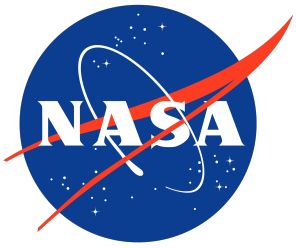
NASA/TM–20250009438



Specifying Convolutions in PVS with an Executable Model in Typed Racket

John V. Siratt
Safety Critical Avionics Systems Branch
Langley Research Center, Hampton, VA

Sarah M. Lehman
Safety Critical Avionics Systems Branch
Langley Research Center, Hampton, VA

J. Tanner Slagel
Safety Critical Avionics Systems Branch
Langley Research Center, Hampton, VA

September 2025

NASA STI Program Report Series

Since its founding, NASA has been dedicated to the advancement of aeronautics and space science. The NASA scientific and technical information (STI) program plays a key part in helping NASA maintain this important role.

The NASA STI Program operates under the auspices of the Agency Chief Information Officer. It collects, organizes, provides for archiving, and disseminates NASA's STI. The NASA STI Program provides access to the NTRS Registered and its public interface, the NASA Technical Report Server, thus providing one of the largest collections of aeronautical and space science STI in the world. Results are published in both non-NASA channels and by NASA in the NASA STI Report Series, which includes the following report types:

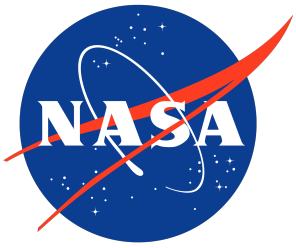
- TECHNICAL PUBLICATION. Reports of completed research or a major significant phase of research that present the results of NASA programs and include extensive data or theoretical analysis. Includes compilations of significant scientific and technical data and information deemed to be of continuing reference value. NASA counterpart of peer-reviewed formal professional papers, but having less stringent limitations on manuscript length and extent of graphic presentations.
- TECHNICAL MEMORANDUM. Scientific and technical findings that are preliminary or of specialized interest, e.g., quick release reports, working papers, and bibliographies that contain minimal annotation. Does not contain extensive analysis.
- CONTRACTOR REPORT. Scientific and technical findings by NASA-sponsored contractors and grantees.
- CONFERENCE PUBLICATION. Collected papers from scientific and technical conferences, symposia, seminars, or other meetings sponsored or co-sponsored by NASA.
- SPECIAL PUBLICATION. Scientific, technical, or historical information from NASA programs, projects, and missions, often concerned with subjects having substantial public interest.
- TECHNICAL TRANSLATION. English-language translations of foreign scientific and technical material pertinent to NASA's mission.

Specialized services also include organizing and publishing research results, distributing specialized research announcements and feeds, providing information desk and personal search support, and enabling data exchange services.

For more information about the NASA STI Program, see the following:

- Access the NASA STI program home page at `http://www.sti.nasa.gov`
- Help desk contact information: `https://www.sti.nasa.gov/sti-contact-form/` and select the "General" help request type.

NASA/TM–20250009438



Specifying Convolutions in PVS with an Executable Model in Typed Racket

John V. Siratt
Safety Critical Avionics Systems Branch
Langley Research Center, Hampton, VA

Sarah M. Lehman
Safety Critical Avionics Systems Branch
Langley Research Center, Hampton, VA

J. Tanner Slagel
Safety Critical Avionics Systems Branch
Langley Research Center, Hampton, VA

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23681-2199

September 2025

Acknowledgments

This work was supported by the NASA Aeronautics Research Mission Directorate, through the Convergent Aeronautics Solutions (CAS) Project.

The use of trademarks or names of manufacturers in this report is for accurate reporting and does not constitute an official endorsement, either expressed or implied, of such products or manufacturers by the National Aeronautics and Space Administration.

Available from:

NASA STI Program / Mail Stop 150
NASA Langley Research Center
Hampton, VA 23681-2199

Abstract

This technical memorandum presents work towards assuring convolutional neural network operations using formal verification methods, specifically within the context of the Aerial AId activity. Aerial AId is an activity under the Convergent Aeronautic Solutions Project focused on developing assured computer vision systems for deployment onboard Uncrewed Aerial Systems (UAS) to enable rapid emergency scene assessment in support of first responder operations.

The primary contributions of this work include: (1) a formalization of convolutional operations in the Prototype Verification System (PVS), and (2) the development of a rapid formal prototyping methodology that integrates PVS with Typed Racket for concurrent specification and implementation development. This approach addresses the challenge of applying formal methods in an agile engineering environment by establishing a correspondence between executable Racket code and formal PVS specifications, enabling iterative development while maintaining mathematical rigor in the verification of neural network components.

While tasks such as property selection for verification, formalization of additional network components, and performance benchmarking remain areas for future investigation, this work demonstrates both the feasibility of formal verification for neural network operations and a practical strategy for integrating formal methods into rapid development cycles. The results provide value both as formal artifacts and as a foundational approach for applying formal verification methods in safety-critical AI/ML systems for aerospace applications.

Contents

1 Introduction **3**
1.1 Aerial **AI**d Overview 4
1.2 AI Assurance Overview 4

2 Mathematical Models of Convolutions **6**
2.1 Convolutional Kernels 7
2.2 Modeling a Convolution 9

3 Formalizing Convolutional Operations **12**
3.1 Tools 13
3.2 Typed Racket/PVS Correspondence 14

4 Results **17**

5 Future Work **20**

List of Figures

1 Aerial perspective of outdoor terrain 5
2 Multilayer perceptron diagram 6
3 Convolution as sliding window 8
4 Formal methods semantic gaps 14
5 Typed Racket/PVS correspondence: monus operation 15
6 Typed Racket/PVS correspondence: finite array 16
7 Specification of a convolution 18

List of Tables

1 PVS proof counts 19

1 Introduction

This report describes work conducted on behalf of the Aerial **AI**d project within the Convergent Aeronautics Solutions (CAS) project to advance safety assurance of Artificial Intelligence and Machine Learning (AI/ML)-enabled systems through formal verification[1] of neural networks. The choice of neural networks as a focus for this research is grounded in the explosive growth and complexity in general purpose, commercially popular neural networks such as the You Only Look Once (YOLO)[2] computer vision system. YOLO in particular has achieved a high degree of popularity due to its high level of performance, achieving up to 85% accuracy[3] when detecting individual humans in group photos [5], as well as its ease of use. As powerful as models such as YOLO are, they can still make mistakes, thanks to the brittle nature of vision models in the face of changing conditions such as unexpected viewing angles or light levels. When this happens, important objects in the scene can be incorrectly labeled or even missed entirely. The difficulty in providing formal guarantees about model behavior in the face of these limitations, therefore, poses a significant barrier to adoption in high-criticality industries such as aerospace and emergency medical response where erroneous or misleading model outputs can lead to property damage, human injury, and other undesirable outcomes.

