compliance and guidance material. A third phase would adapt these outcomes to future developments in AI, towards fully autonomous operations.

1.1 The first Daedalean/EASA IPC (CoDANN)

The first Daedalean/EASA Innovation Partnership Contract (IPC), titled *Concepts of Design Assurance for Neural networks (CoDANN)*, was one of the starting projects of the first phase of the roadmap.

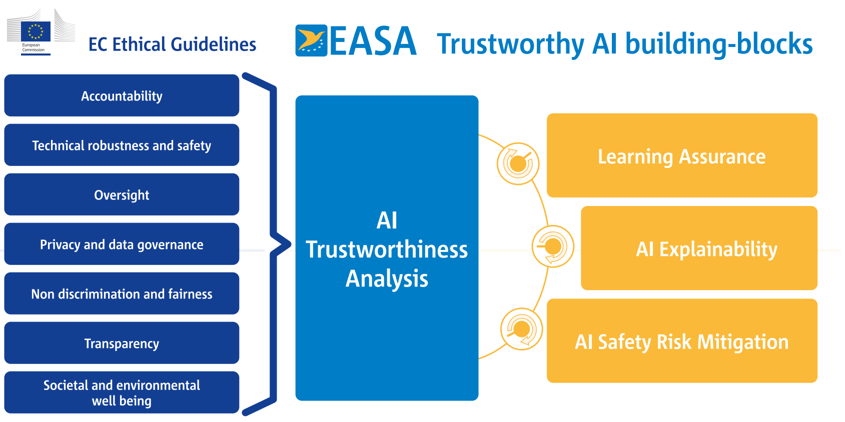

EC Ethical Guidelines
EASA Trustworthy AI building-blocks
Accountability
Technical robustness and safety
Oversight
Privacy and data governance
Non discrimination and fairness
Transparency
Societal and environmental well being
AI Trustworthiness Analysis
Learning Assurance
AI Explainability
AI Safety Risk Mitigation

Figure 1.2: Trustworthy AI building-blocks from [EAS20, Figure 5].

The main goal of the 10-month joint project (July 2019-March 2020) was to

> "*examine the challenges posed by the use of neural networks in aviation, in the broader context of allowing Machine Learning (ML) and more generally Artificial Intelligence (AI) on-board aircraft for safety-critical applications.*"

The focus was put on the *Learning Assurance* and *AI Trustworthiness analysis* building-blocks of the first EASA AI Roadmap (see Figure 1.2), and some of the major outcomes were (see the Executive Summary of [CoDANN20]):

- The definition of the *W-shaped Learning Assurance process* (see Figure 1) as a foundation for future guidance for machine learning applications. It provides an outline of the essential steps for Learning Assurance and their connection with traditional Development Assurance processes.
- The investigation of the notion of generalization of neural networks, with related aspects such as data quality, training, evaluation, verification, etc.
- The approach to accounting for neural networks in safety assessments, on the basis of a realistic use case.

This was discussed thoroughly in a 135-page report, with a 104-page public extract published in March 2020 [CoDANN20].

Most of the considerations are generic and apply to all supervised learning[1] methods, but particular attention is given to (deep) neural networks, as they represent one of the techniques that are both most promising and most complex.

The work happened in parallel with other efforts such as SAE G-34/WG-114 or [UL-4600], and the report contains a comparative survey of those [CoDANN20, Chapter 3]. One year after the publication of the first IPC, a whitepaper has also been published by the DEEL certification workgroup [Wor21] on similar topics, with compatible findings.

The report has been cited multiple times in relevant publications since then (e.g. [For+20; Wor21; Asa+20; Dev+21; Sch+20]).

[1] Supervised learning aims at finding an approximation (model) $\hat{f} : \mathcal{X} \to Y$ to a complex (i.e. that cannot be easily implemented) function $f : \mathcal{X} \to Y$ by using data pairs $(x, f(x))$, or $(x, f(x) + \delta_x)$, where δ_x is a small quantity representing noise in collecting values of f (e.g. from measurement/annotation).

1.1.1 The W-shaped process

The steps of the W-shaped process (illustrated in Figure 1) are:

- *Requirements management* (top left) and *Requirements verification* (top right), covered by traditional system development [ED-79A/ARP4754A].
- *Data management*, where datasets for training, validation and evaluation are created according to the requirements. This might include collection, annotation/labeling, and processing.
- *Learning process management*, which includes all the steps required prior to the training (next step): metrics, strategy to use for model selection, models/architectures to evaluate as well as the setup of software/hardware environment where the actual training takes place.
- *Model training* is self-explanatory, driven mostly by the previous step, in an iterative training/validation cycle, to find a best-performing model (architecture/hyperparameters).
- *Learning process validation*, where the outcome of the previous step, a single trained model, is evaluated on the test dataset[2]. This evaluation includes understanding generalizability (performance guarantees) and failure cases, which can then be fed to a safety assessment.
- *Model implementation*, which includes all the steps required to run the model obtained on the software/hardware platform, that usually differs from the development platform.
- *Inference model verification and integration*[3], where the desired properties of the deployed model are verified, including:
  - Typical software verification aspects such as execution time, memory/stack usage, etc.
  - Performance (in the sense of accuracy) guarantees with respect to metrics from requirements. If these are derived from the trained model, it is in particular fundamental to understand the impact of the transformation in the previous step, as well as that of the development software/hardware platform (whose impact goes beyond writing and compiling source code).
- *Independent data and learning verification*[4], meant to close the data management lifecycle, ensuring that data was correctly used throughout, and corresponds to the requirements (completeness/representativity, see [CoDANN20, Chapter 6]).

In addition to the use cases discussed in the two Daedalean/EASA IPCs, the W-shaped process has been successfully tried in other aviation-specific applications (as reported in [Eur21]).

1.2 Scope of the CoDANN II project

The goal of this second Daedalean/EASA joint project, that took place between July 2020 and May 2021, was to continue developing the *Learning assurance* and *Trustworthiness analysis*

[2]Note that the phrasing in [CoDANN20, p.44] is somewhat misleading: the training/validation iterative process takes place in the previous phase (model training), with no involvement of the test set. While the evaluation of multiple models on the test set are possible, they must be carefully controlled to avoid invalidating generalization guarantees.

[3]Called *Inference model verification* in [CoDANN20]. This was changed in [EAS21].

[4]Called *Data verification* in [CoDANN20]. This was changed in [EAS21].

building-blocks of the EASA AI Roadmap (Figure 1.2), as well as introducing the *Explainability* one. The main topics addressed are:

- *Implementation and inference*, to refer to parts of the W-shaped process related to the software and hardware platforms where models are developed (training, evaluation) and deployed.
  The first project focused mostly on considerations around generalization/performance guarantees abstracted from the development and deployment environments. However, these two environments are often complex platforms that bring additional risks to evaluate and mitigate in the context of the new paradigm of machine learning, where data and code drive the function instead of code only.
- *Explainability*, which is one of the other "AI trustworthiness" building-blocks from EASA's AI Roadmap (see Figure 1.2). "Explainability" and "interpretability" bear many different (sometimes even conflicting) meanings between authors, groups and fields. Two important steps required towards regulatory guidance are to:
  - Survey existing techniques fitting in the field of "explainability/interpretability" of machine learning/neural networks.
  - Identify possible gaps in the concept of learning assurance that these techniques could fill, if any, and discuss possible requirements.
- *Integration of machine-learning subcomponents into a complete system*.
  The first project analyzed how machine learning models integrate into a full system, with generic considerations as well as the use case of a landing guidance system. In particular, [CoDANN20] discussed how machine learning generalization guarantees, complemented by runtime monitoring, filtering etc., would fit into a safety assessment process. Three topics already present in [CoDANN20], but that were deemed to deserve additional considerations due to their importance are:
  - *Runtime monitoring/Out-of-distribution detection*. Ensuring that the input data fits the right distribution (the one identified during development from requirements) is an essential prerequisite to guarantee performance on unseen data.
  - *Integration with filtering/tracking*, including categorization of errors and measure of uncertainties.
  - *DAL level assignments* between the neural network and other components, with definition of the hazards and Failure Conditions to which the neural network contributes.

Use case A new use case of visual traffic detection is presented, complementing the visual landing guidance from [CoDANN20], helping mature both the concepts from [CoDANN20] and the new ones developed therein.

This system developed by Daedalean has been selected for three main reasons: the safety benefit such functions may bring to future airborne application, the complexity of the use case representative of future AI/ML products, and last but not least because this new kind of application is a potential technological enabler of more autonomous aircraft.

Outcome Thanks to the first Daedalean/EASA IPC [CoDANN20] and the current project, all steps of the W-shaped process have now been investigated. Points of interest for future research activities, standards development and certification exercises have been identified, and they will contribute to stir EASA efforts toward the introduction of AI/ML technology in aviation.

1.3 EASA's First usable guidance for Level 1 ML applications

Building on [CoDANN20], the current project, other industry collaborations, and internal work, EASA opened for consultation in April 2021 a first usable guidance for Level 1 machine learning applications [EAS21]. The document

> *"presents a first set of objectives for Level 1 Artificial Intelligence ('assistance to human'), in order to anticipate future EASA guidance and requirements for safety-related ML applications.*
>
> *It aims at guiding applicants when introducing AI/ML technologies into systems intended for use in safety-related or environment-related applications in all domains covered by the EASA Basic Regulation (Regulation (EU) 2018/1139).*
>
> *[. . .]*
>
> *It will serve as a basis for the EASA AI Roadmap 1.0 Phase II ('AI/ML framework consolidation') when formal regulatory development comes into force."*

Several use cases are reviewed, including the one from [CoDANN20] (visual landing guidance).

Link with CoDANN II The development of EASA's concept paper was run in parallel with the current project.

Both activities benefited from each other; the discussions in this project contributed to enrich the development of EASA's guidance with the experience from actual AI/ML product developers.

As the CoDANN II project is now finalized, it is foreseen to integrate the relevant aspect of the visual traffic detection use case as an example in the updated EASA Level 1 Guidance.

1.4 Outline of the report

As in [CoDANN20], the majority of the report is generic to machine learning applications in safety-critical settings (with a focus on neural networks).

However, a specific use case is first presented in Chapter 2 to help the reader contextualize the more abstract discussions in the other chapters. Visual traffic detection was chosen for the interest of the application and inherent challenges. This is also one of Daedalean's products.

Chapter 3 investigates the two environments implicitly present in the W-shaped process (*learning* and *inference* environments) and their interactions:

- Separate sections discuss the specificities and potential risks of each, followed by an overview of what is required to pass from the first to the second.
- Then, a survey of the most important optimizations often needed to use neural networks in real time is provided, again outlining possible challenges.
- Finally, the last sections discuss possible ways to ensure that performance guarantees are maintained for the operational model/platform despite potential major transformations and risks related to the learning environment.

Chapter 4 starts with a discussion on the polysemy of the words "explainability" and "interpretability", noting through academic surveys that they have many, sometimes conflicting, meanings. A definition that is generic enough to encompass all the works of regulatory interest is adopted, therefore avoiding the pitfall of creating yet another definition. A categorization of techniques from academic literature is presented, before surveying major methods and working groups, as well as criticisms and issues of some methods. Given this information, the chapter concludes a reflection on the usefulness of these techniques, and requirements that should be set. Two major uses are identified, matching with [EAS21, Section 4]:

- Strengthening the data–learning assurance link (in particular ensuring that the operating space has been correctly identified), during development and post operations.
- Human-machine interaction, during operations.

Chapter 5 treats various topics around safety assessment:

- Out-of-distribution detection and runtime monitoring, elaborating on Chapter 6 from [CoDANN20].
- Integration with classical filtering/tracking, and the required information to do so on the errors made by the neural network component.
- Connected to the previous items, the estimation and validation of uncertainty of neural networks.

Chapter 6 returns to the use case in the context of the topics discussed in the previous chapters (implementation and inference, Concepts of Operations. . .), with a particular focus on integration into a full system and the related safety assessment.

Finally, Chapter 7 provides a summary of the findings, suggested guidelines, and discusses remaining elements that might be required for certifying Level 1 applications (in the context of [EAS20]), as well as possible next steps and future work.

The notations and conventions from [CoDANN20] will be used throughout.

A note about this document

This document is a public version of the original report.

Some details from the original have been removed for confidentiality reasons (appendices and where indicated in the text).

Interested parties are welcome to contact ipc-feedback@daedalean.ai.

Chapter 2

Concepts of Operations

The application described in these Concepts of Operations (ConOps) is a *vision-based system that detects, tracks and classifies airborne objects such as fixed-wing aircraft, rotorcraft and drones*. It is designed to be integrated within an aircraft to support a Detect and Avoid (DAA) function. Figure 2.1 shows an example output in a form used during development.

While an existing DAA function, based for example on Traffic Collision Avoidance System (TCAS), might provide information on cooperative traffic and relies on other aircraft to have complementary systems, this system requires nothing to be installed in other aircraft allowing non-cooperative traffic to also be identified. This system is intended to supplement an existing DAA system, not replace it.

The use case presented here is to help the reader anchor the considerations in the following generic chapters in a practical application. At the end of the report, Chapter 6 will return to the use case to apply and illustrate the previous chapters.

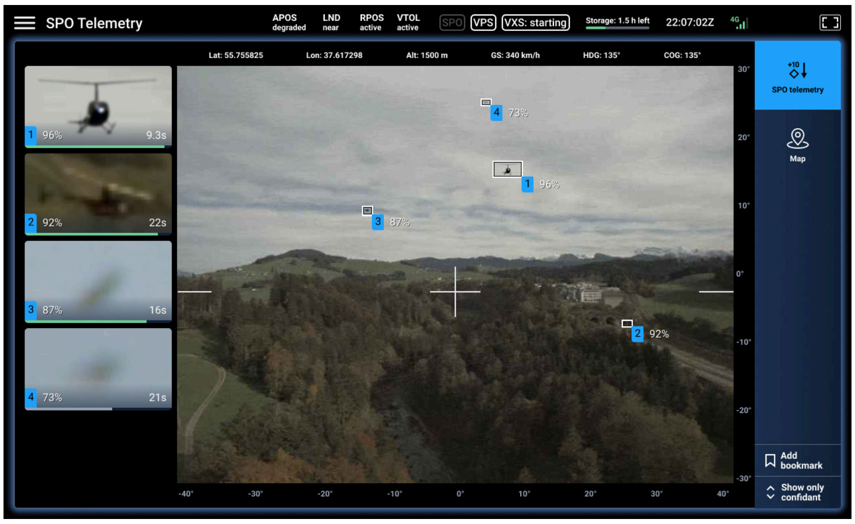


Figure 2.1: Illustration of Daedalean's visual traffic detection system. A 12-megapixel image captured from a camera on an aircraft contains multiple aircraft, successfully detected by the system and displayed as a zoomed-in inlet on the left, in addition to some false positives. Note that this is an internal demonstration user interface, not a proposed display for use by an aircraft's crew.

2.1 System description

One or more avionics-grade cameras are used to capture images of the surrounding space, with as little obstruction as possible. The captured images are then processed by the system to extract all objects (cooperative or non-cooperative traffic) present in the field of view, with relevant information such as category, time-to-collision, etc. This information can be fed to a pilot/operator and/or another system (e.g. to perform an avoidance maneuver when necessary).

Like the system analyzed in [CoDANN20], the proposed visual detection system is a combination of a camera unit, a pre-processing component, a neural network and a tracking/filtering component. See Figure 2.2.

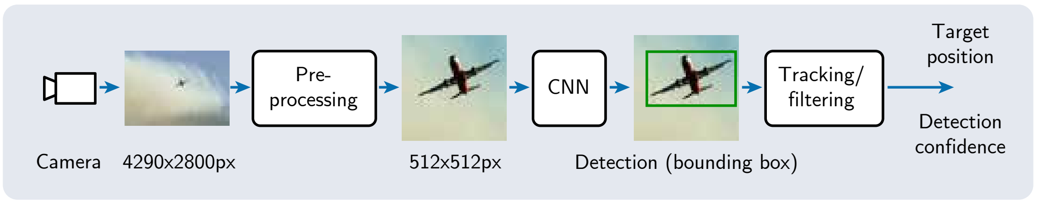


Figure 2.2: Architecture overview.

For the purposes of this document, only one camera is considered, assumed to be fixed on the nose of the aircraft and pointing forward. In practice, the system will likely have multiple cameras for improved field-of-view, robustness, and redundancy.

The next paragraph describe the system outputs more precisely, as well as the different components of the system.

2.1.1 Outputs

The system outputs the following timestamped information at a fixed frequency (e.g. 1 Hz), for consumption by other aircraft systems and/or aircrew (see also [DO-365B; DO-387]):

- A list of current tracks, each with the following information:

Item	**Units**
Track identifier	Increasing integers
Category	Fixed-wing, rotorcraft, drone. . .
Time-to-collision (or τ_{mod})	Seconds
Relative position	Meters
Size	Meters
Output type	Measured or extrapolated
Corresponding uncertainties	–
Target crop (see Section 6.5)	–

  A *track* is a series of measurements that should correspond to a single airborne object in the field of view.

- Status information indicating the health of the system.

2.1.2 Sensing/pre-processing

Sensor The camera unit is assumed to have a global shutter and output 12-megapixel RGB images at a fixed frequency.

Pre-processing The pre-processing component applies a number of operations to the raw camera image data in order to improve the suitability of the image for subsequent processing. This includes operations such as:

- Masking out parts of the image that should not be used for further processing (for example, image regions showing parts of the aircraft itself rather than the environment);
- Normalizing the image channels to a fixed mean and standard deviation;
- Splitting full resolution images into 512×512 tiles for the neural network to process (batching, usually constrained by hardware memory limitations).

This is done with classical software (i.e. no machine learning).

2.1.3 Neural network

A machine learning model operates on the input space $\mathcal{X}$ consisting of 512×512 RGB images obtained after the pre-processing (normalized tiles).

Its output space Y consists of sets of bounding boxes (that should each correspond to an airborne object), with axis aligned to the image, each having:

Quantity	**Properties**	**Range**
Target bounding box	Normalized coordinates of top-left corner Normalized width and height	$[0, 1]^2$ $[0, 1]^2$
Target category	Among a fixed list	See Section 2.1.1
Confidence value		$[0, 1]$

A Convolutional Neural Network (CNN) as shown in Figure 2.3 is a natural choice for this application.

Usually, the "confidence value" and "object category" outputs are implemented by producing a confidence for each class plus a "no object" class. The category is obtained as the one with the largest confidence, and this provides a measure of certainty of both the presence of an object, and of the assigned category. The choice to make a detection and the "confidence" output can then be derived according to the requirements of the next systems. Table 2.1 gives several examples.

Output #	Fixed-wing	Rotorcraft	Drone	Unknown	No object
1	0.9	0.05	0	0	0.05
2	0.4	0.2	0.1	0.1	0.2
3	0.45	0.45	0	0	0.1
4	0.025	0.025	0.025	0.025	0.9

Table 2.1: Neural networks outputs with different per-class probabilities. See Section 5.4 for a discussion around calibration of uncertainties (and their exact definitions).

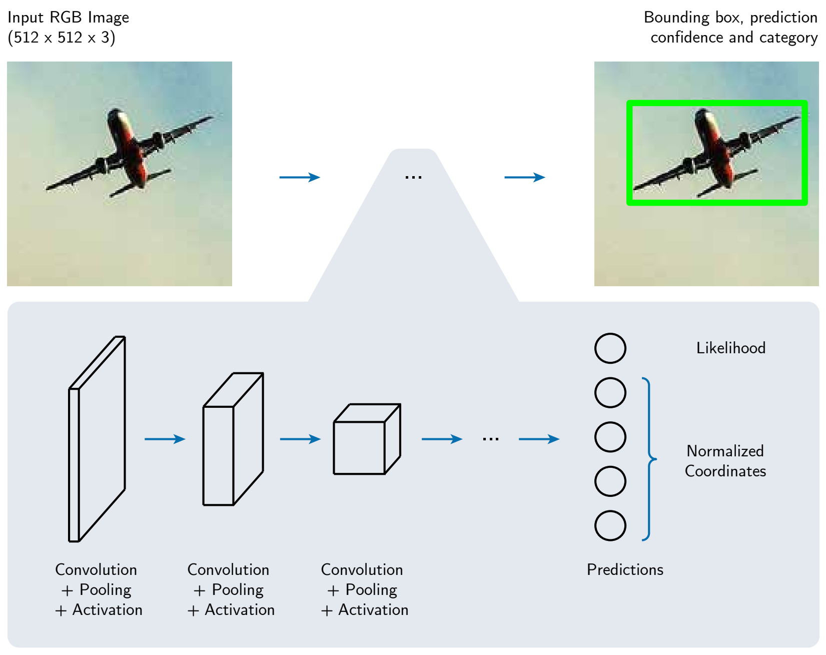

Input RGB Image
(512 x 512 x 3)
Bounding box, prediction confidence and category
...
Convolution + Pooling + Activation
Convolution + Pooling + Activation
Convolution + Pooling + Activation
...
Likelihood
Normalized Coordinates
Predictions

Figure 2.3: Convolutional neural network for object detection.

2.1.4 Tracking/filtering

The tracking/filtering unit post-processes the neural network output ("detections") to turn single frame detections into *tracks*. A track is a collection of single detections from frames with strictly increasing timestamps, with system requirements asking that each airborne object be associated with a unique track during its presence in the field of view of the system.

See also [DO-365B, Appendix F] for tracking in the context of radar.

Filtering At the very least, the movement of the ownship between previous and current frame should be subtracted from the detections, so that the system can distinguish between a target moving and the aircraft moving around a stationary target.

Depending on the kind of neural network used, a Non-Maximum Suppression (NMS) algorithm might be used to first merge overlapping boxes.

The filtering might also be used to reduce noise/variance from the neural networks (see Section 5.3), possibly combining it with a target movement model (within the tracking below).

The height above ground received from aircraft systems might also be used to suppress outputs below a specified height.

Tracking On a fundamental level, the tracking algorithm does the following (see the representation as a state machine in Figure 2.4):

- Track creation: create a new tentative track from a single-frame detection not currently part of a track. These tracks are not part of the outputs.
- Track confirmation: promote a tentative track to a confirmed one, which will be part of the outputs.
- Track association: associate a single frame detection with a current track;

- Track deletion: for tentative tracks that do not get confirmed, or for confirmed tracks that do not get associated with detections anymore.
- Track extrapolation: to handle the case where the neural network has a lower frequency than the desired system frequency, or when detections are missed on some frames, the tracker might extrapolate previous detections according to a 2D or 3D movement model.

In addition to managing tracks, the tracker reduces false positives and false negatives.

An example algorithm will be discussed in Chapter 6.

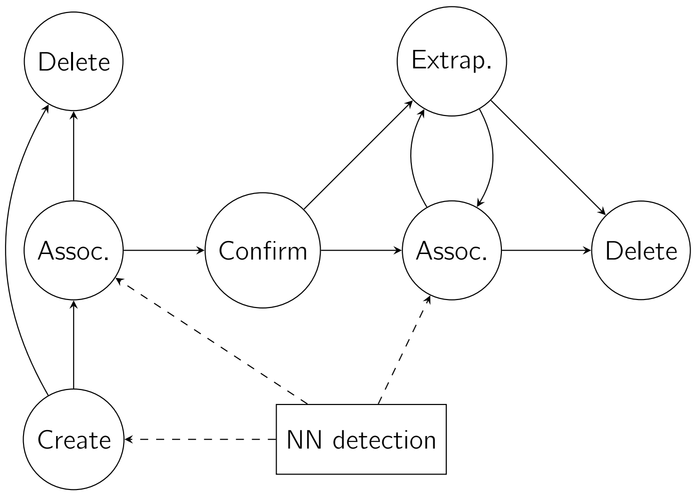


Figure 2.4: The tracker as a state machine.

Implementation As with pre-processing, post-processing is also implemented with "classical software". Even though there exist machine-learned tracking approaches [BML19; Xu+20], decoupling concerns and implementing the non-perception part without machine learning makes the safety analysis easier.

2.2 Operating conditions and performance

Table 2.2 proposes two operational concepts for our visual traffic detection system:

- Operational concept 1 where the traffic information is solely used by the pilot for situational awareness ("Pilot Advisory");
- Operational concept 2 where the traffic information is provided to a fully autonomous flight control system.

	Operational Concept 1	**Operational Concept 2**
Application	Traffic detection	
Aircraft type	General Aviation [CS-23] Class IV[1], Rotorcraft [CS-27; CS-29] or eVTOL [SC-VTOL-01] (cat. enhanced)	
Flight rules and weather conditions	Visual Flight Rules (VFR) in daytime Visual Meteorological Conditions (VMC)	
Class of airspace	Airspace class D, E, F and G [ED-258]	
Level of automation	Pilot advisory	Full autonomy
System interface	Glass cockpit flight director display	Flight computer
Phase of flight	In flight	
	Relevant Operating Parameters	
Minimum ownship altitude	300 ft AGL	
Maximum ownship altitude	At least 15'000 ft AGL and 25'000 ft AMSL	
Temperature range	To [ED-14G/DO-160G, Cat B4] Operating: $-15°$ C to $+55°$ C: Short term: $-20°$ C to $+70°$ C Ground survival: $-25°$ C to $+85°$ C	
Ownship pitch angle	$-30° \dots 30°$	
Ownship roll angle	$-60° \dots 60°$	
Field of view (elevation × azimuth)	$66° \times 220°$	
Time of day (sun position)	1 hour after sunrise to 1 hour before sunset	
Minimum detection distance	10 m	

Table 2.2: Operational concepts.

[1]See the classes defined in [FAA-23.1309-1E, Section 15].

2.3 Aircraft integration

2.3.1 DAA integration

As described above, the visual traffic detection system is integrated into an aircraft's existing DAA system. Track data is provided which is then merged with other traffic data, for example from a TCAS system, before being presented to the pilot (in an advisory system) or fed to a flight control system (autonomous flight). Figure 2.5 presents a simplified view of how the visual traffic detection system integrates with aircraft systems, only connections directly related to the ConOps are shown (e.g. maintenance connections are not shown). The Height Above Ground Level (HAGL) input is used to determine if the output should be inhibited, which is indicated to the pilot or flight control system via the status data output. Power is provided to the camera from the computing platform, not by a direct connection to the aircraft electrical distribution system.

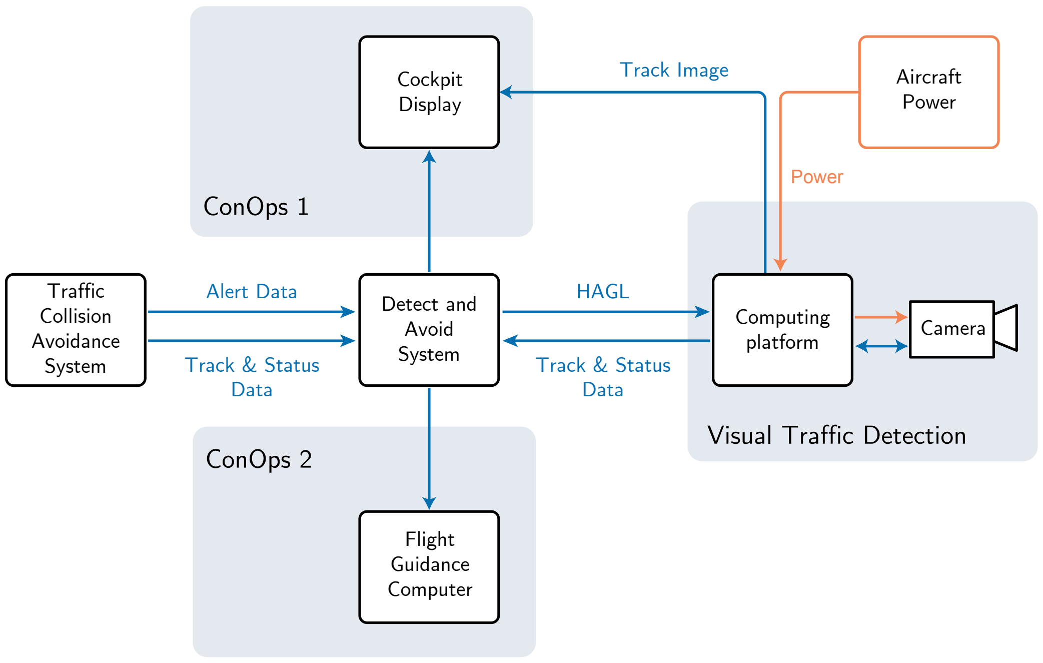


Figure 2.5: Aircraft integration.

2.3.2 Visualization of targets (ConOps 1)

When the visual traffic detection system is integrated within an aircraft as a pilot advisory system (operational concept 1) the display system shall use symbology which differs from that used for standard TCAS traffic so the pilot can mentally separate between them (see also [CM-AS-010]). An example of symbology that could be used for TCAS and non-TCAS traffic is shown in Figure 2.6.

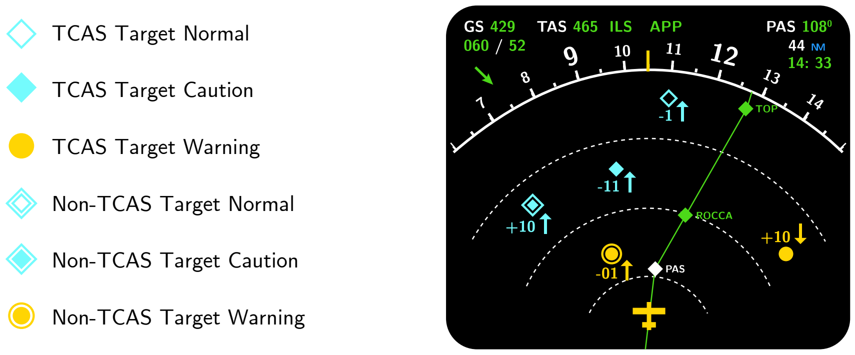


Figure 2.6: How TCAS and non-TCAS traffic might occupy the same display.

In addition to standard traffic information, thumbnail images of visually identified tracks are sent from the visual traffic detection system to the cockpit systems. These may be presented to the aircrew to aid them in visually acquiring traffic and confirming that visually sourced track data is correct. The thumbnail image is provided purely as an aid, the lack of which does not impair the ability of the aircrew to take appropriate action.

2.3.3 Integration with Flight Guidance Computer (ConOps 2)

In the case of a fully autonomous system the visualization of target information is clearly not a concern. However, it is important that the host system consumes and reacts appropriately to status information provided by this system. In particular degradation of the system performance due to malfunction or external factors (such as poor light) should result in an upper level system taking appropriate action to restore appropriate safety margins.

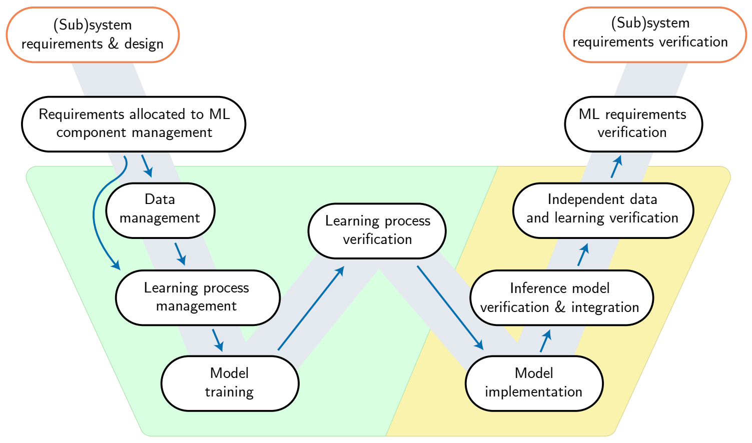
Chapter 3

Learning/inference environments and model implementation

3.1 Introduction

The goal of this chapter is to address the following two important topics remaining from the first report [CoDANN20]:

- The *inference phase* of machine learning components, namely the right-hand side of the W-shaped process (Figure 3.1) up to *Independent data and learning verification*;
- Considerations around supporting tools, software and hardware used throughout the W-shaped development cycle (i.e. both during training and inference).



(Sub)system requirements & design
(Sub)system requirements verification
Requirements allocated to ML component management
ML requirements verification
Data management
Learning process verification
Independent data and learning verification
Learning process management
Inference model verification & integration
Model training
Model implementation

Figure 3.1: The learning (green) and inference (yellow) environments in the W-shaped development cycle for Learning Assurance from [CoDANN20].

