Variability modeling language and tool to represent, configure and evaluate Convolutional Neural Network architectures

Lenguaje y herramienta de modelado de la variabilidad para representar, configurar y evaluar arquitecturas de redes neuronales convolucionales.

JULIAN ALEXANDER MURILLO PORTOCARRERO

Advisor

Raul Mazo Peña PhD.

REQUISITO PARA OPTAR AL TÍTULO DE MAGÍSTER EN INGENIERÍA

UNIVERSIDAD EAFIT
ESCUELA DE CIENCIAS APLICADAS E INGENIERÍA
MAESTRÍA EN INGENIERÍA
MEDELLIN
2024

Abstract

The process of designing Convolutional Neural Network (CNN) architectures currently relies on manual design or the involvement of experts. This process is not only time-consuming but also expensive due to the sheer number of combinations that architects need to do to arrive at the right network hyper-parameters that fit the current problem. In this work, we analyze the current state-of-the-art, in which several approaches have been proposed to automate such processes and explore alternatives for the automated design of Convolutional Neural Networks and Neural Architecture Search (NAS). Additionally, this work proposes a method for hyper-parameter variability generation from a variability model of such convolutional neural networks. The variability model proposed in this master thesis is used to represent, intensively, the valid combinations of parameters corresponding to each convolutional neural network. The language, called CNN variability language, borrows some concepts from Software Product Lines (SPL) and was created on the VariaMos platform to enable architects and engineers not just to create CNN architectures but also to automatically generate configurations, generate executable Jupyter Notebooks for each configuration, and generate comparison reports to speed up the NAS process.

El proceso de diseño de arquitecturas de Redes Neuronales Convolucionales (CNN) actualmente depende del diseño manual o de la participación de personal experto. Este proceso no solo toma tiempo, sino que también es costoso debido a la gran cantidad de experimentos que los arquitectos deben realizar para poder llegar a los hiper-parámetros adecuados de la red neuronal que se ajusten al problema a tratar. En este trabajo se discute el estado del arte de varios enfoques que han sido propuestos para automatizar dichos procesos arquitectónicos, explorando alternativas para el diseño automatizado de Redes Neuronales Convolucionales y la Búsqueda de Arquitectura Neural (NAS). Además, este trabajo propone un método para la generación de variabilidad de hiper-parámetros que permite el diseño automático de dichas redes neuronales convolucionales mediante el uso del lenguaje “CNN variability language”, el cual se basa en los conceptos de las líneas de productos de software orientadas a características (FOSPL). El lenguaje fue creado en VariaMos, una herramienta que no solo nos permite usar los lenguajes creados en ella, sino que también nos permite resolver automáticamente la variabilidad de cada modelo gracias al uso de solvers de restricciones. Además, la herramienta nos permite generar

Jupyter Notebooks ejecutables por cada configuración encontrada, y con cada una de ellas generar informes comparativos que permitan acelerar el proceso de NAS.

CONTENTS

CHAPTER 1. INTRODUCTION

The recent progress in the field of Machine Learning (ML) has opened the door to one of the greatest technological advancements of the last decade, helping to solve one of the most difficult problems in the domains of computer vision, speech recognition, and natural language processing in the first half of the decade [1], and for technologies like generative models create realistic text, voice, sound, image and video for the second half. These advancements have been the product of Deep Neural Networks (DNN), massive amounts of data, and constant training.

The concept of a deep neural network (DNN) is defined as a type of artificial neural network that contains a high number of layers between their input and output layers [2]. Such networks are mainly designed to solve complex tasks such as natural language processing, image processing, and computer vision by transforming the input data through multiple layers of neurons [3].

A DNN architecture is characterized by containing multiple layers to perform specific transformation operations on data [4], besides hierarchical representations on each layer that can help the DNN capture more abstract and complex features and extract more sophisticated patterns and relationships [5].

Some neural network architectures that are very well known in the machine learning field can fall into the following categories: Convolutional Neural Networks [6], Recurrent Neural Networks, and Generative Models [5].

In the next chapters, the reader will find an introduction to the topic in Chapter 1; a revision of the problem statement, hypothesis, and objectives in Chapter 2; a literature review and state-of-the-art in Chapter 3. In Chapter 4, we present the research methodology used to create the method proposed in this master thesis; the design and implementation of the CNN variability language and how to use it are presented in Chapter 5; some proofs of concept in which we show how to model the variability of neural networks and how to configure the resulting models are presented in Chapter 6; the execution and evaluation of the configurations of the CNN variability models are presented in Chapter 7. Finally, the conclusions and future work are presented in Chapter 8.

CHAPTER 2. PROBLEM STATEMENT

One of the challenges that DNN architectural design currently faces is the manual work involved in choosing the right architecture since this labor includes several considerations, like problem complexity, availability of data, and computational resources. Facing these challenges entails experimentation with different architectures. Such kind of experimentation is iterative, which is done by testing different architectural configurations, hyperparameters, and architectural choices by hand to find the optimal balance between model performance and computational resources [7][8].

The area of software product lines (SPL) allows for managing the variability and commonality between different products within a family of related ones by capturing commonalities and differences between them in a set of core assets and variability models [9]. Constraint programming solvers can generate such variations to accelerate the search process [10].

2.1. HYPOTHESIS

One of the challenges of DNN architectures is finding the right architecture parameters considering complexity, data availability, and resources, which can take time given their iterative nature. Considering that random search has been proven to be more efficient than manual search at hyper-parameter optimization [7], and also considering that SPLs can generate random variations by capturing commonalities and differences between products, SPLs can be an appropriate fit to solve the problem of manual iteration for DNN hyper-parameters.

2.2. OBJECTIVES

The proposal, specifically, is to provide a way to automate the iterative processes of CNN architecture experimentation by using SPLs as a form of neural architecture search (NAS) [11] to generate random variations of the different CNN hyper-parameter configurations, which can be tested to provide immediate results for the CNN architect, who can export, experiment on, and continuously refine them, to help the architect find the right architecture hyper-parameter configuration for the problem at hand, reducing the iteration time of the former manual process.

- Provide a state-of-the-art overview of the current progress on the automatic design of CNN architectures and hyper-parameter search.
- Design a variability language for CNN architectures that can be used to do random DNN hyper-parameter generation.
- Design and implement a solution to generate and execute CNN architectural code to provide metrics based on the CNN architectures generated to help the hyper-parameter search.

2.3. SCOPE AND LIMITATIONS

The scope of this work is to design an engineering language to represent the variability of CNN architectures in such a way, that different variations of the architecture can be obtained by configuring such variability models. Thus, the hyper-parameter variations of the deep neural network architectures are encoded in each variability model following an SPL approach. Each configuration corresponds to a CNN architecture that can be executed and validated. This work will not provide an architectural layer or in-depth neuron connection variability for the CNN architectures, and it will only be limited to Convolutional Neural Networks (CNN) due to time, complexity, and resource limitations. Topics like Natural Language Processing (NPL), Generative Adversarial Networks, and other types of architectures can be part of the scope of future works focused on the use of the areas of DNN automatic architecture generation and SPLs.

CHAPTER 3. LITERATURE REVIEW

In automatic Convolutional Neural Network design and Neural Architecture Search (NAS), different research approaches have been proposed to solve the problem of manual neural network architecture design. This research has been done using several mathematical approaches, statistical methods, genetic algorithms, and even the use of neural networks themselves to find the most efficient and cost-effective CNN architectures.

3.1. STATE OF THE ART

The paper “Automatic Design of Convolutional Neural Network for Hyperspectral Image Classification” proposes an idea for automatic CNN classification for Hyperspectral images (HSI), following the premise that manually designed architectures may not fit a specific data set. In that paper, operations like convolution, pooling, identity, and batch normalization were selected, and a gradient-descent-based search algorithm was applied to them to effectively find the optimal architecture evaluated on the validation dataset [12].

Intending to reduce the reliance on manual design or expert experience, in the paper “Automatic Design of Convolutional Neural Network Architectures Under Resource Constraints”, the researchers tried to improve the current state of the art on neural architecture search (NAS) by proposing an automatic method for designing CNN architectures under constraint handling that can search for optimal neural network models meeting the preset constraints, making use of the datasets CIFAR-10 and CIFAR-100 [11].

The paper “Automatic Design of Neural Network Structures” proposes a method for automatically designing deep neural networks based on genetic algorithms (GA) and Lidenmayer systems. According to the paper, such a method is less computationally intensive than existing iterative design procedures. To model the dynamics of a pH neutralization process, a CSTR reactor was used to evaluate the method's performance in nonlinear reactions. According to the authors, the neural network architectures obtained by the algorithm were much simpler and more accurate than those designed by traditional methods [13].

The paper "Automatic Design of Neural Network Structures Using AiS" presented a study that proposes a method to automate neural network design for a regression problem based on the Add-if-Silent (AiS) function. This study shows that by modifying the intermediate function to be a Radial Basis Function (RBF) the researchers found an optimized network structure using the Bike Sharing Dataset as a case study [14].

Combining Deep Neural Networks with variability models in the paper "Automated Search for Configurations of Deep Neural Network Architectures", researchers modeled the variability of DNN architectures using Feature Models (FM) to generalize over existing architectures. Each valid configuration of the FM corresponds to a valid DNN that is then built and trained on top of TensorFlow in an automated fashion. According to the authors, their approach is to train and evaluate the performance of configured models. The evaluation method was applied to MNST and CIFAR-10 datasets to demonstrate that with a small amount of computation and training, it can identify high-performing architectures outperforming state-of-the-art architectures handcrafted by ML researchers [15].

In the paper "Neural architecture search without training", the authors propose a method to partially predict the accuracy of a neural network from its initial state using an algorithm that can run in seconds on a single GPU, following the premise of the time-consuming nature of the hand-design of deep neural network architectures. In addition, in this paper, researchers discuss the slowness and expense of the Neural Architecture Search (NAS) algorithms since they require training vast numbers of candidate networks [16].

Model compression techniques have enabled DNN-based inference on resource-constrained edge devices like mobile phones, drones, robots, medical devices, wearables, and IoT devices. In the paper "Automating Deep Neural Network Model Selection for Edge Inference", researchers found that compressed DNN models can vary significantly, trading off accuracy, latency, and energy consumption. Since no single model is optimal for all the metrics, they propose an automated device-level DNN model selection for Quality of Experience (QoE) edge inference, leveraging machine learning and user feedback. The researchers validated the algorithm by using preliminary simulations to demonstrate the potential of machine learning for automating DNN model selection for these mobile devices [17].

In the area of Automatic Design, the paper “Automatic Design of Artificial Neural Networks for Gamma-Ray Detection” aims to improve gamma/hadron discrimination based on ground-recorded shower patterns by using CNNs due to their capacity to distinguish patterns with automatically designed features. The work used Fast-DENSER++ [19], a variant of Deep Evolutionary Network Structured Representation, to create CNNs to avoid the need for manually crafting topology and learning hyper-parameters. The results show that the best CNN generated by Fast-DENSER++ improves discrimination by a factor of 2.37 compared to classical statistical approaches. [18]

In the paper "Fast-DENSER++: Evolving fully-trained deep artificial neural networks", the authors improve the results reported in the previous paper by allowing training time for candidate solutions to increase as needed, with shorter training times in initial generations and longer cycles in later generations for better performance. This makes the new proposal different from most optimization methods, which evaluate the candidate solutions for a fixed number of epochs, resulting in fully trained DNNs ready for deployment immediately after evolution without needing further training. This approach is an extension of Deep Evolutionary Network Structured Evolution (DENSER) [20].

In the paper "DENSER: Deep Evolutionary Network Structured Representation", the authors propose a novel evolutionary approach for automatically generating DNNs that combines Genetic Algorithms (GA) with dynamic structured grammatical evolution (DSGE). The GA encodes the macrostructure, including layers, learning methods, and data augmentation techniques, and then the DSGE specifies the parameters for each GA evolutionary unit and valid parameter ranges. According to the paper, this method was tested using the CIFAR-10 dataset, achieving up to 95.22% accuracy, and with the CIFAR-100 reaching up to 78.75% classification accuracy, setting a new state-of-the-art method for CNN Automatic Design [20].

3.1. RESEARCH GAPS

The works that were reviewed, besides revealing methods for the automatic design of convolutional deep neural networks, also reveal a need to replace the manual labor and expertise needed for architecting such networks, mainly for solving narrow and specific problems efficiently. With the current manual methods struggling to reach maximum efficiency, some of

the automated approaches reviewed above have yielded promising results and some even have set a new state-of-the-art raising the bar for other novel approaches.

All these methods work very well in isolation, that is, addressing specific problems; however, they don't behave the same for other kinds of networks, like Generative Networks or newer state-of-the-art architectures. As far as the research suggests, the current state-of-the-art Automatic Design of CNNs has yet to find a general and efficient approach that allows the automatic generation of any convolutional neural network that can fit most contexts.

Several approaches, like Fast-DENSER++ which has advanced the state of the art in the areas of evolutionary neural network generation for CNNs [20], offer a positive horizon and a path forward for future research in this area by focusing on the iterative natures of genetic algorithms to find optimal and unexpected combinations. In addition, the use of transformers and generative neural networks can also provide another avenue to explore by using DNNs to generate DNNs, which can in turn become more efficient and less time-consuming than the current strategies, as these kinds of networks have disrupted other areas of research in the past.