CAS is a NASA project which invests in creative and innovative solutions for challenges faced by the aerospace community and similar industries [14]. Furthermore, CAS is focused on execution activities meant to trailblaze research in new directions that contribute to a transformational view of the future. To this end the Aerial **AI**d goals were focused creating a computer vision model for rapid emergency scene assessment and enabling the assurance of such computer vision software as part of safety- and mission-critical Uncrewed Aerial Systems (UAS) operations, with the purpose of producing transferable research. The concurrent development of both a prototype computer vision system and the means to apply formal methods introduced the need for rapid formal specification of different neural networks, tightly coupled with an executable implementation via a general purpose programming language.

This work was motivated by several approaches which were conceived as possible long-term solutions to the assurance problem in such an environment: (1) a formal tool which would facilitate the creation of formal artifacts for use in updating formal arguments, based on the updates on the engineering side, and (2) a "toolbox" enabling the piecing together of verified components into a working model. The result would offer flexible and scalable means to formally assure models in an environment where the model's structure and complexity frequently change, which is often the case in the development of AI/ML systems.

This report describes the first steps towards this vision and the concrete outcomes of the Aerial **AI**d assurance team's work toward formal verification of neural

[1] What is Formal Methods? `https://shemesh.larc.nasa.gov/fm/fm-what.html`

[2] YOLO has had several versions, for this effort versions 8 and 11 developed by Ultralytics `https://docs.ultralytics.com/models/yolov8/`

[3] In object detection tasks, accuracy is measured through intersection-over-union (IOU) which reflects the overlap of the predicted and ground truth bounding boxes of the target object.

networks for computer vision. It includes:

- a formalization of convolutional operations in the Prototype Verification System (PVS), and
- an organically developed method of *rapid formal prototyping* using PVS and Typed Racket.

Tasks such as selection of properties to be verified, formalization of other network components, and benchmarks for determining sufficient runtime performance of the formalized model remain open problems for future work. Despite this, what is reported here has value both as a formal artifact and as a potential strategy for applying formal methods in an agile engineering environment.

1.1 Aerial AId Overview

Aerial **AI**d focuses on the Drones as First Responders (DFR) paradigm, which uses Uncrewed Aerial Systems (UAS) to support first responders (e.g., fire departments, emergency medical services, etc.) during emergency scenarios. Aerial **AI**d assumes that the UAS in this situation is helping to provide a dispatcher, incident commander, or similar personnel with situational awareness before the first responders get to a scene. This could include identifying and triaging injured persons, flagging any hazards or threatened areas within the vicinity, and other functions. This information would then be fed back to the overwatch personnel to review and forward to the first responders if it is deemed relevant. The DFR paradigm has been demonstrated to reduce emergency response time as well as increasing the efficiency and safety of the first responders [7, 11].

Aerial **AI**d uses AI/ML to perform scene assessment, using high-performing Commercial-Off-the-Shelf (COTS) models such as YOLO to identify relevant entities in the scene. YOLO is a publicly available, free-to-use model which has experienced widespread adoption in recent years due to its ease of use and wide range of capabilities. We target this model in Aerial **AI**d to align ourselves with industry interests and pave the way for future technology transfer opportunities. In particular, we leverage YOLO to perform object detection[4] and semantic segmentation[5]. This allows us to identify objects of interest in real-time from the UAS video feed, highlighting the positions of and displaying descriptive labels for each object. This high-level scene assessment concisely summarizes the current scenario for the overwatch personnel, reducing the cognitive burden otherwise imposed by manually piloting a UAS and performing their own scene assessment.

1.2 AI Assurance Overview

Computer vision models are notoriously prone to output errors when dealing with scenarios that produce data different from the data the model was trained on; a trade-off of using a COTS model like YOLO is that the data set on which it was

[4] `https://www.tensorflow.org/hub/tutorials/object_detection`
[5] `https://www.tensorflow.org/tutorials/images/segmentation`

Figure 1. Aerial perspective of outdoor terrain at the City Environment Range Testing for Autonomous Integrated Navigation (CERTAIN) test range at NASA Langley Research Center [3] misidentified by an unmodified YOLOv8 model as a giraffe.

trained primarily had the viewer on the same plane as the subject (i.e., the ground). Applying this model to a system with an aerial perspective often introduces errors where the model would normally perform quite well, such as classifying outdoor terrain as a giraffe (Figure 1). This is risky for a high-criticality domain like emergency response, as output errors can have highly deleterious consequences such as bystander injuries and property damage. To address this, the Aerial **AI**d project included an AI/ML assurance team responsible for researching ways to mitigate these potential errors.

The assurance team focused on two primary areas of investigation: 1) runtime assurance techniques for detecting out-of-domain inputs via scene-level features and volatility approximations, and 2) formal models of neural network components – particularly convolutional operations – to enable deeper analysis of learned behaviors. Here we will be reporting on efforts toward constructing formal models of convolutional processes, with results of other lines of research being presented elsewhere. Convolutions were selected as the operation of focus due to their integral contributions to vision systems; indeed, while various versions of YOLO have different methods of pre-processing and post-processing, the backbone is a relatively stable chain of convolutional blocks. However, the level of assurance necessary for safety-critical systems remains an open problem for convolutional neural networks such as YOLO.

The rest of this technical memorandum is organized as follows. Section 2 gives an overview of mathematical models of convolutions, with a an overview of the formalization with rapid formal prototyping described in Section 3. Section 4 gives details

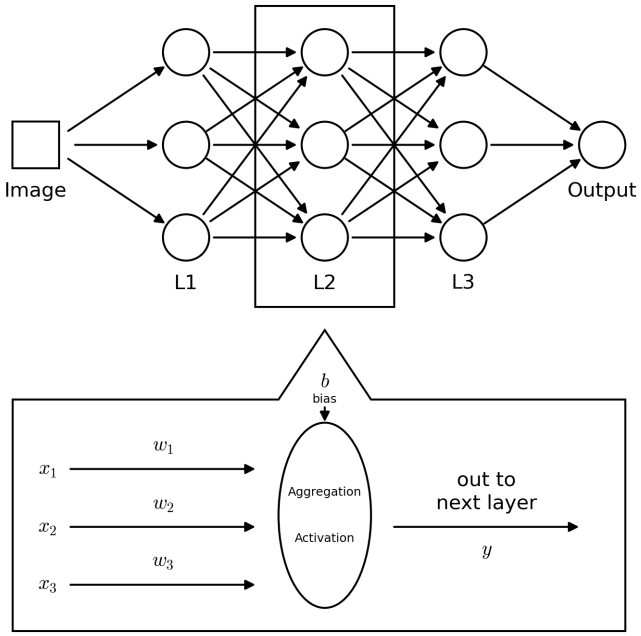


Figure 2. Architecture diagram of a fully connected, feed-forward neural network known as a *multilayer perceptron (MLP)*. Call-out highlights the operations of a given neuron within a given layer.

of the completed formalization, and Section 5 concludes and describes avenues for future work.