The main questions that will be addressed are:

- What are the characteristics (hardware and software) of the inference environment (operational), and how do these differ from "traditional" avionics? What is needed to run neural networks in safety-critical settings, and which platforms might fill these requirements?

- What are the characteristics (hardware and software) of the learning/training environment, and how do these differ from a "traditional" development environment.
- What steps are necessary to pass from the learning to the inference environment? For example, is there an equivalent to compilation? What risks do operations such as model optimizations (quantization, pruning, etc.) carry? How to ensure that learning assurance/generalization is preserved on the inference environment?

3.1.1 The learning and inference environments

As already indicated by the W-shaped process, machine learning models[1] will exist on two different environments (including both software and hardware stacks) during their development and deployment, with different functions and levels of criticality.

- The *learning environment*, in which the evaluation and the training take place, in the left-hand side of the W-shaped process.
- The *inference environment*[2], in which the neural network is provided real time data and produces outputs forwarded to the rest of the system. This is the operational platform where the machine learning model will be deployed. This environment is considered in the right-hand side of the W-shaped process.

Table 3.1 summarizes the features required in the two environments for the case of neural networks.

Minimum ability and support expectation	**Learning environment**	**Inference environment**
Data throughput	Large	Small (real time)
Forward pass	☑	☑
Backward pass	☑	☐
Produce performance metrics	☑	☐
Runs safety-critical applications	☐	☑
Intention to run real-time	☐	☑
Accelerator availability	Varied	Custom

Table 3.1: Example set of minimum abilities for training and running neural networks, and the expectation on whether (and how, if applicable) the ability is supported on each environment.

[1]Here and throughout, a (machine learning) model will refer to an approximation $\hat{f} : \mathcal{X} \to Y$ of a function $f : \mathcal{X} \to Y$, as in [CoDANN20, Chapters 5-10]. In particular, this should not be confused with "model-based development" from [ED-218/DO-331].

[2]"Inference" is commonly used for "computing the output of a neural network on a given input", which takes place on both environments. However, "inference environment" will be used exclusively for the operational/in-flight environment.

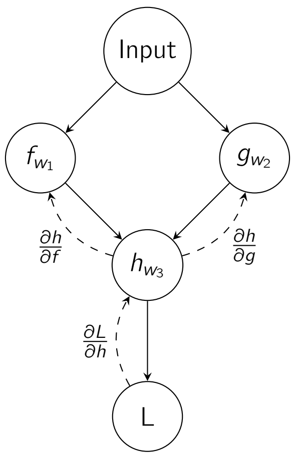

Input
f_{w_1}
g_{w_2}
h_{w_3}
$\frac{\partial h}{\partial f}$
$\frac{\partial h}{\partial g}$
$\frac{\partial L}{\partial h}$
L

Figure 3.2: Forward pass through a neural network (used for training and inference) and backward pass (dashed arrows, used for training only). The functions f, g, h depend on the weights w_1, w_2, w_3 shown as indices, while L is a loss function. The backward pass will compute $\frac{\partial L}{\partial w_i}$, where w_i are the learnable weights. For example, $\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial h}\frac{\partial h}{\partial f}\frac{\partial f}{\partial w_1}$, which is computed bottom-up (hence the "backpropagation"/"backward pass" terminology).

Both environments need the ability to evaluate the model and produce outputs for given inputs. The major differences are that (see also Figure 3.2):

- The learning environment needs the ability to train the model (backpropagation, update of weights, computation of errors);
- The inference environment has the requirements of the operational platform (safety-critical, real-time. . .).

Even though the learning environment is not used during operations, it still needs to be considered carefully, given that the training that produces the final model happens there.

Using the same environment for learning and inference In principle, the two environments could use the same hardware and software stack, but this is unlikely given their different requirements. Typically, the learning environment would use commercial off-the-shelf hardware and software, such as compute clusters with GPUs, software such as TensorFlow, etc., while the inference environment would use custom hardware and software.

The situation might be different for applications with lower requirements on the inference environment such as ground operations or tooling. While this will not be the focus in this chapter, most of the content will apply in both cases.

3.1.2 Structure of the chapter

Addressing the two topics from the start of the chapter will require getting a deeper understanding of the two environments (in Sections 3.2 and 3.3) and the passage between them (Sections 3.4 and 3.5), in particular with respect to learning assurance (Sections 3.6 and 3.7).

As in the rest of the report, the focus will be on (deep) neural networks, which is especially important given that these might require specific hardware and software compared to "classical" machine learning methods.

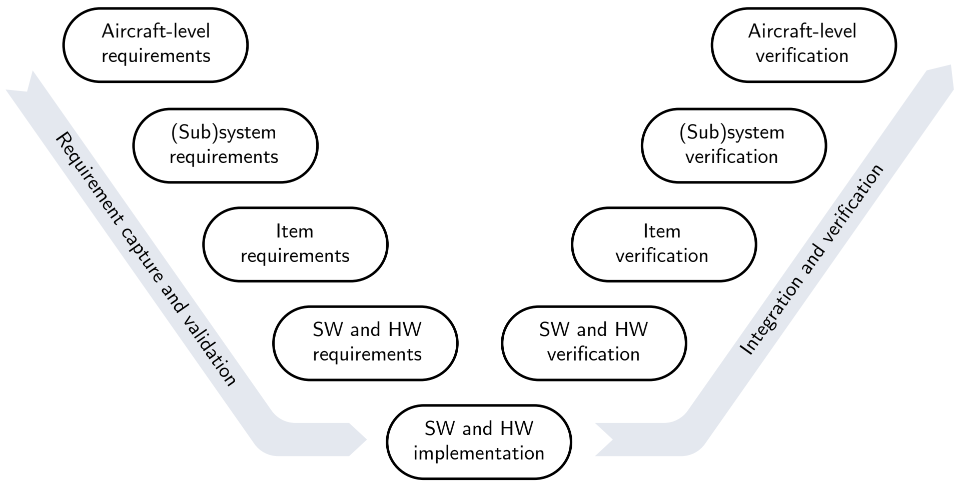


Figure 3.3: V-shaped model for system development of [ED-79A/ARP4754A]. For machine learning components, learning assurance has to be performed in addition to hardware and software assurance. Adapted from [RL13].

3.2 Considerations on the inference environment

The inference environment is the operational platform that will be used in-flight to run the neural network (see Section 3.1.1). It receives live data and produces output that is forwarded to other system components and, in the end, to the pilot. As in [CoDANN20], offline learning is assumed: no changes to the neural networks will be performed during operation/at inference time.

This section discusses considerations and choices for this environment as a safety-critical application that is subject to guidance such as [ED-79A/ARP4754A; ED-14G/DO-160G] and [AMC 20-152A; AMC 20-115D].

3.2.1 System considerations

The inference environment as an overall system or subsystem needs to be planned under a design process such as [ED-79A/ARP4754A] (see Figure 3.3), under which requirements captured and validated for the (sub-)system are systematically allocated to the hardware, the software, as well as the machine learning component. After implementation, requirements are verified, at first on the item level and then, upon integration, on the system level.

The inference environment will for example need to be sufficiently robust, provide sufficient throughput to handle the incoming data, and satisfy limits on the variability of the execution time and the Worst Case Execution Time (WCET). Requirements of the (sub-)system will influence the choice of the machine learning model, the implementation, as well as the selection of the hardware platform. The machine learning model selected will have to find the right trade-off between average execution time and variation in execution time. A recurrent neural network with a variable number of outputs may for example not be suitable for a use case with a hard limit on the execution time. In addition, the soft- and hardware implementation will have to provide a performant implementation of the selected algorithm.

3.2.2 Hardware considerations

A coarse calculation using the data presented in [GLL19] shows that an object detector based on a deep neural network running at 6 frames per second on 12-megapixel images from a single camera can require on the order of ∼10 TOPS in either integer or floating-point performance and can have on the order of 100 million parameters. Implementing a real-time inference environment for such a model is a challenging problem even without considering aviation-specific certification standards such as [AMC 20-152A; ED-14G/DO-160G].

The hardware components typically used for inference can be classified into two main categories:

- *Commercial-off-the-shelf (COTS) hardware* such as commercially available Central Processing Units (CPU) and Graphics Processing Units (GPU) and
- *Customizable or custom hardware* either in the form of logic implemented in Field Programmable Gate Arrays (FPGA) or in the form of Application Specific Integrated Circuits (ASIC).

Commercial off-the-shelf hardware CPUs are generally aimed at performing a wide variety of sequential calculations with minimal latency. Although there is a trend towards providing new instructions that accelerate the typical calculations required by neural networks such as Single Instruction Multiple Data (SIMD) instructions, the performance of CPUs still lags behind the performance of GPUs for these applications.

While off-the-shelf CPUs are available with [AMC 20-152A] certification evidence, they currently do not offer sufficient computational power to run neural network required for computer vision applications. They are however suitable and a preferred solution to run the post-processing computations on the raw output of the neural network, such as object tracking (see Section 2.1.4).

GPUs offer significant computational power for both floating-point and integer operations through massive parallelism at the cost of generality. They are flexible and easy to use due to the libraries and software stacks mentioned in Section 3.3 and therefore a tool of choice when it comes to training and running models such as deep convolutional neural networks used for computer vision.

However, [AMC 20-152A] does not fully cover the specificities of GPUs. Establishing certification evidence as a third party is made difficult by the proprietary nature of the information on the underlying hardware, the drivers, and other tools. This is compounded by abstraction layers that are in place to facilitate cross-device compatibility and simplify the programming of the devices and whose implementation often cannot be inspected.

Current GPUs often consist of many thousand shader processing units managed by complex job scheduling algorithms. These units share hardware-managed caches, software-managed (but hardware-arbitrated) local/global memory banks, as well as register files. Therefore, these GPUs can clearly be defined as multi-core platforms and their core partitioning characteristics are largely considered to be sensitive trade secrets by their vendors. Therefore, care needs to be taken to make sure the considerations discussed in the [CAS16] and the [AMC 20-193] guidance are properly addressed.

In the case of NVIDIA GPUs, an example of such an abstraction layer with a proprietary implementation is CUDA PTX [CUDAPTX], a "low-level virtual machine and instruction set architecture (ISA)". While PTX itself is documented, the mapping from PTX instructions to the actual instructions executed on the device is performed by proprietary functions and may depend on the target hardware.

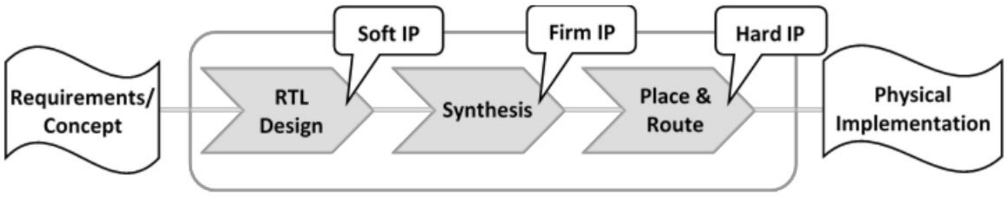


Figure 3.4: Custom hardware can make use of different types of commercial IP. Depending on the type of IP, integration will occur at a different point of the typical hardware design flow as shown in [AMC 20-152A].

In the case of GPUs programmed using OpenCL, an example is SPIR-V [SPIRV] which intends to

> "*Provide a simple binary intermediate language for all functionality appearing in Khronos shaders/kernels*", "*Be the form passed by a client API into a driver to set shaders/kernels*", and "*Support multiple execution environments*".

While itself an open standard, the implementation for a specific GPU is again proprietary.

Customizable or custom hardware Field Programmable Gate Arrays (FPGAs) are prefabricated chips whose logic can be configured to user specifications. FPGA hardware may come with certification evidence for certain aspects of the hardware that is required. This leads to lower up-front costs when compared with ASICs.

Application Specific Integrated Circuits (ASICs) are fabricated with fixed capabilities to user specifications. They are more expensive up-front because of the high costs of lithographic masks, of the required tools, and the need to certify certain aspects of the created chip. A benefit of an ASIC implementation can be that a more efficient implementation of the same high-level logic is possible than in an FPGA.

An example for a custom ASIC for neural network inference is Google's Tensor Processing Units (TPUs) which provides specialized functionality for matrix-matrix multiplications, accumulation, activation functions, and normalization and pooling operations which are commonly used in deep convolutional neural networks [Jou+17].

Both FPGAs and ASICs offer the user the option to implement custom logic in a Hardware Description Language (HDL) such as Verilog [Verilog] or VHDL [VHDL]. This high-level description is translated into logic gates by a synthesis tool, which are in turn implemented in the hardware. Figure 3.4 shows the typical workflow to create custom logic in hardware.

There are different levels of customization possible, ranging from a fully custom logic to the use of commercially available building blocks. Commercially available building blocks can be divided into two subcategories:

- Building blocks that are available in a hardware description language ("Soft IP") and can be inspected. These are used on the same level as custom logic written in an HDL and can be verified with similar means.
- Building blocks that are available on the synthesis/hardware level ("Firm IP"/"Hard IP") and which typically do not offer any insight into the high level logic from which they were generated. They either have to provide the certification evidence needed or have to be assessed with the same methods as other commercial component.

The implementation of an inference environment for a neural network will likely need to retain a programmable model, i.e. Instruction Set Architecture (ISA), on top of the custom hardware

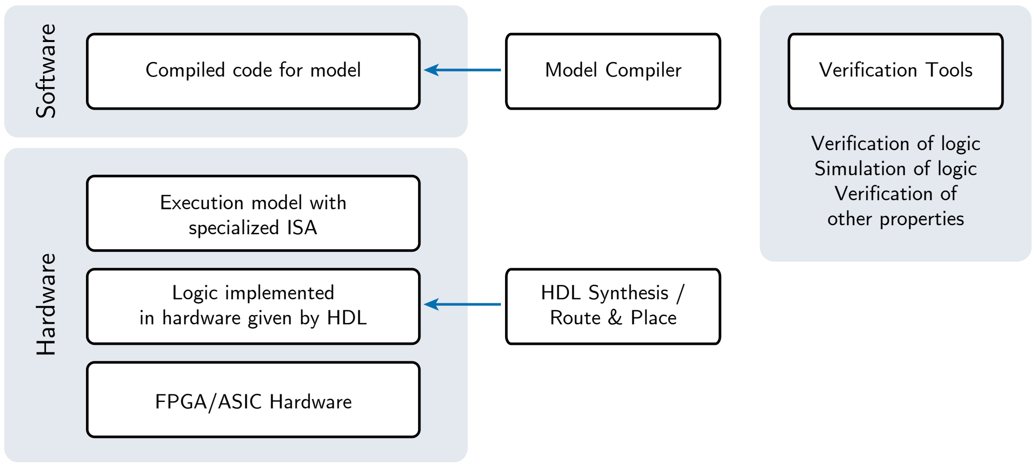


Figure 3.5: Implementation of the inference environment with an FPGA or an ASIC. Custom logic is implemented in a hardware description language. Custom logic likely represents an execution model that is specialized to the model calculations and has a custom instruction set. Instruction set is targeted by a model compiler.

independent of the implementation of the logic in an FPGA or an ASIC. While a statically defined state-machine may be optimal for a fixed and known application, it may be unfeasible for all but trivial applications. Programmability makes life easier during development by:

- Giving applications the flexibility to change.
- Supporting multiple applications on the same hardware.
- Trading memory used to store the instructions for size and complexity of the hardware design, thereby reducing the risk of failures.

In contrast to general purpose CPUs or GPUs, this ISA can be tailored more specifically to the application under consideration. It can also offer additional control over aspects such as memory hierarchy and caching, aspects that are rarely under explicit user control on a CPU or GPU. This programmable model will either be targeted manually or through a custom compiler. This will be discussed further in Section 3.2.3.

Summary COTS hardware that is sufficiently powerful for neural network applications faces steep challenges in terms of certification due to the proprietary nature of the hardware and driver components. This situation is somewhat better for CPUs than for GPUs. FPGAs and ASICs using either custom logic or Soft IP offer significantly more control and insight. This comes at the cost of having to implement and verify the custom logic and ISA on the chip.

The reader is also referred to [AMC 20-152A] for guidance relevant to COTS.

3.2.3 Software considerations

The software used in the inference environment can encompass software such as a Real Time Operating Systems (RTOS) which hosts the application on the inference hardware, drivers thereof, the application code running on the inference hardware itself, or software tools used to produce and verify the application code as illustrated in Figure 3.6. As already mentioned, the software discussions on the inference environment will focus on neural networks.

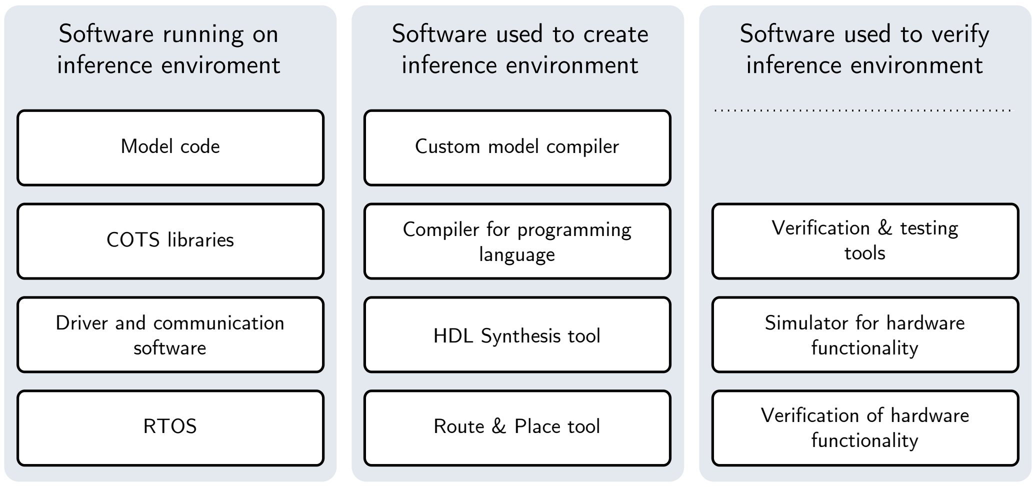

Software running on inference enviroment
Model code
COTS libraries
Driver and communication software
RTOS
Software used to create inference environment
Custom model compiler
Compiler for programming language
HDL Synthesis tool
Route & Place tool
Software used to verify inference environment
Verification & testing tools
Simulator for hardware functionality
Verification of hardware functionality

Figure 3.6: Examples of typical software components used in the context of the inference environment.

The software components in use in the context of the inference environment will depend on the choice of the underlying hardware. In the case of commercially available GPUs for example, the software will include compilers for the GPUs, drivers, and potentially libraries if used as shown in Table 3.1 (see also Section 3.3 below).

In the case of custom hardware, typical software components will include the synthesis/network routing tool for the hardware description language and the model compiler targeting the custom ISA in the hardware. Here, the model compiler is of particular interest as it is normally not a standard component when creating custom logic implementations for logic that is less complex and performance-relevant than the one in the inference environment.

Model compiler The term model compiler refers to the software that is used to create the instructions that are executed on the custom hardware. The compiler can work on the model description that will be introduced in Section 3.4.2. A practical example of such a model compiler is the [Che+18] framework, which can be used to generate optimized low-level code for a number of target platforms from high-level model descriptions. It is tasked with finding an efficient execution of the model on the target hardware. It can be complemented by verification tools for different purposes.

Process considerations In terms of process considerations, the development of the software should be done in accordance to [AMC 20-152A; AMC 20-115D].

3.3 Considerations on the learning environment

The learning environment is where the training of the neural networks take place (see Section 3.1.1). Like the inference environment, it is a combination of a hardware and a software stack.

3.3.1 Hardware stack

In the learning/development phase, it is important to use hardware that allows running a variety of models easily (e.g. different architectures). Unlike in the inference environment, there is no

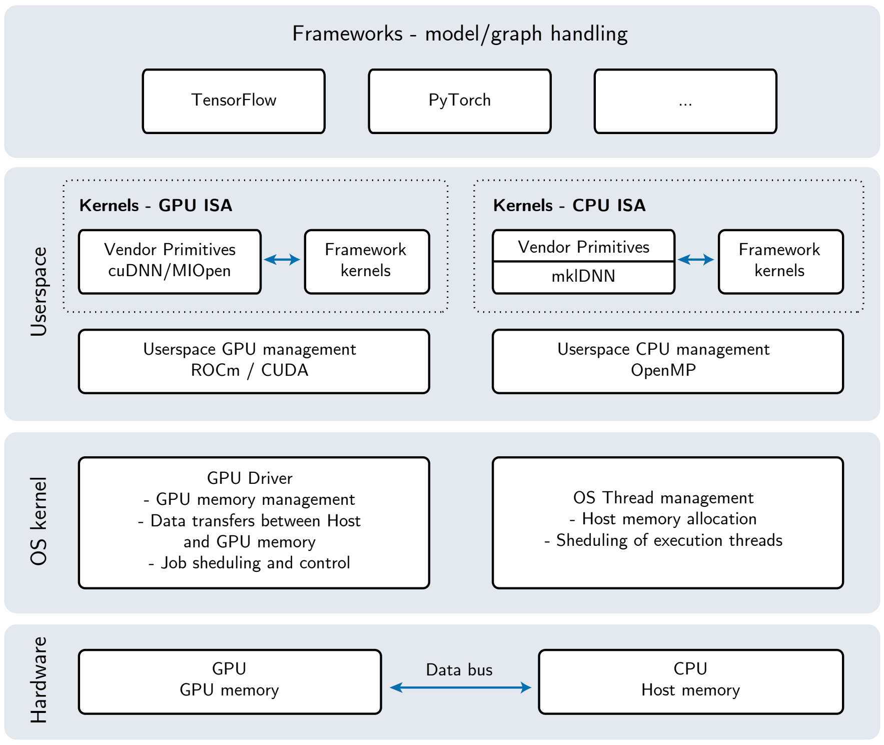

Frameworks - model/graph handling
TensorFlow
PyTorch
...
Userspace
Kernels - GPU ISA
Vendor Primitives cuDNN/MIOpen
Framework kernels
Kernels - CPU ISA
Vendor Primitives
mklDNN
Framework kernels
Userspace GPU management ROCm / CUDA
Userspace CPU management OpenMP
OS kernel
GPU Driver - GPU memory management - Data transfers between Host and GPU memory - Job sheduling and control
OS Thread management - Host memory allocation - Sheduling of execution threads
Hardware
GPU GPU memory
Data bus
CPU Host memory

Figure 3.7: Typical software and hardware stack in the learning environment.

requirement for real-time performance; on the other hand, the hardware should allow to process a large amount of data simultaneously, given that there might be millions of training steps. Moreover, the learning environment hardware should be capable of performing backward passes. See also Table 3.1.

Without considering certified systems, the hardware stack to train machine learning models usually consists of powerful machines (CPU/memory/disks) equipped with either GPUs or (less frequently) ASICs optimized for neural network operations rather than computer graphics (like GPUs originally are). Section 3.2.2 contained detailed descriptions of these components.

Often, these are networked to perform distributed training (i.e. training a single model or several models on several machines), which might happen on-premises or on cloud computing platforms such as Google Cloud Platform[3] (GCP) or Amazon Web Services[4] (AWS). Cloud computing has the advantage of on-demand resources, and some hardware might only be available on such platforms (like Google's TPU ASIC).

3.3.2 Software stack

Over the past decade, powerful software stacks (frameworks, libraries, tools, hardware drivers) have been developed that provide standardized implementations of common neural network components, training algorithms, and data transformation processes. A common software stack layout based on GPU-accelerated hardware is shown in Figure 3.7. Typically, the highest elements of the stack are hardware-agnostic/independent, while the bottom ones are closely tied to it (e.g. GPU drivers and libraries).

[3]https://cloud.google.com/
[4]https://aws.amazon.com/

Layer	Examples
Top-Level Framework	TensorFlow (Apache), PyTorch (BSD)
Kernel library	cuDNN (proprietary), MIOpen (MIT), oneDNN (previously mklDNN, Apache)
Job execution framework	CUDA (proprietary), ROCm (various open source)
Device control	NVIDIA Kernel Driver (Proprietary), AMDGPU/AMDKFD (pseudo-open source), Intel GPU/CPU

Table 3.2: Levels of the inference software stack.

The software stack typically consists of four layers (see Table 3.2):

- On the top-level of the stack are the high-level frameworks such as TensorFlow [Mar+15] or PyTorch [Pas+19], which are popular platforms (de facto standard) for training neural networks and other machine learning algorithms. Both are free and open source, licensed under an Apache License 2.0 and Modified BSD, respectively.
- Platforms like TensorFlow rely on kernel libraries such as NVIDIA's CUDA Deep Neural Network (cuDNN) or Apache's oneDNN, which provide GPU-aware implementations for standard functions, convolutions, activations and other functions relevant to neural networks.
- The frameworks then execute these optimized kernels together with their own on data blocks using acceleration frameworks such as CUDA (Compute Unified Device Architecture) or AMD's ROCm.
- These, in turn, interface with the GPU drivers to control data transfer and kernel execution tasks.

There is a multitude of vendor-specific components implementing this kind of architecture, some of which are mentioned below. These three layers provide the parallel computing capabilities to the benefit of building and subsequently training neural networks.

3.3.3 Requirements on the learning environment

While the learning environment is completely separate from the inference environment (which has all the flight hardware requirements, see Section 3.2), the former should not be ignored as it produces the model that will end on the latter after the conversions and transformations described in Sections 3.4 and 3.5.

A major concern is that it might be difficult to obtain guarantees on what the hardware/software stack computes (see also Section 3.4.1), due to:

- Proprietary drivers, lack of manufacturer support, transparency and guarantees (see also Section 3.2.2).
- Complexity of the software stack: for example, as of April 2021, TensorFlow contains more than 9'500 C++ source code files and 3'000 Python ones.

- Use of cloud computing: how can one ensure that third party machines execute the expected instructions (software and hardware)?

On the other hand, it is important to remember that the learning environment is not subject to requirements such as real-time execution (Table 3.1) or very high reliability (as long as errors are detected).

Possible mitigations and requirements will be discussed further in Section 3.7.

3.4 Passage between environments

As noted in Section 3.1.1 the learning and inference environment are unlikely to be the same given the differences in requirements. Therefore, the model obtained in the learning environment has to be transferred to the inference environment.

$$\begin{tikzcd}\text{Learning environment} \arrow[r] & \text{Inference environment} \\ \hat{f} & \mathcal{T}\hat{f}\end{tikzcd}$$

This raises non-trivial questions, which are discussed in this section.

3.4.1 Models in environments and as abstract functions

Neural networks are sequences of functions (convolutions, poolings, activations), possibly with parameters/weights, represented as *computational graphs* (statically or dynamically, implicitly or explicitly) in the software stacks of the two environments. As in [CoDANN20], the networks considered here all result in directed acyclic computational graphs (e.g. no recurrent neural networks) for simplicity.

Machine learning frameworks in the learning environment (such as PyTorch [Pas+19] and TensorFlow [Mar+15]) allow to create these programmatically using a declarative or imperative syntax: for example, the PyTorch code in Listing 3.1 results in the graph in Figure 3.8.

```
model = nn.Sequential(
    nn.Conv2d(in_channels=1, out_channels=6, kernel_size=3),
    nn.ReLU(),
    nn.MaxPool2d(kernel_size=2),
    nn.Conv2d(in_channels=6, out_channels=16, kernel_size=3),
    nn.ReLU(),
    nn.MaxPool2d(kernel_size=2),
    nn.Flatten(),
    nn.Linear(16 * 6 * 6, 120),
    nn.Linear(120, 84),
    nn.Linear(84, 10),
    nn.Softmax(),
)
```

Listing 3.1: A variant of the LeNet [Lec+98] neural network for handwritten digit recognition implemented in PyTorch with a declarative syntax.

Simplifying concepts from mathematical logic and computability theory, a computational graph represents an abstract function, whose definition and evaluation are mathematically unambiguous.

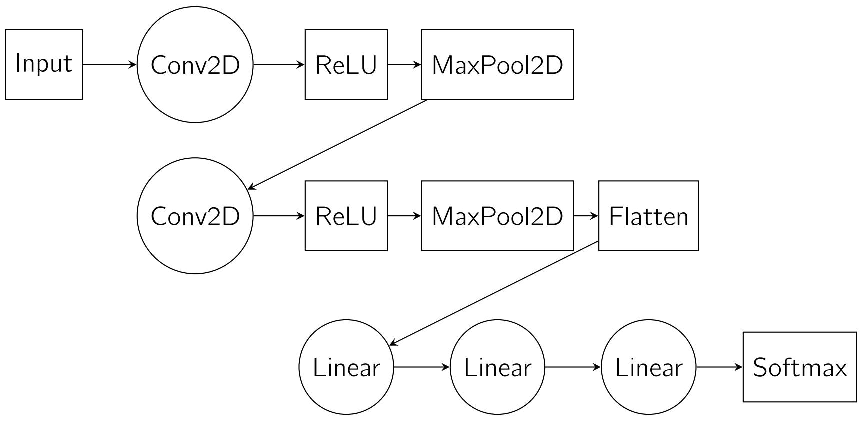


Figure 3.8: Computational graph resulting from the code in Listing 3.1. The circular operators contain learnable parameters.

On the other hand, it is important to note that an actual implementation is strongly tied to the software/hardware environment:

- Implementation of the operators (convolutions, poolings, activations...), as calls to lower-level libraries (e.g. cuDNN, then CUDA on a GPU-powered learning environment).
- Execution on the hardware;
- Representation of the weights;
- Representation of the input;
- etc.

In particular, the fact that the code in Listing 3.1 executes as the function represented by the graph in Figure 3.8 is fully dependent on these aspects. Machine learning practitioners will unanimously confirm the presence of such issues that give rise to different behaviors when migrating a model between environment (even for a change of high-level software stack, e.g. TensorFlow to PyTorch).

In the remaining of the text, a distinction will always be made between an abstract representation of a model and its implementation (see also Figure 3.13 at the end of the chapter).

Differences in implementations As discussed in Section 3.2.2, all hardware is designed with a certain use case in mind. For instance, CPUs are generally designed to execute a large variety of tasks with limited parallelism between the tasks, whereas GPUs are designed to trade genericity for increased parallelism. As a consequence, the optimal way in which a given operation is executed (say, a convolution) can greatly differ between hardware designs.

Listing 3.2 depicts a straightforward implementation of a two-dimensional convolution, where a 3×3 convolution kernel (`weights`) produces a 2×2 `output` array given a 4×4 `input` array.

This might look straightforward, but the lower-level execution kernel of this operation for, say, GPU designs, is more complex. The reason being that the current form of the operator is highly sequential, and left unoptimized for GPU hardware, it will not utilize the high parallelism that it has to offer.

To that end, hardware manufacturers provide libraries that contain multiple low-level operator implementations that best fit the design of the hardware. The choice of the optimal implementation will depend on the data, exact hardware, and other operations.

This will be discussed further in the context of tuning (Section 3.5.1), but in the scope of the current discussion, this implies that even assuming "perfect" execution at the hardware level, it might not be straightforward to guarantee that an implementation matches a target computational graph.

```
void conv(float input[4][4], float weights[3][3], float output[2][2]) {
    for (int row = 0; row < 2; ++row) {
        for (int col = 0; col < 2; ++col) {
            float sum = 0.0;
            for (int y = 0; y < 3; ++y) {
                for (int x = 0; x < 3; ++x) {
                    sum += input[row + y][col + x] * weights[y][x];
                }
            }
            output[row][col] = sum;
        }
    }
}
```

Listing 3.2: An example implementation that performs a convolution operation.

3.4.2 Model conversion

Model conversion covers the process of transferring a model between environments, usually from the learning to the inference environment.

Different environments (software/hardware stacks) will naturally represent differently the computational graphs introduced in Section 3.4.1, containing operations and learned weights. Moreover, as explained in Section 3.1.1, different environments may need to perform different functions (backpropagation for training, forward passes for training and inference), leading to the internal representations having different requirements. Correspondingly, conversion might need to perform both *format* and *mode* conversion, described in the following sections.

Format conversion Conversion is usually done through an intermediate representation of the computational graph that is environment-agnostic.