CHAPTER 4. METHODOLOGY

To achieve the objectives stated in this document, the research methodology Design Science Research Methodology (DSRM) was used. This methodology, based on the engineering design process, consists of six iterative steps aimed at the development of research and solutions related to the area of Information Systems Engineering [24]. The six steps of this methodology are the following.

1. Identify the Problem and Motivate
2. Define the Objectives of a Solution
3. Design and Development
4. Demonstration
5. Evaluation
6. Communication

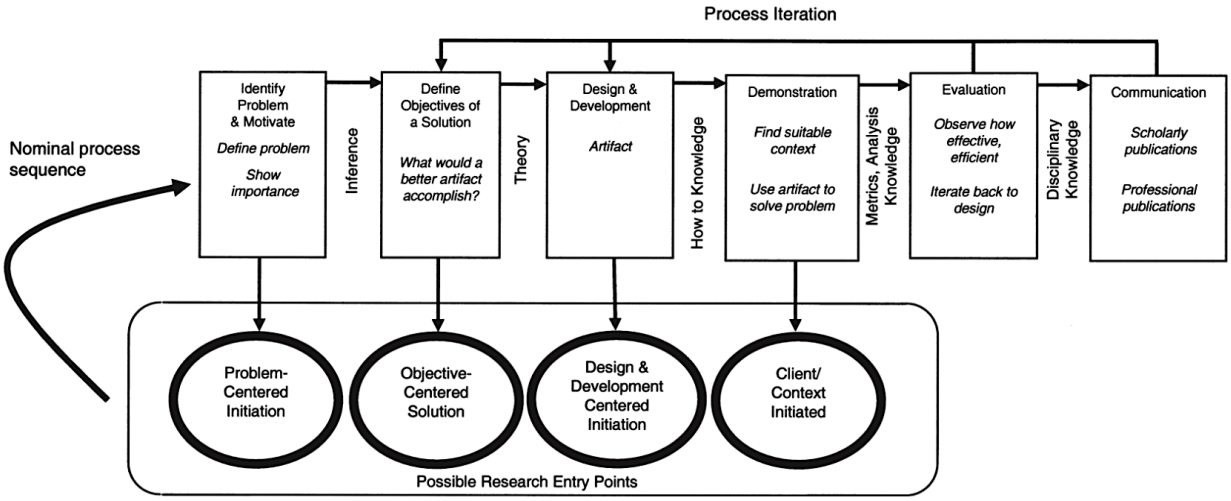


Figure 1. Design Science Research Methodology (DSRM) iterative process

This methodology will be used throughout this document to provide the backbone that can help guarantee the definition, design, development, and verification of the approach stated. Making sure the approach, besides being communicated, gets deployed, tested, and integrated into the VariaMos [10] tool, is available and functioning as expected for future CNN architects.

The methodology process can be found throughout the document, steps for "Identify Problem and Motivate" and "Define Objectives of a Solution" can be found in the introduction section of Chapter 2 and Section 2.2 of the same chapter, respectively; the "Design and Development" step can be found in Chapter 5; the "Demonstration" step can be found in Chapter 6 through a specific case of modeling a CNN; and the "Evaluation" step in Chapter 7.

CHAPTER 5. CNN VARIABILITY MODELING LANGUAGE AND HOW TO USE IT TO GET CNN CONFIGURATIONS

The **CNN variability modeling language** constitutes the primary contribution of this master thesis. The language is intended to be used to (i) represent Convolutional Neural Network (CNN) architectures and (ii) configure particular CNN architectures. The **CNN variability modeling language** has been designed to look like an abstraction per layer of CNNs. As opposed to proposals that externalize variability in a variability model, such as a feature model, with the **CNN variability modeling language,** we decided to take advantage of the architects' mastery of neural network modeling languages to give them the ability to represent the variability of CNNs directly in the architectural models. The benefits of our proposal and its implementation are manyfold. One of the advantages is that it allows the quick creation of CNNs by just dragging and dropping layers instead of, for example, building a model neuron by neuron, or building and federating two models (one for the generic architecture, and another one to represent the variability and commonality present in the architecture). Thus, the resulting CNNs variability model can be easily configured, and each configuration follows the way the Python Keras library creates sequential models [21].

Besides being user-friendly, the model also allows the generation of all the possible hyper-parameter variations of the selected layers, allowing users to customize each of them in case changes need to be made. Additionally, each resulting CNN configuration model includes a Notebook generation service that executes and exports the currently modeled neural network in Jupyter Notebook with Python auto-generated code, which substantially reduces implementation effort.

The generated CNN currently uses CIFAR-10 to produce and run each CNN configuration model, performing classification tasks on such images, using the layers and hyper-parameters of the selected variation.

The **CNN variability modeling language** was implemented on VariaMos [10], which is a web-based tool that makes use of microservices to enable the specification of product lines using multi-language modeling. Inside VariaMos the structure of the model (Concrete and Abstract Syntax) and its semantics are specified using such tool.

5.1. LANGUAGE ELEMENTS

The elements of the **CNN variability modeling language** involve a set of Neural Network Layers commonly used for convolutional classification, and the elements in the language were heavily inspired by the neural network layers used in the Keras project due to their familiarity with the Machine Learning community and pervasiveness in the industry [21]. The Language Model uses the following elements:

- **CNNArchitecture:** The CNNArchitecture layer contains all the configurations related to the Convolutional Neural Network Architecture like the Optimizer used and its possible variations.
- **InputLayer:** The InputLayer represents a standard input layer; it doesn't contain any hyper-parameters since this layer is modified in execution time when the training data is passed through. In execution time, it holds the "input_shape" parameter used in Keras to accommodate the first layer of the coming data.
- **DenseLayer:** This layer represents a standard Keras Dense layer; it contains hyper-parameters like several units and activation.
- **DropoutLayer:** This layer represents the standard Keras dropdown layer; it contains the hyper-parameter of rate.
- **FlattenLayer:** The Flatten Layer represents the standard Keras Flatten layer; it doesn't contain any hyper-parameters since its only function in the model is to reduce the dimensionality of previous layers [21].
- **PoolingLayer:** The PoolingLayer represents the standard Keras AveragePooling layer; it contains the hyper-parameters
- **ConvolutionalLayer:** The ConvolutionalLayer represents the Conv2D layer since 2D convolution is one of the most common types of filters used in convolution [21]. This layer contains hyper-parameters like filters, kernel_size, and activation.
- **OutputLayer:** This layer represents the output layer at the end of every DNN. It contains the attribute of activation, whose possible values are tailored to image classification.

The **CNN variability modeling language** is shaped as a sequential Convolutional Neural Network in which each element contains a set of hyper-parameters that will change in a combination of ways when the model is configured using the constraint solvers of VariaMos.

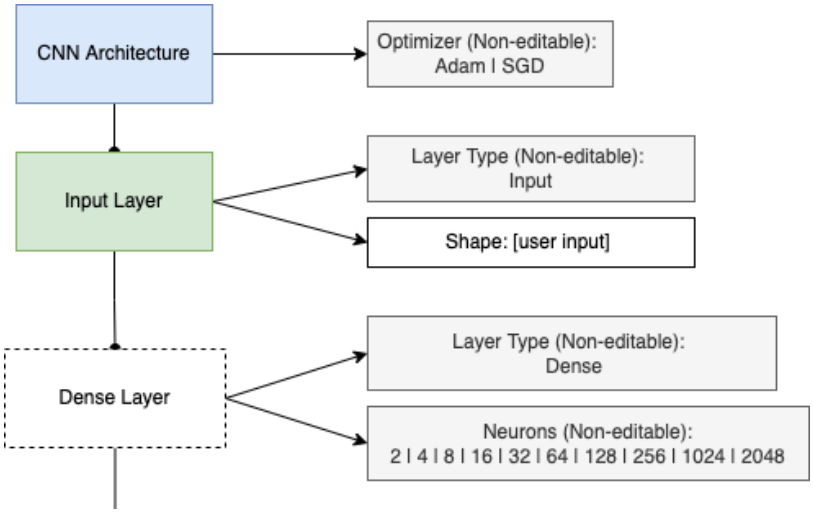


Figure 2. Extract of some hyper-parameters of the CNN variability modeling language

5.2. HYPER-PARAMETERS VARIABILITY

Each of the elements inside the **CNN variability modeling language** contains the most suggested values for each of the CNN layers [22] represented in the following table to generate different CNNs, similar to the ones created and adjusted manually.

Element	Attribute	Possible Values	Meaning
CNNArchitecture	Optimizer	Adam, SGD	Adam and SGD values represent optimizer algorithms used to adjust the weights of neural networks to minimize the loss function during training [21].
InputLayer	N/A	N/A	InputLayer does not have

			any hyper-parameter because is a layer that will be filled in generation time with the values coming from the CIFAR-10 dataset.
OutputLayer	Activation	sigmoid,softmax,softplus,tanh	Activation represents mathematical functions applied to each neuron's output, determining whether it should be activated or not [21].
DenseLayer	Units	1, 2, 8, 16, 32, 64, 128, 256, 512, 1024, 2048	Represents the number of neurons or nodes in a dense (fully connected) layer of a neural network [21].
DenseLayer	Activation	relu, sigmoid, softmax, softplus, softsign, tanh, selu, elu, exponential	Represents the mathematical functions applied to each neuron's output of the Dense layer [21].
DropoutLayer	Rate	0.0, 0.1, 0.2, 0.4, 0.8, 1.0	The rate represents the fraction of input units (neurons) that are randomly set to zero during each update of the training phase

FlattenLayer	N/A	N/A	The “Flatten” layer does not contain any hyperparameter since it is used to reduce the dimensionality of the previous layer in a CNN.
PoolingLayer	PoolSize	None, (2, 2), (3, 3), (4, 4)	The pool size represents the dimensions of the pooling window used in pooling layers, which are employed to reduce the spatial dimensions of the input volume, thus reducing the number of parameters and computations in the network [21].
Convolutional Layer	Filters	8, 16, 32, 64, 128, 256	Filters represent the number of kernels (or filters) applied during the convolution operation [21].
Convolutional Layer	KernelSize	(1, 1), (3, 3), (5, 5), (7, 7), (9, 9)	Kernel size represents the dimensions (width and height) of the convolutional filter [21].
Convolutional Layer	Activation	NoActivation, Relu, Softmax	Represents the mathematical functions

			applied to each neuron's output of the Convolutional layer [21].

Table 1. Table with the possible values of each attribute of the CNN variability modeling language

Each layer inside the CNN architecture, after running through the VariaMos Semantic Translator [23], will have an assigned value per hyper-parameter depending on the variation it is currently in. The variations will be reflected as a combinatorial result of all valid and possible values, as shown in Figure 3.

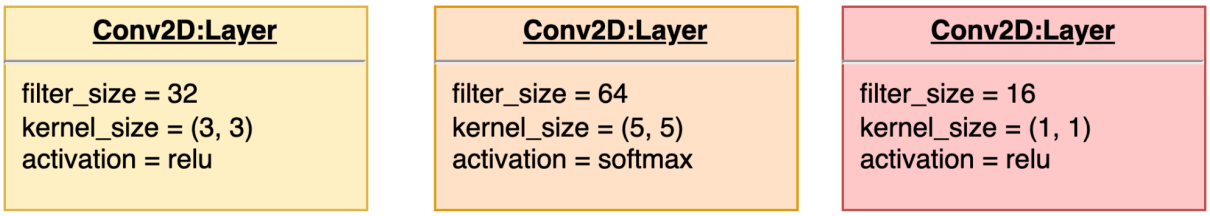


Figure 3. Different variations of the Conv2D layer

To get these variations through the VariaMos reasoning module, it is necessary to perform a query to the model specifying the number of variations wanted. This will make VariaMos generate all the combinations requested in the query, and it will display each found configuration (or solution) as shown in Figure 4.

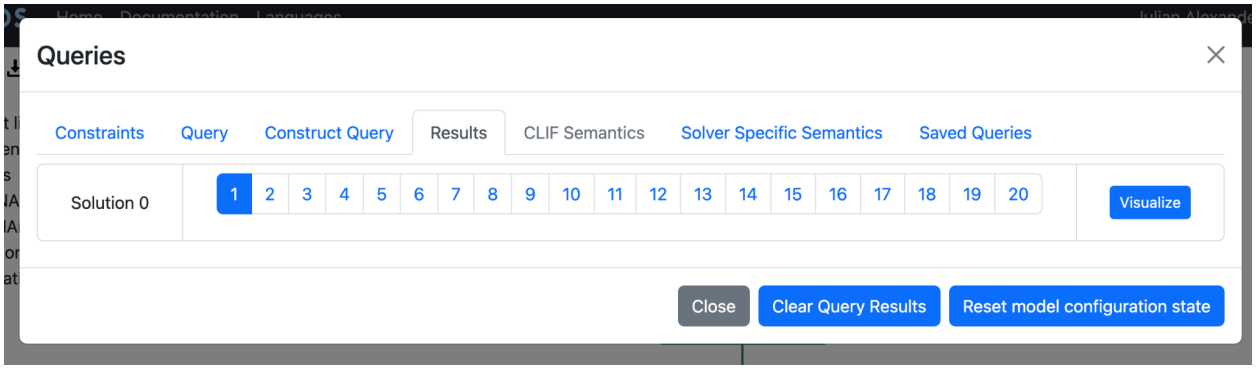


Figure 4. Configurations generated by the VariaMos reasoning module

5.3. CONCRETE AND ABSTRACT SYNTAX

To create the **CNN variability modeling language** in VariaMos we should not just specify the concrete syntax of the language but also its “Abstract Syntax”. On the one hand, the **concrete syntax** of a modeling language refers to the specific notation, symbols, and rules used to visually or textually represent the elements and relationships within a model. On the other hand, the **abstract syntax** represents the entities (elements and attributes) that will be used to create variability-based CNN models and the relationships among these entities [10]. In the case of the **CNN variability modeling language** the “Elements” became layers and “Attributes” became hyper-parameters. The concrete and abstract syntax of the **CNN variability modeling language** is presented in the form of JSON files (the support required by VariaMos) as follows.