2 Mathematical Models of Convolutions

While contemporary, commercially popular neural networks are quite complex, the underlying theory of a basic neural network is rather straightforward. Figure 2 shows an example of a small *multilayer perceptron (MLP)*, that is, a fully connected[6], feed-forward[7] neural network which accepts an input (in this case, an image), transforms it through a series of internal (or "hidden") layers, and then produces a given output. In this example, each internal layer is made up of three neurons. Each neuron accepts the outputs from the previous layer, aggregates the the inputs (and bias) as a weighted sum, applies an activation function, and then passes the output to the next layer. The call-out in Figure 2 shows the operations of a given neuron with greater detail.

In their simplest form, neural networks such as MLPs can be thought of as a

[6] A *fully connected* neural network is one in which the output of each neuron of a given layer is passed as an input to each neuron of the following layer.

[7] A *feed-forward* neural network is one in which the processing pipeline passes linearly from input to output. This is in contrast to other paradigms such as *recurrent* neural networks which loop outputs from previous time steps back to their parent layers use as supplementary inputs in the next time step.

chain of linear algebra matrix operations enacted by the individual neurons, wrapped in non-linear functions known as *activation functions* (discussed in greater detail below). Consider the example shown in the call-out of Figure 2: the neuron accepts three inputs (x_1, x_2, and x_3) and generates some output (y). The neuron also has some internal parameters whose values were set during the training process, one for each connection to neurons in the previous layer (w_1, w_2, and w_3, known as the "weights") and a value for the neuron itself (b, known as the "bias"). The neuron takes all of these values and aggregates them using vector arithmetic ($\vec{w} \cdot \vec{x} + b$) and then normalizes the resulting value with a nonlinear function (also called the "activation function") to produce the final output (y). This output is then passed to each of the neurons in the following layer, and the process continues.

Many changes and improvements have been proposed over the decades to evolve neural networks from MLPs to the incredibly complex and highly performant models available today. In the realm of computer vision, one of the principal contributions in this regard was the convolutional transformation used to identify key features in the image which, when combined with features extracted by other layers, have demonstrated success in tasks such as classification, object detection, and semantic segmentation [4,8,10]. Informally speaking, a convolutional operation performs feature extraction by processing information from the neighborhoods surrounding each pixel, offering a massive improvement in vision task metrics compared to previous state-of-the-art approaches. It is exactly this fundamental significance to today's vision models that led us to focus on the convolutional operation for our formal assurance efforts.

2.1 Convolutional Kernels

For the purposes of computer vision, a convolution refers to the iterative process of applying one matrix of set size (known as the "kernel") in steps across an input matrix, resulting in an output matrix which then contains a set of discriminatory features relevant to the input image. As in Figure 2, where (x_1, x_2, x_3, b) represent internal parameters learned during training, the values of the kernel are also learned parameters. Equation 1 shows two such possible kernels, where S_{vert} and S_{horiz} detect vertical and horizontal edges respectively by approximating the derivative in each respective direction [21]. The kernel is applied to the input image in a series of sliding windows, the results of which each become an entry in the output matrix, as demonstrated in Figure 3.

$$S_{horiz} = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \qquad S_{vert} = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} \tag{1}$$

Mapping a convolutional neural network into an MLP structure, we can see a high amount of regularity and repetition. Aside from the sparseness of non-zero weights, the repeated application of a static kernel across an input channel results in many duplicated weights. This regularity sets convolutional operations apart from the more general problems of analyzing arbitrary feed-forward neural networks, an

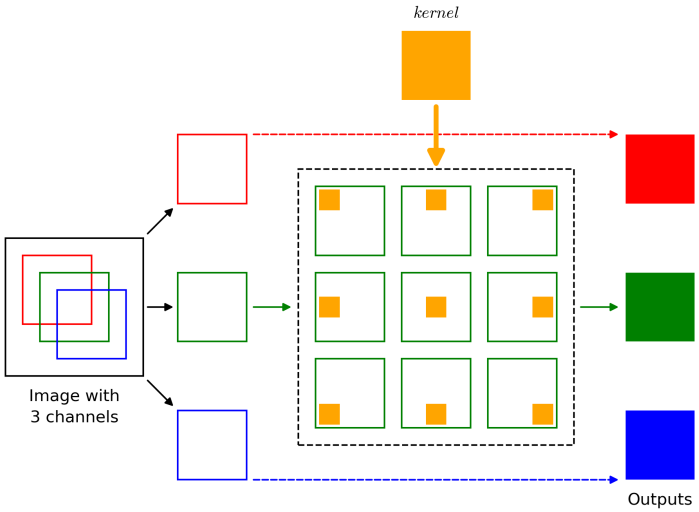


Figure 3. In the "sliding window" interpretation of a convolution, the kernel is systematically applied to different regions of the input. Each application produces one pixel of the output. Above we have illustrated the green channel, but the process occurs for each image channel.

observation which is well-supported in standard literature [22]. Therefore, by mechanically isolating the true machine learning component – the kernel – via formal analogs of the convolution process, we narrow the focus of assuring the result. Formal components would provide a path for comparing against a fully implemented convolutional block extracted from the backbone of a system like YOLO, allowing us to separate problems resulting from learned parameters from those arising from mismatches between the expected behavior represented by formal components and the actual implemented behavior of YOLO. A full, efficient implementation could go further, allowing us to use off-the-shelf solutions to train kernels and then transfer them to a formally verified platform for assured deployment.

If we consider the kernel to be the primary product of the training process for a convolutional operation, we can treat kernels themselves as parameters in the modeling process. The rest of the machinery in a neural net convolution can be formally modeled using the same tools we use for conventional software. In lieu of documented specification for a specific library or tool, we can use the large body of informal description to extract intent. That is, we use the underlying theory of how a convolutional process is *supposed* to work to anchor the model, rather than trying to recreate the exact logic of an existing AI/ML library such as TensorFlow[8] or Scikit-Learn[9]. In the following section, we discuss the concrete mathematical modeling for the convolutional process.

[8]TensorFlow: https://www.tensorflow.org/

[9]Scikit-Learn `https://scikit-learn.org/stable/`

2.2 Modeling a Convolution

Let I be a 4×4 RGB image represented as a matrix of triples (Eq. 2), which in turn naturally decomposes into three image channels (Eq. 3):

$$I = \begin{bmatrix} (r_1, g_1, b_1) & (r_2, g_2, b_2) & (r_3, g_3, b_3) & (r_4, g_4, b_4) \\ (r_5, g_5, b_5) & (r_6, g_6, b_6) & (r_7, g_7, b_7) & (r_8, g_8, b_8) \\ (r_9, g_9, b_9) & (r_{10}, g_{10}, b_{10}) & (r_{11}, g_{11}, b_{11}) & (r_{12}, g_{12}, b_{12}) \\ (r_{13}, g_{13}, b_{13}) & (r_{14}, g_{14}, b_{14}) & (r_{15}, g_{15}, b_{15}) & (r_{16}, g_{16}, b_{16}) \end{bmatrix} \tag{2}$$

$$R = \begin{bmatrix} r_1 & r_2 & r_3 & r_4 \\ r_5 & r_6 & r_7 & r_8 \\ r_9 & r_{10} & r_{11} & r_{12} \\ r_{13} & r_{14} & r_{15} & r_{16} \end{bmatrix} \quad G = \begin{bmatrix} g_1 & g_2 & g_3 & g_4 \\ g_5 & g_6 & g_7 & g_8 \\ g_9 & g_{10} & g_{11} & g_{12} \\ g_{13} & g_{14} & g_{15} & g_{16} \end{bmatrix} \quad B = \begin{bmatrix} b_1 & b_2 & b_3 & b_4 \\ b_5 & b_6 & b_7 & b_8 \\ b_9 & b_{10} & b_{11} & b_{12} \\ b_{13} & b_{14} & b_{15} & b_{16} \end{bmatrix} \tag{3}$$