For neural networks, two examples are:

- The Open Neural Network Exchange (ONNX) [BLZ+19] format, which provides an open standard for neural network interoperability, meaning that it offers a common file format that makes it easier to share models between frameworks and/or custom inference environments. Quoting its documentation, ONNX provides:
  - *A definition of an extensible computation graph model.*
  - *Definitions of standard data types.*
  - *Definitions of built-in operators.*
- Apache TVM's Relay intermediate representation [Che+18], used as input to hardware-agnostic then hardware-specific model compilers, allowing to deploy a model on multiple platforms (CPU, GPU with CUDA or OpenCL, etc.) easily.

The learning environment (usually through the top-level framework, e.g. TensorFlow) should be able to export its internal representation to the environment-agnostic representation, and vice-versa for the inference environment.

The issues noted in Section 3.4.1 apply: an abstraction of the model does not unambiguously represent its execution.

For example, an operation `LE_Conv2D` in the learning environment might be translated into a `Conv2D` operation in the abstract representation, and then mapped into a `IE_Conv2D` operation in the inference environment. It could happen that:

- `LE_Conv2D` and `IE_Conv2D` have different implementations (e.g. because they handle edge cases differently, or simply due to hardware-specific optimizations);
- The parameters (or their defaults values) in `LE_Conv2D` and/or `IE_Conv2D` do not have exactly the same meaning;
- The input data is encoded differently.

Mode conversion Like the format in which the computational graph is represented, the *intended function* of the graph, which will be called *mode* here, may change between the learning and inference environment. For example, recalling Table 3.1, the graph does no longer need certain functionality that is typical for training, such as:

- All operations and functions that support the propagation of gradients through backward passes;
- Some functions change behavior between modes, such as batch normalizations and dropout regularization;
- The ability to store weights as variables, because these no longer need updates;
- Training states, logs and metrics;
- Framework-specific metadata.

Converting a training model into an inference model changes the functionalities like the ones listed above in a certain way. Some operations can be removed entirely, such as gradient propagation functions, and others merely need a change of behavior, such as dropout regularization layers. Deciding which graph components are not needed during inference heavily depends on the function definitions and the training process, therefore having the ability to convert between modes is typically a feature provided by the training framework.

A natural way of creating an abstract representation of an inference model, would be to apply the two conversion in the order described in Figure 3.9 (see also Figure 3.13 later on).

$$\begin{tikzcd}\text{Training model} \arrow[r, "\mathcal{T}_{mode}"] & \text{Inference model} \arrow[r, "\mathcal{T}_{format}"] & \text{Abstract representation}\end{tikzcd}$$

Learning environment Environment-agnostic

Figure 3.9: Converting a trained model to an inference model, and subsequently to an abstract representation.

3.4.3 Requirements on model conversion

In the ideal case, a requirement is that executing the original model in the source environment corresponds pointwise to the execution of the converted model in the target environment, meaning that the same output is obtained for the same input.

However, this might present challenges:

- This would in particular require assurance of the exact execution of the computational graph on the learning environment (see Section 3.3.3).
- Executing the floating-point operations of an operator in the neural network in a different order may lead to a different result due to rounding effects.
- Performing cross-environment verifications (two totally different hardware platforms) might be challenging.

If the conversion can be done bidirectionally, one way to provide evidence of its correctness would be to show that the conversion of the converted model matches the original one, or at least its computational graph.

$$\begin{tikzcd}
\text{Original model} \arrow[r] & \text{Converted model} \arrow[l, bend right] \\
\text{Learning environment} & \text{Inference environment}
\end{tikzcd}$$

Requirements on the learning environment and model conversion, as well as potential methods to meet them, will be discussed further in Sections 3.6 and 3.7.

3.5 Model optimizations

Model optimizations encompasses methods and procedures that aim to reduce the complexity of neural networks, and thereby improving the inference performance (measured by speed/number of operations, rather than by the system performance metrics). This will sometimes be done at some (ideally minimal) cost on the system performance metrics compared to a non-optimized model, for example in the case of pruning and quantization.

These should not be mistaken for mathematical optimization methods such stochastic gradient descent, as they solely aim at improving runtime performance (and may or may not perform mathematical optimization to do so).

Optimizations might be performed:

- As part of the main training, as additional training steps, or post-training (e.g. as part of the *Model Implementation* step of the W-shaped process).
- In the high-level framework (e.g. TensorFlow), on the abstract representation (e.g. TVM Relay), namely after conversion (Section 3.4.2), or on the implementation on the inference platform directly.

An optimization that happens post-training is delicate, as it has the potential of significantly changing the model's behavior. Among the ones that will be surveyed below, most actually admit a better "execution-performance gain vs metrics-performance loss" trade-off when executed during training or with additional training steps.

Again, Sections 3.6 and 3.7 will discuss how properties may or may not be preserved after transformations, and how to mitigate potential risks and ensure that performance guarantees (with respect to metrics and execution) hold on the inference platform.

3.5.1 Tuning

Section 3.4.1 mentioned that (sequences of) neural networks operations could often be implemented in multiple ways.

Model tuning[5] refers to finding the optimal way to do so given a target hardware, to optimize application-specific throughput by increasing parallelism, and reducing memory footprints and workloads. Optimizing for these three properties simultaneously is a complex task, since changing a given property oftentimes influences another.

Of the three techniques reviewed below, the first one is at the level of the implementation, while the next two operate on the computational graph. In particular, it might be easy to prove that the latter do not change the model (and a tool performing the optimization could be qualified, see Section 3.7.1 below), while the former might require more work.

Optimal implementations Kernel libraries (see Table 3.2) usually make available (explicitly or implicitly) multiple implementations of operations when relevant, and possibly information to choose the optimal ones.

For example, cuDNN provides methods

```
cudnnGetConvolutionForwardAlgorithm, cudnnFindConvolutionForwardAlgorithm
```

that search (heuristically and exhaustively) the fastest convolution implementation given input specifications.

Apache TVM [Che+18], already introduced in Section 3.4.2, implements a multi-phase compiler that will perform hardware-agnostic and -specific optimizations on the intermediate representation, so that deployment to multiple platforms can be done transparently to the user. Chen et al. [Che+18] provide an overview of typical kernel library optimizations, two of the most important ones being exposed below.

High-level operator fusion This is an optimization technique on the computational graph level (see Section 3.4.1). Operator fusion aims at increasing execution time performance by reducing memory footprint. It achieves this by fusing operator primitives into single kernels, at the benefit of not having to store intermediary results into memory every time it is executed. The authors of [Che+18] identify four categories of graph operators, which can be:

1. *injective*; one-to-one mappings such as addition,
2. *reduction*; many-to-one mappings such as sums,
3. *complex-out-fusable*; many-to-many, element-wise operations to output mappings, such as two-dimensional convolutions
4. *opaque*; contains operators that cannot be fused.

The strength of operator fusion comes from recursive application of the operator categories. For example, one could fuse a reduction with an input injective operator. The output of this optimization output technique is a transformed version of the original computational graph.

See [Che+18] for additional details.

Constant folding and constant propagation *Constant folding* is a common compiler optimization that seeks to replace expressions with constants, which can be applied to computational graphs as well. In most cases, the substituted expressions are composed of literals

[5]This should not be mistaken for "hyperparameter tuning", which refers to the search of optimal hyperparameters during model selection. In particular, this happens during the training phase, while model tuning happens after, with "tuning" referring to implementation parameters.

or variables that are assigned before– or during compilation time (such as build–, and other kinds of configurations). The primary constraint for substituting expressions is that they are guaranteed to produce the same output for each evaluation. For instance, the expression

```
a = 20 * 3 * 2
```

will always evaluate to 120 no matter how many times a is evaluated. In these cases, constant folding replaces the expression by the constant it evaluates to.

Constant propagation is a similar technique in its ability to substitute expressions with constants (like constant folding) through *reaching definition analysis* on variables.

See [Che+18] for additional details on both aspects.

3.5.2 Quantization

Quantization is the general process of constraining a continuous set of values to a discrete one.

Neural networks with their operations such as convolutions and activation functions are typically specified terms of functions from real numbers to real numbers. On a computer they are encoded in a floating-point representation. In practice 32-bit floating-point numbers are a common choice for inputs, weights, and outputs given that:

- Some functions inherently operate in floating-point arithmetic, such as activation functions and gradient computations, making it hard to define a neural network in integer arithmetic;
- 32-bit floating-point numbers offer a good trade off between the number of significant digits (precision), the dynamic range (exponent) required for the relevant computations.
- GPUs as the de facto standard compute engines for training neural networks offer excellent performance on 32-bit floating-point numbers and the functions typically used in neural networks.

Some of these considerations may change on the inference environment:

- The inference environment may have limited or no support for floating-point arithmetic.
- The inference environment may offer significantly larger performance when integer arithmetic (or other non-floating-point arithmetic) is used, making it easier to meet requirements on throughput, memory, and power consumption.

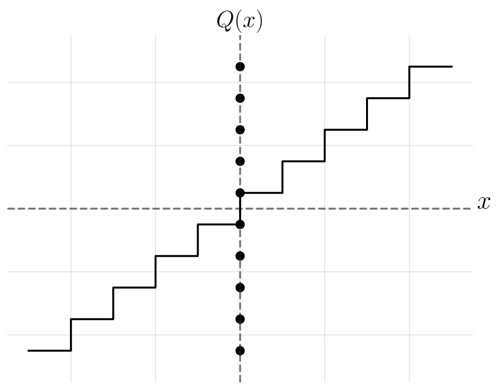
In these cases, quantization is an important tool that transforms the model to operate fully or partially on discrete values using integer arithmetic. This includes inputs, intermediate results and model parameters.

This section contains a high-level description of quantization, its different modes and associated risks.

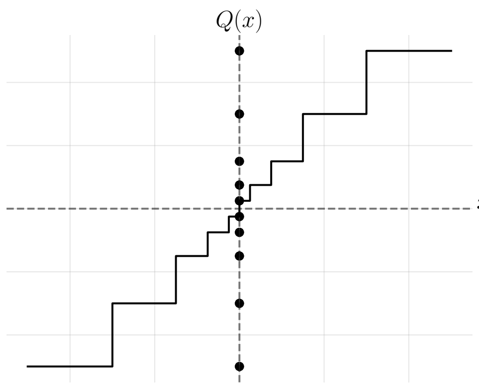
Quantization concept Transforming a set A of real-valued numbers by quantization means finding a function

$$Q : A \to B$$

that maps values in A to a discrete set $B \subset \mathbb{Z}$ together with corresponding transformations of the operations on the numbers in A used in the neural network, i.e. functions $f : A \to A$ have to be mapped to functions $f^Q : B \to B$ that either perform the same operation or an



(a) Uniform quantization



(b) Non-uniform quantization

Figure 3.10: Uniform quantization in (a) assumes equally spaced intervals between the discrete values (sometimes referred to as "quantization levels"). Non-uniform quantization (b) assumes non-equally spaced intervals between the quantization levels.

appropriate approximation. The quantized values in B often have a lower *bitwidth* (i.e. the amount of bits in which the representation is stored):

$$Q : A \to B$$

For example, A can be encoded in a 32-bit single-precision floating-point representation, and B can be a set of 8-bit integers in the range $\{-128, -127, \ldots, 126, 127\}$.

Before addressing the considerations on quantizing neural networks in the context of safety-critical systems, more information on finding such a function Q, as well as its properties is given. The following terminology and explanations are adapted from notable works and surveys on quantization, see [Jac+18; Kri18; Gho+21] for more details.

Uniform quantization means that the discrete values of the target domain are mapped to equally spaced values in real space. The advantage of a uniform quantization is that floating-point addition and multiplication are mapped to integer additions and multiplications, which are either available on the inference environment or relatively straightforward to implement. *Non-uniform* quantization assumes that the targets need not be equally spaced. While this can be more efficient in terms of representing the numbers from the original domain, it comes at the cost of a more complex mapping of operations.

Figure 3.10 illustrates an example of both classes of quantization. Neural networks are generally quantized uniformly due to the good properties arising when operations are mapped as integer additions and multiplications on the system under consideration.

A natural way of quantizing values from A is by setting the quantizer $Q : A \to B$ to:

$$Q(x) = \lfloor x/s \rceil - z \qquad (x \in A),$$

where $\lfloor \cdot \rceil$ is a function that rounds a real number to its nearest integer, $s \in \mathbb{R}$ the *scaling factor* and $z \in \mathbb{Z}$ is a *zero-point* correction. The input domain A can be thought of as vectors of weights or activations. As such, the range of possible floating-point values in A is different from B, which makes the function Q an *affine mapping* of A to B using *quantization parameters* s and z.

The quantization parameters can be chosen to different effect, which will be explained in more detail below. Furthermore, the notion of *de-quantization* is useful to discuss different modes of quantization: it maps quantized representation back to real values. The quantization process is by definition not injective, but an arbitrary preimage can be chosen consistently, e.g. by:

$$x' = s(Q(x) + z),$$

where $x' \approx x$ is the reconstructed real value, which may not exactly match x because of the `round` operation shown earlier. To summarize:

- *Quantization* converts a representation of a real number x to a quantized representation $Q(x)$, e.g. 32-bit single-precision floating-point $\to$ 8-bit integer.
- *De-quantization* converts a quantized representation $Q(x)$ to a real number x', e.g. 32-bit integer $\to$ 32-bit single-precision floating-point.

Choosing the quantization parameters The aforementioned quantization function that maps floating-point values from a set A to a set of integers B contains the parameters s and z, which are the scaling factor and zero-point correction, respectively. A natural way of finding these parameters is to find a linear scaling function:

$$s = \frac{\beta - \alpha}{2^n - 1}$$

where n is the bitwidth of the target representation B, and α and β denote the *clamping* range of A, with $\alpha < \beta$. Choosing values for α and β affects the *symmetry* of the quantizer in of two ways:

- Setting $|\alpha| \neq |\beta|$ makes the quantizer *asymmetric*. For example, asymmetric quantization is likely when choosing α, β such that it covers the complete range of A, i.e. $\alpha = \min(A)$ and $\beta = \max(A)$;
- Setting $|\alpha| = |\beta|$ makes the quantizer *symmetric*. A natural choice for this absolute value could be $\min(|\min(A)|, |\max(A)|)$. Symmetric quantization renders the zero-point correction $z = 0$, and makes the quantizer less sensitive to outliers in A. This approach may become less optimal as the distribution of values in A becomes more skewed.

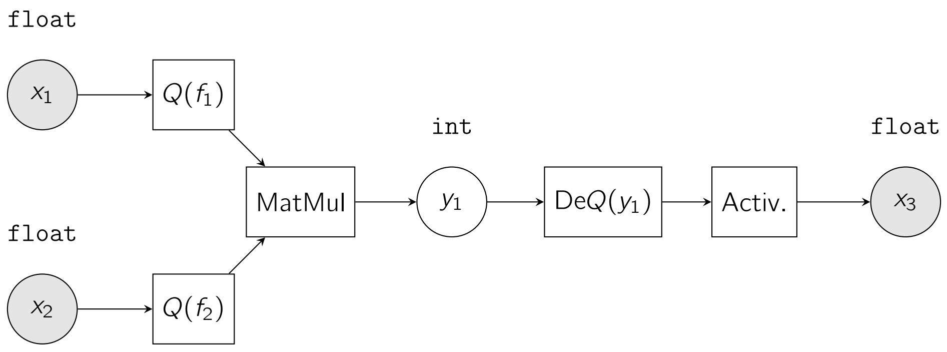


Figure 3.11: An example of a matrix multiplication and a piecewise activation function on the output. The filled circles denote inputs/outputs in floating-point precision, and Q, DeQ stand for quantization and de-quantization respectively.

Quantization modes Up to this point, the discussion revolved around explaining how quantization typically works, and providing an impression of design choices one makes when resorting to a quantization technique. A similarly important, but undiscussed topic is where and when to apply quantization in a function as complex as a neural network. Recall that a neural network is a combination of many functions, some of which may favor integer arithmetic, such as (matrix) multiplication and convolutions, and others are more suitable for floating-point arithmetic, such as softmax, and other types of activations.

As a result, alternating between quantizing floating-point values and de-quantizing integers values is needed to serve each operation with their preferred input representation, depending on the hardware. An alternating sequence is shown in Figure 3.11, which is one that is commonly found in neural networks. Because of this alternation, as well as possible changes in the input space, the quantization parameters (i.e. the scale factor and the zero-point correction, for example) need constant recomputation.

Next, a few examples will be given on *quantization modes* that are commonly described in the literature and supported by training/inference frameworks for neural networks:

- Dynamic quantization;
- Static quantization;
- Quantization-aware training.

Dynamic quantization In dynamic quantization, the scale factor and zero-point parameters are dynamically computed during inference. Only the neural network weights and biases (which become constants), are quantized to integers before runtime.

The benefit of dynamic quantization is that the quantization functions are explicitly computed for each input. As can be expected, the downside is that it comes at the cost of additional overhead. This overhead should not completely diminish the performance gain from switching to integer operations, but this is not guaranteed either. Quantizing a value x by dynamic quantization can be characterized as a transformation

$$x = S(Q(x) + z) + \epsilon_x,$$

where ϵ_x is an unknown residual representing the information lost by rounding, range clamping and other operations.

Dynamic quantization was the baseline by Zafrir et al. [Zaf+19], which have reported that applying dynamic quantization on the BERT architecture [Dev+19] can have a more adverse effect on prediction performance compared to *quantization-aware training*, which is discussed later.

Static quantization Static quantization addresses the overhead drawback of dynamic quantization by pre-computing the quantization parameters before running inference. Specifically, pre-computing the scale and zero-point constants allows the output and activation tensors to be stored as the specified integer representation directly, rather than indirectly as is the case with dynamic quantization.

Static quantization is straightforward. First, it requires composing a representative dataset of the intended operating space. Then, this dataset is fed to the unquantized network, gathering statistics of the activations in the process. These statistics are then used to compute the quantization function, which is fixed during operation. This results in a quantization function that *approximates* the dynamic quantization function Q_D.

The difference between static and dynamic quantization is that dynamic quantization computes the quantization parameters on a per-input basis. This recomputation causes overhead, but the benefit is that the parameters are computed specifically on arbitrary input, making it the mapping relatively precise. On the other hand, static quantization attempts to remove this overhead by finding a set of parameters based on a collection of inputs. The question is however whether these parameters work well on other inputs that are not in the collection. In that sense, for arbitrary input outside the collection, the used parameters under static quantization are an "approximation" of the parameters that one would find when running dynamic quantization.

As such, the resulting transformation by of static quantization can be characterized as:

$$x = S(Q(x) + Z) + \epsilon_x + Q_D(x) - Q(x),$$

adding another unknown residual compared to dynamic quantization, i.e. the approximation error $|Q_T(x) - Q(x)|$, which describes the discrepancy of the approximated quantization function compared to the true quantization function.

Approximating the quantization parameters generally leads to improved runtime performance compared to dynamic derivation, at the cost of prediction accuracy induced by the approximation error.

Quantization-aware training Quantizing neural networks post-training (be it dynamic or static) is a *lossy* transformation, meaning that information gets discarded during the process. Examples include the rounding operation from Section 3.5.2 and the clamping operation in Section 3.5.2, if applied. As a result, the gain in execution performance is likely traded for a penalty in prediction performance. Work by Jacob et al. [Jac+18] proposes a solution to this seemingly inevitable trade-off by introducing *quantization-aware* training.

Quantization-aware training anticipates the information loss by "simulating" the effects of quantization during training. In effect, all convolution computations are carried out using regular 32-bit floating-point operands, but the convolution in- and outputs are quantized in a "simulated".

Once the model has been fully trained, the 32-bit single-precision floating-points can be readily transformed into their target representations without losing information during the conversion.

The reader is referred to the paper [Jac+18] for a detailed description of the method.

Clearly there is still a prediction performance penalty to be paid compared to models that have not been quantized, but the crucial difference compared to post-training quantization is that this penalty is paid during the training phase, instead of after training or during runtime.

Other quantization methods It should be noted that other quantization methods (including variants of the above methods) exist. Examples of these methods are non-uniform quantization, extreme quantization (binarization), mixed-precision quantization, etc. However, discussing all of them in detail would deserve a report on its own. Instead, the reader is referred to survey by Gholami et al. [Gho+21] for an overview on these quantization methods in neural networks.

Risks and benefits of quantization To summarize, quantization might be necessary depending on the inference platform, either because of a lack of support of floating-point operations, or because these would not provide the required performance. However, if it is done post-training, it fundamentally changes the model, which has the risk of invalidating properties of the original floating-point model.

One possible obvious mitigation is quantization-aware training and similar techniques. See also Section 3.7 below for further discussions.

3.5.3 Pruning

Neural networks are complex models in terms of computational and memory requirements. This can make them hard to deploy on devices that lack the resources to run them efficiently, if at all (see also Section 3.2.2).

A common approach to reduce model complexity is by *model pruning*, which is an optimization technique that aims to systematically disable neural network components that have a relatively limited contribution towards prediction performance. Disabling components is typically done by either nullifying or physically removing them from the computational graph entirely, see Figure 3.12 for a basic example.

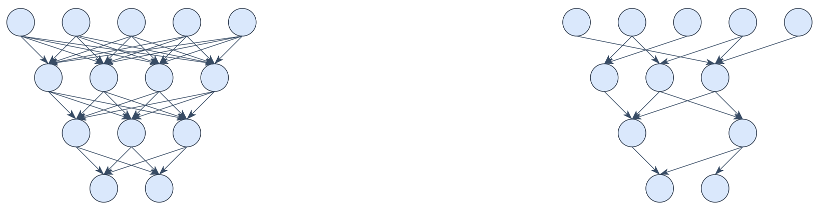

(a) Before pruning (b) After pruning

Figure 3.12: Basic visualization of graph pruning. In this case the graph components of the neural network are the weights (arrows) which are removed from the original graph (a), resulting in a pruned graph (b). Removing all weights towards a neuron (circle), removes the neuron itself.

Basic concept of pruning Making changes to the computational graph by pruning almost always leads to a gain in runtime performance at the cost of prediction performance. To regain the lost prediction performance, many pruning techniques apply additional training steps on the pruned network. Consequently, the high-level algorithm for neural network pruning typically consists of the following steps:

1. Choose a (large) neural network architecture;
2. Train the network until completion;
3. Repeat until a stopping criterion is reached:
    (a) Score components;
    (b) Prune components;
    (c) Additional training of the network.

A survey from Blalock et al. [Bla+20] provides an analysis of different pruning methods, and divides them into the following categories that relate to the above steps:

- *Pruning components*: The structure of the pruning method entails whether individual parameters were removed independently of each other, or groups of parameters were considered, such as convolution filters or channels. These two options are *unstructured* and *structured pruning* methods, respectively.
- *Scoring*: There are many ways in which parameters can be selected for removal. Some methods select at random, others score parameters by absolute value or use functions that approximate importance towards prediction performance. Regardless of the method used, a designer can score parameters *locally*, e.g. by a single layer or group of layers, or *globally*, e.g. over the entire network.
- *Pruning schedule*: The pruning schedule dictates how many parameters are pruned at each pruning step of the high-level algorithm described above. This amount can either be *fixed* or *variable*, if it depends on a pruning rate or a complex function.
- *Additional training*: As mentioned earlier in this section, a common way to recover the prediction performance that was lost in the pruning step is to continue training the

pruned network on the training dataset. The standard approach is to continue training by *fine-tuning* the network on the remaining network parameters, but alternatives are possible, such as restoring the remaining parameters to an earlier state, or re-initializing the parameters entirely (i.e. training "from scratch").

To better understand some of the results and possible side effects that are associated with neural network pruning, it is useful to present observations made by two notable works.

Example 1: Pruning convolution filters Li et al. [Li+17] propose pruning filters from standard CNNs and residual networks, by calculating the sum of their absolute kernel weights for each convolutional layer and removing them accordingly. The method is able to reduce about 30% of floating-point operations for a VGG-16 architecture [SZ15] and a ResNet56 architecture [He+16] without a significant loss in prediction performance[6] on the CIFAR-10 dataset [Kri09]. In addition, the authors report the following observations:

- Pruning filters with the lowest weight magnitude gives better results than pruning filters with the highest weight magnitude, or randomly selected ones.
- Some layers in a residual network are more sensitive to pruning than others; scoring filters *locally* and removing them on a *variable* schedule tends to yield better results.
- Pruning might even *improve* prediction performance in early stages, i.e. when the runtime performance gain is still minimal.
- The prediction performance of a pruned neural network that is fine-tuned is better than taking the ultimate pruned architecture, and training it from scratch.
- The order of pruning and training steps is relevant. The prediction performance of a neural network that is first trained and then fine-tuned through pruning is higher than the performance of network for which the identical pruning is performed first.

Example 2: Pruning by Taylor expansion criterion Molchanov et al. [Mol+17] put an emphasis on examining the different criteria that score feature map activations in convolutional neural networks, for pruning. The authors formulate the goal of finding the least important parameters as a combinatorial optimization problem that seeks to minimize the loss of prediction performance between an original and a pruned model. Then, a set of greedy scoring criteria is taken against a *brute-force* selection criterion (i.e. one that evaluates all possible pruning combinations). Among the results, the authors demonstrate that pruning with a particular criterion can lead to significant runtime speedups across a number of different devices (a factor between 2 and 5 times). The reader is referred to [Mol+17, Table 2] for these results. Other observations are also made:

- Approximating the proposed cost function by Taylor expansion shows superior performance over less complex criterion functions, such as minimum weight magnitude, random selection, mean activation and others.
- The prediction performance of a pruned neural network that has been iteratively fine-tuned is better than that of a network that is 50% smaller and trained from scratch.
- Increasing the number of training updates after pruning increases the model's ability to restore the prediction performance.

[6]CIFAR-10 accuracy error, original model versus pruned model (lower is better). VGG-16: 6.75% → 6.60%, ResNet56: 6.96% → 6.94%

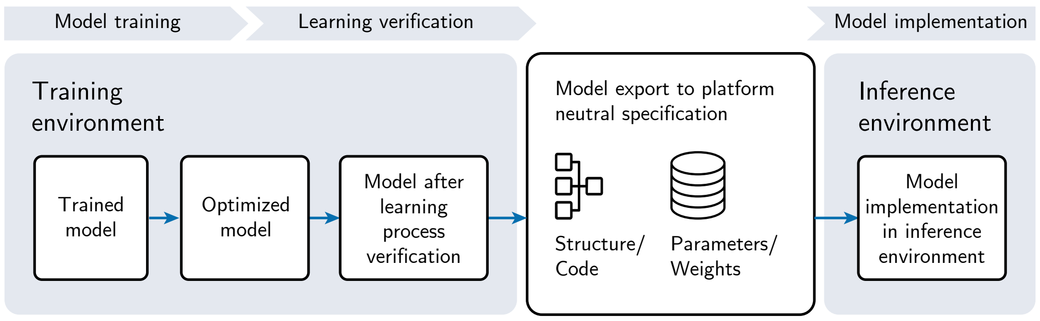


Figure 3.13: Transfer of model structure and model parameters from the training environment to the inference environment.

Risks and benefits of model pruning Except for obvious cases (e.g. identity weights/zero biases), it may be difficult to show pointwise equivalence (see Section 3.6.1 below) between a pruned and an unpruned model.

For iterative pruning schemes as described above, the more iterations are applied, the larger the difference becomes between the pruned and unpruned models. This difference is not just caused by the (possibly heuristic) removal of parameters, but also by the change of the optimal parameter values for the remaining parameters in subsequent training iterations. In addition, it may be hard to demonstrate traceability through these repeated changes from a safety-critical perspective.

Still, optimizing a model by pruning can provide significant benefits:

- Reducing the size of the neural network by pruning relaxes the requirements on the device capabilities in terms of throughput, memory storage and power consumption.
- Systematically lowering the model complexity while maintaining the in-sample error improves the generalizability of the model (see also Section 3.7.3 below).
- Smaller models tend to be more suitable for complex or formal verification methods than larger models.

Parallels may be drawn with removal of "dead code" in classical software (with parts of the architecture/model that are removed rather than actual code). However, it is not trivial how pruning relates to removing dead code from a [ED-12C/DO-178C] perspective. Whether pruning can be considered in a similar context or not is left for future work.

3.6 Transformations and preservations of properties

The previous sections described the full lifecycle of a machine learning model in the W-shaped process, from its training in the learning environment (Section 3.3), optimization (Section 3.5), and transfer (Section 3.4) to the inference environment (Section 3.2). Figure 3.13 summarizes this process.

Each of these actions performs modifications of the original model that can be very complex (e.g. including potentially additional training steps) and significantly change model properties.

Some of the aspects to take into account are:

- Reimplementation on a different hardware and/or software stack, including conversion of the model (weights and computational graph);
- Correctness of the mapping from model on learning environment to abstract representation;
- Transformations such as quantizations or pruning, that might fundamentally change the model;
- Optimizations such as tuning.

The *Learning process verification* step includes understanding generalization capabilities, and other properties, of the trained model.

The *Inference model verification and integration* phase (see Figure 1) is meant to ensure that the transformed model on the inference (operational) environment has the desired properties, which might combine:

- Properties of the trained model such as:
  - Generalization guarantees;
  - Robustness;
  - Explainability.
- Additional properties specific to the inference environment, such as real-time execution guarantees.

If properties are to be carried from the learning to the inference phase of the process, each of the aspects above has the potential to significantly alter the quality of the outputs and invalidating generalization guarantees. The need for traceability also requires understanding the impacts of transformations.

$$\begin{tikzcd} \begin{array}{c}\text{Learning environment}\\ \hat{f}\\ \text{Properties}\end{array} \arrow[r, "\mathcal{T}"] & \begin{array}{c}\text{Inference environment}\\ \mathcal{T}\hat{f}\\ \text{Conserved properties}\\ \text{Additional properties}\end{array} \end{tikzcd}$$

Given the particularities of the learning environment, the assurance level at which properties can be verified therein also has to be discussed. This will be discussed in Section 3.7.

3.6.1 Conditions on transformations

This section formalizes the conservation of properties from a model $\hat{f}$ under a transformation $\mathcal{T}$ (different implementation, on a different platform, functional changes, etc.).

Abstract representations vs implementations A preliminary remark is that given the discussion in Section 3.4.1, the trained model $\hat{f}$ (i.e. output of the *Learning process verification* phase of the W-shaped process) should not be considered as an abstract mathematical function, but as closely tied to the learning environment: the evaluation of $\hat{f}$ is only possible in this environment.[7] Similarly, through the chain of transformations, models will be either abstract representations (as computational graphs) or tied to an environment (learning/inference).

[7]And might not be fully deterministic due to nondeterminism in hardware (e.g. due to the combination of parallelism and floating-point), but this will be assumed for simplicity since this doesn't affect the conclusions.

Conditions on transformations If $\mathcal{T}\hat{f} : \mathcal{X} \to Y$ is the model obtained after applying a transformation $\mathcal{T}$ to $\hat{f} : \mathcal{X} \to Y$, one might require conditions of different strength, depending on the properties that need to be preserved. Ordering by decreasing strength (in the sense of logical implications), where $\varepsilon, \varepsilon'$ are small positive constants:

- $\mathcal{T}\hat{f}$ and $\hat{f}$ are pointwise equivalent:
$$(\mathcal{T}\hat{f})(x) = \hat{f}(x) \qquad \forall x \in \mathcal{X}. \tag{3.1}$$
This is the strongest condition, as all properties applying to $\hat{f}$ will apply to the transformation.

- $\mathcal{T}\hat{f}$ and $\hat{f}$ are pointwise quantifiably close with respect to some metric $m : Y \times Y \to \mathbb{R}_+$:
$$m\left(\hat{f}(x), (\mathcal{T}\hat{f})(x)\right) < \varepsilon \qquad \forall x \in \mathcal{X}. \tag{3.2}$$
This would allow properties of $\hat{f}$ related to m to be propagated to $\mathcal{T}\hat{f}$. For example, if generalization bounds assert that $\hat{f}$ is close to the true function f with respect to m, one might be able to deduce that $\mathcal{T}\hat{f}$ also is, with a larger margin of error controlled by ε.