Concrete Syntax:


```
{
   "elements": {
     "CNNArchitecture": {
       "draw":
"PHNoYXBlIG5hbWU9IlJvb3RGZWF0dXJlIiBhc3BlY3Q9InZhcmlhYmxlIiBzdHJva2V3aWR0aD0iaW5oZXJ
pdCI+Cgk8YmFja2dyb3VuZD4KCQk8c3Ryb2tlY29sb3IgY29sb3I9IiNjYzAwMDAiLz4KCQk8ZmlsbGNvbG9
yIGNvbG9yPSIjZmZmZmZmIi8+CgkJPHN0cm9rZXdpZHRoIHdpZHRoPSIzIiBmaXhlZD0iMSIvPgoJCTxwYXR
oPgoJCQk8bW92ZSB4PSIwIiB5PSIwIi8+CgkJCTxsaW5lIHg9IjEwMCIgeT0iMCIvPgoJCQk8bGluZSB4PSI
xMDAiIHk9IjEwMCIvPgoJCQk8bGluZSB4PSIwIiB5PSIxMDAiLz4KCQkJPGxpbmUgeD0iMCIgeT0iMCIvPgo
JCQk8Y2xvc2UvPiAgCgkJCTxtb3ZlIHg9IjAiIHk9IjMyIi8+CgkJCTxsaW5lIHg9IjEwMCIgeT0iMzIiLz4
gCgkJPC9wYXRoPgkJIAoJPC9iYWNrZ3JvdW5kPgoJPGZvcmVncm91bmQ+CgkJPGZpbGxzdHJva2UvPgoJCTx
0ZXh0IHg9IjUwIiB5PSIxMyIgYWxpZ249ImNlbnRlciIgcHJvcGVydHluYW1lPSJOYW1lIiBzaG93bGFiZWw
9ImZhbHNlIiAvPiAKCQk8dGV4dCB4PSI1MCIgeT0iNDUiIGFsaWduPSJjZW50ZXIiIGR5bmFtaWNwcm9wZXJ
0aWVzPSJ0cnVlIiBzaG93bGFiZWw9ImZhbHNlIiAgLz4gIAoJPC9mb3JlZ3JvdW5kPgo8L3NoYXBlPg==",
       "icon":
"iVBORw0KGgoAAAANSUhEUgAAAEAAAAAgCAYAAACinX6EAAAAAXNSR0IArs4c6QAAAHxJREFUaEPt2cENwCA
MQ9Fm/6HTQwtCVbpA8tjAVmw+JK7hJ179OdSHYIAJeBwQgbMDMnv6ETvxW23dAQwwASLQEgt0gBJc3OcWWA6
4Bsu3AA7AATgAB3R0AAgBISD0TTYQAkLVpygSRILDSbAjBP1oshmyGrMaGxT4SuoNzk9gIT08aqgAAAAASUV
ORK5CYII=",
       "label": "CNNArchitecture",
       "width": 150,
       "height": 80,
       "label_property": "None"
     },
```

```
    "InputLayer": {
      "draw":
"PHNoYXBlIG5hbWU9IlJvb3RGZWF0dXJlIiBhc3BlY3Q9InZhcmlhYmxlIiBzdHJva2V3aWR0aD0iaW5oZXJ
pdCI+Cgk8YmFja2dyb3VuZD4KCQk8c3Ryb2tlY29sb3IgY29sb3I9IiMyNTlCN0UiLz4KCQk8ZmlsbGNvbG9
yIGNvbG9yPSIjZmZmZmZmIi8+CgkJPHN0cm9rZXdpZHRoIHdpZHRoPSIzIiBmaXhlZD0iMSIvPgoJCTxwYXR
oPgoJCQk8bW92ZSB4PSIwIiB5PSIwIi8+CgkJCTxsaW5lIHg9IjEwIiB5PSIwIi8+CgkJCTxsaW5lIHg9Ij
xMDAiIHk9IjEwIiB5PSIwIi8+CgkJCTxsaW5lIHg9IjEwIiB5PSIxMDAiLz4KCQkJPGxpbmUgeD0iMDAiLz4
JCQk8Y2xvc2UvPiAgCgkJCTxtb3ZlIHg9IjAiIHk9IjAiIi8+CgkJCTxsaW5lIHg9IjEwIiB5PSIwIiIgLz4
gCgkJPC9wYXRoPgkJIAoJPC9iYWNrZ3JvdW5kPgoJPGZvcmVncm91bmQ+CgkJPGZpbGxzdHJva2UvPgoJCTx
0ZXh0IHg9IjUwIiB5PSIxMyIgYWxpZ249ImNlbnRlciIgcHJvcGVydHluYW1lPSJOYW1lIiBzaG93bGFiZWw
9ImZhbHNlIiAvPiAKCQk8dGV4dCB4PSIxMyIgYWxpZ249ImNlbnRlciIgcHJvcGVydHluYW1lPSJDb250ZXIiIGR5bmFtaWNwcm9wZXJ
0aWVzPSJ0cnVlIiBzaG93bGFiZWw9ImZhbHNlIiAgLz4gIAoJPC9mb3JlZ3JvdW5kPgo8L3NoYXBlPg==",
      "icon":
"iVBORw0KGgoAAAANSUhEUgAAAEAAAAAgCAYAAACinX6EAAAAAXNSR0IArs4c6QAAAARnQU1BAACxjwv8YQU
AAAAJcEhZcwAADsQAAA7EAZUrDhsAAABmSURBVGhD7ZkxDsAgAIS0//9z2xjs4BMKDJK4SW5z3i9DzIW1fAu
Yc64LC3v4LeBcwC7zV8536hdQAKylAFhLAbCWAmAtBcBaQDspQBYSwGwln6G1imm32EsZYw
H6mkgKD+7AsEAAAAASUVORK5CYII=",
      "label": "InputLayer",
      "width": 150,
      "height": 80,
      "label_property": "None"
    },
    "OutputLayer": {
      "draw":
"PHNoYXBlIG5hbWU9IlJvb3RGZWF0dXJlIiBhc3BlY3Q9InZhcmlhYmxlIiBzdHJva2V3aWR0aD0iaW5oZXJ
pdCI+Cgk8YmFja2dyb3VuZD4KCQk8c3Ryb2tlY29sb3IgY29sb3I9IiNGQkI0MkYiLz4KCQk8ZmlsbGNvbG9
yIGNvbG9yPSIjZmZmZmZmIi8+CgkJPHN0cm9rZXdpZHRoIHdpZHRoPSIzIiBmaXhlZD0iMSIvPgoJCTxwYXR
oPgoJCQk8bW92ZSB4PSIwIiB5PSIwIi8+CgkJCTxsaW5lIHg9IjEwIiB5PSIwIi8+CgkJCTxsaW5lIHg9Ij
xMDAiIHk9IjEwIiB5PSIwIi8+CgkJCTxsaW5lIHg9IjEwIiB5PSIxMDAiLz4KCQkJPGxpbmUgeD0iMDAiLz4
JCQk8Y2xvc2UvPiAgCgkJCTxtb3ZlIHg9IjAiIHk9IjAiIi8+CgkJCTxsaW5lIHg9IjEwIiB5PSIwIiIgLz4
gCgkJPC9wYXRoPgkJIAoJPC9iYWNrZ3JvdW5kPgoJPGZvcmVncm91bmQ+CgkJPGZpbGxzdHJva2UvPgoJCTx
0ZXh0IHg9IjUwIiB5PSIxMyIgYWxpZ249ImNlbnRlciIgcHJvcGVydHluYW1lPSJOYW1lIiBzaG93bGFiZWw
9ImZhbHNlIiAvPiAKCQk8dGV4dCB4PSIxMyIgYWxpZ249ImNlbnRlciIgcHJvcGVydHluYW1lPSJDb250ZXIiIGR5bmFtaWNwcm9wZXJ
0aWVzPSJ0cnVlIiBzaG93bGFiZWw9ImZhbHNlIiAgLz4gIAoJPC9mb3JlZ3JvdW5kPgo8L3NoYXBlPg==",
      "icon":
"iVBORw0KGgoAAAANSUhEUgAAAEAAAAAgCAYAAACinX6EAAAAAXNSR0IArs4c6QAAAARnQU1BAACxjwv8YQU
AAAAJcEhZcwAADsQAAA7EAZUrDhsAAABmSURBVGhD7ZkxDsAgAIS0//9z2xjs4BMKDJK4SW5z3i9DzIW1fAu
Yc64LC3v4LeBcwC7zV8536hdQAKylAFhLAbCWAmAtBcBaQDspQBYSwGwln6G1imm32EsZYw
H6mkgKD+7AsEAAAAASUVORK5CYII=",
      "label": "OutputLayer",
      "width": 150,
```

```
      "height": 80,
      "label_property": "None"
    },
    "DenseLayer": {
      "draw":
"PHNoYXBlIG5hbWU9IkFic3RyYWN0RmVhdHVyZSIgYXNwZWN0PSJ2YXJpYWJsZSIgc3Ryb2tld2lkdGg9Iml
uaGVyaXQiPgoJPGJhY2tncm91bmQ+CgkJPHN0cm9rZWNvbG9yIGNvbG9yPSIjM0Q3NEZGIi8+CgkJPGZpbGx
jb2xvciBjb2xvcj0iI2ZmZmZmZiIvPgoJCTxwYXRoPgoJCQk8bW92ZSB4PSIwIiB5PSIwIi8+CgkJCTxsaW5
lIHg9IjEwMCIgeT0iMCIvPgoJCQk8bGluZSB4PSIxMDAiIHk9IjEwMCIvPgoJCQk8bGluZSB4PSIwIiB5PSI
xMDAiLz4KCQkJPGxpbmUgeD0iMCIgeT0iMCIvPgoJCQk8Y2xvc2UvPiAgCgkJCTxtb3ZlIHg9IjAiIHk9IjM
yIi8+CgkJCTxsaW5lIHg9IjEwMCIgeT0iMzIiLz4gCgkJPC9wYXRoPgkJCiAgICAgICAgICAgICAgICA8ZGF
zaGVkIGRhc2hlZD0iMSIvPgkKICAgICAgICAgICAgICAgIDxkYXNocGF0dGVybiBwYXR0ZXJuPSIxMCAxMCI
vPgoJPC9iYWNrZ3JvdW5kPgoJPGZvcmVncm91bmQ+CgkJPGZvcmVncm91bmQ+CgkJCTx0ZXh0IGFsaWduPSJ
5PSIxMyIgYWxpZ249ImNlbnRlciIgcHJvcGVydHluYW1lPSJOYW1lIiBzaG93bGFiZWw9ImZhbHNlIiAvPiA
KCQk8dGV4dCB4PSI1MCIgeT0iNDUiIGFsaWduPSJjZW50ZXIiIGR5bmFtaWNwcm9wZXJ0aWVzPSJ0cnVlIiB
zaG93bGFiZWw9ImZhbHNlIiAgLz4gIAoJPC9mb3JlZ3JvdW5kPgo8L3NoYXBlPg==",
      "icon":
"iVBORw0KGgoAAAANSUhEUgAAAEAAAAAgCAYAAACinX6EAAAAAXNSR0IArs4c6QAAAARnQU1BAACxjwv8YQU
AAAAJcEhZcwAADsQAAA7EAZUrDhsAAACNSURBVGhD7ZZBEoAgDAOp//+zwkxEr0AbvYSZ+AUU9I4Ow3MIcX
ySkBE6OuhCshf7g2cgO5MbQ0EXALGeKwjMg3IBxTwb4ANkGLBGTBKby0+16AUiw2QYsEZ4E0w4RGQYrEBUgz
eBBMeASkWGyDF4E0wgR+BWYPr36+asUrHzvfuMyegO/G1CQQ8Aa1dKuhPNgYdDSEAAAAASUVORK5CYII=",
      "label": "DenseLayer",
      "width": 150,
      "height": 80,
      "label_property": "None"
    },
    "DropoutLayer": {
      "draw":
"PHNoYXBlIG5hbWU9IkFic3RyYWN0RmVhdHVyZSIgYXNwZWN0PSJ2YXJpYWJsZSIgc3Ryb2tld2lkdGg9Iml
uaGVyaXQiPgoJPGJhY2tncm91bmQ+CgkJPHN0cm9rZWNvbG9yIGNvbG9yPSIjM0Q3NEZGIi8+CgkJPGZpbGx
jb2xvciBjb2xvcj0iI2ZmZmZmZiIvPgoJCTxwYXRoPgoJCQk8bW92ZSB4PSIwIiB5PSIwIi8+CgkJCTxsaW5
lIHg9IjEwMCIgeT0iMCIvPgoJCQk8bGluZSB4PSIxMDAiIHk9IjEwMCIvPgoJCQk8bGluZSB4PSIwIiB5PSI
xMDAiLz4KCQkJPGxpbmUgeD0iMCIgeT0iMCIvPgoJCQk8Y2xvc2UvPiAgCgkJCTxtb3ZlIHg9IjAiIHk9IjM
yIi8+CgkJCTxsaW5lIHg9IjEwMCIgeT0iMzIiLz4gCgkJPC9wYXRoPgkJCiAgICAgICAgICAgICAgICA8ZGF
zaGVkIGRhc2hlZD0iMSIvPgkKICAgICAgICAgICAgICAgIDxkYXNocGF0dGVybiBwYXR0ZXJuPSIxMCAxMCI
vPgoJPC9iYWNrZ3JvdW5kPgoJPGZvcmVncm91bmQ+CgkJPGZvcmVncm91bmQ+CgkJCTx0ZXh0IGFsaWduPSJ
5PSIxMyIgYWxpZ249ImNlbnRlciIgcHJvcGVydHluYW1lPSJOYW1lIiBzaG93bGFiZWw9ImZhbHNlIiAvPiA
KCQk8dGV4dCB4PSI1MCIgeT0iNDUiIGFsaWduPSJjZW50ZXIiIGR5bmFtaWNwcm9wZXJ0aWVzPSJ0cnVlIiB
zaG93bGFiZWw9ImZhbHNlIiAgLz4gIAoJPC9mb3JlZ3JvdW5kPgo8L3NoYXBlPg==",
      "icon":
"iVBORw0KGgoAAAANSUhEUgAAAEAAAAAgCAYAAACinX6EAAAAAXNSR0IArs4c6QAAAARnQU1BAACxjwv8YQU
```

```
AAAAJcEhZcwAADsQAAA7EAZUrDhsAAACNSURBVGhD7ZZBEoAgDAOp//+zwkxErP0AbvYSZ+AUU9I4Ow3MIcX
ySkBE6OuhCshf7g2cgO5MbQ0EXALGeKwjMg3IBxTwb4ANkGLBGTBKby0+16AUiw2QYsEZ4E0w4RGQYrEBUgz
eBBMeASkWGyDF4E0wgR+BWYPr36+asUrHzvfuMyegO/G1CQQ8Aa1dKuhPNgYdDSEAAAAASUVORK5CYII=",
        "label": "DropoutLayer",
        "width": 150,
        "height": 80,
        "label_property": "None"
      },
      "FlattenLayer": {
        "draw":
"PHNoYXBlIG5hbWU9IkFic3RyYWN0RmVhdHVyZSIgYXNwZWN0PSJ2YXJpYWJsZSIgc3Ryb2tld2lkdGg9Iml
uaGVyaXQiPgoJPGJhY2tncm91bmQ+CgkJPHN0cm9rZWNvbG9yIGNvbG9yPSIjM0Q3NEZGIi8+CgkJPGZpbGx
jb2xvciBjb2xvcj0iI2ZmZmZmZiIvPgoJCTxwYXRoPgoJCQk8bW92ZSB4PSIwIiB5PSIwIi8+CgkJCTxsaW5
lIHg9IjEwMCIgeT0iMCIvPgoJCQk8bGluZSB4PSIxMDAiIHk9IjEwMCIvPgoJCQk8bGluZSB4PSIwIiB5PSI
xMDAiLz4KCQkJPGxpbmUgeD0iMCIgeT0iMCIvPgoJCQk8Y2xvc2UvPiAgCgkJCTxtb3ZlIHg9IjAiIHk9IjM
yIi8+CgkJCTxsaW5lIHg9IjEwMCIgeT0iMzIiLz4gCgkJPC9wYXRoPgkJCiAgICAgICAgICAgICAgICA8ZGF
zaGVkIGRhc2hlZD0iMSIvPgkKICAgICAgICAgICAgICAgIDxkYXNocGF0dGVybiBwYXR0ZXJuPSIxMCAxMCI
vPgoJPC9iYWNrZ3JvdW5kPgoJPGZvcmVncm91bmQ+CgkJPGZpbGxzdHJva2UvPgoJCTx0ZXh0IHg9IjUwIiB
5PSIxMyIgYWxpZ249ImNlbnRlciIgcHJvcGVydHluYW1lPSJOYW1lIiBzaG93bGFiZWw9ImZhbHNlIiAvPiA
KCQk8dGV4dCB4PSI1MCIgeT0iNDUiIGFsaWduPSJjZW50ZXIiIGR5bmFtaWNwcm9wZXJ0aWVzPSJ0cnVlIiB
zaG93bGFiZWw9ImZhbHNlIiAgLz4gIAoJPC9mb3JlZ3JvdW5kPgo8L3NoYXBlPg==",
        "icon":
"iVBORw0KGgoAAAANSUhEUgAAAEAAAAAgCAYAAACinX6EAAAAAXNSR0IArs4c6QAAAARnQU1BAACxjwv8YQU
AAAAJcEhZcwAADsQAAA7EAZUrDhsAAACNSURBVGhD7ZZBEoAgDAOp//+zwkxErP0AbvYSZ+AUU9I4Ow3MIcX
ySkBE6OuhCshf7g2cgO5MbQ0EXALGeKwjMg3IBxTwb4ANkGLBGTBKby0+16AUiw2QYsEZ4E0w4RGQYrEBUgz
eBBMeASkWGyDF4E0wgR+BWYPr36+asUrHzvfuMyegO/G1CQQ8Aa1dKuhPNgYdDSEAAAAASUVORK5CYII=",
        "label": "FlattenLayer",
        "width": 150,
        "height": 80,
        "label_property": "None"
      },
      "PoolingLayer": {
        "draw":
"PHNoYXBlIG5hbWU9IkFic3RyYWN0RmVhdHVyZSIgYXNwZWN0PSJ2YXJpYWJsZSIgc3Ryb2tld2lkdGg9Iml
uaGVyaXQiPgoJPGJhY2tncm91bmQ+CgkJPHN0cm9rZWNvbG9yIGNvbG9yPSIjM0Q3NEZGIi8+CgkJPGZpbGx
jb2xvciBjb2xvcj0iI2ZmZmZmZiIvPgoJCTxwYXRoPgoJCQk8bW92ZSB4PSIwIiB5PSIwIi8+CgkJCTxsaW5
lIHg9IjEwMCIgeT0iMCIvPgoJCQk8bGluZSB4PSIxMDAiIHk9IjEwMCIvPgoJCQk8bGluZSB4PSIwIiB5PSI
xMDAiLz4KCQkJPGxpbmUgeD0iMCIgeT0iMCIvPgoJCQk8Y2xvc2UvPiAgCgkJCTxtb3ZlIHg9IjAiIHk9IjM
yIi8+CgkJCTxsaW5lIHg9IjEwMCIgeT0iMzIiLz4gCgkJPC9wYXRoPgkJCiAgICAgICAgICAgICAgICA8ZGF
zaGVkIGRhc2hlZD0iMSIvPgkKICAgICAgICAgICAgICAgIDxkYXNocGF0dGVybiBwYXR0ZXJuPSIxMCAxMCI
vPgoJPC9iYWNrZ3JvdW5kPgoJPGZvcmVncm91bmQ+CgkJPGZpbGxzdHJva2UvPgoJCTx0ZXh0IHg9IjUwIiB
```

```
5PSIxMyIgYWxpZ249ImNlbnRlciIgcHJvcGVydHluYW1lPSJOYW1lIiBzaG93bGFiZWw9ImZhbHNlIiAvPiA
KCQk8dGV4dCB4PSI1MCIgeT0iNDUiIGFsaWduPSJjZW50ZXIiIGR5bmFtaWNwcm9wZXJ0aWVzPSJ0cnVlIiB
zaG93bGFiZWw9ImZhbHNlIiAgLz4gIAoJPC9mb3JlZ3JvdW5kPgo8L3NoYXBlPg==",
      "icon":
"iVBORw0KGgoAAAANSUhEUgAAAEAAAAAgCAYAAACinX6EAAAAAXNSR0IArs4c6QAAAARnQU1BAACxjwv8YQU
AAAAJcEhZcwAADsQAAA7EAZUrDhsAAACNSURBVGhD7ZZBEoAgDAOp//+zwkxErP0AbvYSZ+AUU9I4Ow3MIcX
ySkBE6OuhCshf7g2cgO5MbQ0EXALGeKwjMg3IBxTwb4ANkGLBGTBKby0+16AUiw2QYsEZ4E0w4RGQYrEBUgz
eBBMeASkWGyDF4E0wgR+BWYPr36+asUrHzvfuMyegO/G1CQQ8Aa1dKuhPNgYdDSEAAAAASUVORK5CYII=",