Let K be a 2×2 kernel (Eq. 4) which, in order to be applied to an image channel, must be embedded within larger frames (Eq: 5):

$$K = \begin{bmatrix} k_1 & k_2 \\ k_3 & k_4 \end{bmatrix} \tag{4}$$

$$\begin{aligned} K_{0,0} &= \begin{bmatrix} k_1 & k_2 & 0 & 0 \\ k_3 & k_4 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \quad K_{0,1} = \begin{bmatrix} 0 & k_1 & k_2 & 0 \\ 0 & k_3 & k_4 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \quad K_{0,2} = \begin{bmatrix} 0 & 0 & k_1 & k_2 \\ 0 & 0 & k_3 & k_4 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \\ K_{1,0} &= \begin{bmatrix} 0 & 0 & 0 & 0 \\ k_1 & k_2 & 0 & 0 \\ k_3 & k_4 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \quad K_{1,1} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & k_1 & k_2 & 0 \\ 0 & k_3 & k_4 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \quad K_{1,2} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & k_1 & k_2 \\ 0 & 0 & k_3 & k_4 \\ 0 & 0 & 0 & 0 \end{bmatrix} \\ K_{2,0} &= \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ k_1 & k_2 & 0 & 0 \\ k_3 & k_4 & 0 & 0 \end{bmatrix} \quad K_{2,1} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & k_1 & k_2 & 0 \\ 0 & k_3 & k_4 & 0 \end{bmatrix} \quad K_{2,2} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & k_1 & k_2 \\ 0 & 0 & k_3 & k_4 \end{bmatrix} \end{aligned} \tag{5}$$

We define the Frobenius inner product of two matrices $\langle M, N \rangle_F$ as the sum of the elements of the Hadamard product $M \odot N$ (pointwise multiplication). Notationally, we might write this definition as

$$\langle M, N \rangle_F = \sum_{i,j} M_{i,j} \cdot N_{i,j}.$$

The result of a convolution, such as applying kernel K to channel input R, is naturally represented as applications of the Frobenius inner product (Eq. 6), with any additive bias implied by the neural net parameters applied to the result.

$$R * K = \begin{bmatrix} \langle R, K_{0,0} \rangle_F & \langle R, K_{0,1} \rangle_F & \langle R, K_{0,2} \rangle_F \\ \langle R, K_{1,0} \rangle_F & \langle R, K_{1,1} \rangle_F & \langle R, K_{1,2} \rangle_F \\ \langle R, K_{2,0} \rangle_F & \langle R, K_{2,1} \rangle_F & \langle R, K_{2,2} \rangle_F \end{bmatrix}$$
$$= \begin{bmatrix} (r_1k_1 + r_2k_2 + r_5k_3 + r_6k_4) & (r_2k_1 + r_3k_2 + r_6k_3 + r_7k_4) & (r_3k_1 + r_4k_2 + r_7k_3 + r_8k_4) \\ (r_5k_1 + r_6k_2 + r_9k_3 + r_{10}k_4) & (r_6k_1 + r_7k_2 + r_{10}k_3 + r_{11}k_4) & (r_7k_1 + r_8k_2 + r_{11}k_3 + r_{12}k_4) \\ (r_9k_1 + r_{10}k_2 + r_{13}k_3 + r_{14}k_4) & (r_{10}k_1 + r_{11}k_2 + r_{14}k_3 + r_{15}k_4) & (r_{11}k_1 + r_{12}k_2 + r_{15}k_3 + r_{16}k_4) \end{bmatrix} \tag{6}$$

It is mathematically natural to represent convolutions as aggregates of inner products. In a more general mathematical setting over a domain D we can express a convolution as

$$[f * g](x) = \int_D f(z)g(z - x)dz.$$

In our setting, we would interpret f as the image and g as the kernel. This becomes a sum in a discrete domain, so in a two-dimensional discrete domain we might write

$$[f * g](x, y) = \sum_{(i,j) \in D} f(i, j)g(i - x, j - y).$$

In a finite discrete domain such as ours, where matrix notation is appropriate, we can use the compact form

$$[R * K]_{i,j} = \langle R, K_{i,j} \rangle_F \tag{7}$$

to express the same computations as a function over the indices.

This accounts for the level of detail we have modeled, with the top layer exposing both the channel convolution operation $C * K$ – where C is a channel and K is the kernel being applied – and the image convolution operation

$$I \cdot \overline{K} = (C_1 * K_1, \ldots, C_n * K_n)$$

where C_i are an arbitrary number of channels in the input I, and K_i are the kernels to be applied to the corresponding channels. These operations are mathematically clear and do not change with training, giving us a clean, high-level surface through which to interface the formal model with iterative production results (i.e., the kernels).

The inner product representation is neither novel nor surprising. The Frobenius inner product here is performing the same function as the dot product (which itself is an inner product) does in a neural network. We can see this correspondence more clearly when we represent these same operations as MLPs. This requires us to flatten out the two-dimensional structures via vectorization, such as $\vec{R} = (r_1, r_2, \ldots, r_{16})$ and $\vec{K_2} = (0, k_1, k_2, 0, 0, k_3, k_4, 0, 0, 0, 0, 0, 0, 0, 0, 0)$. Combining the vectors representing kernel embeddings into a single matrix, we obtain:

$$W = \begin{bmatrix} k_1 & k_2 & 0 & 0 & k_3 & k_4 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & k_1 & k_2 & 0 & 0 & k_3 & k_4 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & k_1 & k_2 & 0 & 0 & k_3 & k_4 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & k_1 & k_2 & 0 & 0 & k_3 & k_4 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & k_1 & k_2 & 0 & 0 & k_3 & k_4 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & k_1 & k_2 & 0 & 0 & k_3 & k_4 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & k_1 & k_2 & 0 & 0 & k_3 & k_4 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & k_1 & k_2 & 0 & 0 & k_3 & k_4 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & k_1 & k_2 & 0 & 0 & k_3 & k_4 \end{bmatrix} \tag{8}$$

Using this consolidated matrix of kernel weights, the corresponding neural network could then be expressed by $W \cdot \vec{R} + B$, where the bias B is not present in our current model but may be treated as an additive filter applied post-convolution. We note that, in this representation, there are 144 separate elements in the 9×16 matrix W, but only five distinct values – k_1, k_2, k_3, k_4, and 0. In fact, unless the kernel is large relative to the target channel, the most common element in the convolution matrix would be 0, making this a sparse matrix with highly repetitive diagonal entries (see [22] for further discussion and examples).