- $\mathcal{T}\hat{f}$ and $\hat{f}$ are pointwise equivalent or close, but only on average over the operational probability space $\mathcal{X}$, e.g.
$$P_{\mathcal{X}}\Big((\mathcal{T}\hat{f})(x) = \hat{f}(x)\Big) > 1 - \varepsilon' \quad \text{or} \quad \mathbb{E}\Big[m\left(\hat{f}(x), (\mathcal{T}\hat{f})(x)\right)\Big] < \varepsilon.$$

- Other conditions are naturally possible, depending on the requirements for the end model, and how properties are expected to be transferred between the learning and inference environments. In particular, the examples above focused on pointwise relationships, but higher-level conditions (e.g. "$\hat{f}$ verifies $A \Rightarrow \mathcal{T}\hat{f}$ verifies A") can also be envisioned.

3.6.2 Applicability of the conditions

In some cases, it might be difficult to prove a condition strong enough on the relationship between a function and its transformation so that properties can be carried over.

For example:

- It might prove very challenging to meaningfully measure the impact (in terms of any of the conditions above) of a complete change of the hardware and software stacks.
While it is reasonable to expect that the inference environment guarantees that the execution of the inference model corresponds to the mathematical function represented by the computational graph (up to controlled rounding errors), this might be much harder in the learning environment (see Sections 3.2 and 3.3).

- Pruning (see Section 3.5.3) usually involves additional training steps of the model (learning environment);

- Quantization (see Section 3.5.2) similarly might require training-aware mitigations to only carry an acceptable loss of performance (for a continuous output space, bounds such as (3.1) might even be impossible);

Moreover, even if functional equality could be verified, as mentioned earlier, ensuring that desired properties are met in the original model with the right assurance level might as well be difficult given the needs and therefore particularities of the learning environment (Section 3.3), in particular flexible hardware and software.

In that situation, an alternative to the obvious methodology of carrying properties from the original model to the inference model in the inference environment is required.

3.7 Ensuring properties on the inference platform

This final section investigates means to meet the goal of guaranteeing properties (such as performance on unseen data) for the inference model on the inference environment met, despite:

- The training environment likely being one where it is difficult to give strict guarantees (GPUs with proprietary drivers, operating system, etc.), see Section 3.3.
- The complex transformations from the training to the inference environments, see Section 3.6.

To do so:

- Section 3.7.1 provides a reminder about tool qualification, and puts in this context the learning environment, tools to transfer the model to the inference environments, as well as tools to verify the latter.
- Section 3.7.2 provides a useful reminder on how performance guarantees on unseen data are obtained for machine learning models, to help drive the rest of the discussion.
- Section 3.7.3 will analyze how to obtain performance guarantees on models despite transformations or risks related to the learning environment.
- Finally, Section 3.7.4 provides a summary of a possible generic approach mitigating these risks and ensuring that the W-shaped process meets the above goal.

3.7.1 Tool qualification

The software and hardware used to train/evaluate a model, and then transfer it to the inference environment (including additional evaluations) are tools akin to compilers and development environments in [ED-12C/DO-178C]. As described in Section 3.2, the development of the inference environment itself (hardware and software) might also use various tools; this is however close to classical development.

Any tool used should be documented appropriately and assessment should be made if tool qualification is required for specific uses of the tool. The purpose of the tool qualification is to detect and report errors that may have been introduced in the system due to the use of the tool. According to [ED-12C/DO-178C, Section 12.2.1],

> "*Qualification of a tool is needed when processes of [ED-12C/DO-178C] are eliminated, reduced, or automated by the use of a software tool without its output being verified as specified in [ED-12C/DO-178C, Section 6] (Software Verification Process).*"

Only if this is the case then the Tool Qualification Level (TQL) needs to be determined. The TQL depends on the following criteria in combination with the software assurance level of [ED-12C/DO-178C, Section 12.2.2]:

Criteria	Tool properties
1	Output is part of the airborne software and thus could insert an error.
2	Automates verification process(es) and thus could fail to detect an error.
3	Within the scope of its intended use, could fail to detect an error.

Whether the qualification of tools throughout the W-shaped process (learning environment, transfer to the inference environment, verification of the latter...) is required can thus not be answered in general. This will depend on how they are used in practice, and what the surrounding processes to verify the output are.

The learning environment (software and hardware) produces a model (as an abstract computational graph after conversion, see Section 3.4) which could be seen as a criteria 1 tool.

However, it might be impossible to meet the requirements on a criteria 1 tool due to particularities of the environment. On the other hand, extensive tests are performed on the model in the training and inference environments, as developed below, which might change the requirements on the learning environment and passage between environments.

3.7.2 Reminder on learning assurance/performance guarantees

To get a better understanding of the requirements on the learning and inference environments, and the flow between them, towards performance guarantees, it is useful to briefly recall some aspects of learning assurance.

A model $\hat{f} : \mathcal{X} \to Y$ approximating a function $f : \mathcal{X} \to Y$ is obtained after running a learning algorithm (e.g. gradient descent on a fixed architecture, starting with random weights) on a training dataset D_{train}, possibly selected from several other models using a validation dataset D_{val}. By definition, this happens in the learning environment (see Section 3.3).

Generalization guarantees (see [CoDANN20, Section 5.3]) typically provide confidence bounds for the out-of-sample error (performance on unseen data) as a function of the in-sample error

$$E_{\text{in}}(\hat{f}, D_{\text{train}}, m) = \frac{1}{|D_{\text{train}}|} \sum_{(x,f(x))\in D_{\text{train}}} m\left(\hat{f}(x), f(x)\right),$$

i.e. the error observed during training.

Bounds on the out-of-sample error (equivalently on the *generalization gap*, see [CoDANN20, 5.3.1]) precisely provide performance guarantees on the machine learning model, as explained in [CoDANN20] (see also Chapters 5 and 6). The fact that it is possible to ensure correct behavior on unseen data (but satisfying the correct distribution) is the basis for deploying machine learning in safety-critical settings.

As described in [CoDANN20, Section 5.3], these bounds might come from complexity-based approaches (complexity/capacity of the model class and/or of the data, etc.), testing, or a combination of both.

Role of the training process An important remark is that the bounds will usually *not depend on how the model was obtained from the class of models considered* (e.g. all possible weights on a given architecture), but only on *characteristics of the class* (e.g. number of layers) and the data (size, structure...): see [CoDANN20, Section 5.3].

For example, classical Vapnik–Chervonenkis (VC) bounds have the form:

with probability $> 1 - \delta$
over all iid-sampled datasets of size $|D|$,
the generalization gap is

$$\leq \sqrt{\frac{d_{\text{vc}} \cdot \log(2|D|/d_{\text{vc}}) + \log(1/\delta)}{|D|}},$$

where d_{vc} is the VC-dimension of the model class. For a neural network with L layers, W weights and ReLU activations, Bartlett et al. [Bar+19] proved that $d_{\text{vc}} = O(WL \log W)$ (see the paper for the exact hypotheses). Therefore, the bound above only depends on the model architecture (namely W and L), $|D|$ and δ, but not on the training process. This is similar to the fact that the final bounds do not depend on the process followed during the training/validation cycle, only on the number of such cycles (assuming a classical process).

In particular, the training process (optimizer, number of epochs, etc.) is in theory irrelevant. However:

- This does *not* mean that an arbitrary function $\hat{f} : \mathcal{X} \to Y$ achieving an acceptable in-sample error can be used: the model has to come from the class of models to which the generalization bounds apply.
- Some generalization bounds (see [CoDANN20, Section 5.3.6]), such as the ones based on compression and/or PAC-Bayes bounds, require additional optimizations or transformations of the models. Nonetheless, the final bounds will usually still not depend on the behavior of these steps, but only on their result.
- One might want to retain the optimization process as part of the argument.
  - For models searched through convex optimization, one might be able to bound the error directly from parameters such as the number of iterations.
  - For models obtained through a non-convex optimization problem (e.g. neural networks), it is significantly more difficult to obtain guarantees from the optimization process. Nonetheless, the behavior during training (e.g. smoothly decaying training and validation losses) could be part of a qualitative safety argument, as sanity checks. Generalization bounds might actually be derived similarly (e.g. noise stability around the weights obtained after gradient descent [DR17]).

3.7.3 Applying generalization guarantees to the transformed models

In Section 3.7.2, it was noted that generalization guarantees usually depended on the in-sample error $E_{\text{in}}(\hat{f}, D, m)$, the dataset D and the class of models to which $\hat{f}$ belongs, but *not* on the training process or how $\hat{f}$ was obtained.

This implies that these guarantees could be also obtained for the final model $\mathcal{T}\hat{f}$ operating the inference environment directly, e.g. by computing

$$E_{\text{in}}(\mathcal{T}\hat{f}, D, m) \tag{3.3}$$

on the *inference environment*, with $\mathcal{T}$ a composition of multiple transformations (abstracting the model, optimizations, etc.; see Sections 3.4 and 3.5).

The *Independent data and learning verification* step of the W-shaped process as described in [CoDANN20, Section 6.1] includes

- the evaluation of $E_{\text{in}}(\mathcal{T}\hat{f}, D, m)$
- the investigation of possible discrepancies with $E_{\text{in}}(\hat{f}, D, m)$

but one could use the latter (with one or several of the datasets) instead of the former for generalization guarantees, see Figure 3.14.

Abstract function vs implementation Section 3.4.1 explained the important distinction between an abstract representation of a computational graph and its implementation. Generalization guarantees apply to mathematical representations and not to implementations, so the above makes the assumption that the inference environment provides the guarantee that the abstract representation of $\mathcal{T}\hat{f}$ and its execution correspond (up to precision-related errors that can be taken into account in the generalization bounds, see [CoDANN20, Chapter 5]).

$$\begin{tikzcd}
\text{Learning environment} \arrow[r, "\mathcal{T}", "\text{Traceability}"'] & \text{Inference environment} \\
E_{\text{in}}(\hat{f}, D, m) \arrow[d, dotted] & E_{\text{in}}(\mathcal{T}\hat{f}, D, m) \arrow[d] \\
\text{Generalization guarantees} \arrow[r, dotted] & \boxed{\text{Generalization guarantees}}
\end{tikzcd}$$

Figure 3.14: Two ways of obtaining generalization guarantees for the inference model in the inference environment (boxed): Transferring generalization bounds from the learning to the inference environment (dotted arrows), or directly obtaining them for the target model/environment (solid arrows).

Advantages of the approach This has the advantage that:

- The bounds side-step the risks related to post-training transformations, as they apply directly to the model/platform to be used during operations;
- Similarly, this might prevent part of the concerns on the learning environment, in particular the need of ensuring that the hardware and software implement exactly the abstract computational graph declared in the top-level framework (see Section 3.4.1);
- The preservation of properties after transformations therefore does not have to be analyzed with the same level of rigor (which might be impossible in the cases shown above). Only some level of traceability is required.
- Generalization bounds are derived on the class of transformed models. For example, if the learning environment considered all possible weights on a given architecture, the inference environment might have to consider pruned or quantized versions. As the transformed models are usually simpler (this is part of the goals of the post-training transformations), the bounds might require less data to arrive at the desired performance. The survey in [CoDANN20, Section 5.3] mentioned generalization approaches based on compression (such as [Aro+18]), that actually force such transformations.

Possible risk or difficulties On the other hand, possible drawbacks are that:

- It might prove difficult to compute (3.3), given that the inference platform is not designed for high data throughput, see Table 3.1.
- Generalization bounds must be derived on the class of transformed model, and might not be readily available or simple to obtain in all cases (to contrast with the corresponding advantage above).
- The traceability between the learning and inference models/environments is not as direct. Section 3.7.2 explained that the training process is in theory irrelevant, but might provide additional points to the overall safety argument.

It is important to remark that even if the link with training is changed, the link with data (and therefore with the first 2 steps of the W-shaped process) remains, as (3.3) is evaluated on the data.

3.7.4 Summary

From the considerations in Section 3.7 and the information surveyed in this chapter, it appears that a possible sound strategy to handle the inevitable post-training transformations and complexity of the learning environment is to:

1. (*Model training*) Optimizations that require additional training such as quantization or pruning are done directly during the original training phase(s).
2. (*Learning process verification*) Evaluate the generalization ability of the trained model and analyze the behavior of the learning processes.
3. (*Model implementation*) Evaluate the impact of transformations that bring the trained model to the inference model (including environment-to-abstraction and vice-versa), e.g. through one of the conditions from Section 3.6.1. This is mostly done for traceability reasons.
4. (*Inference model verification and integration*) Properties that cannot be shown on the learning environment or be carried over through transformations with enough assurance (e.g. generalization guarantees) can be verified on the inference environment, which is possible and valid given the discussion in Section 3.7.3.
   In any case, differences between the trained model and the inference model should still be analyzed (e.g. values of the in-sample errors on the various datasets).[8]

In all cases, the link between the learning and inference environment is maintained. The stringency of the verification in the second step will depend on whether these properties (e.g. generalizability) are also directly demonstrated on the inference model or indirectly (through conservation of model properties).

The focus above was put on generalization guarantees, but similar considerations would apply to other properties.

3.8 Conclusion

This chapter analyzed the particularities of the *learning* and *inference* environments, where machine learning models are respectively developed and deployed.

The learning environment resembles a classical software development environment, but unlike compilation of code into a binary, the whole training process influences the output, from data and model implementation on high-level frameworks to training on specialized hardware.

Naturally, the inference environment is very similar to classical avionics software and hardware, however there are particularities due to the specialized hardware that might be required to run neural networks and their massively parallelized operations.

The steps required to pass from the learning to the inference environment were reviewed: passing from a model tied to an environment to an abstract representation and vice-versa, possibly with optimizations such as quantization, pruning and tuning. See Figure 3.13.

Throughout the chapter, risks attached to both environments and related processes were discussed.

Finally, approaches to mitigate these risks and meet the requirement of performance guarantees during operations were analyzed, in particular by reviewing machine learning assurance theory.

[8]For example, if performance guarantees are obtained from the in-sample error on the inference environment, this value being close to the corresponding one for the original model in the learning environment is a useful sanity check. This would be implied by the (likely unreachable) functional equality.

Chapter 4

Explainability

4.1 Introduction

The concept of explainability of AI/ML systems is one of EASA's building-blocks for AI trustworthiness [EAS20], along with *Trustworthiness Analysis*, *Learning Assurance*, and *Safety Risk Mitigation*.

These buildings-blocks aim in particular at fulfilling the AI Ethics Guidelines from the European Commission High Level Expert Group [EGTA]: *accountability*, *technical robustness and safety*, *oversight*, *privacy and data governance*, *non-discrimination and fairness*, *transparency*, *societal and environmental well-being*.

Definition from EASA's roadmap

The EASA concept paper [EAS21] gives the following preliminary definition:

> *[Explainability] deals with the capability to provide the human with understandable and relevant information on how an AI/ML application is coming to its results.*

As noted by Lipton in his seminal article [Lip18], an issue with the notion of explainability is that it has been widely studied by different communities (philosophy, statistics, machine learning. . .) and therefore possesses several, sometimes conflicting, definitions. Some of these also show the circularity observed here (explainability $\rightarrow$ explanation). To progress towards regulatory guidelines, it is therefore important to reach a definition that is as precise as possible, while being generic enough to not fundamentally contradict existing ones or rule out use cases.

The preliminary definition already suggests that the meaning of "human understandable" will need to be agreed on.

The following sections inquire about the definition of explainability, review existing academic works, discuss requirements in the context of ML-based avionics software, identify remaining steps to achieve explainability in the context of the W-shaped assurance process from [CoDANN20]. Possible solutions for the use case analyzed in this report will be discussed in Section 6.5.

4.2 Defining explainability/interpretability

The Merriam–Webster dictionary [MW] defines "explain" as "*to make known / plain or understandable, to give the reason for or cause of*", "*to show the logical development or relationships*

of", and "interpret" as "*to explain or tell the meaning of; present in understandable terms*".
The following will make a conscious effort to not assign a more specific meaning to these terms than the above, unless explicitly mentioned. Indeed, the goal of this chapter is not to create yet another definition (see again [Lip18] and the widely cited survey [Gui+18] for discussions about the overabundance of specific definitions), but to categorize the existing and possibly future ones, and understand the relevance to certification of machine learning applications.

For the same reason, no explicit distinction will be made between "explainability" and "interpretability", as their respective definitions will vary between groups and authors. Rather, the considerations below will apply to both, regardless of the definition.

4.2.1 Categorization of explanations

Three dimensions of *explanations* emerge naturally from the generic dictionary definitions. See Table 4.1 for the categorization of explanations that follows from these.

Object	System/Output
Recipient	Regulator/system designer/user
Understandability/ transparency	Time and expertise required Simulatability, decomposability, algorithmic

Table 4.1: Properties of an explanation.

The object of the explanation It could be either:

1. The system itself (*a priori*/*global* explanation).
2. An output of the system (*post-hoc*/*a posteriori*/*local* explanation).

Naturally, one can expect that an important part of the explanation of the system itself (and the performance thereof) comes from the argument outlined in [CoDANN20], namely following the W-shaped process. In other words, correct data and learning assurance provide a mathematical proof of the correctness of the system. This proof is human-understandable given that the data and learning assurance are. However, this might not be fully satisfactory with respect to other dimensions, and might need to be complemented with other considerations.

The second object, the outputs, was not considered in [CoDANN20] and will be a major focus here.

The terminology local/global will be avoided, since these might suggest that the local explanations are minor compared to the global ones. Similarly, the *a priori*/*a posteriori* is not well-suited to the aviation context, since *a posteriori* suggests "after system operations", while output of the systems might need to be explained both during and after operations.

The recipient of the explanation The *recipient* of the explanation. In the aviation certification setting, this could be:

- Regulators (who act as domain expert proxies for the public);
- System designers;
- Pilots/operators;
- Investigators in the case of accidents.

In this setting, output-level explanations are likely to be used by all recipients, and system-level by all except the operators.

The meaning of plain/understandable Namely, the complexity of the explanation. Of course, any output from a neural network can be explained by going through the millions of computations performed, but this might not be enlightening or even possible at the human scale. Similarly, an explanation given to a pilot during operations needs to be simpler than the one provided to regulators during an in-depth certification or investigation process. As in [DK17; Gui+18], two sub-dimensions can be identified:

- The *time* required for the recipient to comprehend the explanation (e.g. operations vs certification vs investigation);
- The *expertise* of the recipient (e.g. pilot vs machine learning expert vs regulator).

There is a natural trade-off between explanation power and simplicity, and the choice of the explanation to provide will depend mostly on these two dimensions.
The understandability of an explanation is also called *transparency*. Lipton [Lip18] also distinguishes transparency of AI systems explanations at several levels:

- *Simulatability*: the entire system/model;
- *Decomposability*: components (model parameters, inputs, computations);
- *Algorithmic transparency*: the training algorithm.

4.2.2 Examples

It is useful at this point to illustrate the above dimensions.

Human decisions As Lipton [Lip18] observes, the human brain can usually provide fairly transparent explanations of its decisions/outputs. However, at least in the current state of neuroscience, global explanations are only available when considering decisions that follow from a precise algorithm. In other words, the human brain is a "black box" outside of cases where an AI system would also possess a global explanation.
When humans are involved in safety-critical processes, it is assumed that:

- Their training will provide a global and adequately transparent explanation of their decisions (cf. for example the wide use of checklists both in aviation and in the medical field);
- Even though the human brain has no global explanation, experience over the 200'000 years of existence shows that humans are generally capable of making correct decisions even for situations outside their training, or at least of explaining their thought process.

System versus output explanations Any deterministic model possesses a system explanation (its definition, namely architecture and learned parameters), which also provides output explanations, by simply going through the computations. However, these might not be transparent/understandable, even to a domain expert. This is the case e.g. for neural networks with their millions of weights. Usually, output explanations trade generality for simplicity.

4.3 Explanations for ML-based systems

4.3.1 System-level explanations

Given that deterministic ML-based systems (such as those discussed in [CoDANN20]) always admit system-level explanations, one should examine the transparency and recipients of such possible explanations.

Simple models A small linear model (say with less than ten variables, all having an explicit meaning) and a small decision tree are two examples of machine learning models whose definition themselves provide an almost fully transparent system-level explanation. More precisely, the expertise and time required are minimal, the explanation is simulatable and decomposable, and common training algorithms are simple and understandable by anyone with knowledge in statistical inference.

Object	System-level, therefore output-level
Recipient	Minimal expertise needed
Understandability/ transparency	Fully transparent: Simulatable, decomposable, algorithmically transparent

Table 4.2: Explanations for a "simple" machine learning model.

Complex models/neural networks On the other hand, a deep neural network usually has millions of parameters/weights without explicit/straightforward meaning, so that the explanation given by its definition cannot be deemed understandable by a human, and making sense of how the weights generically lead to a decision is beyond the limits of the human mind.

However, it is possible to provide a global explanation different from the model itself that provides better transparency. This is actually at the core of the [CoDANN20] report. Indeed, the association of data (that drives the function), the W-shaped process and the mathematical arguments explaining generalization/learning assurance together provide a global explanation that has algorithmic transparency and partial decomposability.

- The recipient would be any domain-expert for the data (e.g. pilot) and a machine learning specialist for the learning assurance argument. The latter is performed by the applicant, who could rely on results accepted by the scientific community.
- The time required to understand the explanation is the time to go through the data (which has to be annotated by humans anyway) and to go through the mathematical/statistical arguments and the training/evaluation processes.
- There is no decomposability of the parameters, but there is clear decomposability with respect to inputs and computations (a deep neural network is usually a clear sequence of operations such as convolutions, max poolings, etc.).
  Still, activities such as analyzing features of convolutional neural networks might provide enhanced transparency from the decomposability point of view.

Altogether, the learning assurance process described in [CoDANN20] provides stronger global explanations than those available for complex human decisions. Humans overtake machines in their tested ability to reason, but empirical evidence does not provide global explanations.

Object	System-level, therefore output-level (but there might be more transparent output-level explanations)
Recipient	Domain expert (data) and ML specialist (learning assurance)
Understandability/ transparency	Time to go through data/mathematical argument Decomposability: inputs/computations but not parameters Algorithmic transparency

Table 4.3: System-level explanation for a "complex" machine learning model provided by learning assurance.

4.3.2 Output-level explanations

Given the above discussion on the availability of beyond-human system-wide explanations for machine learning models, this section discusses *output-level* explanations. While system-level explanations yield output-level ones, it is still desirable to have separate output-level explanations, since they can usually be simpler/more transparent by the power/simplicity trade-off.

It is important to identify the recipients of output-level explanations. They are mainly:

- The user/operator, when the AI system is in use. In this case, the requirements are also driven by Human-Machine interface considerations;
- Investigation bodies, when analyzing accidents;
- The regulator, during testing phases of certification.

4.4 Classifying and surveying methods

Several challenges arise when surveying the literature on explainability in the setting of AI/ML:

- As mentioned above, Lipton [Lip18] and Guidotti et al. [Gui+18] recall how multiple authors and/or works have different, sometimes conflicting definitions of explainability.
- The latter survey also posits that most works do not explicitly state what properties their explanations possess (local/global, recipient, transparency; see Table 4.1).
- As a corollary of the previous point, it is sometimes unclear what benefits/requirements certain techniques provide/fulfill. Works such as [Ade+18] demonstrated that some approaches to explainability do not actually provide information on the model, but on the data itself (see Section 4.6.4 below).
- Methods applying to deep learning might be significantly different from "classical" machine learning models. Similarly, techniques often apply to a specific type of input (images, text, sound, sequences of images. . .), and sometimes even to a specific architecture.

4.4.1 The classification of Guidotti et al. (2018)

While not giving an explicit definition, the survey of Guidotti et al. [Gui+18] defines *black box models* as models that are too complex to admit a transparent system-level explanation (given specific users). This qualification should be put in contrast to the observation in Section 4.3 that these models might possess a system-level partially transparent explanation to domain experts, but the considerations in the survey still apply, to seek more transparent explanations.

Categories Guidotti et al. classify research works based on the four different types of problems they might be trying to solve:

1. *Model explanation*: providing system-level explanations through a transparent model that approximates the original model.
2. *Outcome explanation*: finding transparent output-level explanations;
3. *Inspection*: giving a visual representation of a model on any dataset;
4. *Transparent box design*: creating models that, by design, admit a locally or globally transparent explanation (instead of having such models approximating the model of interest).

The last category will not be discussed further here, under the assumption that applications considered could not be engineered with the required performance level using simpler transparent models.

Explanators In each category, the authors also analyze the "explanator" (i.e. the human-understandable explanation) used, among which:

- Simple/transparent models for local approximation: Linear models [ESL, Chapters 3, 4], decision trees [ESL, Section 9.2], decision rules;
- Inputs importance:
  - Numerical: Feature importance [ESL, Sections 3.3-3.4, Chapter 7], sensitivity analysis [Sal02];
  - Visual: Saliency masks, see Section 4.6.4 below.
- Prototype selection [ESL, Chapter 13]: a datapoint, either from the development datasets or derived from them, similar to the input.
- Neural network activations/filters: looking inside the neural networks (by itself, or for a given input). See Sections 4.5 and 4.6 for examples.

The reader is referred to [Gui+18] for the full classification.

4.4.2 Focus of this chapter

The following will focus (as the rest of the report and [CoDANN20]) on *convolutional neural networks applied to images*.

However, one may easily believe that there are simpler techniques for simpler models and similar techniques for different kinds of inputs. In addition to [Gui+18], the book [Mol19] gives an overview of methods that apply to numerical and textual data.

4.5 System-level methods

A convolutional neural network model can generally be seen as the composition of:

- An architecture: a sequence of convolutions, activations and other operations (e.g. pooling).
- Parameters of the relevant operations, most importantly the weights of the convolutional filters.

Given that the number of parameters of deep neural networks can easily reach millions, it is unfeasible to provide a human-comprehensible explanation for all of them. Instead, one can rely on techniques to visualize them (indirectly or directly), given that humans can easily process large amounts of visual information. There exist many methods (see the survey [Qin+18]), and the next sections review three main categories.

4.5.1 Filters visualization

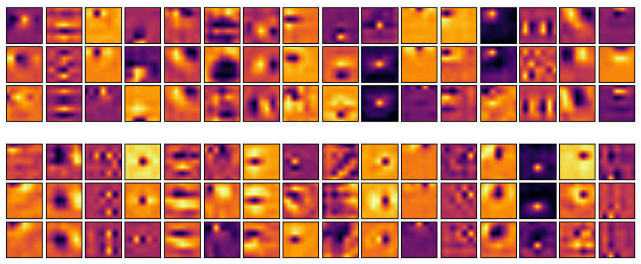
It should be recalled that one of the reasons to consider convolutions instead of fully connected layers is to share parameters and reduce their overall number. Each convolutional layer is determined by f filters/kernels of size $w \times w$ and depth d, where d corresponds to the depth of the previous layer ($d = 3$ on the first layer for RGB images). See Table 4.4 for an example.

Layer	Dimension	Filter size	Number of filters
Input	$224 \times 224 \times 3$	$11 \times 11 \times 3$	96
2	$55 \times 55 \times 96$		
3	$27 \times 27 \times 96$	$5 \times 5 \times 96$	256
4	$27 \times 27 \times 256$		
5	$13 \times 13 \times 256$	$3 \times 3 \times 256$	384
6	$13 \times 13 \times 384$	$3 \times 3 \times 256$	384
7	$13 \times 13 \times 384$	$3 \times 3 \times 384$	256
8	$13 \times 13 \times 256$		

Table 4.4: The convolutions in AlexNet [KSH12].

A straightforward idea is therefore to visualize these filters as 2D images. This is elementary for a convolutional layer applied on the input, as it will be composed of a set of filters of shape $w \times w \times 3$, which can be seen as a set of 2D RGB images (or triplets of monochromatic images). The example in Figure 4.1 makes use of that fact to visualize the filters in an early layer of a CNN. However:

- After the first layer, there will usually be more filters, of higher depth and smaller width/height (but with higher receptive field). For example (see Table 4.4), the fifth and sixth layer of AlexNet contain 384 filters of depth 256 and width/height 3. Some more complex visualization schemes might be used, but this quickly loses human understandability.
- It is not clear how that improves the understanding of the network. Even if all filters were visualized, the way they act and interact to produce the final output from a given input might not be more transparent.


(a) Trained model.

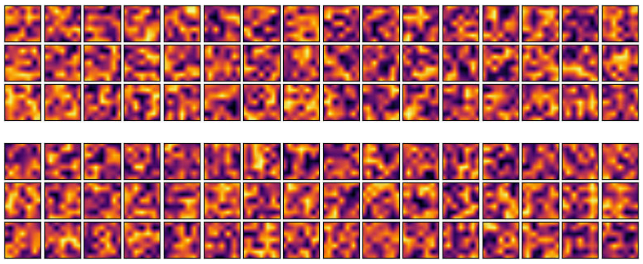

(b) Untrained model.

Figure 4.1: Visualization of the 32 filters of size $7 \times 7 \times 3$ on the first convolutional layer of a CNN trained for aircraft detection. One colum shows the three channels of a filter. For comparison, the filters of the same architecture, but without training (random initializations of parameters). In both cases, the images have been upsampled for clarity.

- Filters that contain noisy patterns might be an indication of a network that has been undertrained, but the converse is not an indication that the model has reached any kind of convergence. One should not acquire a false sense of trust by looking at filters only.

In conclusion, filter visualization tends to suffer from the same dimensionality restrictions as inspections of the weights themselves, and therefore hardly provides a more human-understandable explanation of the behavior of the network. While not providing real explainability, they can still serve as a useful sanity check. Understanding filters might also be relevant in the context of pruning (see Section 3.5.3, [Bla+20]), given that some techniques remove parts of the network based on filter values.

4.5.2 Generative methods

A popular method to understand the role/importance of a part of a deep neural network, be it the output or an intermediary activation, is to generate inputs that maximize this element. Formally, for a model $\hat{f} : X \to Y$ and an intermediary value represented by $g : X \to \mathbb{R}$, one computes

$$\boldsymbol{x}_0 = \operatorname*{arg\,max}_{x \in X} \big(g(\boldsymbol{x}) - R(\boldsymbol{x})\big),$$

where R is a regularizer that incentivizes the creation of "natural images" rather than inputs that artificially maximize g. For example, R can simply be the ℓ_2 norm. The optimization problem can be solved by gradient ascent, similarly to the way the model is trained in the first place (optimizing over the input rather than the weights).

This has the advantage that the input space is by definition interpretable.

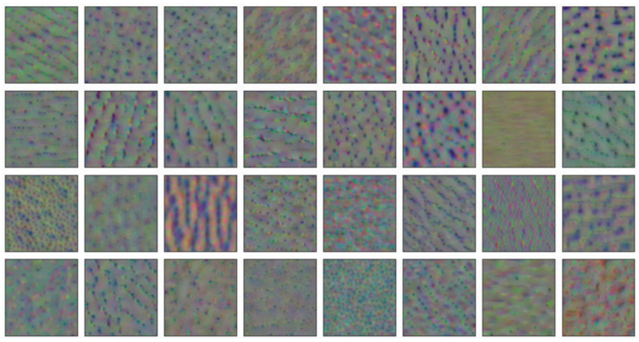


Figure 4.2: Maximally activating images generated from random noise: Of the images above, each one has been designed to yield a particularly high value for one of the 32 activations at an early stage of a CNN, which was trained to detect aircraft. The corresponding optimization process starts with a random noise image.

For example, g could be:

- In the case of a model $\hat{f} : X \to [0,1]^n$ performing classification into n classes and returning probabilities for each class, $g = \hat{f}_i$, the score of the ith class;
- The value of an activation on some intermediary layer.

This method was introduced in [Erh+09], and applications to modern CNNs for supervised learning were studied in [SVZ14]. Figure 4.3 is based on an example from the latter in the classification case, while Figure 4.2, inspired by [Yos+15], illustrates the second case for activations on an early layer of a CNN. In both cases, the network is similar to AlexNet.
Important observations are that (See [Yos+15]):

- Features appear even when looking at single activations (“locality”);
- Several aspects of the objects seem to be learned rather than only discriminative ones;
- Secondary features are learned implicitly.

However:

- Only one (or a finite number) of images are sampled, which does not give a complete view/explanation of all inputs that might maximize g/yield a given output.
- The optimization problem does not have a unique solution, and multiple images, sometimes fairly different can be obtained by changing hyperparameters. Usually, papers present carefully selected examples that are “most interpretable”, which presents the risk of human selection bias.
- In the case of activations visualization, this suffers the same dimensionality problem as before.