      "label": "PoolingLayer",
      "width": 150,
      "height": 80,
      "label_property": "None"
    },
    "ConvolutionalLayer": {
      "draw":
"PHNoYXBlIG5hbWU9IkFic3RyYWN0RmVhdHVyZSIgYXNwZWN0PSJ2YXJpYWJsZSIgc3Ryb2tld2lkdGg9Iml
uaGVyaXQiPgoJPGJhY2tncm91bmQ+CgkJPHN0cm9rZWNvbG9yIGNvbG9yPSIjM0Q3NEZGIi8+CgkJPGZpbGx
jb2xvciBjb2xvcj0iI2ZmZmZmZiIvPgoJCTxwYXRoPgoJCQk8bW92ZSB4PSIwIiB5PSIwIi8+CgkJCTxsaW5
lIHg9IjEwMCIgeT0iMCIvPgoJCQk8bGluZSB4PSIxMDAiIHk9IjEwMCIvPgoJCQk8bGluZSB4PSIwIiB5PSI
xMDAiLz4KCQkJPGxpbmUgeD0iMCIgeT0iMCIvPgoJCQk8Y2xvc2UvPiAgCgkJCTxtb3ZlIHg9IjAiIHk9IjM
yIi8+CgkJCTxsaW5lIHg9IjEwMCIgeT0iMzIiLz4gCgkJPC9wYXRoPgkJCiAgICAgICAgICAgICAgICA8ZGF
zaGVkIGRhc2hlZD0iMSIvPgkKICAgICAgICAgICAgICAgIDxkYXNocGF0dGVybiBwYXR0ZXJuPSIxMCAxMCI
vPgoJPC9iYWNrZ3JvdW5kPgoJPGZvcmVncm91bmQ+CgkJPGZpbGxzdHJva2UvPgoJCTx0ZXh0IHg9IjUwIiB
5PSIxMyIgYWxpZ249ImNlbnRlciIgcHJvcGVydHluYW1lPSJOYW1lIiBzaG93bGFiZWw9ImZhbHNlIiAvPiA
KCQk8dGV4dCB4PSI1MCIgeT0iNDUiIGFsaWduPSJjZW50ZXIiIGR5bmFtaWNwcm9wZXJ0aWVzPSJ0cnVlIiB
zaG93bGFiZWw9ImZhbHNlIiAgLz4gIAoJPC9mb3JlZ3JvdW5kPgo8L3NoYXBlPg==",
      "icon":
"iVBORw0KGgoAAAANSUhEUgAAAEAAAAAgCAYAAACinX6EAAAAAXNSR0IArs4c6QAAAARnQU1BAACxjwv8YQU
AAAAJcEhZcwAADsQAAA7EAZUrDhsAAACNSURBVGhD7ZZBEoAgDAOp//+zwkxErP0AbvYSZ+AUU9I4Ow3MIcX
ySkBE6OuhCshf7g2cgO5MbQ0EXALGeKwjMg3IBxTwb4ANkGLBGTBKby0+16AUiw2QYsEZ4E0w4RGQYrEBUgz
eBBMeASkWGyDF4E0wgR+BWYPr36+asUrHzvfuMyegO/G1CQQ8Aa1dKuhPNgYdDSEAAAAASUVORK5CYII=",
      "label": "ConvolutionalLayer",
      "width": 150,
      "height": 100,
      "label_property": "None"
    }
  },
  "relationships": {
    "DenseLayer_Feature": {
      "styles": [
        {
```

```
            "style":
"strokeColor=#446E79;strokeWidth=2;endArrow=oval;endFill=1;endSize=12;",
            "linked_value": "Mandatory",
            "linked_property": "Type"
          },
          {
            "style":
"strokeColor=#446E79;strokeWidth=2;endArrow=oval;endFill=0;endSize=12;",
            "linked_value": "Optional",
            "linked_property": "Type"
          },
          {
            "style": "strokeColor=#446E79;strokeWidth=2;endArrow=open;",
            "linked_value": "Includes",
            "linked_property": "Type"
          },
          {
            "style":
"strokeColor=#446E79;strokeWidth=2;startArrow=open;endArrow=open;",
            "linked_value": "Excludes",
            "linked_property": "Type"
          },
          {
            "style": "strokeColor=#446E79;strokeWidth=2;endArrow=none;",
            "linked_value": "IndividualCardinality",
            "linked_property": "Type"
          },
          {
            "style": "strokeColor=#446E79;strokeWidth=2;"
          }
        ],
        "label_property": "Type"
      },
      "InputLayer_Feature": {
        "styles": [
          {
            "style":
"strokeColor=#446E79;strokeWidth=2;endArrow=oval;endFill=1;endSize=12;",
            "linked_value": "Mandatory",
            "linked_property": "Type"
          },
```

```
        {
          "style":
"strokeColor=#446E79;strokeWidth=2;endArrow=oval;endFill=0;endSize=12;",
          "linked_value": "Optional",
          "linked_property": "Type"
        },
        {
          "style": "strokeColor=#446E79;strokeWidth=2;endArrow=open;",
          "linked_value": "Includes",
          "linked_property": "Type"
        },
        {
          "style":
"strokeColor=#446E79;strokeWidth=2;startArrow=open;endArrow=open;",
          "linked_value": "Excludes",
          "linked_property": "Type"
        },
        {
          "style": "strokeColor=#446E79;strokeWidth=2;endArrow=none;",
          "linked_value": "IndividualCardinality",
          "linked_property": "Type"
        },
        {
          "style": "strokeColor=#446E79;strokeWidth=2;"
        }
      ],
      "label_property": "Type"
    },
    "OutputLayer_Feature": {},
    "DropoutLayer_Feature": {
      "styles": [
        {
          "style":
"strokeColor=#446E79;strokeWidth=2;endArrow=oval;endFill=1;endSize=12;",
          "linked_value": "Mandatory",
          "linked_property": "Type"
        },
        {
          "style":
"strokeColor=#446E79;strokeWidth=2;endArrow=oval;endFill=0;endSize=12;",
          "linked_value": "Optional",
```

```
            "linked_property": "Type"
          },
          {
            "style": "strokeColor=#446E79;strokeWidth=2;endArrow=open;",
            "linked_value": "Includes",
            "linked_property": "Type"
          },
          {
            "style":
"strokeColor=#446E79;strokeWidth=2;startArrow=open;endArrow=open;",
            "linked_value": "Excludes",
            "linked_property": "Type"
          },
          {
            "style": "strokeColor=#446E79;strokeWidth=2;endArrow=none;",
            "linked_value": "IndividualCardinality",
            "linked_property": "Type"
          },
          {
            "style": "strokeColor=#446E79;strokeWidth=2;"
          }
        ],
        "label_property": "Type"
      },
      "FlattenLayer_Feature": {
        "styles": [
          {
            "style":
"strokeColor=#446E79;strokeWidth=2;endArrow=oval;endFill=1;endSize=12;",
            "linked_value": "Mandatory",
            "linked_property": "Type"
          },
          {
            "style":
"strokeColor=#446E79;strokeWidth=2;endArrow=oval;endFill=0;endSize=12;",
            "linked_value": "Optional",
            "linked_property": "Type"
          },
          {
            "style": "strokeColor=#446E79;strokeWidth=2;endArrow=open;",
            "linked_value": "Includes",
```

```
            "linked_property": "Type"
          },
          {
            "style":
"strokeColor=#446E79;strokeWidth=2;startArrow=open;endArrow=open;",
            "linked_value": "Excludes",
            "linked_property": "Type"
          },
          {
            "style": "strokeColor=#446E79;strokeWidth=2;endArrow=none;",
            "linked_value": "IndividualCardinality",
            "linked_property": "Type"
          },
          {
            "style": "strokeColor=#446E79;strokeWidth=2;"
          }
        ],
        "label_property": "Type"
      },
      "PoolingLayer_Feature": {
        "styles": [
          {
            "style":
"strokeColor=#446E79;strokeWidth=2;endArrow=oval;endFill=1;endSize=12;",
            "linked_value": "Mandatory",
            "linked_property": "Type"
          },
          {
            "style":
"strokeColor=#446E79;strokeWidth=2;endArrow=oval;endFill=0;endSize=12;",
            "linked_value": "Optional",
            "linked_property": "Type"
          },
          {
            "style": "strokeColor=#446E79;strokeWidth=2;endArrow=open;",
            "linked_value": "Includes",
            "linked_property": "Type"
          },
          {
            "style":
"strokeColor=#446E79;strokeWidth=2;startArrow=open;endArrow=open;",
```

```
            "linked_value": "Excludes",
            "linked_property": "Type"
          },
          {
            "style": "strokeColor=#446E79;strokeWidth=2;endArrow=none;",
            "linked_value": "IndividualCardinality",
            "linked_property": "Type"
          },
          {
            "style": "strokeColor=#446E79;strokeWidth=2;"
          }
        ],
        "label_property": "Type"
      },
      "CNNArchitecture_Feature": {
        "styles": [
          {
            "style":
"strokeColor=#446E79;strokeWidth=2;endArrow=oval;endFill=1;endSize=12;",
            "linked_value": "Mandatory",
            "linked_property": "Type"
          },
          {
            "style":
"strokeColor=#446E79;strokeWidth=2;endArrow=oval;endFill=0;endSize=12;",
            "linked_value": "Optional",
            "linked_property": "Type"
          },
          {
            "style": "strokeColor=#446E79;strokeWidth=2;endArrow=open;",
            "linked_value": "Includes",
            "linked_property": "Type"
          },
          {
            "style":
"strokeColor=#446E79;strokeWidth=2;startArrow=open;endArrow=open;",
            "linked_value": "Excludes",
            "linked_property": "Type"
          },
          {
            "style": "strokeColor=#446E79;strokeWidth=2;endArrow=none;",
```

```
          "linked_value": "IndividualCardinality",
          "linked_property": "Type"
        },
        {
          "style": "strokeColor=#446E79;strokeWidth=2;"
        }
      ],
      "label_property": "Type"
    },
    "ConvolutionalLayer_Feature": {
      "styles": [
        {
          "style":
"strokeColor=#446E79;strokeWidth=2;endArrow=oval;endFill=1;endSize=12;",
          "linked_value": "Mandatory",
          "linked_property": "Type"
        },
        {
          "style":
"strokeColor=#446E79;strokeWidth=2;endArrow=oval;endFill=0;endSize=12;",
          "linked_value": "Optional",
          "linked_property": "Type"
        },
        {
          "style": "strokeColor=#446E79;strokeWidth=2;endArrow=open;",
          "linked_value": "Includes",
          "linked_property": "Type"
        },
        {
          "style":
"strokeColor=#446E79;strokeWidth=2;startArrow=open;endArrow=open;",
          "linked_value": "Excludes",
          "linked_property": "Type"
        },
        {
          "style": "strokeColor=#446E79;strokeWidth=2;endArrow=none;",
          "linked_value": "IndividualCardinality",
          "linked_property": "Type"
        },
        {
          "style": "strokeColor=#446E79;strokeWidth=2;"
```

```
        }
      ],
      "label_property": "Type"
    }
  }
}
```