Yet, this accounts for a convolution between the filter and only one channel. For multiple channels we have to consider performing this in parallel, which gives rise to even more sparseness. Consider two, possibly indistinct, convolutional operations $W_1 \cdot \vec{x}$ and $W_2 \cdot \vec{y}$. To embed these within a fully-connected feed-forward neural network for concurrent computation, we would concatenate $\vec{x}^\frown\vec{y}$ and combine the convolution weights in the manner of the block matrix W' (Eq. 9), which more than doubles the number of trivial weights.

$$W' = \begin{bmatrix} W_1 & 0 \\ 0 & W_2 \end{bmatrix} \tag{9}$$

If we represent the original (unvectorized) channels as X, Y, and the kernels represented by the matrices as K_1, K_2, we have the correspondence below:

$$\begin{bmatrix} W_1 & 0 \\ 0 & W_2 \end{bmatrix} \cdot \vec{x}^\frown\vec{y} \simeq (X * K_1, Y * K_2) \tag{10}$$

We can see that this is not a technical distinction as much as it is a conceptual one, but conceptual simplification is often important for reasoning about complex systems. Two immediate benefits are:

- The Frobenius inner product retains the two-dimensional structure of the input, suppressing the need to vectorize in order to use the traditional dot product and de-vectorize to recover expected dimensionality; and
- The process of embedding the kernel within sparse framing matrices is an internal detail which can be abstracted away, allowing proof reasoning to easily bypass the abundance of trivial weights (the same simplifying insight which makes convolutional neural networks more practical for image processing than MLPs).

MLPs are very well understood, more so when we are able to represent or sufficiently approximate the activation function with a piece-wise linear function and the problem can be posed as one of classification [19]. Embedding convolutions into this kind of network does present the opportunity to analyze them with existing tools; however, such a low level of analysis makes establishing high level properties more complicated.

Our choices in formalization may be seen as a convenient way to simplify low-level details, but it is not arbitrary. As demonstrated in the justification of Equation 7, the Frobenius inner product is a natural way to mathematically express the fundamental core of convolutions, even generalizing into non-Euclidean spaces [2].

These mathematical concepts will serve as background for the formalization which follows. The next section will describe the methodology developed to realize these concepts using theorem-proving tools.

3 Formalizing Convolutional Operations

This portion of the report discusses the tool selections, design decisions, and other implementation details relevant to the formalization of convolutions and the verification of neural networks.

The execution effort of Aerial **AI**d (and thus the assurance team's work) was well underway before YOLOv8 was identified as the engineering target. Even so, details such as the intention to use a neural net solution, and even the more specific expectation of a convolution-based network, were known earlier. Other concerns, such as how the model would interact with broader systems, remained unknown, so the specification of assurance properties was deferred until the project reached a more mature stage. The formal verification strategy needed to be tailored to capitalize on the available information and work around unknowns to increase research utility.

To maintain flexibility, high-level abstractions based on intended function were used when details were not available, and tight development loops incorporating formal models and executable models were used. The resulting process allowed proof complexity to guide code clarity, and failed tests to refine the models – all while maintaining a close syntactic correspondence between formal and executable models to allow easy inspection.

This is reminiscent of existing methodologies, such as

- the use of PVSio [12] for rapid prototyping within PVS;
- the MINERVA software development process [13], which utilizes differential testing between an implementation and the symbolic execution of the formal solution; and even
- the practice of digital twinning [9], which uses digital representations of real processes in real-time.

However, we are unaware of this particular combination of tools and approach being reported previously, and so will expound on it here.

3.1 Tools

The primary tools used in formalizing convolutions were the Prototype Verification System (PVS) [17] and the multi-paradigm programming language Racket [6]. PVS is an industrial proof assistant developed by the Computer Science Laboratory at SRI International, and is an important tool for the work of the Langley Formal Methods Group [20]. PVS has a long research lineage with its immediate predecessor being the Enhanced Hierarchical Development Methodology (EHDM), from which it inherits its specification language [15]. The maturity of the platform and NASA's long association with its use and development made it the best choice as a verification tool. The specification language allows for clear and readable descriptions while its rich dependent typing allows for complex properties about an artifact to be carried by its type. This form of inline encoding can make advanced proofs more manageable. PVS was originally built to explore ideas in interactive theorem proving with respect to the EHDM specification language because the expressive type system is undecidable algorithmically.

To address the concurrent engineering and assurance work, we considered creating tools to more closely integrate the teams. Repetitious work in verification is often generalized and abstracted into proof strategies (also known as "tactics" in some proof assistants). These scripted automation tools target common structures across similar arguments and modularize proof construction. Externalizing this approach through custom tools would allow the engineering team to provide up-to-date formal artifacts for assurance purposes. Complete automation would likely not be feasible, but even something as fundamental as extracting weights and constructing a formal model would benefit the process. This approach would require processing models from complex, commercial neural networks such as YOLO and constructing PVS representations.

We also considered a complimentary approach involving the development of formally verified components for object detection tasks. Parameterized components could be optimized and verified in a general way, and then composed into larger frameworks for machine vision tasks. A particular framework and its parameters could be developed traditionally, or the framework could mimic the layout of an existing solution like YOLO and be instantiated by parameters extracted from a trained model. This approach is similar to the one above in that it front-loads as much verification as possible and defers only that which is most subject to change – the parameters resulting from training (e.g., kernels).

The intersection of these two approaches lies in creating modular formal models. Since convolutions are central to contemporary object detection tasks, it was determined that formal convolutions would be the foundation for our future verification efforts. Although PVS supports symbolic execution of a subset of its specification language through PVSio [12], this is not an easily deployable solution. A general purpose programming language was desired in which to mirror the formal PVS model. There are a number of reasons Racket was chosen to serve this function:

- Typed Racket can represent much of the type information contained in the PVS specification;

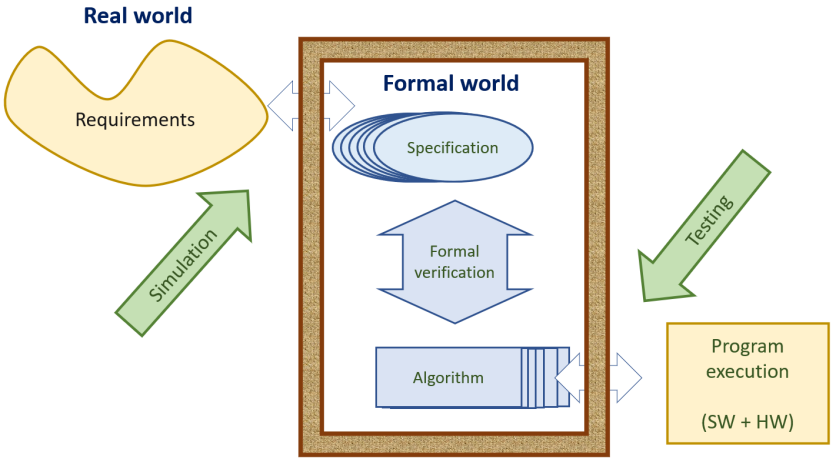


Figure 4. Illustration of well-known semantic gaps in formal verification (requirement/specification and algorithm/execution).