Note that this method and the one in the previous paragraph are in-between *Inspection* and *Model explanation* in the classification from Section 4.4.1, in the sense that they do not really provide a system-level explanation, but also do not depend on a dataset (outside of the training/validation datasets). Rather than reflecting an issue in the classification of [Gui+18], this illustrates that these two methods might not be of significant help towards providing missing transparency.

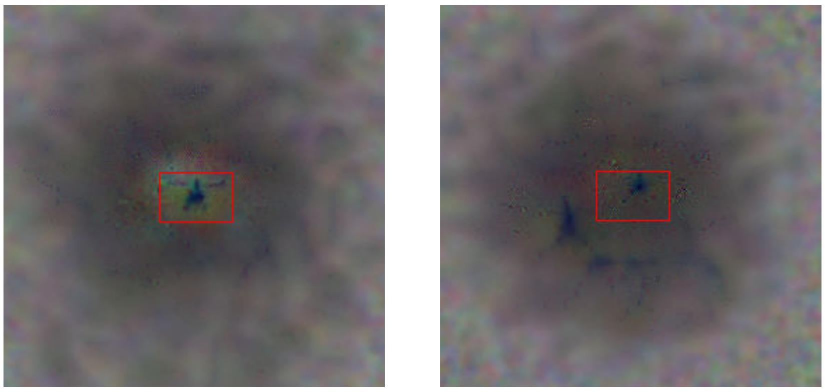

(a) Class "helicopter". (b) Class "background".

Figure 4.3: Input images maximizing the probability of a given class in a fixed bounding box (in red), for an aircraft detection neural network, starting from random noise. Note that the right-hand side image still contains a small object that might be assimilated to a helicopter, because the optimization is also affected by other bounding boxes than the one selected.

Nonetheless, both can act as sanity checks, in the sense that their output is not an explanation/guarantee of correct behavior, but might help uncover faulty behaviors (e.g. if the image that maximally activates a class contains representations of a totally different class).

4.5.3 Maximally activating inputs

Given a quantity $g : X \to \mathbb{R}$ of interest inside the network, instead of the optimization outlined in Section 4.5.2, a simpler method is to feed a set of images (e.g. the training, validation or test sets) into the network, and record the inputs for which g is maximal.

An example for this is given in Figure 4.4, where g is the average over a group of activations. In the case of classification, a similar technique can be used to find images where the network is the most confident in the predicted class.

4.6 Output-level methods

The previous methods were working on the model level, and it was noted that they might provide useful sanity checks, but presented challenges to serve as transparent system-level explanations. The following techniques solve either the *Model explanation* (*locally*), *Outcome explanation* or *Model inspection* problems from the classification of Guidotti et al. from Section 4.4.1, therefore in the category of output-level explanation according to Table 4.1.

4.6.1 Local approximation

As explained in Section 4.3, simpler models are more transparent by definition, but they will often not be good enough for complex use cases. Similarly, approximations of a complex model by a simpler model (i.e. the *Model explanation* task in the classification of Guidotti et al. from Section 4.4.1) might not be possible with having both a faithful and transparent approximation.

However, the problem becomes easier if one relaxes the faithfulness of the explanation to be only true *locally*. The locality of the approximation can be around a given input point, or in a

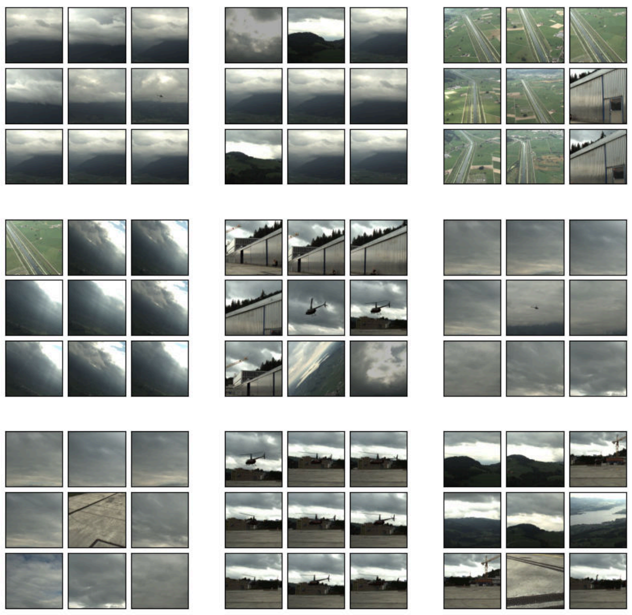


Figure 4.4: Maximally activating input images for a CNN tasked with the detection of aircraft: Each cluster of 9 maximizes the mean over a certain group of activations in an intermediary layer of the CNN.

region of interest for an explanation.

There is usually a natural trade-off between:

- The quality of the approximation (how close the original and explanator model are);
- The locality of the approximation (the size of the region where the approximation is satisfactory);
- The transparency of the explanator model.

This trade-off is expressed by Ribeiro et al. [RSG16] as

$$E(g) = L(\hat{f}, g, \pi_{x_0}) + \Omega(g), \qquad \text{where:}$$

- $\hat{f} : X \to Y$ is the complex model under analysis;
- g is a transparent model/explanator;
- Ω a measure of the complexity (and therefore transparency) of g;
- π_{x_0} a density function expressing locality around a point $\boldsymbol{x}_0 \in X$;
- L a measure of the quality of the approximation of $\hat{f}$ by g around $\boldsymbol{x}_0$ (with respect to π_{x_0}).

For example, if $X = \mathbb{R}^n$ and $Y = \mathbb{R}^m$, one may set

$$L(\hat{f}, g, \pi_{x_0}) = \int_X ||\hat{f}(\boldsymbol{x}) - g(\boldsymbol{x})||^2 \pi_{x_0}(\boldsymbol{x}) d\boldsymbol{x},$$

i.e. the difference between $\hat{f}$ and g weighted by the locality measure, and if g is a linear model, $\Omega(g)$ might be the number of non-zero variables or their norms. If g is a decision tree, $\Omega(g)$ might be its size.

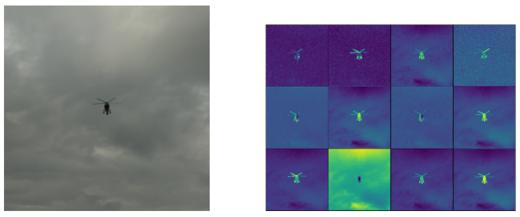

(a) Original image (b) Resulting activations

Figure 4.5: CNN activations visualized: The picture to the right shows a subset of the activations of a CNN's intermediary layer. The CNN at hand has been trained to detect aircraft. Light pixels correspond to high activations. The input image can be seen on the left. The input image is of size $512 \times 512 \times 3$ pixels (3 color channels are present) and the resulting activation has shape $1 \times 32 \times 128 \times 128$, which is due to the convolutional mechanics of the network. Each of the 12 pictures to the right visualizes one of these 32 activation matrices of size 128×128.

This is best understood when g lies in a family of transparent models G, with the other quantities fixed, and one tries to find the best trade-off with respect to the above, i.e. $\arg\min_{g\in G} E(g)$. This formulation has the advantage of allowing to control the trade-off between approximation power and transparency explicitly. In most cases, the optimization problem is intractable, but its solution can be approximated.

LIME The Local Interpretable Model-agnostic Explanations (LIME) framework of [RSG16] gives a general method for doing so when $\hat{f} : \mathbb{R}^n \to \mathbb{R}$ and g is a linear model with at most K features, without requiring internal knowledge of $\hat{f}$: one simply samples points around $\boldsymbol{x}_0$ (according to $\pi_{\boldsymbol{x}_0}$), obtain their predictions from $\hat{f}$, and train a linear model using these and the predictions, limiting to the use of only K features from the original data.

Selecting locality The definition of the neighborhood (equivalently of the density $\pi_{\boldsymbol{x}_0}$) is delicate and depends on the input space X. For images, it would not make sense to look at pixel-level perturbations, as the resulting data will be too complex for the simpler model, and these might be too fine to even show changes in the classification. Instead, Ribeiro et al. [RSG16] suggest using r clusters of connected pixels ("super-pixels") in $\boldsymbol{x}_0$ that can be grayed out. The neighborhood of $\boldsymbol{x}_0$ is identified to $\{0,1\}^r$, with $\boldsymbol{x}_0$ identified to the unit vector $\mathbf{1}$.

4.6.2 Activations visualization

Given an input image, a straightforward method to delve into the computations happening in the CNN that yield the output is to visualize the activation layers. This is dual to the filters visualization exposed in Section 4.5.1, with a lower dimensionality (according to Table 4.4, there are between 96 and 384 activations in the intermediary layers, with a maximum size of 55×55).

This allowed Yosinski et al. [Yos+15] to create a tool that allows visualizing activations layer by layer in real time (see Figure 4.5). They compared cases where the network performs well to ones where it does not, and observed different behaviors in the activations.

Like with filters, analyzing activations can also provide insights towards pruning (Section 3.5.3, [Bla+20]) and about the training process (see the references in [Lu+20]), but this is off-topic for explainability.

4.6.3 Activations visualization in the input space

Instead of visualizing activations themselves, a category of methods attempts to provide a visualization in the input space, similar to Section 4.5.2.

Given an input image $\boldsymbol{x}_0$, one would like to understand which input pixels have the largest effect on the activation $g : X \to \mathbb{R}$.

One of the first methods is the DeconvNet of Zeiler and Fergus [ZF14], which attempts to reverse the forward operations to obtain an input that produced the activation. The authors discuss the evolution of such features during training, invariance to transformation, and the general type of each feature at each level. The first layers tend to have high-level features such as corners, colors or patterns, while the last layers might show complete objects.

The DeconvNet method was shown by [SVZ14] to be very close (up to the effect on the activations functions) to the following simple idea: one may approximate g linearly (as in Section 4.6.1) as

$$g(\boldsymbol{x}_0) \approx \boldsymbol{w}(\boldsymbol{x}_0)^T \boldsymbol{x} + b, \qquad \boldsymbol{w}(\boldsymbol{x}_0) = \frac{\partial g}{\partial \boldsymbol{x}}\Big|_{\boldsymbol{x}_0}$$

and the magnitude of each element of $\boldsymbol{w}(\boldsymbol{x}_0)$ denotes the influence of the corresponding pixel in x on the value of g, around $\boldsymbol{x}_0$. The derivative can be found by back-propagation (over the input instead of the weights), and [SVZ14, Section 4] shows the quasi-equivalence of the DeconvNet and this back-propagation.

In the classification case $\hat{f} : X \to [0,1]^n$, this method can also be applied when g is the class score f_i, providing an idea of which pixels in the input have the largest influence on the score, for the given image. This is an example of saliency map, discussed in the next section. See [SVZ14, Figure 2] for an example.

4.6.4 Saliency maps

A natural question to ask for models operating on complex inputs is

> *Which parts of the input were used, and how/to which extent, to arrive at the output?*

The importance of this question is particularly clear for models working on images, see Figure 4.6.

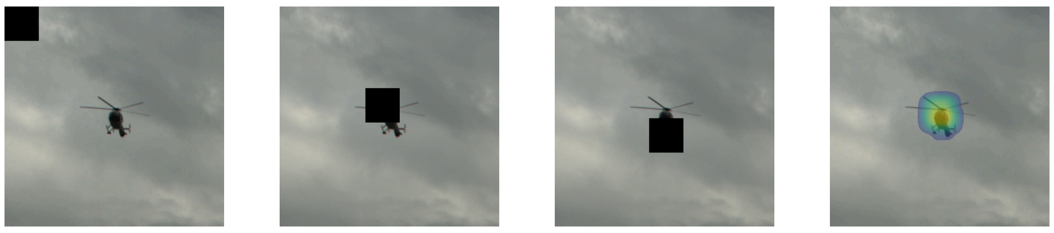

(a) Prob. 1.00 (b) Prob. 0.91 (c) Prob. 0.99 (d) Saliency map.

Figure 4.6: Saliency map computed from occlusion. A rectangle filled with the mean image color (represented as black here for clarity) is moved over the image, occluding a small region at a time. The intensity of each pixel in the saliency map represents the decay of the probability that the image contains a helicopter (as measured by the neural network) when this pixel is occluded.

These output-level/outcome explanations can have multiple uses, such as providing an additional input to the operator or uncovering potentially faulty behaviors. For example, Ribeiro et al. [RSG16] trained a small dog vs wolf classifier that turned out to use snow as a discriminative feature instead of the animal, and were able to uncover this on an unseen image using a saliency map (see Figure 4.7).

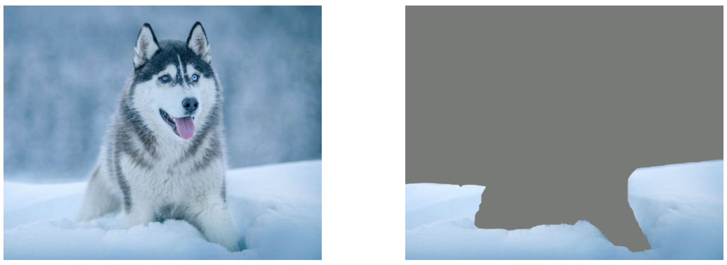

(a) Husky classified as wolf. (b) Explanation.

Figure 4.7: Saliency map detecting faulty discriminative features, adapted from [RSG16, Figure 11].

Some popular techniques There exist many techniques to compute saliency maps, and these usually depend on:

- The meaning of the visualization (class probabilities, top class, distance between top 2 classes, gradient. . .);
- The "visual quality" of the saliency maps produced;
- Whether they are model-agnostic, or specific to certain types of models/architectures;
- Whether they need access to the model internals;
- Whether they are based on optimization or not.

The simplest method, discussed by [ZF14], is to perturb parts of the image (e.g. replacing by a constant color, blur or noise), and observe whether the prediction changes. In the context of classification, that might mean observing the probability of the originally predicted class, or the predicted class for the occluded image. This is illustrated in Figure 4.6.

The *gradient/backpropagation* method of [SVZ14] was already discussed in Section 4.6.3. It visualizes the gradient of the class probability with respect to the image, and therefore shows the local influence of each pixel on the probability.

Guided backpropagation [Spr+15] is a variant that only backpropagates positive gradients and activations, the idea being to improve the aspect of the saliency maps by focusing on what makes the given prediction, rather than what does not. Like [SVZ14], this can also be used for features visualization.

CAM [Zho+16] is another popular method for a specific architecture (global average pooling followed by a fully connected layer at the end of the network), that was then adapted to generic architectures as *Grad-CAM* [Sel+17], as well followed by several variants.

The LIME method on images presented in Section 4.6.1, using super-pixels, also provides a way to create saliency maps: the explanator linear model will use a small number K of super-pixels to arrive at the predicted value at $\boldsymbol{x}_0$, and the K coefficients can be inspected to understand the influence of each super-pixel; see Figure 4.8.

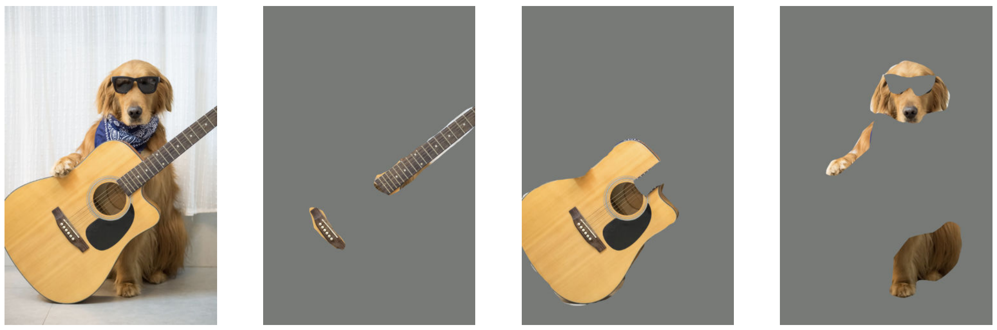

(a) Original image. (b) "Electric guitar". (c) "Acoustic guitar". (d) "Retriever".

Figure 4.8: Using LIME on super-pixels for saliency maps, adapted from [RSG16, Figure 4]: original image and explanation for three classes.

Criticism and risks Given the large amount of saliency map techniques that are available and keep being developed, Adebayo et al. [Ade+18] analyzed methods including the ones discussed above, and came to the conclusion that several did not provide any actual insight on the models. Quoting the first page of the paper:

> "*some widely deployed saliency methods are independent of both the data the model was trained on, and the model parameters*"

The authors performed a set of qualitative and quantitative tests on methods, including:

1. Model parameters randomization: This compares the saliency maps for a trained model and an untrained one, and therefore indicates whether the saliency maps depend on the learned weights.
2. Data randomization: This compares the saliency maps between a trained model and a model trained on the same data, but with (non-consistently) permuted ground truth labels, showing whether the saliency map depends on the relationship between input and labels and generalizable aspects (the second model memorizing the data). See [Ade+18, Figure 6].
3. Comparing the saliency map to the output of a single edge detector. See [Ade+18, Figure 1].

In short, it turns out that the gradient method from [SVZ14] and Grad-CAM from [Sel+17] pass the sanity checks, while in particular Guided-backprop [Spr+15] does not.

The paper of Rudin [Rud19] underlines similar issues, see Figure 4.9.

As in the previous methods, this emphasizes a risk of explainability methods, being that they might give a false sense of confidence or understanding through human confirmation bias, and the need for precise requirements a priori.

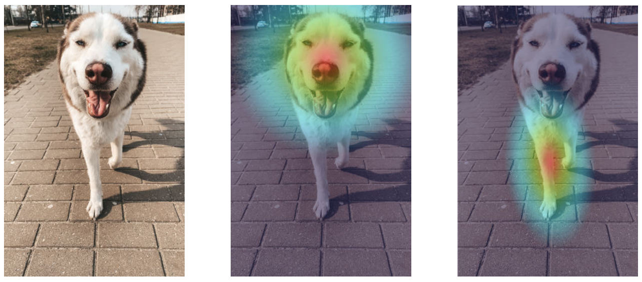

(a) Original image. (b) "Husky". (c) "Flute".

Figure 4.9: Adapted from [Rud19, Figure 2]: *"Saliency does not explain anything except where the network is looking"*.

4.7 Working groups around explainability

In the following, an overview of the most important working groups in the field of explainability is provided. Given the context of this report, the focus is put on groups that research either model-agnostic methods that apply to image processing or methods targeting images specifically.

Each working group is presented with regard to the background of their research, a description of their contribution and a classification of the latter with respect to the classification framework presented in Section 4.4. The latter is contained in the columns "Object of Explanation" and "Explanation Effect" of the respective summary tables.

4.7.1 Fraunhofer Heinrich Hertz Institute: XAI Group

This group around researchers Wojciech Samek (head of the XAI group[1] at Fraunhofer Heinrich Hertz Institute) and Grégoire Montavon (Technical University of Berlin) has published one book [Sam+19] and several papers on explainability, especially but not only in the context of deep neural networks and image classification. Three of their papers in the area are highly cited ones [MSM18; Mon+17; Bac+15]. Regarding the motivation behind their research, they identify four main benefits of explainable AI systems [SWM17]:

- Verification: Such systems allow humans to understand the decision-making process and therefore to see whether it is generally logically sound. In particular, obviously unreasonable decision processes that do not generalize well can be identified at an early stage.
- Improvement: Understanding a system's decision-making process could give a better understanding of its weaknesses and therefore provide a promising starting point for improving it.
- Learning: There are tasks in which AI systems significantly outperform humans. Understanding the processes that make AI systems so good might therefore generate novel insights into these tasks.
- Compliance to regulation: In many use cases, notably safety-critical ones, the missing understanding of an AI system's inner workings is problematic from a regulatory standpoint. In these use cases, making AI systems explainable is a necessary step in employing them at all.

[1] https://www.hhi.fraunhofer.de/en/departments/ai/research-groups/explainable-artificial-intelligence.html

Approach	**Object of Explanation**	**Explanation Effect**
Layer-wise relevance propagation (LRP) [Bac+15]	Particular outputs	General explanation framework. Can be used to create saliency maps for images
Deep Taylor decomposition [Mon+17]	Particular outputs	Can be used to create saliency maps for images

Table 4.5: Summary and classification of contributions by Samek's explainable AI group and its collaborators.

The group has published multiple surveys that give an overview on some popular explainability techniques. Among the ones they mention are *activation maximization* [MSM18, pg. 2], *sensitivity analysis* [SWM17, pg. 3] and *simple Taylor decomposition* [MSM18, pg. 4].
In addition, they have also presented two entirely new approaches concerning explainability, namely:

- *Layer-wise relevance propagation (LRP)* [Bac+15]: This is a somewhat model-agnostic method as it applies to any model that computes its decision in different layers (e.g. deep neural networks). The relevance of different input variables to specific predictions is explained by propagating a measure of relevance from output to input in a layer-wise manner while conserving the total relevance in between different layers. This can be used to generate saliency maps.
- *Deep Taylor decomposition* [Mon+17]: This is a specific explanation approach, though it also explains the relevance of different input variables to specific model predictions. It can also be used to generate saliency maps.

4.7.2 DARPA's Explainable AI program

In May 2017, DARPA (Defense Advanced Research Projects Agency, a branch of the US government) launched a 4-year research program into explainable AI [GA19]. The program's focus is on explainability as a means to create AI systems that can interact with human users and is an umbrella for a multitude of different explainability research projects by different research groups.

Approach	**Object of Explanation**	**Explanatory Effect**
Varies between subgroups	Varies between subgroups	Varies between subgroups

Table 4.6: Summary and classification of contributions by the subgroups within DARPA's explainable AI program.

The range of topics covered by DARPA's program is quite broad, and while some projects target computer vision problems, others examine more general approaches to explainability or even psychological considerations. In general, the contribution of each subgroup is broken down into two areas: "explainable models" and "explanation interfaces", referring to the underlying machine learning models and the way in which a result is explained to the user, respectively.

4.7.3 Duke University: Prediction Analysis Lab

Cynthia Rudin and her Prediction Analysis Lab[2] at Duke University (formerly at MIT) have produced multiple highly cited articles on explainability. Rudin's research focuses on applications of machine learning in healthcare, criminology and power infrastructure. She has emphasized the importance of deploying transparent (ergo explainable) models instead of black-box ones in high-stakes use cases [Rud19].

Approach	**Object of Explanation**	**Explanatory Effect**
Bayesian rule sets (BRS) [Wan+17]	Model itself	Inherently explainable model (simulatable and decomposable)
Prototype-based NNs [Let+15]	Model itself	Increased model explainability (insight into a learned aspect of the model)

Table 4.7: Summary and classification of contributions by Rudin and her Prediction Analysis Lab.

Rudin and her group have proposed different designs for such inherently explainable ML models. Two notable ones are:

- *Bayesian rule sets (BRS)* [Wan+17]: Here, the authors propose a technique to generate a classification model consisting of a manageable number of if-else-then rules (see [Wan+17, Figure 1]). The advantage of this is that these if-else-then rules are easily comprehensible by humans.
- *Prototype-based neural networks* [Let+15]: This is a special neural network architecture in which the model is enforced to learn different prototypes and make decisions based on how inputs relate to these different prototypes. Since the learned prototypes can be inspected, this generates additional insights into the model.

In terms of classification, models consisting of Bayesian rule sets allow interpretation at the model-level. Due to their limited complexity and their if-else-then character, the generated models are simulatable: It is easy to take an input and go through a number of if-else-then rules. Decomposability is also given since such models can be decomposed into a limited number of concrete and inherently mutually exclusive rules. The prototype-based neural network design also seeks to clarify aspects of the model itself. However, the proposed class of models are still not simulatable as they contain neural networks. They are also neither decomposable (no natural decomposition presents itself) nor algorithmically transparent (the training process still involves training a neural network).

4.7.4 Shanghai Jiao Tong University: Explainable AI group

Quanshi Zhang's explainable AI group[3] at Shanghai Jiao Tong University has contributed to multiple highly cited articles regarding explainability of convolutional neural networks. They argue that an explanation of a CNN's logic is necessary if humans are to really trust its decisions [ZWZ18].

[2]https://users.cs.duke.edu/~cynthia/lab.html
[3]https://sites.google.com/site/quanshizhang/home/explainableaigroup

Approach	Object of Explanation	Explanatory Effect
Interpretable CNNs [ZWZ18]	Model (filters)	CNN filters tend to represent decoupled and hence more interpretable object parts
Explanatory graphs for CNNs [Zha+18]	Model	Component and subcomponent relationships that drive detection are revealed
Decision trees for CNNs [Zha+19]	Model	Decision mechanism is described from top-level to elementary decisions
Quantification of NN input relevance [Che+19]	Particular decisions	The relevance of each input factor to a particular decision is quantified

Table 4.8: Summary and classification of contributions by Zhang and his exlainable AI group.

Zhang and his co-authors have proposed multiple methods to increase the explainability of CNNs:

- *Interpretable CNNs* [ZWZ18]: Here, an altered architecture incentivizes CNNs filters to represent decoupled and therefore more easily interpretable object parts. For example, such an interpretable CNN that is trained to detect the presence of cats on images may have e.g. one filter that corresponds to a face and another one that corresponds to a leg (see [ZWZ18, Figure 1]). An ordinary CNN, on the other hand, might have filters that represent mixtures of parts.
- *Explanatory graphs for CNNs* [Zha+18] are automatically constructed from an already trained CNN to explain how it detects an object. Simply put, the goal of such a graph is to explain what object components need to be present for the object to be recognized as present. In turn, the recognition of object components might require the presence of object subcomponents, etc. See [Zha+18, Figure 10].
- *Decision trees for CNNs* [Zha+19]: In this technique, a decision tree is automatically generated to explain a CNNs decision-making. For this, "decisions" are arranged in a tree-like structure (see [Zha+19, Figure 2]) that the authors call "explanatory tree", with decisions starting at a coarse-grained level (e.g. concerning high-level image features) that in turn depend on increasingly fine-grained decisions (e.g. concerning more elementary features) when considering decisions at a deeper level of the tree.
- *Quantification of NN input relevance* [Che+19]: This technique attempts to explain the influence of different decision factors of a CNN by assigning their influence on the final decision a concrete numeric value.

4.7.5 Microsoft Research's Adaptive Systems and Interaction group

Microsoft Research's Adaptive Systems and Interaction Group[4] researches ways of developing AI systems that are trustworthy and seeks to increase mankind's understanding of the principles behind computational intelligence. They describe their work as "motivated by the goal of

[4]https://www.microsoft.com/en-us/research/group/adaptive-systems-and-interaction/

developing systems that can perform well amidst the complexities of the open world, either via autonomous execution or in their collaboration with people". While the group comprises more than a dozen researchers, the works of Marco Tulio Ribeiro and Scott Lundberg are deemed particularly important to this report.

Approach	**Object of Explanation**	**Explanatory Effect**
Local interpretable model-agnostic explanations (LIME) [RSG16]	Behavior of model around a particular input	Model is explained locally through a simpler, but only locally valid approximation; can be used to generate saliency maps
Shapley additive explanations (SHAP) [LL17]	Particular predictions	The relevance of each output to the model decision is quantified

Table 4.9: Summary and classification of contributions by Microsoft Research's Adaptive Systems and Interaction group.

Techniques proposed by researchers within this group include:

- *Local interpretable model-agnostic explanations (LIME)* [RSG16]: This is a broadly applicable interpretation framework and has already been covered in some detail in Section 4.6.1.
- *Shapley additive explanations* [LL17]: Here, a framework is presented that can explain particular model predictions by revealing which inputs made the prediction more or less probable.

4.7.6 Google

Google employs a large number of researchers that work on a plethora of different machine learning topics. Three notable ones in the context of explainability are Been Kim (Google Brain), Karen Simonyan (Google DeepMind) and Mukund Sundararajan.
Contributions by researchers at Google include:

- *MMD-critic* [KKK16]: This is a technique that aims to provide a better understanding of a model by generating prototype samples and criticism samples of that model (see [KKK16, Figure 1]). While prototype samples are samples that can be classified well by the model, criticism samples are data points that do not fit the model well.
- *Testing with concept activation vectors (TCAV)* [Kim+18]: This technique aims at explaining specific model predictions. It specifically applies to image classification and quantifies the relevance of specific user-defined visual concepts (e.g. stripiness) for the classification decision (e.g. picture contains zebra). See [Kim+18, Figure 1].
- *PatternAttribution* [Kin+18]: This is a way of tracing a model's particular classification decision back to specific pixels of an image, which gives rise to a saliency map.
- *Bayesian case model (BCM)* [KRS14]: A clustering framework that both finds clusters in an unsupervised manner and explains them. The explaining is done by automatically providing prototypes and important features for each cluster.

Approach	**Object of Explanation**	**Explanatory Effect**
MMD-critic [KKK16]	Model	Insight into the kinds of inputs that fit the model well vs. the kinds that do not
Testing with concept activation vectors (TCAV) [Kim+18]	Model	Insight into the relevance of visual concepts to model predictions
PatternAttribution [Kin+18]	Particular predictions	Extraction of the most relevant inputs to particular model predictions
Bayesian case model (BCM) [KRS14]	Particular outputs (clusters)	Clustering outcome is explained by providing prototypes and relevant features for each cluster
Visualization of classes learned by CNNs [SVZ14]	Model	Learned classes are visualized
Class saliency extraction method [SVZ14]	Particular predictions	Extraction of most relevant inputs to particular model predictions
Integrated Gradients [STY17]	Particular predictions	Extraction of most relevant inputs to particular model predictions; can be used to generate saliency maps

Table 4.10: Summary and classification of contributions by Google's researchers.

- *Visualization of classes learned by CNNs* [SVZ14]: The idea here is to visualize a class learned by a CNN by generating an image that is representative of this class.
- *Class saliency extraction method* [SVZ14]: This technique concerns the generation of saliency maps. It therefore explains specific model predictions by highlighting the pixels on the input image that are most relevant to the classification decision, see [SVZ14, Figure 2].
- *Integrated Gradients* [STY17]: This is yet another technique to attribute particular model decisions to specific input parameters and can also be used to generate a kind of saliency map.

4.7.7 DEEL: Dependable and Explainable Learning

DEEL is a collaboration between institutions from Toulouse (France) and Québec (Canada) and pursues the goal of AI systems that can be employed in safety-critical applications like avionics. The project is funded with a budget of 30 million euros and is divided into four areas, one them being explainability.

Approach	**Object of Explanation**	**Explanatory Effect**
Entropic variable projection [Bac+18]	Particular predictions	Quantification of how the status of specific inputs generally affects class probabilities predictions

Table 4.11: Summary of contributions by DEEL and affiliated researchers.

Entropic variable projection [Bac+18] is a model-agnostic technique proposed by researchers affiliated with DEEL. To explain particular model predictions, the method relies on quantifying the relevance of each input variable to each class of model decisions. For example, in the case of image classification, this technique can assign each pixel a probability for each class that expresses how much more likely it is that an image be assigned to this respective class when this pixel is activated (vs. having background intensity) [Bac+18]. The visualized outcome has similarities to a saliency map, but explains a model's classification characteristics in general rather than explaining one particular prediction. A Python implementation of this technique is available as part of the package "Ethik AI"[5].

4.7.8 NIST's explainable AI group

The United States of America's National Institute of Standards and Technology (NIST) runs a working group on explainable AI [6] as part of their effort to "measure and understand the capabilities and limitations of [AI] technologies"[7]. In this context, NIST recently released a draft whitepaper [Phi+20] in which they describe what they consider the four fundamental principles behind explainable AI and reflect whether / how existing explainable AI methods and human explanations fulfill these four principles.

[5] https://xai-aniti.github.io/ethik/
[6] https://www.nist.gov/artificial-intelligence/ai-foundational-research-explainability
[7] https://www.nist.gov/artificial-intelligence/ai-research

The presented principles are:

- *Explanation*: Systems deliver accompanying evidence or reason(s) for all outputs.
- *Meaningful*: Systems provide explanations that are understandable to individual users.
- *Explanation Accuracy*: The explanation correctly reflects the system's process for generating the output.
- *Knowledge Limits*: The system only operates under conditions for which it was designed or when the system reaches a sufficient confidence in its output.