Abstract Syntax:

```
{
 "elements": {
   "CNNArchitecture": {
     "properties": [
       {
         "name": "Selected",
         "type": "String",
         "comment": "Selected",
         "possibleValues": "Undefined,Selected,Unselected"
       },
       {
         "name": "Optimizer",
         "type": "String",
         "display": true,
         "comment": "Optimizer used for the DNN Architecture",
         "possibleValues": "Adam,SGD"
       }
     ]
   },
   "InputLayer": {
     "properties": [
       {
         "name": "Selected",
         "type": "String",
         "comment": "Selected",
         "possibleValues": "Undefined,Selected,Unselected"
       }
     ]
   },
   "OutputLayer": {
     "properties": [
```

```
      {
        "name": "Selected",
        "type": "String",
        "comment": "Selected",
        "possibleValues": "Undefined,Selected,Unselected"
      },
      {
        "name": "Activation",
        "type": "String",
        "comment": "The type of layer activation",
        "possibleValues": "sigmoid___,softmax___,softplus___,tanh___"
      }
    ]
  },
  "DenseLayer": {
    "properties": [
      {
        "name": "Selected",
        "type": "String",
        "comment": "Selected",
        "possibleValues": "Undefined,Selected,Unselected"
      },
      {
        "name": "Units",
        "type": "String",
        "comment": "The layer units",
        "display": true,
        "possibleValues":
"SingleNeuron,CoupleNeurons,SmallLayer,SmallHiddenLayer,MediumHiddenLayer,LargeHidde
nLayer,VeryLargeHiddenLayer,ExtraLargeHiddenLayer,UltraLargeHiddenLayer,MegaLargeHid
denLayer"
      },
      {
        "name": "Activation",
        "type": "String",
        "comment": "The type of layer activation",
        "display": true,
        "possibleValues":
"relu_,sigmoid_,softmax_,softplus_,softsign_,tanh_,selu_,elu_,exponential_"
      }
    ]
```

```
  },
  "DropoutLayer": {
    "properties": [
      {
        "name": "Selected",
        "type": "String",
        "comment": "Selected",
        "possibleValues": "Undefined,Selected,Unselected"
      },
      {
        "name": "Rate",
        "type": "String",
        "display": true,
        "comment": "Size of the rate",
        "possibleValues":
"NoDropout,VeryLightDropout,LightDropout,ModerateDropout,StrongDropout,HeavyDropout,
CompleteDropout"
      }
    ]
  },
  "FlattenLayer": {
    "properties": [
      {
        "name": "Selected",
        "type": "String",
        "comment": "Selected",
        "possibleValues": "Undefined,Selected,Unselected"
      }
    ]
  },
  "PoolingLayer": {
    "properties": [
      {
        "name": "Selected",
        "type": "String",
        "comment": "Selected",
        "possibleValues": "Undefined,Selected,Unselected"
      },
      {
        "name": "PoolSize",
        "type": "String",
```

```
        "comment": "Size of the pool",
        "possibleValues":
"LightAveragePooling, moderateAveragePooling,StrongAveragePooling"
      }
    ]
  },
  "ConvolutionalLayer": {
    "properties": [
      {
        "name": "Selected",
        "type": "String",
        "comment": "Selected",
        "possibleValues": "Undefined,Selected,Unselected"
      },
      {
        "name": "Filters",
        "type": "String",
        "comment": "Number of filters",
        "possibleValues":
"VerySmallNumber,SmallNumber,ModerateNumber,Many,VeryMany,ExtremelyMany"
      },
      {
        "name": "KernelSize",
        "type": "String",
        "comment": "Size of the filter",
        "possibleValues": "VerySmall,Small,Medium,Large,VeryLarge"
      },
      {
        "name": "Activation",
        "type": "String",
        "comment": "Activation of the Layer",
        "possibleValues": "NoActivation,relu__,softmax__"
      }
    ]
  }
},
"restrictions": {
  "quantity_element": [
    {
      "max": 1,
      "min": 0,
```

```
      "element": "CNNArchitecture"
    },
    {
      "max": 1,
      "min": 0,
      "element": "InputLayer"
    },
    {
      "max": 1,
      "min": 0,
      "element": "OutputLayer"
    }
  ]
},
"relationships": {
  "CNNArchitecture_Feature": {
    "max": 1,
    "min": 0,
    "source": "CNNArchitecture",
    "target": ["InputLayer"],
    "properties": [
      {
        "name": "Type",
        "type": "String",
        "possibleValues": "Mandatory,"
      }
    ]
  },
  "DenseLayer_Feature": {
    "max": 1,
    "min": 0,
    "source": "DenseLayer",
    "target": [
      "DenseLayer",
      "ConvolutionalLayer",
      "PoolingLayer",
      "FlattenLayer",
      "DropoutLayer",
      "OutputLayer"
    ],
    "properties": [
```

```
      {
        "name": "Type",
        "type": "String",
        "possibleValues": "Mandatory,Optional"
      }
    ]
  },
  "InputLayer_Feature": {
    "max": 1,
    "min": 0,
    "source": "InputLayer",
    "target": [
      "DenseLayer",
      "ConvolutionalLayer",
      "PoolingLayer",
      "FlattenLayer",
      "DropoutLayer",
      "OutputLayer"
    ],
    "properties": [
      {
        "name": "Type",
        "type": "String",
        "possibleValues": "Mandatory,Optional"
      }
    ]
  },
  "OutputLayer_Feature": {
    "max": 0,
    "min": 0,
    "source": "OutputLayer",
    "target": [],
    "properties": [
      {
        "name": "Type",
        "type": "String",
        "possibleValues": "Mandatory,"
      }
    ]
  },
  "DropoutLayer_Feature": {
```

```
    "max": 1,
    "min": 0,
    "source": "DropoutLayer",
    "target": [
      "DenseLayer",
      "ConvolutionalLayer",
      "PoolingLayer",
      "FlattenLayer",
      "DropoutLayer",
      "OutputLayer"
    ],
    "properties": [
      {
        "name": "Type",
        "type": "String",
        "possibleValues": "Mandatory,Optional"
      }
    ]
  },
  "FlattenLayer_Feature": {
    "max": 1,
    "min": 0,
    "source": "FlattenLayer",
    "target": [
      "DenseLayer",
      "ConvolutionalLayer",
      "PoolingLayer",
      "FlattenLayer",
      "DropoutLayer",
      "OutputLayer"
    ],
    "properties": [
      {
        "name": "Type",
        "type": "String",
        "possibleValues": "Mandatory,Optional"
      }
    ]
  },
  "PoolingLayer_Feature": {
    "max": 1,
```

```
    "min": 0,
    "source": "PoolingLayer",
    "target": [
      "DenseLayer",
      "ConvolutionalLayer",
      "PoolingLayer",
      "FlattenLayer",
      "DropoutLayer",
      "OutputLayer"
    ],
    "properties": [
      {
        "name": "Type",
        "type": "String",
        "possibleValues": "Mandatory,Optional"
      }
    ]
  },
  "ConvolutionalLayer_Feature": {
    "max": 1,
    "min": 0,
    "source": "ConvolutionalLayer",
    "target": [
      "DenseLayer",
      "ConvolutionalLayer",
      "PoolingLayer",
      "FlattenLayer",
      "DropoutLayer",
      "OutputLayer"
    ],
    "properties": [
      {
        "name": "Type",
        "type": "String",
        "possibleValues": "Mandatory,Optional"
      }
    ]
  }
 }
}
```