- PVS specification is primarily functional, and PVS itself is built on top of Common Lisp, so a modern Lisp dialect such as Racket made sense for maintaining correspondence between the two encodings;
- the language-oriented programming philosophy behind Racket seems to support the desire to eventually integrate this work into a domain specific tool for convolutional neural network assurance; and
- there was existing time investment in learning to use the language within this particular domain due to team composition and exploratory work on interpreting PVS specification within Racket.

In practice, the level of code correspondence obtained facilitated rapid iterative development, as noted in Section 3.2 below.

3.2 Typed Racket/PVS Correspondence

We will use the diagram in Figure 4 as an anchor for this discussion. The diagram illustrates the relation of formal verification to real world systems. We pay particular attention to the two notable semantic gaps that occur (1) in the interpretation of system requirements as formal specifications, and (2) the realization of an algorithm as a computational process. These gaps are unavoidable because transitioning between the formal world and the real world involves abstraction, which can never be guaranteed to be complete and correct.

In the beginning of Aerial **AI**d, detailed requirements had not been developed. Initial PVS specification was based on descriptions of how convolutional neural networks were expected to behave. We had no engineering artifact to validate the specification against, so it was helpful to create an executable model in Typed Racket. By writing the Racket model in such a way that it closely corresponded

Typed Racket	**PVS Specification**

(: monus (→ Natural Natural Natural))
(define (monus n m)
  (let ([temp (- n m)])
  (if (Natural? temp)
  temp
  0)))
	monus(n,m:nat): nat =
	  LET temp = n - m IN
	  IF temp ⩾ 0
	  THEN temp
	  ELSE 0 ENDIF

Figure 5. A basic example of direct correspondence between Typed Racket and the PVS specification language.

with the PVS specification, we were able to build confidence in the specification despite the unknowns.

Testing the executable model allowed us to refine the formal specification. In turn, proving properties about the specification identified issues in the executable model. The had the effect of creating a closed loop where the behavior of the executable model served as a stand-in for the real system:

$$\text{Concrete System} \underset{\textit{prescriptive}}{\overset{\textit{descriptive}}{\rightleftharpoons}} \text{Formal System}$$

We note that the actual Racket code resides in the formal world, along with the PVS specification. The computational behavior of the code is the real-world observable.

PVS's higher-order classical logic provides a natural language to express mathematics. The ability to avoid cumbersome encodings helps us manage the descriptive gap. Since Racket is a relatively mainstream language, we expect errors in its semantics to be uncovered by its user community. This gives us confidence in how the prescriptive gap is managed. By closing the loop, we compose these two gaps – PVS describes what Racket prescribes.

This creates a situation where the assumptions made for each gap cross-check. Discrepancies may arise in failed tests in the executable model, or unprovable properties in the specification. While this can not eliminate gaps, it does facilitate the discovery of many classes of errors and brings the two models into semantic correspondence. Remaining errors are more likely to arise from fundamental misunderstandings of the real system which exist in both models. In the absence of a stable real-world model, this helps mitigate drift in the formal specification and accelerates development.

To complement semantic correspondence we also strive for syntactic correspondence. In Figure 5, we see a simple example of syntactic correspondence between Typed Racket and PVS specification illustrated by definitions of the monus function (often encountered in foundational recursion theory and denoted with the $\dot{-}$ symbol). Since PVS specification is typically written functionally, its control structures have a high degree of similarity with Racket.

One notable semantic difference is the many-case conditional, denoted 'cond' in Racket and 'COND' in PVS. The PVS version must have provably disjoint conditions

Typed Racket

```
(struct (A) finite-array
  ([width : Natural]
   [height : Natural]
   [valuation : (→ Natural Natural A)]
   [default-value : A])
  #:transparent



  #:guard (λ (wid hgt val def name)
            (values wid
                    hgt
                    (restrict-array wid hgt val def)
                    def)))

(: restrict-array
  (All (A) (→ Natural Natural
               (→ Natural Natural A) A
               (→ Natural Natural A))))
(define (restrict-array wid hgt arry default-value)
  (λ (x y) (if (and (< x hgt) (< y wid))
               (arry x y)
               default-value)))
```

PVS Specification

```
finite_array: TYPE =
[# width : nat,
   height : nat,
   default_value : A,
   valuation :
     {x:[[nat,nat]→A] |
       ∀(n,m:nat):
         (n<height ∧ m<width) ∨ x(n,m) = default_value}
#]
construct_finite_array(wid,hgt:nat,val:[[nat,nat]→A],def:A):
  finite_array =
(# width := wid, height := hgt, default_value := def,
    valuation := restrict_array(wid,hgt,val,def)
#)

restrict_array(wid,hgt:nat,arry:[[nat,nat]→A],default_value:A):
{x:[[nat,nat]→A] |
  ∀(i,j:nat):
    ((i<hgt ∧ j<wid) ⟹ x(i,j) = arry(i,j)) ∧
    ((i⩾hgt ∨ j⩾wid) ⟹ x(i,j) = default_value)} =
  λ(x,y:nat): IF (x < hgt ∧ y < wid)
              THEN arry(x,y)
              ELSE default_value ENDIF
```

Figure 6. Comparison of finite array definitions.

while no such requirement is (or even can be) placed on the Racket version. This works in our favor as it allows the formal specification to guarantee a stronger constraint than what will actually be required at runtime.

The monus example is straightforward because it only uses the natural numbers, which is a fundamental type in both Racket and PVS. Things become more complicated because PVS and Racket utilize different type systems. For example, Typed Racket supports the supertype 'Any', but has only limited, experimental support for subtypes [23]; on the other hand, PVS specifications often lean heavily on predicate subtyping and abstract datatypes, but in general two arbitrary types will not share a common supertype [16, 17].

Despite these differences, we can still leverage language features to create analogous structures. The definitions of the finite arrays used for convolutions provides a good example. In Figure 6, we see an example of a PVS record type with internal dependent subtyping used to define finite arrays in PVS. This is very similar to defining a structure in Typed Racket (which conveniently creates a type as well). Additionally, by using guard functions with the structure we can create a version of the PVS subtyping.

In this example the guard function actively intercepts and modifies data provided at construction. In the PVS specification shown in Figure 6, we restrict the function encoding the values of the array in such a way that there are only a finite number of index pairs which have non-default values. A guard function in the Racket structure takes an arbitrary valuation function and wraps it in a lambda term that prevents any non-default value from appearing outside of the provided dimensions. To reflect this, we introduce a PVS function modeling the finite array constructor in Racket with this guard. This is neither perfectly equivalent nor is it possible in general to represent subtypes with guard functions. It is a bespoke correspondence tailored for this construction.