Rather than proposing concrete techniques, their contribution aims at laying the conceptual groundwork for future advances in explainable AI and seeks to initiate a discourse around the topic.
To compare the four principles proposed in the whitepaper with the classification framework presented in Section 4.4, the former are more concerned with what constitutes an explanation in general while the latter distinguishes kinds of explanations based on the explained aspect of the system and the kind of transparency provided.
Generally speaking, it can be noted that there is a degree of correspondence between the principles of explainable AI proposed in [Phi+20] and the considerations on explainability in this report.
The first principle ("explanation") is inherently fulfilled by any of the explanation methods described in Section 4.5 and Section 4.6, as they all try to show how AI systems determine their output, either by providing system-level (Section 4.5) or output-level (Section 4.6) explanations.
The second principle ("meaningful") has been thematized in the considerations on the classification of explainability techniques Section 4.2.1. There, it has been recognized that different explainability techniques yield differing kinds and degrees of transparency, and this insight is also addressed in the description of different explainability techniques in Sections 4.5 and 4.6.
Regarding the third principle ("explanation accuracy"), it has been recognized that there "is a natural trade-off between explanation power and simplicity" (see Page 54).
The fourth principle ("knowledge limits") has been covered in Chapter 5 and Section 4.8.1: Instead of looking into ways of detecting insufficient confidence of a ML model, the considerations therein address knowledge limits through a discussion on how one can guarantee that ML model inputs fall within them in the first place.

4.8 Understanding needs for explainability

The sections above illustrated that there exist many methods to visualize neural networks and their predictions, but that it is fundamental to set requirements, in particular in light of [Ade+18; Rud19].

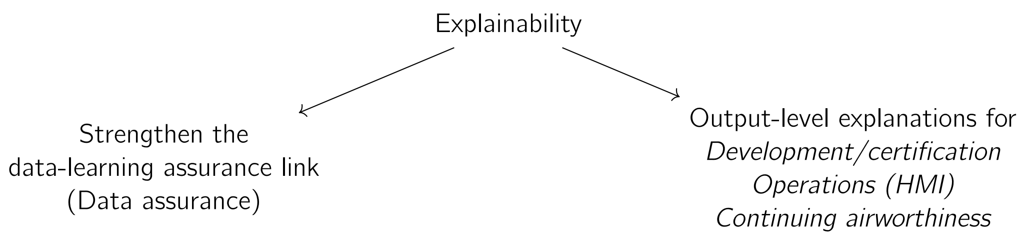
In particular, applying techniques blindly might just produce information that does not actually provide any arguments towards trust in the system, and worse, might lead to an incorrect sense of additional confidence.

System-level explanations through generalization As argued in Section 4.3, an analysis of the neural network as in [CoDANN20, Chapters 6, 9], based on statistical generalization guarantees (and possibly other methods such as simulation), will provide detailed probability distributions for the different failure cases of the model. Altogether, this provides a system-level explanation of its correctness, transparent to the recipient (designer/regulator/investigator).

Additional activities On the other hand, the following activities would help bridge the remaining gaps (discussed in Section 4.3) between "classical" software and software where the function is mostly learned from data:

- As discussed in Section 5.1 and [CoDANN20], a crucial assumption for the learning assurance guarantees is that the operational space has been correctly identified, and the training/validation/testing datasets have been correctly sampled. Explainability, as defined in Section 4.2, could be used to strengthen this data–learning assurance link as part of the *Independent data and learning verification* of the W-shaped process.
- Then, while it is not possible to give global transparency to the model, output-level explanations can be provided:
  - During the system design/certification (this also fits into the previous objective of verifying the data–learning link).
  - During operations, to give additional insights to the pilot/operator about the system outputs, helping in decision-making. This is closely tied to Human-Machine interaction (HMI) considerations.
  - Understand failure cases more deeply than them just falling into the admissible failure probability from learning guarantees. This might be useful during development (correct systematic errors), after operations (continuing airworthiness) and for potential investigations.

The first category of activities is discussed further in the next section, while the second, more dependent on the use case, is discussed in the context of the traffic detection system in Section 6.5.


Explainability
Strengthen the data-learning assurance link (Data assurance)
Output-level explanations for
Development/certification
Operations (HMI)
Continuing airworthiness

4.8.1 Strengthening the data–learning assurance link

An incomplete specification of the operational space turns out to be one of the most common pitfalls in real-world machine learning applications, and an important share of "domain bias" errors arise from the fact that the model uses parts of the data that do not lend themselves to generalization, unlike what the system designers were believing.

The folklore example (according to [Bra19], which contains an extensive list of real-world cases) is the tank detection system that relies solely on time-of-day, and Section 4.6.4 introduced the "wolf vs dog" variant. It is important to note that these errors would definitely violate the requirements on data from [CoDANN20, Chapter 6]. The "tank detection" example does not satisfy the very simple assumption that the data should be (in particular) uniformly distributed in the explicit operating parameters

$$\{\text{time of day}\} \times \{\text{presence of tank}\},$$

assuming that the system is meant to work at any time of day.

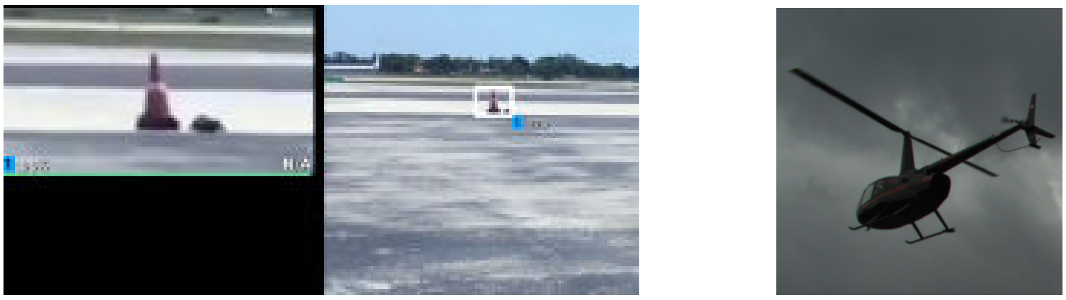


Figure 4.10: Another example of the kind of unexpected and undesirable input-output relationships that can be identified with explainability methods. Here, an object detection network tasked with the detection of aircraft falsely identified a traffic cone as a helicopter (left, screenshot from an internal demonstration user interface). Interestingly, the network was trained using images of R66-type helicopters (right), whose rotors rest on a part that resembles the triangular shape of a traffic cone.

The "explainability" methods surveyed above could be applied in the *Independent data and learning verification* part of the W-shaped process (or earlier) as a complement to the methods discussed in Section 5.1. These do *not* add missing hypotheses to learning assurance, but rather provide additional techniques to recognize a misidentification of the operational space. They focus on using the trained model rather than ad hoc methods working on the data only.

Like classical testing, these methods should be seen as testing against the null hypothesis that the model uses the datasets in a way that does not match the operational space, and gathering enough evidence to reject it.

System-level methods Section 4.5 provided a short survey of system-level methods, and they all can be used to collect evidence towards the above.

Filters visualization (Section 4.5.1) suffers from dimensionality issues, but the first layers can still be analyzed, where noisy patterns might reveal a model that might have been undertrained for the complex task at hand.

Maximally activating inputs (Section 4.5.3) can be used to visualize training/testing datapoints where the model has extremal behaviors (e.g. maximal confidence among all outputs with a certain property). Errors uncovered there should however also have been detected during training/validation/testing.

On the other hand, generative methods (Section 4.5.2) create maximally activating inputs not present in the original datasets. They might uncover:

- Undesirable artifacts that the model might be using from the data (e.g. brand logo to classify helicopters as such, or time of day in the context of the tank detection example [Bra19]).
- Inputs that maximize some outputs while not containing data that should trigger such behavior.

An arbitrary number of images can be generated (e.g. by varying hyperparameters of the optimization), and one should pay attention to confirmation bias[8].

Finally, the output-level methods from Section 4.6 can all be applied on training/validation/testing datasets. Given their sizes, sampling or automatic analysis might be useful.

[8] "*The tendency to search for, interpret, favor, and recall information in a way that confirms or supports one's prior beliefs or values*" [Nic98].

Chapter 5

Safety Assessment

5.1 Out-of-Distribution (OOD) detection

As discussed in [CoDANN20, Chapters 5,6,9], a crucial assumption to guarantee performance of a machine learning model on unseen data is that the operational space has been correctly identified. This is a double-sided requirement, concerning multiple stages of the system lifecycle, where both:

- the design data (training, validation and testing datasets) and
- the operational data

must be sampled uniformly from the operational probability space $\mathcal{X}$, as illustrated below. Remember from [CoDANN20, Section 5.2] that this is the probabilistic statement "$D \sim \mathcal{X}^{|D|}$", including the fact that the points are sampled independently. The term "D iid in $\mathcal{X}$" will be used interchangeably.

$$\begin{tikzcd}
\text{Train, val., test datasets} \arrow[r] & \text{Operational space} & \text{Input data during operation} \arrow[l] \\
D_{\text{train}},\ D_{\text{val}},\ D_{\text{test}} \arrow[r, "\subset"] & \mathcal{X} & x_{t_1},\ x_{t_2},\ x_{t_3},\ \ldots \arrow[l, "\ni"']
\end{tikzcd}$$

To ensure that these assumptions hold, [CoDANN20, Chapter 6] discussed methods such as:

- Enumerating explicit operating parameters [CoDANN20, Section 6.2.7] (see also Section 5.1.2), akin to "classical" requirements design, connected to the system requirements/ConOps, and verifying that the parameters for this data are distributed according to the right distribution.
- Creating a "distribution discriminator" fulfilling pre-set requirements [CoDANN20, Chapter 6.2.8], which might be less transparent/interpretable (in the sense of decomposability, see Chapter 4) than the operating parameters, but more powerful and less subject to human bias.

This section aims at giving additional details about both approaches.

Section 4.8.1 has also discussed how what are usually called "explainability methods" can be seen as additional methods to verify these hypotheses. The techniques discussed there might also help ensure that the operational space has been correctly identified (as part of the *Data assurance* part of the W-shape), which is often an implicit assumption for OOD detection.

Additionally, Chapter 6 will give specific examples related to the use case from Chapter 2.

5.1.1 General remarks on analyzing distributions of images

Goodness-of-fit or anomaly/outlier/out-of-distribution detection are classical and well-studied topics in statistics/machine learning [Pim+14; Hub+12].

Naturally, it is not possible to provide definitive *yes/no* answers, and these techniques are statistical tests that provide evidence for or against hypotheses.

Available methods can in particular be characterized by:

- Whether they are *parametric* (assuming the distribution is known, or at least some properties thereof) or not. The distribution of some operating parameters relating directly to the ConOps might be explicit (e.g. altitude or object sizes), but it would be impossible to parameterize most image aspects.
- Whether they use the model that will receive the data or operate on the data only (*model-agnostic*).
- The *dimensionality* of the data on which the method can operate.
- The *properties of the underlying statistical test*: the probabilities of classifying an in-distribution sample as out-of-distribution and vice-versa.

A challenge arises however when $\mathcal{X}$ consists of images, like in the use case considered here (see Chapters 2 and 6).

Indeed, many methods do not generalize well in high dimensions, in particular due to the curse of dimensionality (see [ESL, Section 2.5]), with RGB images of size 512×512 spanning 786'432 dimensions. Moreover, another challenge is that the possible images (e.g. "all images that can be captured over Switzerland within the ConOps") occupy only a small subspace/manifold of this space that is difficult to characterize.

Dimensionality reduction A natural mitigation is to project the full space $\mathcal{X}$ onto a smaller one, with the induced distribution:

$$\mathcal{X} \xrightarrow{\pi} \mathcal{F} = \pi(\mathcal{X}). \tag{5.1}$$

If a dataset $D \subset \mathcal{X}$ is uniformly sampled from $\mathcal{X}$ (with respect to the corresponding probability measure), then $\pi(D) \subset \mathcal{F}$ will be uniformly distributed in $\mathcal{F}$ (with respect to the probability measure induced by π and $\mathcal{X}$).

$$D \text{ iid in } \mathcal{X} \implies \pi(D) \text{ iid in } \mathcal{F} \implies \text{statement from OOD test in } \mathcal{F} \tag{5.2}$$

Equivalently, if $\pi(D)$ does *not* have the right distribution in $\mathcal{F}$, then the same holds for D in $\mathcal{X}$. The reverse implication does *not* hold in general, in the same way that tests for out-of-distribution do not provide a definite yes/no answer (see above). However, this provides a way to test the desired assumption in the lower-dimensional space $\mathcal{F}$. The likelihood of the reverse implication to hold will depend on the "quality/expressivity" (ideally in a quantifiable way) of the embedding.

Explicit parameters The "explicit operating parameters" method mentioned above is the simplest example of that: with a careful enumeration of system parameters, one might expect to create an embedding into a product of simple spaces (intervals, categorical variables...) with a clear probability distribution (e.g. from the ConOps), where classical methods can be applied. With a sufficiently rich embedding, this might catch a non-negligible amount of out-of-distribution samples. A detailed example will be given in Section 6.1. As already explained, this might however suffer from human bias, and for the same reason is constrained to a reasonably small number of dimensions.

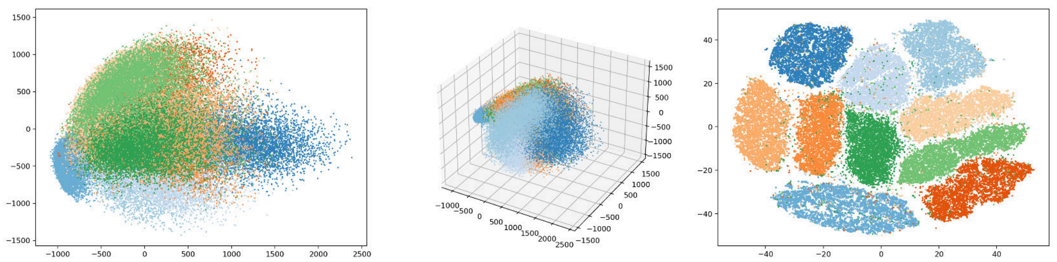


Figure 5.1: Dimensionality reduction of the MNIST-784 dataset [Lec+98], comprised of 8-bit grayscale 28 × 28 images, from left to right: 2-dimensional PCA, 3-dimensional PCA, and 2-dimensional t-SNE [MH08; Maa09]. Each color represent one of the digits from 0 to 9. Note that the parameters of the t-SNE embedding are learnt through gradient descent, while PCA follows from matrix decompositions.

More advanced techniques More generally, there is a large body of work on dimensionality reduction techniques (see [ESL, 13.4.2, 14] for an introduction), whose goal is to find embeddings as in Equation (5.1) that preserve the main properties (depending on the context) of the data in arbitrary lower dimensions. For example:

- A popular method is Principal Component Analysis (PCA) [ESL, 14.5], which can be seen as fitting to the data an ellipsoid of the requested (lower) dimension. See Figure 5.1 for an example. An obvious issue is the linearity and globality of the embedding.
- Autoencoders (see [GBC16, Chapter 14] for an overview) are unsupervised neural networks aimed exactly at learning low-dimensional representation. Their design can be very intuitive, with an encoder network, reducing the dimension until a bottleneck layer providing the representation, followed by a decoder network that tries to recover the original image from the representation (see Figure 5.2).

A common issue for these, due to the complexity of the embedding, is that it might be difficult to identify both

- The image $\mathcal{F} = \pi(\mathcal{X})$ of the embedding inside the lower-dimensional space $\mathbb{R}^p$ (which is well-defined when using explicit parameters);
- The induced probability distribution on $\pi(\mathcal{X})$.

In particular, the embedding might lose transparency/interpretability in the sense of Chapter 4.

However, unlike hand-crafted approaches, it is generally possible to quantify how well the embedding expresses the data (e.g. from a test dataset; see Figures 5.1 and 5.2), therefore the likelihood that the converse of the first implication in Equation (5.2) holds. For example, one might look at the reprojection error of an autoencoder on (a subset of) the test dataset.

Ensembling was also discussed in [CoDANN20, 6.3.3], drawing a parallel with multiple version dissimilarity: it is likely that different models will behave differently on out-of-distribution data.

Ultimately, the level of confidence at which one needs to verify the conditions recalled at the beginning of the chapter will depend on the safety argument of the whole system, in addition to whether it is applied during development, for operations, or other phases of the system lifecycle.

See also Section 5.4 on the definition and calibration of "uncertainty".

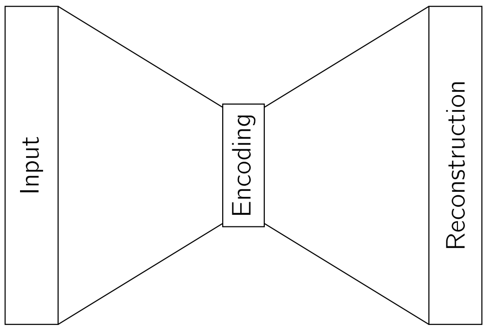


Figure 5.2: Generic architecture for an autoencoder. The left-hand side is the encoder, the right-hand side the decoder, while the middle part (often called "bottleneck") provides the low(er)-dimensional encoding. The network is trained by minimizing the difference between the input and its reconstruction.

5.1.2 Image conditions and quality metrics

Explicit operating parameters can be categorized along two directions, characterizing in further details:

- The external environment (distribution of targets, location...);
- The input received by the visual sensors.

The former is usually more classical, as it readily relates to the system requirements/ConOps, so this section focuses on the latter.

Image quality metrics Given a vision-based system (classical or machine learned), it is important to carefully characterize the algorithms' behavior and limitations with respect to properties of the input images. Changes in the operating environment (lighting, position of sun, visibility) would directly influence what the visual sensor receives, and in turn, changes in the image conditions would influence the performance and behavior of the algorithm. For example, this was one of the issues noted by Google and their partners in the real-world deployment of their diabetic retinopathy diagnostic system [Bee+20].
In the context of learning assurance, this simply means:

- Understanding which image parameters might vary in the images received by the sensors (i.e. in the operating space $\mathcal{X}$);
- Defining which ranges are expected and acceptable, and under which distribution;
- Understanding how they might impact the intended function;
- Adding these as explicit operating parameters, and ensure that the design datasets (training/validation/testing) cover them under the adequate distribution.

Operating parameters that focus on the image itself rather than what it represents will be named Image Quality Metrics (IQM). These will generally apply to most vision-based systems.

Normalization/standardization It is often possible to normalize/standardize the input to ensure that an image parameter falls within a range (or is even equal to a fixed value), but it should not be forgotten that the normalized out-of-range images will *still* have a different normalization: all that happened is a switch to another (perhaps currently untracked) parameter.

For example, a very dark image can be normalized to have a certain brightness, but the resulting image might contain less or different information than an image captured with the target brightness.

Examples The following provides various examples of IQMs which can be computed per image frame.

- Brightness: average value of channel (V in HSV spectrum of the image) intensity. The value should be within a range; too low/high indicates under-/overexposure.
- Contrast: standard deviation of pixel intensities of the gray-scaled image. The value should be within a range; too low may indicate under-/overexposure.
- Entropy: measured as the amount of randomness or infrequent pixel values found in the gray-scaled image. It is indicative of the presence of rich texture in the image.

These metrics can also be computed locally (dividing the image into several parts and computing metrics on each). This is for example appropriate for the visual traffic detection use case from Chapter 2, where detections happen on (a priori unknown) local parts of the input image.

5.2 Assurance level of the neural network component

Some approaches to provide runtime assurance on complex systems such as neural networks have put the focus on runtime monitoring rather than the system itself. In other words, a lower criticality is set on the complex system (that might even be seen as a "black box"), with the monitor supposed to bring runtime assurance.

This section discusses possible issues with this methodology, as well as the criticalities to assign to neural networks (possibly integrated with tracking/filtering) and external monitors.

5.2.1 Runtime monitoring for runtime assurance

The most prominent example of this approach is [ASTM F3269-17] *Standard Practice for Methods to Safely Bound Flight Behavior of Unmanned Aircraft Systems Containing Complex Functions* (not recognized by EASA as an Acceptable Mean of Compliance (AMC)), which proposes an architecture where:

- a "complex function" for vehicle control is monitored by
- a *Safety Monitor* which can operate
- a *Runtime Assurance (RTA)* switch to put
- an alternate *Recovery Control Function* (RCF) in control to restore an appropriate level of safety.

If the *Monitor*, *Switch* and alternative *Recovery Control Function* are "pedigreed", this might allow a "non-pedigreed" complex function to be sufficiently bounded.

This is similar to the *Doer/Checker* architecture described in [KKB19].

Examples The ASTM standard further describes in an appendix an application to the case where the non-pedigreed complex function is, in fact, a human pilot, and the monitor, switch and recovery system prevents flight into terrain.

A more relevant example of this architecture was recently described in [Cof+20], for an automated taxiing system following a runway centerline. The complex system at hand is a neural network identifying the centerline, and the monitor can stop the aircraft if:

- GPS and inertial navigation detect the aircraft to go too far outside of the safety margins;
- A classical computer vision algorithm identifying the centerline disagrees significantly with the neural network;
- A variational autoencoder (see Section 5.1.1) detects the input images to be out-of-distribution. From [Cof+20], "It is not a trusted component and is therefore not used to enforce system safety".

When such an architecture is possible it can certainly be a valid solution, but some criticism might be raised at the feasibility of the simple safe and certified recovery system, the feasibility of a reliable and certifiable monitoring system, and at the assumptions and guarantees of the resulting composite system.

The Recovery Control Function First, defining the recovery strategy to a safe state may in fact be the most challenging part of the entire operation.

For a driving car or a taxiing aircraft, braking until standstill is certainly an option that can be controlled by a relatively simple special purpose system, but standing still may not be the safest state when on a highway or a runway track.

Beyond the safety risk, such a faulty reaction may have some operational impact on third parties and a limited acceptance by other users (airlines using the same airport, Air Traffic Control. . .).

The Safety Monitor Recognizing unsafe conditions may itself be a task that can only be performed by a system as complex as the one to be monitored, for which establishing "pedigreed guarantees" is at least as hard. In the examples of runway landing guidance and traffic detection, recognizing runway incursions and traffic belong to the situational awareness sensing block of the architecture diagram, not to the control part. The ability of a human pilot to recognize certain unsafe conditions could very well be harder to replicate in a machine than the control function used for recovery.

The overall system and its safety guarantees Safety is not a property of a sensor, a monitor or a controller, but of the entire system in its operational conditions in an aircraft in an environment. The hazards and sources of uncertainty are present at all these levels, and to assume that the uncertainty can be attributed to a subsystem like a monitor or a controller is to presuppose a simplified universe of possible engineering problems.

If one takes the view that maintaining a safe state is a control problem [STPA], one can see that the ASTM proposes a classical control-monitor architecture:

> at each instant in time,
> when some safety-metric drops below an acceptable threshold,
> activate a safety-restoring-function.

This larger controller-plant system, that includes the original controller as a subcomponent, has to be carefully analyzed for its safety guarantees: complex and unexpected interactions between the components may very well make the resulting system more complex, harder to prove safe, and in fact less safe. For example, the switchover function may introduce hard-to-analyze or unbounded and unexpected behavior[1].

[1] A failing RTA architecture of this nature is what the nuclear disasters at Three Mile Island and Chernobyl had in common [Per84], triggered by maintenance and safety testing procedures.

Conclusion Installing a safety monitoring system and fallback controls can not be a general alternative to providing safety guarantees on a system or subsystem that has a sufficiently complex task. In short, if a monitoring system able to detect faults becomes sufficiently complex, a more reliable system can likely be built by putting the complexity directly into the monitored system. Moreover, even if the safety monitor was perfect, this architecture would provide no guarantees that the complex system performs its function, only that its failures (as frequent as they are) can be detected.

5.2.2 Proposed architecture and assurance level allocation

Limiting the scope of complex components to the minimum necessary Limiting complex and hard to guarantee functions to the smallest possible block in the architecture diagram is good practice.

This is why the examples presented in [CoDANN20] and Chapters 2 and 6 limit the machine learnt component to single images, deliberately leaving out time dependence, as well as other dependencies that can be treated with traditional systems. This is opposed to so called "end-to-end" architectures where a neural network for example directly controls the elevators, rudder and ailerons given a camera input.

Section 5.3 below will extend the discussions from [CoDANN20, Chapter 9] around the integration of classical tracking/filtering algorithms to take into account these dependencies, preventing in particular error rates from accumulating over time.

Learning assurance Learning assurance, as described in [CoDANN20], provides bounds on the behavior of a machine learning component, so the objective of bounding behavior is already met within that framework.

The out-of-distribution detection presented in [CoDANN20] and Section 5.1 as an essential component is in fact a runtime safety monitor, subject to similar statistical analysis as the main function. Note that this goes beyond the example above from [Cof+20] where the distribution check is used as a minor part of the monitor.

This link between learning assurance and out-of-distribution detection has also been discussed in Chapter 4 as part of explainability: the link between the operational and training data provides the explanation for the system performance.

Assurance level allocation In particular, the joint neural network, tracking/filtering and OOD monitoring should be assigned a high (relative) assurance level that cannot be reduced by combining it with a "bounded behaviour" system as described in [ASTM F3269-17].

An example will be given in the setting of the use case in Section 6.3. Future work may investigate ways of deriving credit from an RTA.

5.3 Integration within classical filtering/tracking

A machine learning model $\hat{f} : \mathcal{X} \to Y$ can be seen as a sensor that approximates the true function $f : \mathcal{X} \to Y$.

All sensors, classical ones or machine learning models seen as such, are imperfect and noisy, and dealing with this is a classical topic in engineering (see e.g. [ISO98-3; TBF05]).

Typically, sensor inputs can be:

- *Filtered*, to take into account the state of the system, the time dependencies, and uncertainties. This can be especially useful when the neural network has no knowledge of state or previous inputs/outputs, as in the example system studied in this report (Chapters 2 and 6).
- *Fused* with other sensors having different types of errors (see e.g. [DO-365B, Appendix F] for a system fusing radar, ADS-B and AST tracks).

These techniques usually require making and validating assumptions on the sensors, such as means, covariances, independence of errors, etc. This section discusses these for machine learning models.

5.3.1 Learning assurance and mean errors

The learning assurance guarantees (see [CoDANN20, Chapter 5]) provide a statement of the form[2]

$$\mathbb{E}_{x\sim\mathcal{X}}\left[m\left(f(x),\hat{f}(x)\right)\right] < E_{\text{in}}(\hat{f},D,m)+\varepsilon \qquad (m\in\mathcal{M}), \tag{5.3}$$

where $\mathcal{M}$ is the set of metrics of interest. The right-hand side is determined during development as the performance observed during development plus a summand to account for generalizing on unseen data (with $\varepsilon \to 0$ as $|D| \to \infty$).

This asserts that the average errors (as measured with the metrics from $\mathcal{M}$) of the "neural network sensor" are minimal on average. For metrics m that can be written $m(x,y)=\tilde{m}(x-y)$ (e.g. squared residual), then one can write

$$f(x)=\hat{f(x)}+e(x),$$

and Equation (5.3) states that $\mathbb{E}_{x\sim\mathcal{X}}\left[\tilde{m}(e)\right]$ is bounded by a known small quantity.

Systematic errors This is a guarantee that the model generalizes on unseen data, but one might need the stronger assumption that this mean is actually zero, i.e. that the model does not make systematic errors, or to consider higher moments. This is a crucial hypothesis for the analysis of most filtering or fusing techniques.

This might be difficult to ensure from (5.3), given that the right-hand side will always be strictly positive. However, the mean being nonzero would imply that the model makes systematic errors, which would be managed by:

- Correcting the training/training data to prevent these systematic errors. This corresponds to the *epistemic uncertainty* discussed in [CoDANN20, Section 6.6.4] (categorized by Der Kiureghian and Ditlevsen [DD09]);
- If this is not possible, identifying possible subspaces of the input space where the systematic errors are made, and mitigating these differently. This corresponds to the *aleatory uncertainty* discussed in [CoDANN20, Section 6.6.4].

The assumption that the mean is zero with these conditions can then be statistically tested.

[2]The conventions from [CoDANN20] are followed. In particular, metrics are of the type "lower is better".

5.3.2 Covariances and independence

Considering model outputs over time

$$\hat{f}(x_0), \hat{f}(x_1), \dots, \hat{f}(x_t), \dots,$$

viewing $x_0, \dots, x_t$ as random variables, an additional assumption required by filtering techniques is that the errors

$$e(x_0), \dots, e(x_t)$$

are:

- Of known covariances $\mathbb{E}\left[e(x_t)e(x_t)^t\right]$ (assuming zero mean);
- Uncorrelated;
- Uncorrelated with the process noise and the initial state.

There is in general no hard requirement for independence or normality: the Kalman filter is the optimal filter under the assumption of normality, but is still the best *linear* estimator (which might be enough) without this hypothesis, see [Sim06, Chapter 5].

Covariances and independence / types of errors The (un-)correlation of errors in time series was briefly mentioned in [CoDANN20, Section 9.5]. Assuming that the model has an elevated error $e(x_t)$ on frame x_t, one might argue that the likelihood to also have an elevated error on the next frame x_{t+1} is either:

1. Not different from the average case, given that x_t is close to but different from x_{t+1}. For example, if x_t is an isolated singularity/discontinuity of the model $\hat{f}$ that produces an erroneous output, changing a single pixel might be enough to get an adequate result. In this case, errors between frames should be independent and/or uncorrelated.
2. Higher than the average case, if the model is being operated in a region where less training data was available or where the learning task is intrinsically harder (e.g. different behavior of the target function f).

These cases should be related respectively to the aleatory and epistemic uncertainties mentioned above.

In particular, epistemic uncertainties should be reduced as much as possible: [DD09] also discusses the issues they pose for systems operating over time such as the setting considered here, as they violate the assumptions above, yielding sequences of heightened errors.

This also means that methods to verify the absence of significant errors of the second case have to be designed and applied, for example by studying means as in Equation (5.3) over subsets of the whole space. In other words, given that learning assurance only provides results on average over the whole space, one wants to ensure that means do not hide weakened performance on some regions of the operating space. This is often called "bias", which should not be confused with the "bias" from the "bias-variance decomposition", see e.g. [Meh+19].

5.4 Uncertainty estimation and validation

Section 5.3 viewed a machine learning model, approximating a true function that is too complex to implement explicitly, as a sensor. This analogy is appropriate as guarantees on the performance of a machine learning model will be probabilistic in nature, as with any sensor. In other words, "measurements" are guaranteed to be correct on average, but individual ones are subject to inevitable statistical noise.

The [ISO98-3] standard defines the *uncertainty of a measurement* as

- *"a parameter, associated with the result of a measurement, that characterizes the dispersion of the values that could reasonably be attributed to the measurand."*
- *"a measure of the possible error in the estimated value of the measurand as provided by the result of a measurement."*
- *"an estimate characterizing the range of values within which the true value of a measurand lies."*

The two types of uncertainties, aleatory and epistemic, were recalled in Section 5.3. A discussion on detecting and measuring epistemic uncertainty was started in Section 5.1.

The inputs to the model will usually themselves come from "classical" sensors (camera, IMU, RADAR, etc.), bringing additional possibilities of errors/uncertainties in the system. Figure 5.4 presents two cases where a visual sensor captures aircraft present in the field of view, but the resulting images might not contain enough information for a detection to achieve the same precision as in easier cases.

$$\begin{tikzcd} \text{World} \arrow[r] & \text{Sensor} \arrow[r, "x \in \mathcal{X}"] & \text{Model} \arrow[r] & \hat{f}(x) \\ & \text{Uncertainty} \arrow[u, dashed] & \text{Uncertainty} \arrow[u, dashed] & \end{tikzcd}$$

Figure 5.3: Uncertainties in sensor and model.