*

The previous abstract and concrete syntax can be used to create CNN variability models using the VariaMos tool. To create such a variability model, an **example of using an image classification CNN** on which hyper-parameters needed to be variated using only Convolutional and Dense layers can be seen in Figure 5. When a Notebook is generated, the classification process will use the CIFAR-10 dataset, which contains images classified into different categories.

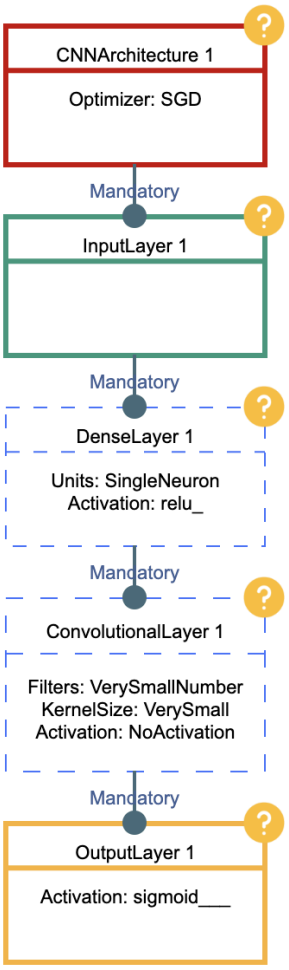


Figure 6. CNN variability model example, includes a configuration element (CNNArchiteture), an Input layer (Input Layer 1) which will be filled with the dataset input shape in the Notebook generation process, a Dense layer (DenseLayer 1) with the number of units and activation hyperparameters, a Conv2D (ConvolutionalLayer 1) containing the filters, kernel size and

activation hyperparameters, and an Output layer (OutputLayer 1) containing an activation hyperparameters. Each of the hyperparameters uses a textual representation of their values to avoid colliding with the semantic variation process.

5.4. LANGUAGE SEMANTICS

The semantics of the **CNN variability modeling language** were written using the standard Common Logic Interchange Format (CLIF), which is a Lisp-like standard constraint language that can be used to add constraints to language constructs with the added benefit that they can be interpreted by any solver like MiniZinc or Prolog [23]. The semantics of the **CNN variability modeling language** can be seen in the following code.

```
{
 "elementTypes": [
   "CNNArchitecture",
   "InputLayer",
   "OutputLayer",
   "DenseLayer",
   "DropoutLayer",
   "FlattenLayer",
   "PoolingLayer",
   "ConvolutionalLayer"
 ],
 "relationTypes": ["Mandatory", "Optional"],
 "attributeTypes": [
   "CNNArchitecture",
   "InputLayer",
   "OutputLayer",
   "DenseLayer",
   "DropoutLayer",
   "FlattenLayer",
   "PoolingLayer",
   "ConvolutionalLayer"
 ],
 "hierarchyTypes": [],
 "typingRelationTypes": [],
 "relationPropertySchema": {
   "type": {
```

```
    "key": "value",
    "index": 0
  }
},
"elementTranslationRules": {
  "CNNArchitecture": {
    "param": "F",
    "constraint": "(and (bool UUID_F) (= UUID_F 1))",
    "selectedConstraint": "(and (bool UUID_F) (= UUID_F 1))",
    "deselectedConstraint": "(= 0 1)"
  },
  "InputLayer": {
    "param": "F",
    "constraint": "(and (bool UUID_F) (= UUID_F 1))",
    "selectedConstraint": "(and (bool UUID_F) (= UUID_F 1))",
    "deselectedConstraint": "(= 0 1)"
  },
  "OutputLayer": {
    "param": "F",
    "constraint": "(and (bool UUID_F) (= UUID_F 1))",
    "selectedConstraint": "(and (bool UUID_F) (= UUID_F 1))",
    "deselectedConstraint": "(= 0 1)"
  },
  "DenseLayer": {
    "param": "F",
    "constraint": "(and (bool UUID_F) (= UUID_F 1))",
    "selectedConstraint": "(and (bool UUID_F) (= UUID_F 1))",
    "deselectedConstraint": "(= 0 1)"
  },
  "DropoutLayer": {
    "param": "F",
    "constraint": "(and (bool UUID_F) (= UUID_F 1))",
    "selectedConstraint": "(and (bool UUID_F) (= UUID_F 1))",
    "deselectedConstraint": "(= 0 1)"
  },
  "FlattenLayer": {
    "param": "F",
    "constraint": "(and (bool UUID_F) (= UUID_F 1))",
    "selectedConstraint": "(and (bool UUID_F) (= UUID_F 1))",
    "deselectedConstraint": "(= 0 1)"
  },
```

```
  "PoolingLayer": {
    "param": "F",
    "constraint": "(and (bool UUID_F) (= UUID_F 1))",
    "selectedConstraint": "(and (bool UUID_F) (= UUID_F 1))",
    "deselectedConstraint": "(= 0 1)"
  },
  "ConvolutionalLayer": {
    "param": "F",
    "constraint": "(and (bool UUID_F) (= UUID_F 1))",
    "selectedConstraint": "(and (bool UUID_F) (= UUID_F 1))",
    "deselectedConstraint": "(= 0 1)"
  }
},
"relationReificationTypes": [],
"relationTranslationRules": {
  "Optional": {
    "params": ["F1", "F2"],
    "constraint": "(>= UUID_F1 UUID_F2)"
  },
  "Mandatory": {
    "params": ["F1", "F2"],
    "constraint": "(= UUID_F1 UUID_F2)"
  }
},
"attributeTranslationRules": {
  "String": {
    "param": "id",
    "parent": "F",
    "values": "Xs",
    "template": "A",
    "constraint": "(enum (Xs) UUID_A)"
  },
  "Boolean": {
    "param": "id",
    "parent": "F",
    "template": "A",
    "constraint": "(and (bool UUID_A) (>= UUID_F UUID_A))"
  },
  "Integer": {
    "param": "id",
    "value": "V",
```

```
      "parent": "F",
      "template": "A",
      "constraint": "(and (int UUID_A) (= UUID_A V))",
      "unset_constraint": "(and (int UUID_A) (if (= UUID_F 0) (= UUID_A 0)) (if (>=
UUID_F 1) (>= UUID_A 1)))"
    }
  },
  "hierarchyTranslationRules": {},
  "relationReificationExpansions": {
    "params": ["Xs"]
  },
  "typingRelationTranslationRules": {},
  "relationReificationPropertySchema": {},
  "relationReificationTranslationRules": {},
  "relationReificationTypeDependentExpansions": {}
}
```


Let's look at an example of a concept formalized in the semantics of the CNN variability modeling language. In the case of the hyper-parameter generations, the constraint applied is “attributeTranslationRules” > “String” > “constraint”. Since most of the hyper-parameters are represented as Strings, this representation may differ in the Notebook generation since the Notebook will be able to execute and validate the whole CNN code. The String constraint “(enum (Xs) UUID_A)” replaces “Xs” with a list of “Possible values” and selects one of these values for each element hyper-parameter generation. The “enum” clause ensures that these values do not appear twice and that each variation remains unique.

Multiple variations can be generated by querying the language using a semantic translator tool using these semantics [23]. For example, to generate 20 configurations, we have two options in variaMos: write the query by ourselves using the JSON-based query language (Query tab in Figure 6), or use the graphical interface for query generation (Construct Query tab in Figure 6). “Solve N times” is selected next to the type of solver wanted, in this case, “MiniZinc” was chosen as a solver, as seen in Figure 6.

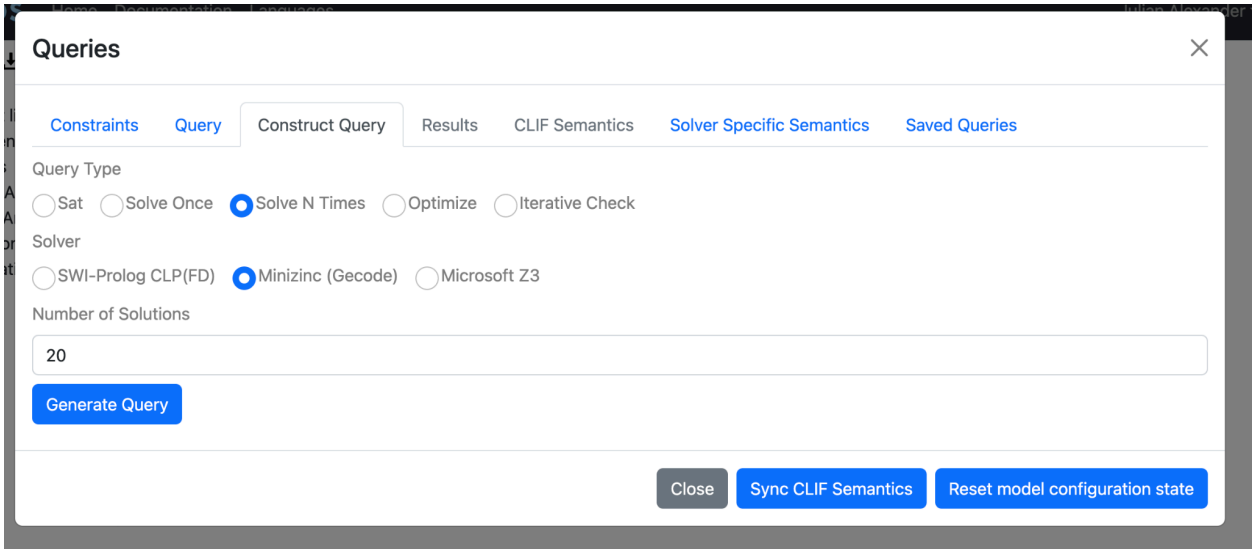


Figure 6. Query construction using the MiniZinc solver.

After the query is constructed and generated as presented in Figure 7, a list of variations will appear available as "Results" as shown in Figure 4.

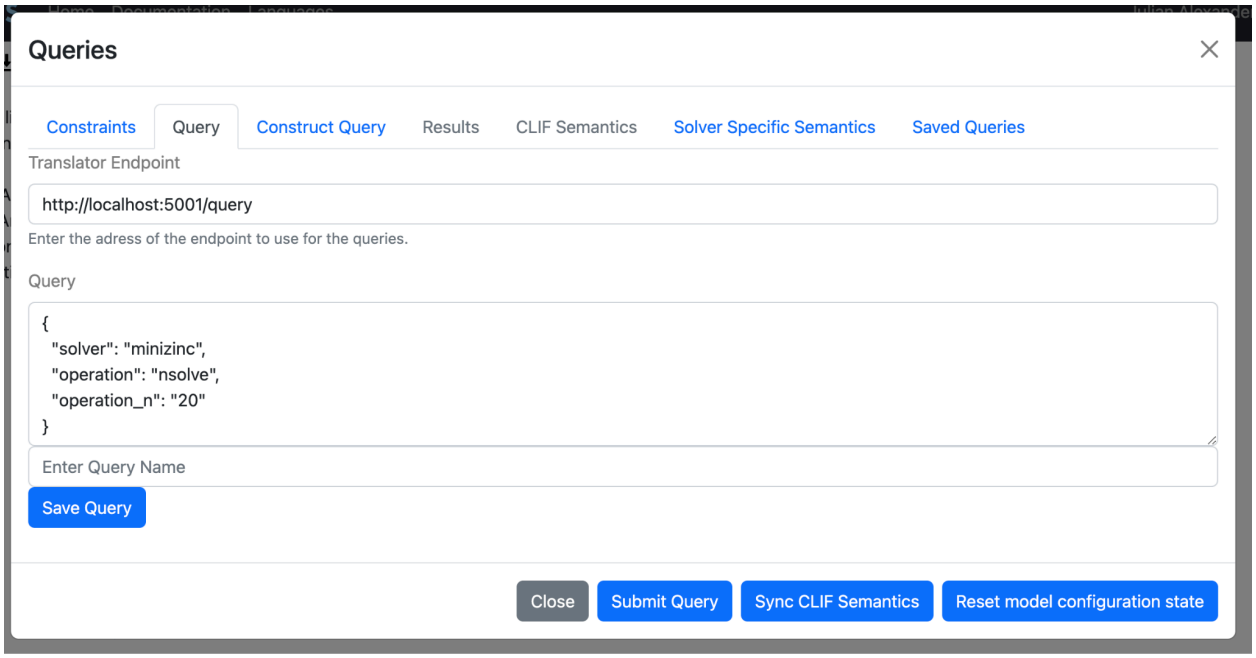


Figure 7. A query is generated using the Construct Query tab.

When any of the generated variations are selected, an architecture will be displayed with the hyper-parameter variations attributed to it as shown in Figure 8. From there, the architect can

choose to change any of the parameters or go right into generating a notebook based on architecture.