In practice, more than one iteration through the formal-concrete development cycle was used to refine the result seen in Figure 6. As an illustrating example,

we might start by sketching out the structure roughly in Typed Racket without a guard. Moving to PVS, we realize we can tighten up the specification by using predicate subtyping on the valuation, thereby requiring us to place the valuation last in the record since it depends on the default value. Returning to Racket, we then implement the helper function to restrict the array and the guard function. Back in PVS, we define the array restriction function and a function which models the implicit Racket finite array constructor.

This circular process encourages us to find compromises between the languages and can yield identifiable code correspondence, with clear connections between how *what we want* relates to the specification language (the interpretation), and between how the implementation relates to *what is being done* (the computation). It should be noted that, while this iterative process is highly effective, the example in Figure 6 does not reflect a finished product because there are still things we could do to bring the two into closer correspondence. Possible improvements include: reordering the components of the Racket structure to match the order in the PVS type; and using PVS's local type polymorphism to reflect polymorphic typing (ex. 'All (A)') seen in the Racket code.

The example in Figure 5 was chosen for its simplicity, and that in Figure 6 was chosen for its diversity and how it illustrates iterative development. Similar examples of varying complexity and character occurred throughout the formalization process, but we see no need to continue at length at the code level. In the next section we instead move into a high level summary of how the formalism is structured.

4 Results

The central components of the Typed Racket/PVS development are separated into *Core*, *Finite Array*, and *Convolutions*. Each consists of a PVS theory and the corresponding Racket module. For *Convolutions*, we define the Frobenius inner product of two matrices $\langle M, N \rangle_F$ as the sum of the elements of the Hadamard product $M \odot N$ – that is,

$$\langle M, N \rangle_F = \sum_{i,j} M_{i,j} \cdot N_{i,j}.$$

Here one matrix represents an image channel, and the other represents a kernel embedded at a specific window location. The operation results in a single pixel of the channel's output.

The remainder of the material in *Convolutions* involves defining how to:

- Perform a convolution, given an image channel, a kernel, and offsets for the kernel's starting position, by embedding the image channel in a larger buffered matrix (if the offsets require it), generating the matrices corresponding to each application of the kernel at a particular window, taking the Frobenius inner product for each, and combining the pixels into a channel output (see Figure 7); and
- Apply convolutions to all channels given a list of image channels and a list of kernels to be applied, respectively.

```
perform_convolution
  (input_array,kernel_array:finite_array,
   row_offset,col_offset:{i:int | i<=0}): finite_array =
LET host_roff = abs(row_offset),
    host_coff = abs(col_offset),
    new_wid = input_array`width - 2*col_offset,
    new_hgt = input_array`height - 2*row_offset,
    crop_wid = monus[real](new_wid,kernel_array`width),
    crop_hgt = monus[real](new_hgt,kernel_array`height),
    padded_host =
      pad_array(input_array,host_coff,host_roff,new_hgt,new_wid) IN
list2finarr(map((LAMBDA (x:finite_array):
                  frobenius_inner_product(x,padded_host)),
                generate_embeddings(padded_host,kernel_array)),
            crop_wid+1,
            crop_hgt+1,
            0)
```

Figure 7. This is the raw PVS specification of a convolution. Note that much of this is concerned with padding the input array, generating the kernel embeddings, and transforming the list of Frobenius results into a new array. As noted elsewhere, much of this can be eliminated by bringing the specification more in line with the functional form of Equation 7 – an example of the ongoing improvements and refinements characterizing the iterative development process.

	TCCs	Lemmas	Total Proofs
Convolutions	21	1	22
Finite Array	31	5	36
Core	16	3	19
Dictionary	18	10	28
Finite Sequence	11	4	15
Neuron	2	0	2
Totals	99	23	122

Table 1. The PVS development contains a total of 122 proofs. The current formalization of finite arrays is only dependent on *Convolutions*, *Finite Array*, and *Core*. The stronger the guarantees encoded into the formalization's types are, the more type-correctness conditions (TCCs) are generated. Additional properties can then be stated as Lemmas.

Test cases were performed in both PVS and in Racket to verify agreement. This involved running a test in Racket, then proving a theorem in PVS which stated the Racket result was correct. The PVS formalization of *Convolutions* includes 21 proven type-correctness conditions (TCCs) and a proof indicating that the test convolution gave the correct output (see Table 1 for comprehensive counts).

In *Finite Array* we define the finite array structure polymorphically as a value-carrying function on two indices, a default value, and a pair of index bounds. The definitions are made in such a way that any indexing out of bounds must return the default value. Additionally convenience operations have been defined such as:

- Direct indexing of finite arrays;
- Mapping functions across the elements of a finite array;
- Constructing finite arrays with a list of elements, bounds, and a default value; and
- Converting between lists of lists and finite arrays.

Some special operations involving finite arrays which are required for convolutions are also defined here. For example, generating the index pairs for each kernel window and embedding a kernel into a larger matrix are both described in *Finite Array*.

We note that for future work the formalization may be simplified by leveraging the internal functional representation of the array values to implement convolutions more closely to the form shown in Equation 7. This would abstract away many of these low-level details related to generating kernel embeddings.

Some of the other definitions in *Finite Array* involve Racket functions `max`, `range`, `quotient`, and `modulo`. There were a few options for how to handle these in the PVS formalization. One approach is to import versions of these from the PVS libraries. This is opaque and carries the implicit promise that the PVS library version faithfully replicates the Racket version, but slight changes in definition can yield incompatible type signatures.

Safer alternatives would include either modeling the underlying behavior of the Racket function and types in PVS, or defining a PVS function in terms of the presumed behavior of the Racket version. This illustrates the semantic embedding [1] choices that underlie much of this process. The former approaches a *deep embedding*, while the latter is a more *shallow embedding*.

The shallower embedding is chosen here because it is transparent in the assumptions it is making on Racket's behavior – indeed, inline comments in the specification indicate that these are presumed behavior. In general, this entire process is guided by making *in situ* mappings between the two languages as shallow as possible in order to aid human review of the correspondence.

Global mappings between PVS and Racket have been avoided because the disparate type systems do not seem to allow (and probably preclude) a general shallow embedding. A deep embedding is almost certainly possible, but the overhead added by the embedding would discourage direct human review. To reach a similar level of confidence would require a resource-intensive program to formally verify an automated deep embedding. The current clarity of the syntactic correspondence contra-indicates such an intensive effort at this time.

In part due to the expression of semantic assumptions, the formal version of *Finite Array* contains a total of 36 PVS proofs. Five of these are for stated theorems of behavior and the remaining are to satisfy TCCs.

The final components central to the development are contained within *Core*. This consists of a collection of low-level operations, the most notable of which is the method of array restriction which can accept valuation functions of arbitrary content and encapsulate them in a lambda form that will return the default value if out-of-bound indices are provided.