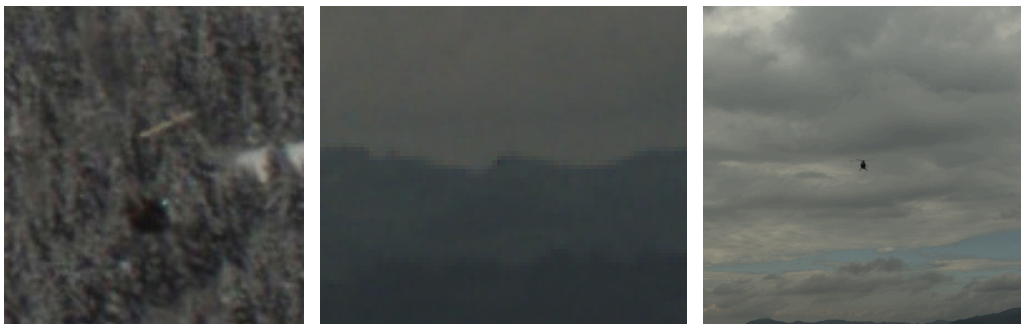


Figure 5.4: Limitations of a visual sensor for object detection: two objects that are hard to distinguish from the background and/or very distant. A helicopter is present in the center of the middle image, but is only represented by 1 pixel on a 12-megapixel image. The image on the right presents an easy case (black helicopter against light background).

This section discusses the meaning of measuring/predicting uncertainty as well as methods to do so and requirements to set. It would be important to discuss both the frequentist and Bayesian points of view, but the latter is left for future work. The reader is referred to the recent survey [Abd+21] (focusing on deep neural networks).

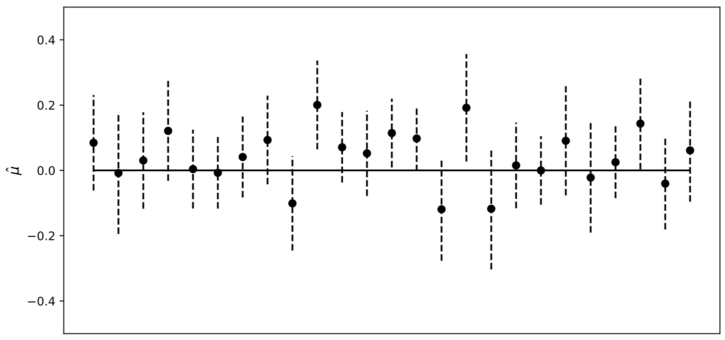


Figure 5.5: Repeated measurement $\hat{\mu}$ of a parameter $\mu = 0$ with their respective confidence intervals. A confidence level of β would mean that the ratio of intervals intersecting the horizontal axis $\mu = 0$ would converge to β.

5.4.1 Expressing uncertainties

Uncertainties are usually reported as confidence intervals of the form

$$\hat{\mu} \pm a \qquad \text{or} \qquad [\hat{\mu} - a, \hat{\mu} + b] \qquad (a, b > 0),$$

with $\hat{\mu}$ the estimated value of a target parameter μ. These are usually not absolute statements, but reported for a confidence level β (e.g. 95% or 99%).

Confidence levels and probability spaces This confidence β is commonly misinterpreted [WL16] as the probability that the true value μ lies in the confidence interval. Such a statement does not make sense as μ is (in the frequentist interpretation) fixed: the probability that it belongs to an interval is either zero or one.

Instead of considering the parameter μ random, the probability space of all measurements is considered: each measurement M has a confidence interval $C(M)$, and

$$\beta = P\big(\mu \in C(M)\big).$$

By the law of large numbers, when performing a large number of measurements, the ratio of confidence intervals that contain μ will converge to β. See Figure 5.5 for an illustration.

More generally, when discussing uncertainties, it is important to always make the underlying probability space, or priors in the Bayesian setting, clear.

Relationship with variance Section 5.3.2 discussed the importance of understanding (co)variance of model errors. This is related to uncertainty and confidence intervals: concentration inequalities such as Chebyshev's can provide confidence intervals for sample means as a function of the variance. In the case of a normal distribution with variance σ^2, it is well-known that more than 95% of observations will fall within 2σ of the mean asymptotically.

5.4.2 Measuring/predicting uncertainty

The [ISO98-3] standard distinguishes two ways of measuring uncertainty in measurements:

- Type A: statistical methods on repeated observations;
- Type B: other methods.

For both, the probability space in which the uncertainties will apply (see Section 5.4.1) needs to be determined. Once the probabilistic setting is clear, so will be possible means to verify the chosen technique.

Type A methods Under the analogy of machine learning models as sensors, type A methods do not make sense on a single input (collected e.g. from a sensor), as the model is assumed to be deterministic[3].

However, it is possible to obtain a meaningful measure of uncertainty around a single input within the Type A setting of repeated measurements, for example:

- In the case of highly correlated time series inputs (such as images in the use cases from Chapter 2 and [CoDANN20]), one point of view might be to see successive images as a repeated measurement. Often, the outputs between successive frames even admit an explicit transformation (in the example, depending on the relative movement of the two aircraft). Note that this will also include uncertainty from the sensor (see Figure 5.3).
- Injecting noise into the input, in line with techniques from explainability (Section 4.6.4), adversarial attacks, or generalization bounds based on weight perturbations (see [CoDANN20, 5.3.6]).
- Ensembling, which combines several[4] models into one, and therefore performs several "measurements" for each input. This can help quantify both types of uncertainty as different models might behave differently on out-of-distribution data, and have different errors on in-distribution data. See [CoDANN20, 6.3.3] on multiple version dissimilarity.

Type B methods Some type A methods can be seen as type B ones when applied to test data; other examples of type B methods are:

- Uncertainty as an output of the model itself.
- Analyzing the overall model uncertainty from errors on test data, possibly making and verifying assumptions on their distribution at the same time. While this uses repeated measurements, these are of different inputs.
- In simulation, type A methods are easier to apply, as small variations of an image that leave the output fixed can be easily generated. If the gap between reality and simulation ([CoDANN20, Chapter 7]) is well understood, this information can be used during operations.

The first method is discussed further in the next section.

5.4.3 Uncertainty as a model output

This section discusses the natural method of trying to predict uncertainties as part of the model output, meaning this task is learnt as part of the training. This a priori only makes sense assuming in-distribution data, i.e. assuming that epistemic uncertainty is controlled[5].

[3] On the other hand, for humans tasked to annotate training data, or simply for measured "ground truth", it is likely that each measurement will be slightly different. Recall from [CoDANN20, Section 5.2.2] that the "true function" is only an abstraction, and that measuring/collecting values is tainted by random noise.

[4] Possibly variants of the same base model, see variational dropout or stochastic ensembles.

[5] An interesting discussion is present in [Sae00, Section 4] on possible issues in past research.

#	Class probability			Predicted class
	Background	Rotorcraft	Fixed-wing	
1	0.3	0.3	0.4	Fixed-wing
2	0.1	0.7	0.2	Rotorcraft
3	0.8	0.1	0.1	Background

Table 5.1: Categorical predictions from probability distributions over three classes.

Output types It is useful to distinguish to two general types of outputs of machine learning models:

- Categorical outputs, where the output is among a fixed finite set, such as the airborne object category in Section 2.1.3;
- Numerical outputs, where the output is a continuous variable (or considered as such), such as the bounding box coordinates in Section 2.1.3.

Categorical outputs Categorical outputs are usually implemented with a built-in probabilistic viewpoint: for each input, a probability distribution over the input classes is computed. This is often achieved with the softmax function

$$\boldsymbol{v} \in \mathbb{R}^n \mapsto \boldsymbol{p} = \sigma(\boldsymbol{v}) \in [0,1]^n, \qquad p_i = \frac{e^{v_i}}{\sum_{i=1}^n e^{v_i}}, \qquad \sum_{i=1}^n p_i = 1$$

that transforms a real-valued vector into a probability distribution. The output class is selected as the one having maximum probability.

It is tempting to see the distribution $\boldsymbol{p}$ as a straightforward measure of uncertainty: if the highest probability is close to 1, the prediction is very confident; if all probabilities are close to $1/n$, the prediction is not better than random guessing; two close scores denote uncertainty between two classes. This is illustrated in Table 5.1.

For this to be valid, following Section 5.4.1, the probabilities produced by the neural network should satisfy

$$P\Big(\text{Correct prediction} \mid \text{Predicted confidence} = p\Big) = p,$$

where "correct prediction" means "predicted confidence of the true class is > 0.5". This property is called *calibration*. By the law of large numbers, p is, in that case, equal to the asymptotic proportion of correctly classified samples with predicted confidence equal to p. For example, on a large amount of samples, roughly 30% of those where the neural network outputs a 70% confidence should be misclassified.

This is true for models such as logistic regression on linearly separable data, but miscalibration is a known issue for more complex models such as neural networks [NC15; Guo+17]. See Figure 5.6 for an illustration. Interestingly, [NC15] observed in 2005 that neural networks tended to be well-calibrated, but [Guo+17] noted that this was not true anymore with recent training and regularization techniques as well as increased model capacity.

Therefore, the calibration of confidences should always be carefully analyzed, in particular when it will be used during filtering (Section 5.3).

There exist multiple techniques (see e.g. [NC15; Guo+17]) to recalibrate models post-training while conserving prediction quality. An effective but simple method, already mentioned in [CoDANN20], is *Platt scaling*, which simply rescales probabilities with a linear model on the validation set.

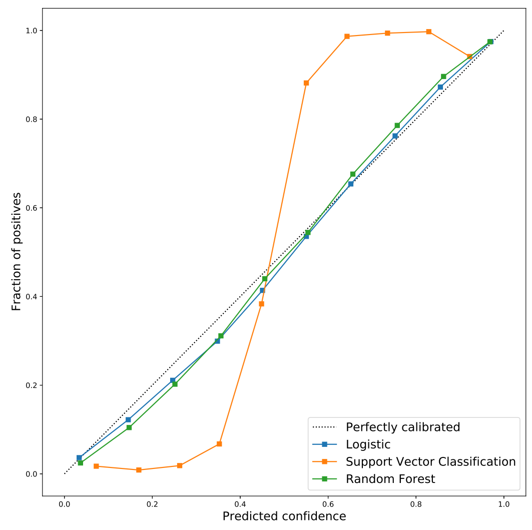


Figure 5.6: Calibration curve for binary classification models (logistic, support vector classifier, random forest) on a synthetic dataset (10^5 samples, 20 dimensions, 10^4 samples used for training). The x-axis contains predicted confidences, while the y-axis displays the proportion of samples correctly classified. A perfectly calibrated classifier should fit the diagonal. Adapted from the documentation of [Ped+11].

Numerical outputs Unlike categorical outputs, numerical ones do not usually have a straightforward interpretation of confidence.

Assuming "perfect training" (global minimum reached and adequate model capacity, see [Sae00]), using the natural mean squared error as a loss function will result in predicting the conditional mean of the output given the input. It is therefore natural to attempt to predict other statistics.

A well-known example is the method of *quantile regression*, where the loss is modified to yield a model that predicts conditional quantiles: see [Sae00] for a proof in "perfect training" conditions.

A similar recent example can be found in Pearce et al. [Pea+18], which, building on previous work, provides a method to produce prediction intervals for deep neural networks through a specific loss function (and ensembling to take epistemic uncertainty into account).

The potential issues noted above for categorical outputs carry to numerical ones. It is important to carefully verify techniques, given that perfect training conditions required for theoretical results usually do not hold. Experimental verification frameworks are present in most relevant publications (see e.g. [Pea+18, Section 6]).

5.4.4 Conclusion

This section showed that care had to be taken in the definition of uncertainties (probability spaces, epistemic or aleatory), and surveyed common issues and techniques. One aspect that should be kept in mind is that a probabilistic interpretation (such as "probabilities" in classification, even when minimizing a cross-entropy) does not necessarily satisfy probabilistic properties (e.g. calibration).

The reader is referred to the literature and surveys such as [Abd+21] for more details on specific techniques, in particular on the Bayesian point of view.

Chapter 6

Detailed analysis of the Visual Traffic Detection use case

This chapter returns to the use case described in Chapter 2 and applies the considerations from Chapters 3 to 5. While [CoDANN20, Chapter 10] examined the overall W-shaped process, this section focuses on runtime monitoring (Section 6.1), neural network/tracking integration (Sections 6.2 and 6.4), full system integration (Sections 6.3 and 6.4) and explainability (Section 6.5).

6.1 Runtime monitoring and out-of-distribution detection

Sections 4.8.1, 5.1 and 5.2 emphasized the need to ensure that the inputs to the system match the probability space specified by requirements. Not doing so would render impossible to guarantee performance on unseen data for a system with a machine-learnt component.

For the visual detection use case, several distribution monitors can be envisioned.

Explicit operating parameters From Table 2.2, examples are altitude, speed, time of day, location, etc.

As they are usually low-dimensional, classical goodness-of-fit testing methods can be used to ensure that distributions match the expected ones.

Image quality metrics Described in Section 5.1.2, these are based on the input received by the visual sensors, and are particular examples of explicit operating parameters. They might be directly included in the system requirements or not.

The ability to visually detect airborne objects from images will depend on

- How much the objects stand out from the background (e.g. black helicopter on black background is less distinguishable than a black helicopter against a bright background);
- How much information is present on the object itself (e.g. a camera blinded by the sun will "hide" information about the object).

Figure 6.1 presents an easy and a challenging example (with the system studied detecting the aircraft in both cases); see also Figure 5.4.

Some obvious metrics are brightness, contrast, etc., computed locally (on parts of the images) or globally (on the whole image). The Grey Level Co-occurence Matrix (GLCM) [HSD73] can provide additional metrics such as uniformity, homogeneity and correlation.

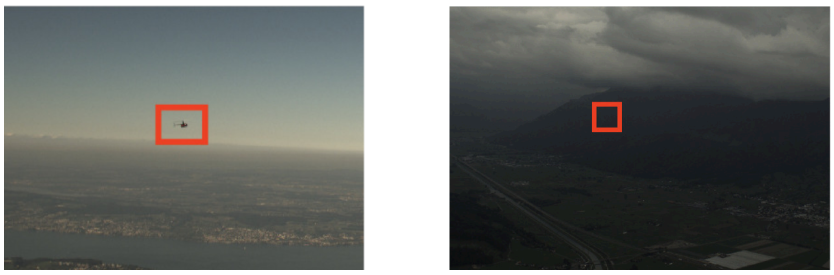

(a) High contrast and brightness. (b) Low contrast and brightness.

Figure 6.1: Easy and hard example for visual traffic detection, characterised by brightness and contrast.

Computing metrics locally is particularly relevant for an object detection system, as this might provide insight about regions of the image where the system might not be able to detect a target, even if the overall image metrics are within adequate ranges.

Complex distribution discriminator As discussed in Section 5.1, explicit operating parameters and image quality metrics might be too shallow to characterize the expected input space distribution and detect deviations from it during operations.

Section 5.1.1 presented several techniques based on dimensionality reduction. The framework presented in [CoDANN20, Section 6.2.8] provides requirements and means of verification that are agnostic to the exact techniques used.

A careful analysis should be carried out to quantify the residual probability of out-of-distribution data (where no generalization guarantees can be given) not being detected by the out-of-distribution detection system.

Section 4.8.1 also showed how explainability techniques provide means to ensure that the operating space is correctly identified.

6.2 Tracking/neural network integration

This section applies the considerations from Section 5.3 to the post-processing described in Section 2.1.4. The discussions therein will also be an input to the functional hazard assessment in Section 6.4. In particular, failure conditions and fault trees must carefully take into account the interactions between the neural network and the post-processing/tracking/filtering component. This might include the time dimension given that post-processing is important to prevent a small error rate from exponentially increasing over time.

[CoDANN20, Chapter 9] examined how learning assurance and filtering can be combined to obtain performance guarantees on a post-processed neural network output. The focus here will therefore be put on the analysis of the tracking component, assuming that probabilistic failure rates on the neural network component are known.

This section has been removed in the public version due to confidentiality reasons. Interested parties are welcome to contact ipc-feedback@daedalean.ai.

6.3 Full system integration

Within this section the integration of the neural network within the whole system is explored further in order to highlight any necessary mechanisms to support the desired safety case.

This section has been removed in the public version due to confidentiality reasons. Interested parties are welcome to contact ipc-feedback@daedalean.ai.

6.4 Functional Hazard Assessment (FHA)

In the ConOps examples, for both identified use cases, the following functional decomposition has been identified:

- F1: To detect aircraft in the surrounding airspace.
- F2: To provide track information on detected aircraft in the surrounding airspace.
- F3: To monitor the system.
- F4: To interface with the aircraft systems.
- F5: To predict potential collisions with other aircraft and respond to traffic alerts to avoid collisions.

6.4.1 Functional allocation

This section has been removed in the public version due to confidentiality reasons. Interested parties are welcome to contact ipc-feedback@daedalean.ai.

6.4.2 Functional analysis

Failure conditions list

See the following pages.

ID	Function description	Failure condition title	Phase of operation	Effect of the Failure on the Aircraft, Crew and Occupants	FC classi-fication	Rationale for classification	Notes (assumptions)
FC1-1	F1 + F2 (traffic detection and tracking)	**Total loss** of function, **indicated** to the crew (advisory system): system is unavailable and crew is warned by the system	In Flight	Alert and associated procedure are used to mitigate FC	MIN	Classified Minor due to the impact on safety margin/crew workload. Spurious alerting falls under CS29/25.1322, CS23/VTOL requirements related to alert (see AMC/VTOL MOC)	Assumption #1: Flight crew, alerted to system in operation, will be ready to visually acquire potential hazardous traffic. Assumption #2: Alert and AFM procedure will be developed to cover this case.
FC1-2-1	F1, F2 + F3 (system monitor)	**Total loss** of function, **not indicated** to the flight crew (advisory system): system is unavailable and crew is not warned by the system, **without threat**	In Flight	Flight crew may notice system malfunction in due time and recover	MAJ	Classified Major due to the impact on safety margin/crew workload. Per the ConOps, ATC may not be available to detect potential collision.	Assumption #3: Flight crew do not reduce their alertness to potential hazardous traffic, despite the presence of this system. Note this FC also cover incorrect inhibition above 300 ft (e.g. system always inhibited). No dedicated entry has been created for an incorrect inhibition as the effect are similar to this particular FC. Incorrect inhibition is thus to be considered included in this FC.
FC1-2-2	F1, F2 + F3	**Total loss** of function, **not indicated** to the flight crew (advisory system): system is unavailable and crew is not warned by the system, **with threat**	In Flight	Flight crew may notice system malfunction in due time and recover	HAZ	Classified Hazardous due to potential that threat is not detected by crew or other system in due time. Per the ConOps, ATC may not be available to detect potential collision. Classification assumes no credit from ATC	As per FC1-2-1
FC1-3	F1 + F2	**Erroneous guidance** but **detected** by the crew: system is not providing proper guidance and crew is warned by the system	In Flight	Flight crew may identify that system has failed in due time and recover as per AFM procedure	MIN	As per FC1-1	As per FC1-1
FC1-4-1	F1, F2 + F3	**Erroneous guidance**, but **not detected** by the crew (Misleading : system is not providing proper guidance and crew is not warned by the system, **without threat**	In Flight	Flight crew may notice system malfunction in due time and recover	MAJ	As per FC1-2-1	As per FC1-2-1

ID	Function description	Failure condition title	Phase of operation	Effect of the Failure on the Aircraft, Crew and Occupants	FC classi-fication	Rationale for classification	Notes (assumptions)
FC1-4-2	F1, F2 + F3	**Erroneous guidance**, but **not detected** by the crew (Misleading : system is not providing proper guidance and crew is not warned by the system, **with threat**	In Flight	Flight crew may notice system malfunction in due time and recover	HAZ	As per FC1-2-2	As per FC1-2-1
FC1-5	F4 (interface with aircraft)	Loss of inhibition below 300 ft : The system keeps running below 300 ft	Take Off, Initial Climb, Approach, Landing	Spurious alerts due to ground traffic may occur during take-off or landing procedures.	MAJ	Classified Major due to increased crew workload, due to spurious alerting, during critical phases of flight.	
FC1-6-1	F4	Loss of the video link to the display	In Flight	Flight crew is unable to acquire/confirm the target through the avionics display	MIN	Classified Minor due to slight increase in crew workload.	Assumption #4: Video image is discretionary information and does not effect ability of flight crew to make avoidance action decisions.
FC1-6-2	F1, F2 + F3	Misleading video on the display	In Flight	Flight crew is unable to acquire/confirm the target through the avionics display	MIN	Classified Minor due to slight increase in crew workload.	As per FC1-6-1 The video does not correspond to the detected target: e.g. there is no intruder displayed because the system is not sending the right part of the image.
FC1-7-1	F4	ACAS Data is not displayed (Loss of data)	In Flight	Flight crew is not alerted to uncooperative visible traffic.	MAJ	Classified Major due to the impact on safety margin/crew workload. Per the ConOps, ATC may not be available to detect potential collision.	As per FC1-2-1
FC1-7-2	F1, F2 + F3	ACAS Data is incorrectly displayed	In Flight	Flight crew is given misleading alerts to uncooperative visible traffic.	MAJ	Classified Major due to the impact on safety margin/crew workload. Per the ConOps, ATC may not be available to detect potential collision.	As per FC1-2-1
FC2-1	F1 + F2	**Loss** of traffic alerts on an autonomous system, **indicated** to an upper level system **without threat**	In Flight	For fully autonomous system, having alerts / Flight deck effect in such a case are not necessarily relevant	MIN	Classified Minor due to slight decrease in safety margin (host aircraft having to take action as a result of system failure).	Assumption #5: Once system failure reported to host aircraft appropriate action is taken.

ID	Function description	Failure condition title	Phase of operation	Effect of the Failure on the Aircraft, Crew and Occupants	FC classi-fication	Rationale for classification	Notes (assumptions)
FC2-2	F1 + F2	**Loss** of traffic alerts on an autonomous system, **indicated** to an upper level system **with threat**	In Flight	For fully autonomous system, having alerts / Flight deck effect in such a case are not necessarily relevant	HAZ	Classified Hazardous due to potential that threat is not detected by other systems in due time. Per the ConOps, ATC may not be available to detect potential collision. Classification assumes no credit from ATC.	As per FC2-1
FC2-3	F1, F2 + F3	**Loss** of traffic alerts on an autonomous system, **not indicated** to an upper level system **without threat**	In Flight	For fully autonomous system, having alerts / Flight deck effect in such a case are not necessarily relevant	HAZ	Classified Hazardous based on the reduction of safety margin considering that system is one external event (external traffic on collision course) away from a CAT event	
FC2-4	F1, F2 + F3	**Loss** of traffic alerts on an autonomous system, **not indicated** to an upper level system **with threat**	In Flight	For fully autonomous system, having alerts / Flight deck effect in such a case are not necessarily relevant	CAT		Assumption #6: No reliance on other A/C avoidance system as mitigation (non co-operative system submitted so similar operating condition assumed), or ATC to ensure separation.
FC2-5	F1, F2 + F3	Erroneous traffic detection but detected by an upper level system/the system itself on an autonomous system	In Flight	For fully autonomous system, having alerts / Flight deck effect in such a case are not necessarily relevant	HAZ	Classified Hazardous based on the reduction of safety margin considering that system is one external event (external traffic on collision course) away from a CAT event	
FC2-6	F1, F2 + F3	Misleading traffic detection traffic on an autonomous system	All	For fully autonomous system, having alerts / Flight deck effect in such a case are not necessarily relevant	CAT		This could result in inappropriate pitch down command close to ground

Table 6.1: Failure conditions. FC1 rows are related to use case 1 (pilot advisory), FC2 rows are related to use case 2 (full autonomy). NOTE: Supporting material and Verification methods for each Failure Condition are not discussed in the scope of the IPC.

- NSE = No Safety Effect, as defined in applicable guidance.
- MIN = Minor Failure condition as defined in applicable guidance.
- MAJ = Major Failure condition as defined in applicable guidance.
- HAZ = Hazardous Failure condition as defined in applicable guidance.
- CAT = Catastrophic Failure condition as defined in applicable guidance.
  For example, an applicable guidance for VTOL is AMC VTOL.2510.

6.4.3 Fault Tree Analysis

This section has been removed in the public version due to confidentiality reasons. Interested parties are welcome to contact ipc-feedback@daedalean.ai.

6.4.4 Development Assurance Level Assignment

This section has been removed in the public version due to confidentiality reasons. Interested parties are welcome to contact ipc-feedback@daedalean.ai.

6.5 Explainability

This section finally looks at explainability for the use case, in the context of Chapter 4.

Focus on explainability for detection The focus will be put on the traffic detection part, and not on the avoidance system where the former might be integrated in.

The avoidance algorithm is usually built on classical software for which [ED-12C/DO-178C] applies. It can be thought of as a set of rules that trigger maneuvers depending on the relative position and movement of potential obstacles. However, this does not mean that explainability does not need to be considered for this part, as modern systems might make use of optimization algorithms generating complex lookup tables, not dissimilar to how neural networks work (see [KHC12] for an overview; also [JKO19]). Moreover, this might move the level from "aid to decision" to "decision-making".

Components of the detection system Section 2.1 explained that the visual traffic detection system is composed of:

- A neural network producing a set of detections (essentially bounding boxes with confidences) on each frame;
- A tracking algorithm creating "tracks" from the single detections. Track points might be composed of detections from the neural network, or extrapolated from previous detections.

Categorization As in Section 4.2.1, for explanations one should distinguish the:

- Object: system or output;
- Recipient: regulator, system designer, pilot/operator, or investigator.

System-level explanations through generalization These will not be discussed further here as they were extensively discussed in [CoDANN20], and have been the object of Sections 6.2 and 6.4. The analysis should provide detailed probability distributions for the different cases where the model might fail to output a target (false negatives) or produce spurious detections (false positives). These distributions can be fed into the tracking algorithm and the overall safety analysis.

Requirements for explainability Following Section 4.8, the following sections discuss additional activities: output-level explanations and strengthening the data-learning assurance link (Section 4.8.1), starting with the first given that it can be used for the second too.

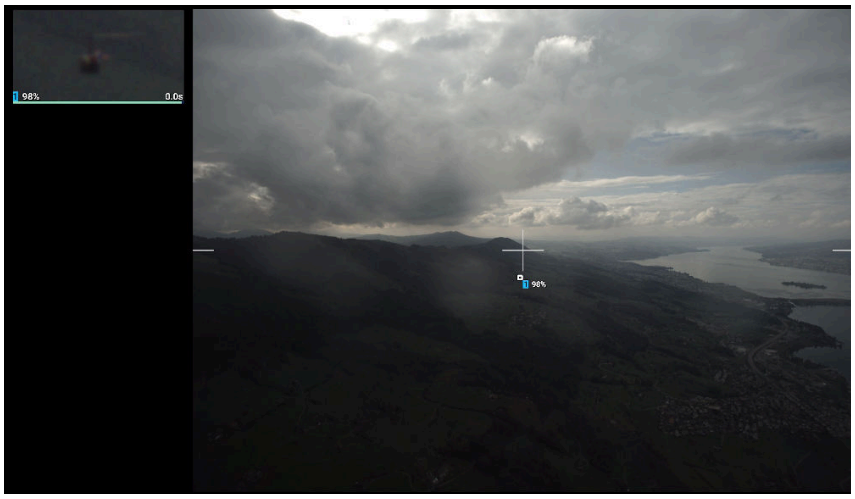


Figure 6.2: Screenshot from an internal demonstration user interface, providing a zoomed-in inlet of a detection, allowing confirmation.

6.5.1 Output-level/local explanations

For simplicity, one might view the detection component (neural network, before the tracking component) as having two states (locally on the input image, e.g. by splitting it into a grid):

1. Object detected, with a given class and confidence;
2. No object detected, with a given confidence.

It is important to realize that many methods focus on the first case (analyzing a detection), while in the application considered, not detecting an actual target might be a more critical failure than a false positive (e.g. might lead to a collision). One might even argue that understanding why the system does not detect an object is a harder problem than explaining a detection: while a zoomed-in picture of an aircraft is a clear proof of correctness (see Figure 6.2), it is harder to provide a meaningful and transparent justification for the absence of a detection.

To be able to put forward requirements, it is helpful to distinguish further between true/false positives/negatives (see Table 6.2).

	Object present	**No object**
Detection	True positive	False positive
No detection	False negative	True negative

Table 6.2: Reminder about true/false positives/negatives.

Providing local explanations for detections There are two subcases to consider, with the following main objectives:

- True positives: confirm that the detection was mostly triggered by the object itself, and not by its surroundings (see also the discussion in Section 4.8.1).
- False positives: understand which parts of the input image triggered a detection and why; distinguish between systematic errors (e.g. cars always mistaken for aircraft) and spurious ones.

Activations visualization (directly or in the input space, see Sections 4.6.2 and 4.6.3) and saliency maps (Section 4.6.4) would provide useful information towards both objectives. Figures 4.5 and 4.6 provided examples within the use case. See also [Yos+15] for a real-time visualization framework.

In the case of false positives, activation visualization also provides a way to generate inputs where the state of the model (e.g. its activations) is similar to the current one. This might allow detecting systematic problems or provide evidence against these. Alternatively, one might also proceed as in Section 4.5.3 to obtain inputs from the training/validation/testing datasets. This can allow to make the "systematic vs. spurious" distinction. For example, in the context of Figure 4.10, this could have helped to understand why traffic cones are consistently detected as helicopters.

Regarding saliency maps, the concerns from [Ade+18] should be kept in mind, and clear requirements should be imposed before choosing a specific technique. As in [Ade+18], one might ask in particular that the saliency maps:

1. Depend on the models: the saliency maps of untrained models should be qualitatively and quantitatively different (e.g. close to random noise) than the ones from the trained model.
2. Depend on the data: the saliency maps of untrained models should be qualitatively and quantitatively different from the ones from the trained model when data is randomized, to verify that the explanation actually rely on the data/label relationship.
3. Do not make use (implicitly or explicitly) of the ground truth.
4. Have adequate properties on the training/validation/verification and/or (part of) the training set (e.g. they are maximal around the objects and minimal else where, and possibly their values are proportional to the confidences).

Providing local explanations for non-detection The subcases with their objectives are:

- False negatives: understand why the object was not detected.
- True negatives. This is more delicate, since this is the double negative (no object/no detection) case. Two objectives could be to:
  - Show in which parts of the images the detection confidence was highest (albeit still below the detection threshold).
  - As above, show inputs where the state of the model (e.g. its activations) is similar to the current one, and confirm that these also do not contain detections.

As above, activations visualization (Sections 4.6.2 and 4.6.3) and saliency maps (Section 4.6.4) can be useful tools towards these objectives.

Displaying detection confidences over the image can be done using only the system outputs (Section 2.1.1), recalling that these are required to be calibrated (see Section 5.4).

Explanations for pilots/operators and HMI In this context, the goal is to give additional insights about the system outputs to help in decision-making, e.g. how much trust to put in the current output or how to cross-check it.

Possibilities were already mentioned in Sections 2.1.1 and 2.3.2 (system outputs provided along tracks and their visualization). For example:

- A zoomed-in extract of the image with a target is a clear proof of a correct detection (see Figure 6.2);

- The display of track points as extrapolated or detected with their confidences (e.g. using a color coding) provides more context.

Human factors aspects have been discussed as part of this project, but are out of scope for this report. See also [EAS21, Section 4.5].

6.5.2 Strengthening the data–learning assurance link (system level)

Section 4.8.1 already contained an outline of possible techniques as part of the *Independent data and learning verification* step of the W-shaped process.

Good candidates for the target of the maximally activating inputs (Section 4.5.3) and generative methods (Section 4.5.2) are the confidences:

- At a given location;
- For a given object class;
- That there is an object in the image;
- etc.

Figures 4.1 to 4.4 provided examples in the context of the use case.

Chapter 7

Conclusion

Building on the first EASA/Daedalean Innovation Partnership Contract (IPC) and the resulting report [CoDANN20], this second project matured the *Learning assurance* and *Trustworthiness analysis* EASA AI trustworthiness building-blocks [EAS20], in addition to introducing the *Explainability* block.

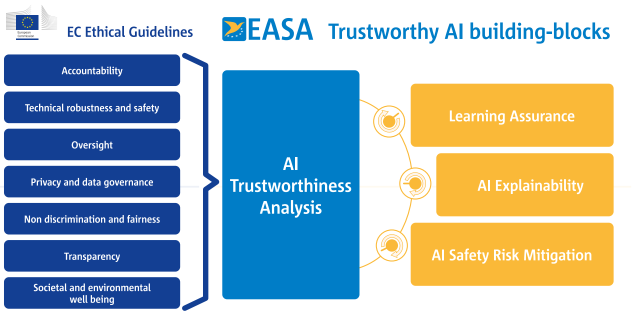

EC Ethical Guidelines
EASA Trustworthy AI building-blocks
Accountability
Technical robustness and safety
Oversight
Privacy and data governance
Non discrimination and fairness
Transparency
Societal and environmental well being
AI Trustworthiness Analysis
Learning Assurance
AI Explainability
AI Safety Risk Mitigation

Figure 7.1: Trustworthy AI building-blocks from [EAS20, Figure 5].