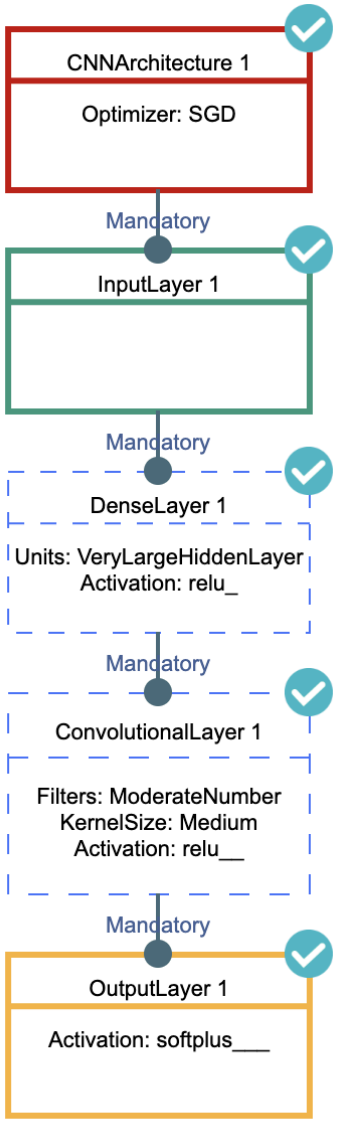


Figure 8. Selected architecture variation.

5.5. SEMANTIC TRANSLATION

The semantics written in CLIF are used to represent each CNN model (using the **CNN variability modeling language)** in an executable language such as MiniZinc or Prolog. The

corresponding codes of the current CNN architecture are automatically generated in MiniZinc and Prolog as presented in Figures 9 and 10.

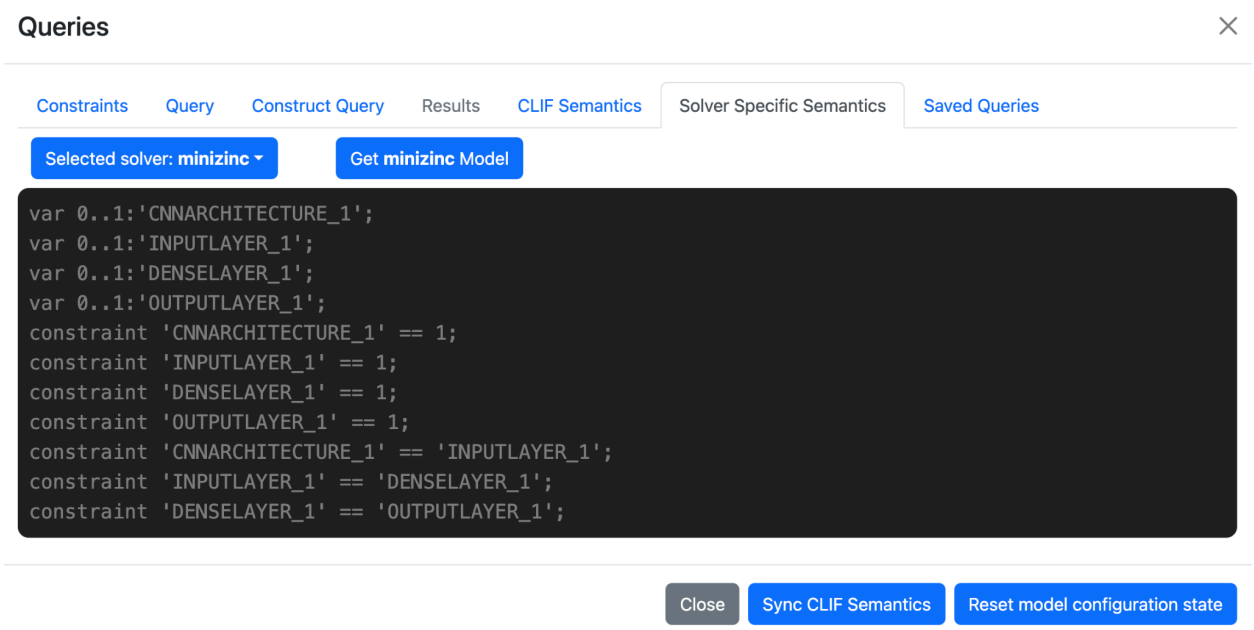


Figure 9. MiniZinc code of the current architecture.

Figure 10. Prolog code of the current architecture.

5.6. GENERATING NOTEBOOKS AND REPORTS

The VariaMos tool contains ways to hook up an external service per modeling language. The service was created and configured for the generation of CNNs and was named Architecture Generation System (AGS). Such a service provides the following functionality:

1. Convert CNN configuration models into Keras code
2. Create and Run a Jupyter Notebook
3. Generate and store accuracy reports

To convert the CNN models into Keras code, the service exposes an endpoint running a Flask server to receive the CNN Model Project in JSON format to extract the model elements from it as shown in Figure 11.

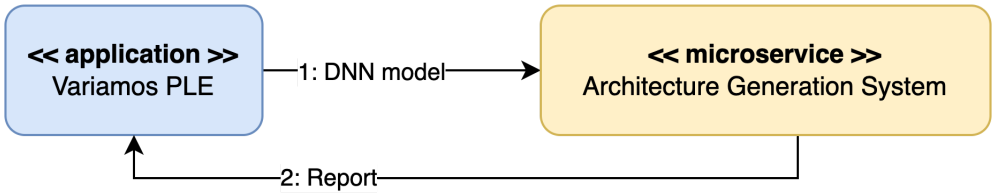


Figure 11. Architecture showing the VariaMos tool connecting with the AGS

5.6.1. CONVERT CNN LANGUAGE MODEL INTO KERAS CODE

After receiving the **CNN variability modeling language** project in JSON format, the service recreates a Keras Neural Network architecture by using the structure of the language elements. After that, the service replaces some of the selected values in the **CNN variability modeling language** with the proper hyper-parameter values using the specific mappings found in Table 2.

Language Model Attribute Value	Hyper-parameter Value
NoDropout	0.0

VeryLightDropout	0.1
LightDropout	0.2
ModerateDropout	0.4
StrongDropout	0.6
HeavyDropout	0.8
Full Dropout	1.0
NoPooling	None
LightAveragePooling	(2, 2)
ModerateAveragePooling	(3, 3)
StrongAveragePooling	(4, 4)
VerySmallNumber	8
SmallNumber	16
ModerateNumber	32
Many	64
VeryMany	128
ExtremelyMany	256
VerySmall	(1, 1)
Small	(3, 3)
Medium	(5, 5)
Large	(7, 7)
VeryLarge	(9, 9)
VerySmallStride	1
SmallStride	2
MediumStride	3
LargeStride	4
VeryLargeStride	5
NoActivation	None
SingleNeuron	1

CoupleNeurons	2
SmallLayer	10
SmallHiddenLayer	32
MediumHiddenLayer	64
LargeHiddenLayer	128
VeryLargeHiddenLayer	256
ExtraLargeHiddenLayer	512
UltraLargeHiddenLayer	1024
MegaLargeHiddenLayer	2048
BinaryClassification	2
DecaClassification	10
HectaClassification	100

Table 2. Attribute / Hyper-parameter mapping.

The output of the model recreation imports the CIFAR-10 data from the Keras dataset, converts the images into the proper format, passes the data to the Keras DNN model, and prints the accuracy and loss scores at the end. The Keras model code should look like the following code.

```
import keras
from keras import models
from keras.layers import Input, Dense, Conv2D, Flatten
from tensorflow.keras.utils import to_categorical

# Load the CIFAR-10 dataset
(x_train, y_train), (x_test, y_test) = keras.datasets.cifar10.load_data()

num_classes = 10
input_shape = (32, 32, 3)
x_train = x_train.astype('float32') / 255.0
x_test = x_test.astype('float32') / 255.0

# Convert labels to one-hot encoded vectors
y_train = to_categorical(y_train, num_classes)
```

```
y_test = to_categorical(y_test, num_classes)

print("x_train shape:", x_train.shape)
print(x_train.shape[0], "train samples")
print(x_test.shape[0], "test samples")

model = models.Sequential([
    Input(shape=input_shape),
    Dense(units=2, activation='relu'),
    Conv2D(filters=8, kernel_size=(1, 1), activation=None),
    Flatten(),
    Dense(num_classes, activation='sigmoid')
])
model.compile(optimizer='Adam', loss="categorical_crossentropy",
metrics=["accuracy"])

model.summary()
batch_size = 128

model.fit(x_train, y_train, batch_size=batch_size, epochs=3, validation_split=0.1)

score = model.evaluate(x_test, y_test, verbose=0)
print("Test loss:", score[0])
print("Test accuracy:", score[1])
```

5.6.2. CREATE AND RUN A JUPYTER NOTEBOOK

After the conversion process, the service creates a notebook including the source code and some Markdown documentation, this is possible by using the Python “Nbformat” library which allows the generation of Jupyter notebooks programmatically [25]. The finished notebook is stored as a file to be executed and sent back to the VariaMos tool, which will appear as a downloadable file to the user. The downloaded Notebook should look like Figure 12 when opened.

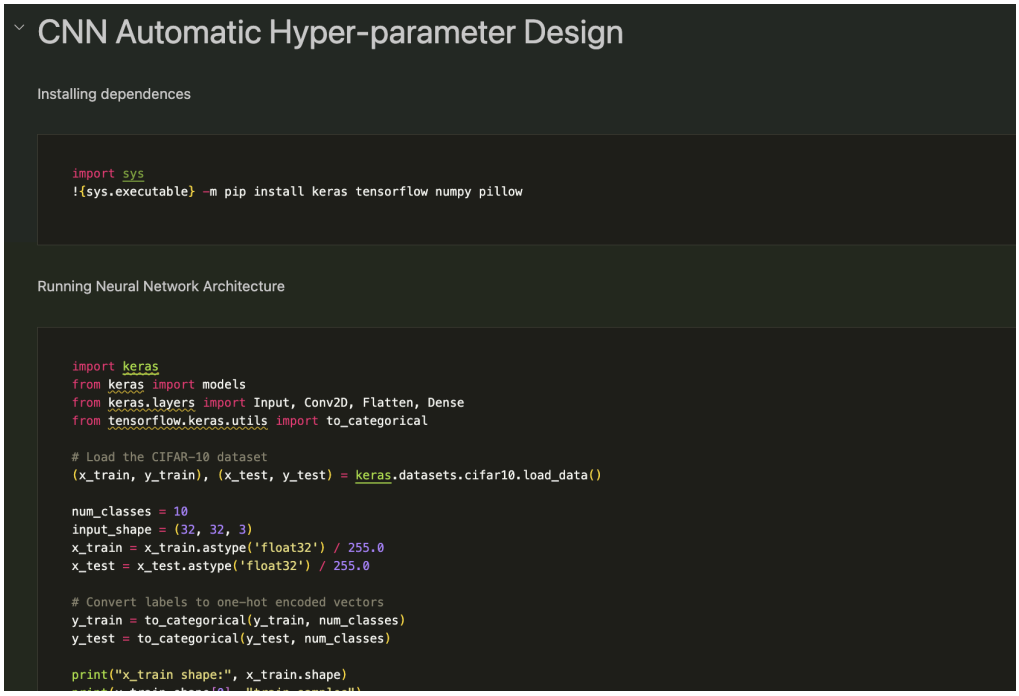


Figure 12. Generated Jupyter Notebook output.

5.6.3. GENERATE AND STORE ACCURACY REPORTS

While the notebook file is being executed, a report is made and stored in the service. This report contains the last execution date, timestamp, loss, and accuracy scores. This is for displaying a top 5 of results to compare to the current accuracy in subsequent architecture generations.

To show the best reports, the AGS uses a Pareto optimization algorithm to display the best reports based on the lower “loss” and the higher “accuracy” score. An example of the list of accuracy reports can be shown in Figure 13.

```
[
    {
        "loss": 1.889296293258667,
        "accuracy": 0.3488999903202057,
        "date": "2024-06-12 01:30:43.273515",
        "timestamp": 1718173843.273515
    },
    {
        "loss": 1.7950654029846191,
        "accuracy": 0.3813000023365021,
        "date": "2024-06-12 01:34:05.419634",
        "timestamp": 1718174045.419634
    },
```


Figure 13. Two entries of the accuracy report.

After the accuracy report is generated, a top 5 of the best reports is taken from the list of stored reports, and it is placed in the first cell of the Notebook (Figure 14). This provides a way for the architect to compare with previous executions.

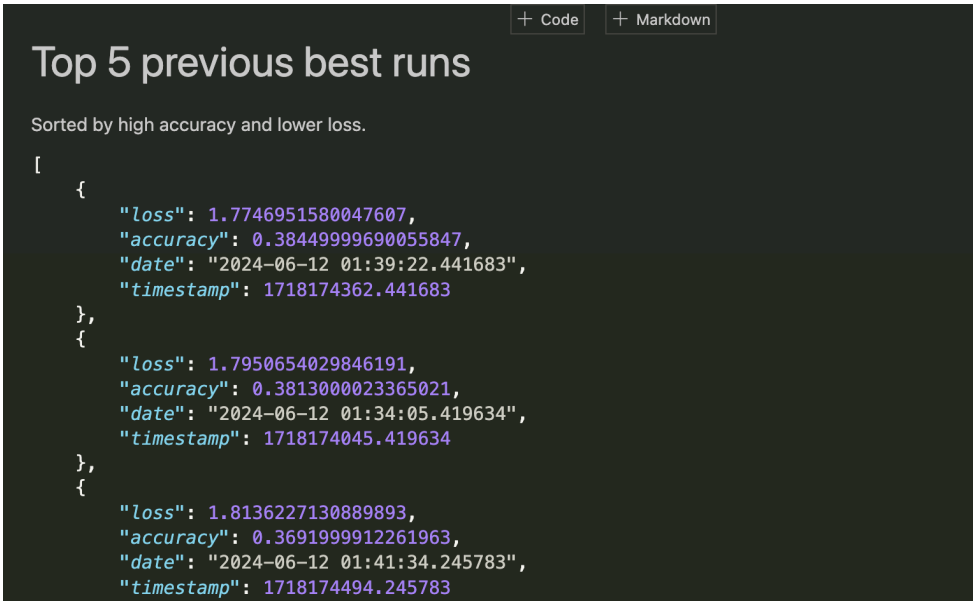


Figure 14. Top 5 accuracy report.