There are three proven properties and 16 satisfied TCCs, but, because *Core* contains definitions used by components which are not central to the development of convolutions, it would be misleading to attribute all of these to the formalization of convolutions. These non-central components include

- *Dictionary*, an early alternative for representing sequences and arrays by creating a common, type-safe dictionary data structure for Racket and PVS;
- *Finite Sequence*, the one-dimensional version of *Finite Arrays*; and
- *Neuron*, prior to deciding on the Frobenius product as the fundamental level of abstraction, we modeled neurons with *Finite Sequence*.

These developments are potentially useful formalizations on their own or as part of a larger neural net library.

5 Future Work

This is part of the closeout for Aerial **AI**d and represents the state of research when the project went into orderly shutdown. It does not represent a natural or planned stopping point for the work. The progress reported here largely falls under two headings:

- the ongoing formalization of convolutions in a proof assistant, and
- the methods developed to adapt and integrate our formal methods workflow with an agile development process.

From a formal methods perspective the project is incomplete. The formalization is provably consistent but no high-level properties of convolutions are proven. Indeed, it's not the properties of an arbitrary convolution we were interested in. The next step would have been to extract a real kernel from production, define a convolution operator using that kernel, and prove properties related to that operator. The strength of this approach is that, rather than attempting to verify a large, monolithic convolutional neural net, we can instead focus on verifying the most important parts. Kernels present a much more tractable verification target with a common size being 3×3 [18].

In this report we have documented the mathematics behind the formalization to support future work. Such work would include the simplification of the existing work to abstract away mechanical details which may differ between implementations, the formalization of other network components for composition with convolutional operations (e.g., downsampling, normalization), performance benchmarking, and the selection of relevant safety properties.

The need to integrate assurance efforts into what was essentially a product development cycle presented challenges. It was not immediately clear what technical solutions would be chosen, what the precise operational domain would be, or what safety requirements would be required. This was difficult enough for runtime assurance methods, but for formal verification it made our normal approaches almost impossible. By beginning at a high abstraction level and checking against an executable model, we could focus on the expected mathematical behavior of general solutions and quickly refine as new information came in.

Specification, refinement, and basic verification were greatly sped up by the self-imposed drive for semantic and syntactic correspondence between the specification and executable model. This created an almost adversarial relationship between PVS and Typed Racket that surfaced errors early, before a lot of time could be invested in a dead end. Partially because of this, the process did not noticeably slow down progress.

This provides an example of how formal verification can be integrated into an agile development process without imposing prohibitive overhead. With the close-out of Aerial **AI**d there remains work to be done. Future work would include further integration with the engineering process and the development of tools and documentation to refine the approach.

These early results establish a foundation for future verification efforts. Despite the short window for Aerial **AI**d research, the efforts have produced promising lines of research which may be built upon. The logical extension of this work is a full neural network library in PVS that allows formal specifications of neural networks to be written and formally reasoned about, and corresponding components in Typed Racket which implement the assured behavior. Additionally, the obstacles for translating PVS into Typed Racket do not apply to untyped Racket. An automated translation of verified specifications into Racket would provide an alternate

avenue of code generation which could support the use of PVS as a platform for quickly developing and deploying assurance tools.

References

1. Richard J Boulton, Andrew D Gordon, Michael JC Gordon, John Harrison, John Herbert, and John Van Tassel. Experience with embedding hardware description languages in hol. In *TPCD*, volume 10, pages 129–156, 1992.

2. Michael M Bronstein, Joan Bruna, Yann LeCun, Arthur Szlam, and Pierre Vandergheynst. Geometric deep learning: going beyond euclidean data. *IEEE Signal Processing Magazine*, 34(4):18–42, 2017.

3. Jill Brown. CERTAIN: City environment range testing for autonomous integrated navigation. Technical report, NASA Langley Research Center, 2016.

4. John Denker, W Gardner, Hans Graf, Donnie Henderson, R Howard, W Hubbard, Lawrence D Jackel, Henry Baird, and Isabelle Guyon. Neural network recognizer for hand-written zip code digits. *Advances in neural information processing systems*, 1, 1988.

5. Miral Desai, Hiren Mewada, Ivan Miguel Pires, and Sparsh Roy. Evaluating the performance of the yolo object detection framework on coco dataset and real-world scenarios. *Procedia Computer Science*, 251:157–163, 2024.

6. Matthew Flatt et al. The Racket reference. `https://docs.racket-lang.org/reference`, 2014. Accessed: 2025-08-05.

7. Dr. Steve King, Shay Major, and Marlis McCollum. Drone as first responder programs: A new paradigm in policing, August 2023.

8. Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25, 2012.

9. Andreas Löcklin, Manuel Müller, Tobias Jung, Nasser Jazdi, Dustin White, and Michael Weyrich. Digital twin for verification and validation of industrial automation systems–a survey. In *2020 25th IEEE international conference on emerging technologies and factory automation (ETFA)*, volume 1, pages 851–858. IEEE, 2020.

10. Jonathan Long, Evan Shelhamer, and Trevor Darrell. Fully convolutional networks for semantic segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 3431–3440, 2015.

11. MITRE Corporation. First responder drone initiatives, April 2023.

12. César A Muñoz. Rapid prototyping in PVS. Technical report, NASA/CR-2003-212418, 2003.

13. Anthony Narkawicz, César A Muñoz, and Aaron M Dutle. The MINERVA software development process. In *NASA Formal Methods Symposium (NFM) 2017*, number NF1676L-26800, 2017.

14. NASA. Convergent aeronautics solutions (CAS). `https://www.nasa.gov/directorates/armd/tacp/cas/`, 2024.

15. Sam Owre, John Rushby, Natarajan Shankar, and Friedrich Von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE transactions on software engineering*, 21(2):107–125, 2002.

16. Sam Owre and Natarajan Shankar. The formal semantics of PVS. Technical report, NASA/CR-1999-209321, 1999.

17. Sam Owre, Natarajan Shankar, John M Rushby, and David WJ Stringer-Calvert. PVS language reference. *Computer Science Laboratory, SRI International, Menlo Park, CA*, 1(2):21, 1999.

18. Joseph Redmon and Ali Farhadi. Yolo9000: better, faster, stronger. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 7263–7271, 2017.

19. John V Siratt. *Some Applications of Formal Mathematics*. PhD thesis, University of Notre Dame, 2024.

20. John V Siratt. Interactive theorem proving and applications at NASA. `https://ntrs.nasa.gov/api/citations/20250001586/downloads/siratt\_presentation.pdf`, 2025.

21. Irwin Sobel. History and definition of the Sobel operator. *Retrieved from the World Wide Web*, 1505, 2014.

22. Gilbert Strang et al. *Linear algebra and learning from data*, volume 4. Wellesley-Cambridge Press Cambridge, 2019.

23. Sam Tobin-Hochstadt, Vincent St-Amour, Eric Dobson, and Asuma Takikawa. The Typed Racket reference. `https://docs.racket-lang.org/ts-reference`, 2014. Accessed: 2025-06-16.