Learning/inference environments and model implementation While [CoDANN20] focused on more theoretical aspects of learning assurance, this report pushed the discussion to taking into account the software/hardware platforms implementing neural networks and related tools in the development and operational environments. Chapter 3 surveyed possible platforms, particularities of the *learning* and *inference* environments, as well as means to provide performance guarantees for the end system despite the major changes that might take place during the development cycle, and the complexity of usual training environments.

Explainability Explainability is a recurrent topic for artificial intelligence in safety-critical systems, and Chapter 4 provided a survey of definitions, techniques, and working groups. As there exist many methods, whose contributions towards providing additional data are sometimes disputed, it is important to identify what gaps might exist in learning assurance, and what requirements should be put on mitigations. In line with [EAS21], the report identified two main objectives: strengthening the learning assurance–data link, and human-machine interaction.

Safety assessment Regarding the topic of safety assessment, Chapter 5 developed topics from [CoDANN20]: out-of-distribution detection, integration with classical filtering/tracking to handle time dependencies, as well as uncertainties estimation/validation and assurance level assignments for neural network components.

Use case As in [CoDANN20], a practical application was analyzed, corresponding to Daedalean's visual traffic detection product. Chapter 2 presented this system within the scope of two operational concepts (pilot advisory and full autonomy), while Chapter 6 illustrated the concepts developed in the previous chapters on this use case, culminating in a safety assessment of the full detect-and-avoid system.

Next steps / future work

With [CoDANN20] and the present project, the full W-shaped development cycle for learning assurance from [CoDANN20] has been investigated.

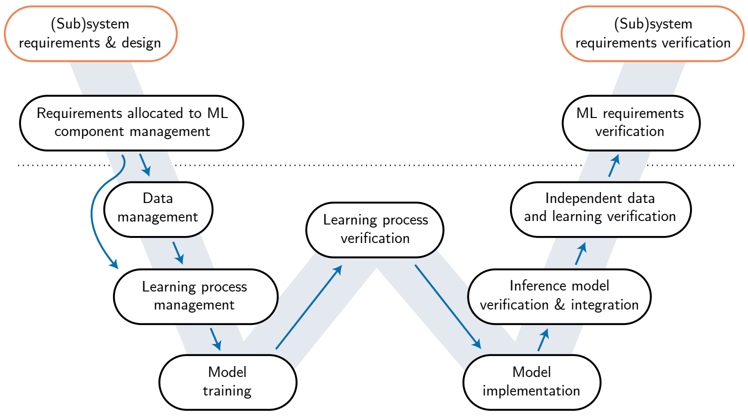

(Sub)system requirements & design
(Sub)system requirements verification
Requirements allocated to ML component management
ML requirements verification
Data management
Learning process verification
Independent data and learning verification
Learning process management
Inference model verification & integration
Model training
Model implementation

Figure 7.2: W-shaped development cycle for *Learning assurance* from [CoDANN20].

The two EASA/Daedalean IPC projects have been used as inputs by the EASA AI project team when preparing EASA's Level 1 Guidance [EAS21].

The visual landing application from [CoDANN20] is integrated as a use case in the first draft of the document, and a future update might include the visual traffic detection system studied in this second project.

Some of the topics that have not yet been addressed are:

- Post type certificate changes (e.g. model retraining);
- Proportionality;
- Variants of the non-adaptive supervised learning setting considered throughout. Although this setting covers a large number of uses cases, future applications might require adaptive learning, recurrent neural networks, unsupervised learning, etc.

Although it is highly desirable to investigate these aspects in the near future to streamline upcoming certification applications, the current state of the Level 1 Guidance [EAS21] and the flexibility built into the certification process enable the first applications to be submitted.

EASA's roadmap [EAS20] foresees to tackle these topics in the near future. The research community, industry and standardization working groups also have a crucial role to play in these areas. To address these challenges, EASA will continue to support and even expand its collaborations and partnerships with the AI community in the coming months and years.

References

[Abd+21]	Moloud Abdar et al. *A Review of Uncertainty Quantification in Deep Learning: Techniques, Applications and Challenges*. Preprint. 2021. URL: https://arxiv.org/abs/2103.13630.
[Ade+18]	Julius Adebayo et al. "Sanity Checks for Saliency Maps". In: NIPS'18. Montréal, Canada: Curran Associates Inc., 2018.
[AMC 20-115D]	EASA. *AMC 20-115D Airborne Software Development Assurance Using EUROCAE ED-12 and RTCA DO-178*. Amendment 14. Nov. 2018.
[AMC 20-152A]	EASA. *AMC 20-152A Development Assurance for Airborne Electronic*. Amendment 19. Oct. 2020.
[AMC 20-193]	EASA. *AMC 20-193 Use of multi-core processors (MCPs)*. Proposed amendment. Sept. 2020.
[Aro+18]	Sanjeev Arora, Rong Ge, Behnam Neyshabur, and Yi Zhang. "Stronger generalization bounds for deep nets via a compression approach". In: *35th International Conference on Machine Learning (ICML)*. Ed. by Andreas Krause and Jennifer Dy. Stockholm, Sweden: International Machine Learning Society (IMLS), 2018, pp. 390–418.
[Asa+20]	Erfan Asaadi et al. "Assured Integration of Machine Learning-based Autonomy on Aviation Platforms". In: *2020 AIAA/IEEE 39th Digital Avionics Systems Conference (DASC)*. IEEE. 2020, pp. 1–10.
[ASTM F3269-17]	*F3269-17, Standard Practice for Methods to Safely Bound Flight Behavior of Unmanned Aircraft Systems Containing Complex Functions*. Standard. ASTM, Sept. 2017.
[Bac+15]	Sebastian Bach et al. "On pixel-wise explanations for non-linear classifier decisions by layer-wise relevance propagation". In: *PLOS ONE* 10.7 (2015).
[Bac+18]	Francois Bachoc, Fabrice Gamboa, Max Halford, Jean-Michel Loubes, and Laurent Risser. "Entropic Variable Projection for Explainability and Intepretability". Unpublished. 2018. URL: https://arxiv.org/abs/1810.07924.
[Bar+19]	Peter L. Bartlett, Nick Harvey, Christopher Liaw, and Abbas Mehrabian. "Nearly-tight VC-dimension and Pseudodimension Bounds for Piecewise Linear Neural Networks". In: *Journal of Machine Learning Research* 20.63 (2019), pp. 1–17.

[Bee+20] Emma Beede et al. "A Human-Centered Evaluation of a Deep Learning System Deployed in Clinics for the Detection of Diabetic Retinopathy". In: *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*. New York, NY, USA: Association for Computing Machinery, 2020, pp. 1–12.

[Bla+20] Davis Blalock, Jose Javier Gonzalez Ortiz, Jonathan Frankle, and John Guttag. "What is the State of Neural Network Pruning?" In: *Proceedings of Machine Learning and Systems*. Mar. 2020, pp. 129–146.

[BLZ+19] Junjie Bai, Fang Lu, Ke Zhang, et al. *ONNX: Open Neural Network Exchange*. 2019. URL: https://onnx.ai/.

[BML19] Philipp Bergmann, Tim Meinhardt, and Laura Leal-Taixé. "Tracking Without Bells and Whistles". In: *The IEEE International Conference on Computer Vision (ICCV)*. Seoul, Korea, Oct. 2019.

[Bra19] Gwern Branwen. *The Neural Net Tank Urban Legend*. https://www.gwern.net/Tanks. Aug. 2019.

[CAS16] FAA Certification Authorities Software Team (CAST). *Position Paper CAST-32A: Multi-core Processors*. Nov. 2016.

[Che+18] Tianqi Chen et al. "TVM: An Automated End-to-End Optimizing Compiler for Deep Learning". In: *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. Carlsbad, CA: USENIX Association, 2018, pp. 578–594. URL: https://www.usenix.org/conference/osdi18/presentation/chen.

[Che+19] Runjin Chen, Hao Chen, Jie Ren, Ge Huang, and Quanshi Zhang. "Explaining neural networks semantically and quantitatively". In: *IEEE/CVF International Conference on Computer Vision (CVPR)*. Long Beach, CA, USA, 2019, pp. 9187–9196.

[CM-AS-010] EASA. *Certification Memorandum: FLARM system installations in CS-23, CS 27 and CS-29 aircraft*. May 2019.

[CoDANN20] EASA and Daedalean AG. *Concepts of Design Assurance for Neural Networks*. Tech. rep. 2020. URL: https://www.easa.europa.eu/sites/default/files/dfu/EASA-DDLN-Concepts-of-Design-Assurance-for-Neural-Networks-CoDANN.pdf.

[Cof+20] Darren Cofer et al. "Run-Time Assurance for Learning-Enabled Systems". In: *NASA Formal Methods – DARPA Assured Autonomy*. Aug. 2020, pp. 361–368.

[CS-23] EASA. *Certification Specification for Normal, Utility, Aerobatic, and Commuter Category Aeroplanes*. Standard. Amendment 5. Mar. 2017.

[CS-27] EASA. *Certification Specifications and Acceptable Means of Compliance for Small Rotorcraft*. Standard. Amendment 7. June 2020.

[CS-29] EASA. *Certification Specifications for Large Rotorcraft*. Standard. Amendment 7. July 2019.

[CUDAPTX] NVIDIA. *Parallel Thread Execution ISA*. Accessed on 2021-03-23. URL: https://docs.nvidia.com/cuda/pdf/ptx_isa_7.2.pdf.

[DD09] Armen Der Kiureghian and Ove Ditlevsen. "Aleatory or epistemic? Does it matter?" In: *Structural Safety* 31.2 (2009), pp. 105–112.

[Dev+19] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding". In: *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. Minneapolis, Minnesota: Association for Computational Linguistics, June 2019, pp. 4171–4186.

[Dev+21] S. K. Devitt et al. *Robotics Roadmap for Australia V2 – Trust and Safety*. Forthcoming. 2021.

[DK17] Finale Doshi-Velez and Been Kim. "Towards A Rigorous Science of Interpretable Machine Learning". In: *Explainable and Interpretable Models in Computer Vision and Machine Learning*. Challenges in Machine Learning. to appear. Springer, 2017.

[DO-365B] *DO-365B, Minimum Operational Performance Standards (MOPS) for Detect and Avoid (DAA) Systems*. Standard. EUROCAE/RTCA, Mar. 2021.

[DO-387] *DO-387, Minimum Operational Performance Standards (MOPS) for Electro-Optical/Infrared (EO/IR) Sensors System for Traffic Surveillance*. Draft. EUROCAE/RTCA, Mar. 2020.

[DR17] Gintare Karolina Dziugaite and Daniel M. Roy. "Computing Nonvacuous Generalization Bounds for Deep (Stochastic) Neural Networks with Many More Parameters than Training Data". In: *Proceedings of the Thirty-Third Conference on Uncertainty in Artificial Intelligence, UAI 2017, Sydney, Australia, August 11-15, 2017*. 2017.

[EAS20] EASA. *Artificial Intelligence Roadmap: A human-centric approach to AI in aviation*. Feb. 2020. URL: https://www.easa.europa.eu/sites/default/files/dfu/EASA-AI-Roadmap-v1.0.pdf.

[EAS21] EASA. *First usable guidance for Level 1 machine learning applications*. Concept paper for consultation. Apr. 2021. URL: https://www.easa.europa.eu/newsroom-and-events/news/easa-releases-consultation-its-first-usable-guidance-level-1-machine.

[ED-12C/DO-178C] *ED-12C/DO-178C, Software Considerations in Airborne Systems and Equipment Certification*. Standard. EUROCAE/RTCA, Jan. 2011.

[ED-14G/DO-160G] *ED-14G/DO-160G: Environmental Conditions and Test Procedures for Airborne Equipment*. Standard. EUROCAE/RTCA, Dec. 2010.

[ED-218/DO-331] *ED-218/DO-331, Model Based Development and Verification Supplement to DO-178C and DO-278A*. Standard. EUROCAE/RTCA, Dec. 2011.

[ED-258] *ED-258, Operational Services and Environment Description for Detect and Avoid [Traffic] in Class D-G Airspaces under VFR/IFR*. Standard. EUROCAE, Jan. 2019.

[ED-79A/ARP4754A] *ED-79A/ARP4754A, Guidelines for Development of Civil Aircraft and Systems*. Standard. EUROCAE/SAE, Dec. 2011.

[EGTA] *Ethics and Guidelines on Trustworthy AI*. Tech. rep. European Commission's High-Level Expert Group on Artificial Intelligence, Apr. 2019. URL: https://ec.europa.eu/digital-single-market/en/news/ethics-guidelines-trustworthy-ai.

[Erh+09] Dumitru Erhan, Yoshua Bengio, Aaron Courville, and Pascal Vincent. *Visualizing Higher-Layer Features of a Deep Network*. Tech. rep. 1341. Université de Montréal, 2009.

[ESL] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning*. Springer Series in Statistics. Springer, 2001.

[Eur21] Eurocontrol. *FLY AI webinar – EASA AI Trustworthiness Guidance: paving the way to safety-related AI certification*. Apr. 2021. URL: https://www.eurocontrol.int/event/easa-ai-trustworthiness-guidance-paving-way-safety-related-ai-certification.

[FAA-23.1309-1E] FAA. *23.1309-1E - System Safety Analysis and Assessment for Part 23 Airplanes*. Nov. 2011.

[For+20] Håkan Forsberg, Joakim Lindén, Johan Hjorth, Torbjörn Månefjord, and Masoud Daneshtalab. "Challenges in Using Neural Networks in Safety-Critical Applications". In: *2020 AIAA/IEEE 39th Digital Avionics Systems Conference (DASC)*. 2020, pp. 1–7.

[GA19] David Gunning and David Aha. "DARPA's explainable artificial intelligence (XAI) program". In: *AI Magazine* 40.2 (2019), pp. 44–58.

[GBC16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

[Gho+21] Amir Gholami et al. *A Survey of Quantization Methods for Efficient Neural Network Inference*. Preprint. 2021. URL: https://arxiv.org/abs/2103.13630.

[GLL19] Golnaz Ghiasi, Tsung-Yi Lin, and Quoc V. Le. "NAS-FPN: Learning Scalable Feature Pyramid Architecture for Object Detection". In: *IEEE Conference on Computer Vision and Pattern Recognition, CVPR*. Long Beach, CA, USA: Computer Vision Foundation / IEEE, 2019, pp. 7036–7045.

[Gui+18] Riccardo Guidotti et al. "A Survey of Methods for Explaining Black Box Models". In: *ACM Comput. Surv.* 51.5 (Aug. 2018).

[Guo+17] Chuan Guo, Geoff Pleiss, Yu Sun, and Kilian Q Weinberger. "On calibration of modern neural networks". In: *Proceedings of the 34th International Conference on Machine Learning*. Vol. 70. 2017, pp. 1321–1330.

[He+16] K. He, X. Zhang, S. Ren, and J. Sun. "Deep Residual Learning for Image Recognition". In: *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. Las Vegas, NV, USA, 2016, pp. 770–778.

[HSD73] Robert M. Haralick, K. Shanmugam, and Its'Hak Dinstein. "Textural Features for Image Classification". In: *IEEE Transactions on Systems, Man, and Cybernetics* SMC-3.6 (1973), pp. 610–621.

[Hub+12] Catherine Huber-Carol, Narayanaswamy Balakrishnan, M Nikulin, and M Mesbah. *Goodness-of-fit tests and model validity*. Springer, 2012.

[ISO98-3] ISO/IEC GUIDE 98-3:2008. *Uncertainty of measurement – Part 3: Guide to the expression of uncertainty in measurement*. Standard. International Organization for Standardization, Nov. 2008.

[Jac+18] Benoit Jacob et al. "Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. Long Beach, CA, USA, June 2018.

[JKO19] Kyle D. Julian, Mykel J. Kochenderfer, and Michael P. Owen. "Deep Neural Network Compression for Aircraft Collision Avoidance Systems". In: *Journal of Guidance, Control, and Dynamics* 42.3 (2019), pp. 598–608.

[Jou+17] Norman P. Jouppi et al. "In-Datacenter Performance Analysis of a Tensor Processing Unit". In: *SIGARCH Comput. Archit. News* 45.2 (June 2017), pp. 1–12.

[KHC12] Mykel J. Kochenderfer, Jessica E. Holland, and James P. Chryssanthacopoulos. "Next-Generation Airborne Collision Avoidance System". In: *Lincoln Laboratory Journal* 19.1 (2012).

[Kim+18] Been Kim et al. "Interpretability beyond feature attribution: Quantitative testing with concept activation vectors (tcav)". In: *International Conference on Machine Learning (ICML)*. PMLR. Stockholm, Sweden, 2018, pp. 2668–2677.

[Kin+18] Pieter Jan Kindermans et al. "Learning how to explain neural networks: Patternnet and Patternattribution". In: *6th International Conference on Learning Representations (ICLR)*. Vancouver, Canada, 2018.

[KKB19] P. Koopman, A. Kane, and J. Black. "Credible Autonomy Safety Argumentation". In: *Safety-Critical Systems Symposium*. 2019.

[KKK16] Been Kim, Rajiv Khanna, and Oluwasanmi O Koyejo. "Examples are not enough, learn to criticize! Criticism for Interpretability". In: *Advances in Neural Information Processing Systems*. Ed. by D. Lee, M. Sugiyama, U. Luxburg, I. Guyon, and R. Garnett. Vol. 29. Barcelona, Spain: Curran Associates, Inc., 2016.

[Kri09] Alex Krizhevsky. *Learning multiple layers of features from tiny images*. Tech. rep. University of Toronto, 2009.

[Kri18] Raghuraman Krishnamoorthi. *Quantizing deep convolutional networks for efficient inference: A whitepaper*. Whitepaper. 2018. URL: https://arxiv.org/abs/1806.08342.

[KRS14] Been Kim, Cynthia Rudin, and Julie Shah. "The Bayesian Case Model: A Generative Approach for Case-Based Reasoning and Prototype Classification". In: *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2*. NIPS'14. Montreal, Canada: MIT Press, 2014, pp. 1952–1960.

[KSH12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "ImageNet Classification with Deep Convolutional Neural Networks". In: *Advances in Neural Information Processing Systems*. Ed. by F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger. Vol. 25. Curran Associates, Inc., 2012, pp. 1097–1105.

[Lec+98] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. "Gradient-based learning applied to document recognition". In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324.

[Let+15] Benjamin Letham, Cynthia Rudin, Tyler H McCormick, David Madigan, et al. "Interpretable classifiers using rules and bayesian analysis: Building a better stroke prediction model". In: *Annals of Applied Statistics* 9.3 (2015), pp. 1350–1371.

[Li+17] Hao Li, Asim Kadav, Igor Durdanovic, Hanan Samet, and Hans Peter Graf. "Pruning Filters for Efficient ConvNets". In: *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. 2017.

[Lip18] Zachary C. Lipton. "The Mythos of Model Interpretability: In Machine Learning, the Concept of Interpretability is Both Important and Slippery." In: *Queue* 16.3 (June 2018), pp. 31–57.

[LL17] Scott M. Lundberg and Su-In Lee. "A Unified Approach to Interpreting Model Predictions". In: NIPS'17. Long Beach, California, USA: Curran Associates Inc., 2017, pp. 4768–4777.

[Lu+20] Lu Lu, Yeonjong Shin, Yanhui Su, and George Em Karniadakis. "Dying ReLU and Initialization: Theory and Numerical Examples". In: *Communications in Computational Physics* 28.5 (2020), pp. 1671–1706.

[Maa09] Laurens van der Maaten. "Learning a Parametric Embedding by Preserving Local Structure". In: *Proceedings of the Twelth International Conference on Artificial Intelligence and Statistics*. Ed. by David van Dyk and Max Welling. Vol. 5. Proceedings of Machine Learning Research. PMLR, Apr. 2009, pp. 384–391.

[Mar+15] Martin Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: http://tensorflow.org/.

[Meh+19] Ninareh Mehrabi, Fred Morstatter, Nripsuta Saxena, Kristina Lerman, and Aram Galstyan. "A Survey on Bias and Fairness in Machine Learning". Unpublished. Aug. 2019. URL: https://arxiv.org/abs/1908.09635.

[MH08] Laurens van der Maaten and Geoffrey Hinton. "Visualizing Data using t-SNE". In: *Journal of Machine Learning Research* 9.86 (2008), pp. 2579–2605.

[Mol+17] Pavlo Molchanov, Stephen Tyree, Tero Karras, Timo Aila, and Jan Kautz. "Pruning Convolutional Neural Networks for Resource Efficient Inference". In: *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. 2017.

[Mol19] Christoph Molnar. *Interpretable Machine Learning. A Guide for Making Black Box Models Explainable*. https://christophm.github.io/interpretable-ml-book/. 2019.

[Mon+17] Grégoire Montavon, Sebastian Lapuschkin, Alexander Binder, Wojciech Samek, and Klaus-Robert Müller. "Explaining nonlinear classification decisions with deep taylor decomposition". In: *Pattern Recognition* 65 (2017), pp. 211–222.

[MSM18] Grégoire Montavon, Wojciech Samek, and Klaus-Robert Müller. "Methods for interpreting and understanding deep neural networks". In: *Digital Signal Processing* 73 (2018), pp. 1–15.

[MW] *Merriam–Webster dictionnary*. https://www.merriam-webster.com/. Oct. 2020.

[NC15] Alexandru Niculescu-Mizil and Rich Caruana. “Predicting Good Probabilities with Supervised Learning”. In: *Proceedings of the 22nd International Conference on Machine Learning (ICML)*. Bonn, Germany: Association for Computing Machinery, 2015, pp. 625–632.

[Nic98] Raymond S. Nickerson. “Confirmation Bias: A Ubiquitous Phenomenon in Many Guises”. In: *Review of General Psychology* 2.2 (1998), pp. 175–220.

[Pas+19] Adam Paszke et al. “PyTorch: An Imperative Style, High-Performance Deep Learning Library”. In: *Advances in Neural Information Processing Systems 32*. Ed. by H. Wallach et al. Curran Associates, Inc., 2019, pp. 8024–8035.

[Pea+18] Tim Pearce, Alexandra Brintrup, Mohamed Zaki, and Andy Neely. “High-Quality Prediction Intervals for Deep Learning: A Distribution-Free, Ensembled Approach”. In: *Proceedings of the 35th International Conference on Machine Learning*. Ed. by Jennifer Dy and Andreas Krause. Vol. 80. Proceedings of Machine Learning Research. PMLR, July 2018, pp. 4075–4084.

[Ped+11] F. Pedregosa et al. “Scikit-learn: Machine Learning in Python”. In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[Per84] Charles Perrow. *Normal Accidents*. Basic Books, 1984.

[Phi+20] P. Jonathon Phillips, Carina A. Hahn, Peter C. Fontana, David A. Broniatowski, and Mark A. Przybocki. *Four Principles of Explainable Artificial Intelligence (NISTIR 8312)*. Draft. National Institute of Standards and Technology, Aug. 2020.

[Pim+14] Marco A.F. Pimentel, David A. Clifton, Lei Clifton, and Lionel Tarassenko. “A review of novelty detection”. In: *Signal Processing* 99 (2014), pp. 215–249.

[Qin+18] Zhuwei Qin, Fuxun Yu, Chenchen Liu, and Xiang Chen. “How convolutional neural networks see the world — A survey of convolutional neural network visualization methods”. In: *Mathematical Foundations of Computing* 1 (2018), p. 149.

[RL13] Leanna Rierson. *Developing Safety-Critical Software, A Practical Guide for Aviation Software and DO-178C Compliance*. CRC Press, 2013.

[RSG16] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. “"Why Should I Trust You?": Explaining the Predictions of Any Classifier”. In: KDD ’16. San Francisco, CA, USA: Association for Computing Machinery, 2016, pp. 1135–1144.

[Rud19] Cynthia Rudin. “Stop Explaining Black Box Machine Learning Models for High Stakes Decisions and Use Interpretable Models Instead”. In: *Nature Machine Intelligence* 1 (2019), pp. 206–215.

[Sae00] M. Saerens. “Building cost functions minimizing to some summary statistics”. In: *IEEE Transactions on Neural Networks* 11.6 (2000), pp. 1263–1271.

[Sal02] Andrea Saltelli. “Sensitivity Analysis for Importance Assessment”. In: *Risk Analysis* 22.3 (2002), pp. 579–590.

[Sam+19] Wojciech Samek, Grégoire Montavon, Andrea Vedaldi, Lars Kai Hansen, and Klaus-Robert Müller. *Explainable AI: interpreting, explaining and visualizing deep learning*. Vol. 11700. Springer Nature, 2019.

[SC-VTOL-01] EASA. *Special Condition for small-category VTOL aircraft*. Standard. July 2019.

[Sch+20] Gesina Schwalbe et al. "Structuring the Safety Argumentation for Deep Neural Network Based Perception in Automotive Applications". In: *Computer Safety, Reliability, and Security. SAFECOMP 2020 Workshops*. Ed. by António Casimiro, Frank Ortmeier, Erwin Schoitsch, Friedemann Bitsch, and Pedro Ferreira. Springer International Publishing, 2020, pp. 383–394.

[Sej18] Terrence J. Sejnowski. *The Deep Learning Revolution*. MIT Press, 2018.

[Sel+17] R. R. Selvaraju et al. "Grad-CAM: Visual Explanations from Deep Networks via Gradient-Based Localization". In: *2017 IEEE International Conference on Computer Vision (ICCV)*. 2017, pp. 618–626.

[Sim06] Dan Simon. *Optimal State Estimation: Kalman, H Infinity, and Nonlinear Approaches*. USA: Wiley-Interscience, 2006.

[SPIRV] John Kessenich, Boaz Ouriel, and Raun Krisch. *SPIR-V Specification*. Accessed on 2021-03-23. URL: https://www.khronos.org/registry/spir-v/specs/unified1/SPIRV.html.

[Spr+15] J.T. Springenberg, A. Dosovitskiy, T. Brox, and M. Riedmiller. "Striving for Simplicity: The All Convolutional Net". In: *ICLR (workshop track)*. San Diego, CA, USA, 2015.

[STPA] Nancy G. Leveson and John P. Thomas. *STPA Handbook*. Mar. 2018. URL: http://psas.scripts.mit.edu/home/materials/.

[STY17] Mukund Sundararajan, Ankur Taly, and Qiqi Yan. "Axiomatic Attribution for Deep Networks". In: *Proceedings of the 34th International Conference on Machine Learning*. 2017, pp. 3319–3328.

[SVZ14] Karen Simonyan, Andrea Vedaldi, and Andrew Zisserman. "Deep Inside Convolutional Networks: Visualising Image Classification Models and Saliency Maps". In: *2nd International Conference on Learning Representations, ICLR, Workshop Track Proceedings*. Ed. by Yoshua Bengio and Yann LeCun. Banff, Canada, Apr. 2014.

[SWM17] Wojciech Samek, Thomas Wiegand, and Klaus-Robert Müller. "Explainable artificial intelligence: Understanding, visualizing and interpreting deep learning models". In: *ITU Journal: ICT Discoveries* 1.1 (Oct. 2017).

[SZ15] Karen Simonyan and Andrew Zisserman. "Very Deep Convolutional Networks for Large-Scale Image Recognition". In: *International Conference on Learning Representations (ICLR)*. San Diego, CA, USA, 2015.

[TBF05] Sebastian Thrun, Wolfram Burgard, and Dieter Fox. *Probabilistic Robotics (Intelligent Robotics and Autonomous Agents)*. The MIT Press, 2005.

[UL-4600] Edge Case Research. *Standard for Safety for the Evaluation of Autonomous Vehicles and Other Products*. standard. Underwriter Laboratories, Apr. 2020.

[Verilog] 1800_WG - SystemVerilog Language Working Group. *IEEE 1800-2017 - Standard for SystemVerilog*. Dec. 2018.

[VHDL] P1076 - VHDL Analysis and Standardization Group. *IEEE 1076-2019 - Standard for VHDL Language*. Dec. 2019.

[Wan+17] Tong Wang et al. "A bayesian framework for learning rule sets for interpretable classification". In: *The Journal of Machine Learning Research* 18.1 (2017), pp. 2357–2393.

[WL16] Ronald L. Wasserstein and Nicole A. Lazar. "The ASA Statement on p-Values: Context, Process, and Purpose". In: *The American Statistician* 70.2 (2016), pp. 129–133.

[Wor21] DEEL Certification Workgroup. *Machine Learning in Certified Systems*. Whitepaper. IRT Saint Exupéry, Mar. 2021.

[Xu+20] Yihong Xu et al. "How To Train Your Deep Multi-Object Tracker". In: *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. Seattle, USA, June 2020, pp. 1–14.

[Yos+15] Jason Yosinski, Jeff Clune, Anh Nguyen, Thomas Fuchs, and Hod Lipson. "Understanding Neural Networks Through Deep Visualization". In: *Deep Learning Workshop, International Conference on Machine Learning (ICML)*. Lille, France, 2015.

[Zaf+19] Ofir Zafrir, Guy Boudoukh, Peter Izsak, and Moshe Wasserblat. "Q8BERT: Quantized 8Bit BERT". In: *5th Workshop on Energy Efficient Machine Learning and Cognitive Computing - NeurIPS 2019*. Vancouver, Canada, 2019.

[ZF14] Matthew D. Zeiler and Rob Fergus. "Visualizing and Understanding Convolutional Networks". In: *Computer Vision – ECCV 2014*. Ed. by David Fleet, Tomas Pajdla, Bernt Schiele, and Tinne Tuytelaars. Springer International Publishing, 2014, pp. 818–833.

[Zha+18] Quanshi Zhang, Ruiming Cao, Feng Shi, Ying Nian Wu, and Song-Chun Zhu. "Interpreting CNN knowledge via an explanatory graph". In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 32. 1. New Orleans, USA, 2018.

[Zha+19] Quanshi Zhang, Yu Yang, Haotian Ma, and Ying Nian Wu. "Interpreting CNNs via decision trees". In: *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. Long Beach, CA, USA, 2019, pp. 6261–6270.

[Zho+16] B. Zhou, A. Khosla, A. Lapedriza, A. Oliva, and A. Torralba. "Learning Deep Features for Discriminative Localization". In: *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. Las Vegas, NV, USA, 2016, pp. 2921–2929.

[ZWZ18] Quanshi Zhang, Ying Nian Wu, and Song-Chun Zhu. "Interpretable Convolutional Neural Networks". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. Salt Lake City, UT, USA, June 2018.

Acronyms

Acceptable Mean of Compliance (AMC) . . . 81
Application Specific Integrated Circuits (ASIC) . . . 25
Artificial Intelligence (AI) . . . 8

Central Processing Units (CPU) . . . 25
Commercial-off-the-shelf (COTS) . . . 25
Concepts of Design Assurance for Neural Networks (CoDANN) . . . 5
Concepts of Operations (ConOps) . . . 13
Convolutional Neural Network (CNN) . . . 15

Detect and Avoid (DAA) . . . 13

European Union Aviation Safety Agency (EASA) . . . 5

Field Programmable Gate Arrays (FPGA) . . . 25
Functional Hazard Assessment (FHA) . . . 93

Graphics Processing Units (GPU) . . . 25

Hardware Description Language (HDL) . . . 26
Height Above Ground Level (HAGL) . . . 19
Human-Machine interaction (HMI) . . . 75

Identically independently distributed (iid) . . . 77
Image Quality Metrics (IQM) . . . 80
Innovation Partnership Contract (IPC) . . . 5
Instruction Set Architecture (ISA) . . . 26

Machine Learning (ML) . . . 8

Neural Network (NN) . . . 15
Non-Maximum Suppression (NMS) . . . 16

Open Neural Network Exchange (ONNX) . . . 33
Out-of-Distribution (OOD) . . . 77

Principal Component Analysis (PCA) . . . 79

Real Time Operating Systems (RTOS) . . . 27
Runtime Assurance (RTA) . . . 81

Single Instruction Multiple Data (SIMD) . . . 25
Systems Theoretic Process Analysis (STPA) . . . 82