CHAPTER 6. PRELIMINARY EVALUATION WITH A PROOF OF CONCEPT

This chapter is about going through the process of modeling, generating, selecting, and exporting a CNN Architecture implemented on the VariaMos platform. This is to provide CNN architects with useful Notebooks that they can use and to help them experiment and test variations on a model more effectively. To do this, each section is going to explain in detail each of the steps it entails, which is to obtain a successful CNN model and a Notebook export in the process.

6.1. LANGUAGE ELEMENTS

To first create a CNN model in the VariaMos tool, the architect must right-click their VariaMos project and select “New Model” > “CNNArchitecture” and then select the new model file that will appear inside “Models”. Once there, the architect can select any of the language elements available, as shown in Figure 15. These elements represent a list of standard layers used in a CNN architectural model [21].

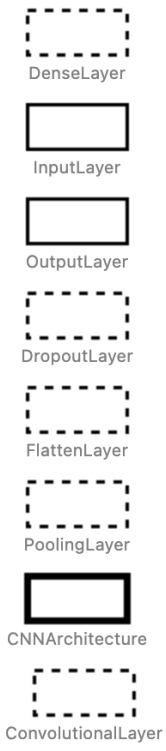


Figure 15. List of CNN Variability Modeling Language elements

When dragging each of the elements into the VariaMos playground, the layers are going to take a color related to their final position in a CNN architecture, the configuration layer (CNNArchitecture) will become red, the InputLayer green, the OutputLayer yellow, and the deep layers are going to be outlined blue with dotted lines. To simulate color, they are represented in the literature as shown in Figure 16.

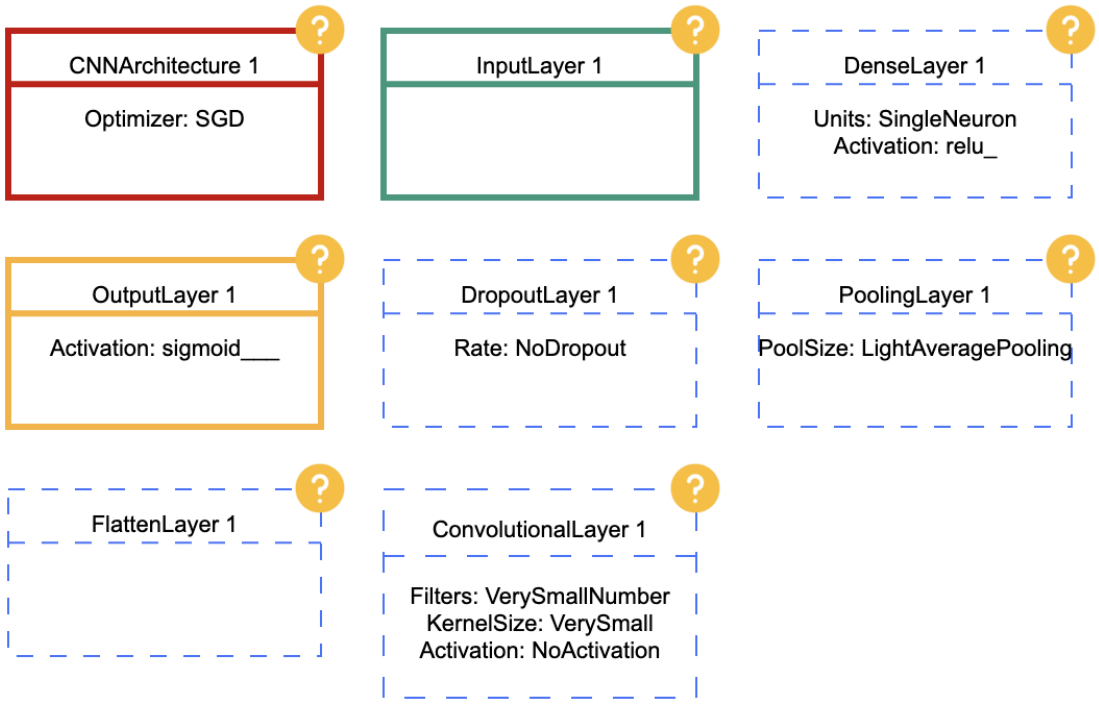


Figure 16. List of CNN variability modeling Language elements implemented

6.2. CONNECTING THE CNN

To connect the CNN layers, the architect can click them and drag the connection to another layer. Certain restrictions were put in place to avoid connecting incompatible layers (e.g., connecting CNNArchitecture layers to deep layers without an input layer). The current premise of this **CNN variability modeling language** is that Layers will not have branched behavior; although it is supported by the tool and some types of CNN architectures, this aspect is intended more for future iterations.

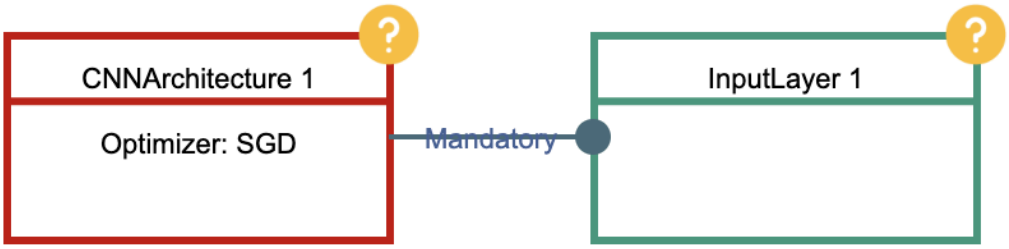


Figure 17. Connecting two compatible layers.

After knowing the basics of connecting layers, the architect can start creating a standard CNN architecture to test its variations. A good example of this architecture can be seen in Figure 18.

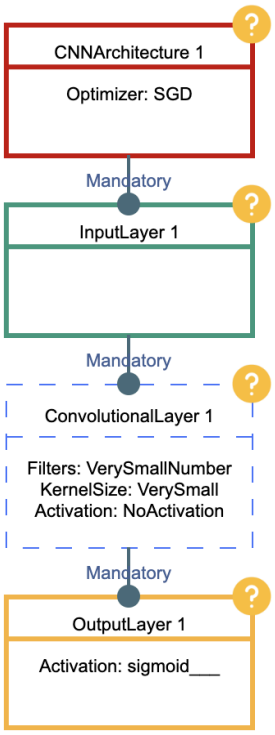


Figure 18. Simple convolutional architecture.

6.3. GENERATING AND SELECTING CNN ARCHITECTURE VARIATIONS

To generate CNN architecture variations, once the CNN Model is created, the architect needs to open the “Queries” tool, identified by the screw with the wrench icon. Here, the architect needs to go to the “Construct Query” tab, select “Solve N times”, select either “MiniZinc” or “SWI-Prolog” and type the number of solutions in the input text of such a name. An example is shown in Figure 5, and their results are shown in Figures 9 and 10.

After that, a query should be generated on the “Queries” tab, where the architect can select and submit requests and wait for the service to respond. If the service returns a successful answer, then the resulting variations will be shown in the “Results” tab, where they can be selected to see the respective variation rendered in the VariaMos playground (Figure 19).

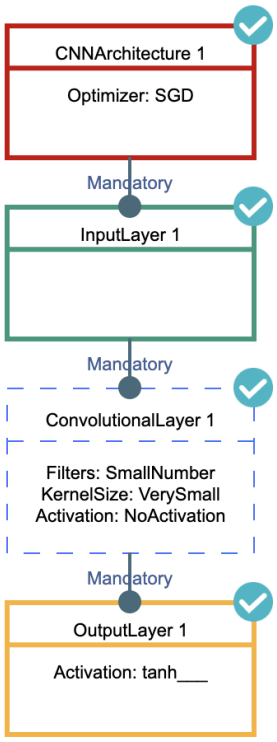


Figure 19. Simple convolutional architecture.

6.4. GENERATING A REPORT

To generate the notebook of the selected variation, the architect must right-click the model file created with the name “CNNArchitecture” under “Models/CNNArchitecture”, then select “Tools” > “Generate Report”. This will immediately generate and execute the resulting Keras Notebook and prompt a download.

After downloading and opening the Notebook, the architect should see a notebook with the CNN model that appears in Figure 20.

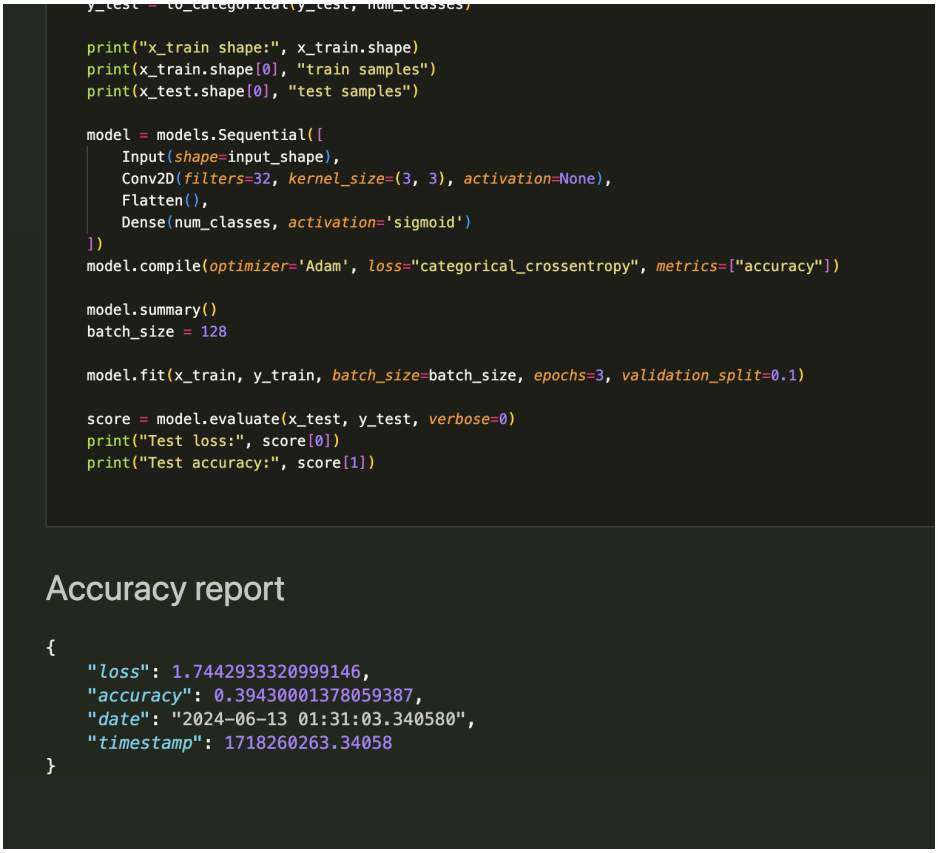


Figure 20. Simple convolutional architecture.

CHAPTER 7. PRELIMINARY EVALUATION OF THE LANGUAGE WITH A COMPARATIVE STUDY OF THREE CNN VARIABILITY MODELS

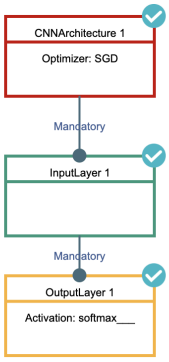
7.1. LANGUAGE MODEL EVALUATION

The model evaluation was done by generating three different models and comparing elements like the number of layers, variations number, code generated, and accuracy reports. For this preliminary evaluation, only three examples and a low number of neurons per layer were used, mainly due to the time constraints complex Notebook runs can take. Although low in accuracy, these low-resolution models can help showcase the differences in execution for each of the models.

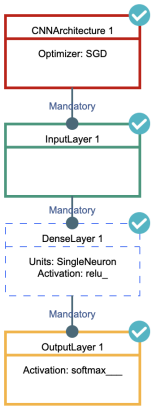
7.1.1. MODEL COMPARISON

To compare different CNN architectures, three models were created with distinct elements as presented in Table 3. For model 1, only two layers were used, mainly InputLayer and OutputLayer just to have a benchmark to compare against. For model 2, a DenseLayer was added between the InputLayer and the Output layer. And for model 3, a ConvolutionalLayer was added between the DenseLayer and the OutputLayer.

Model 1



Model 2



Model 3

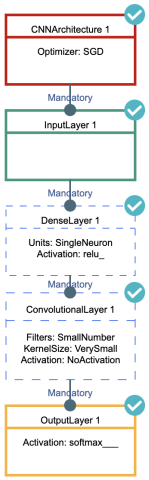
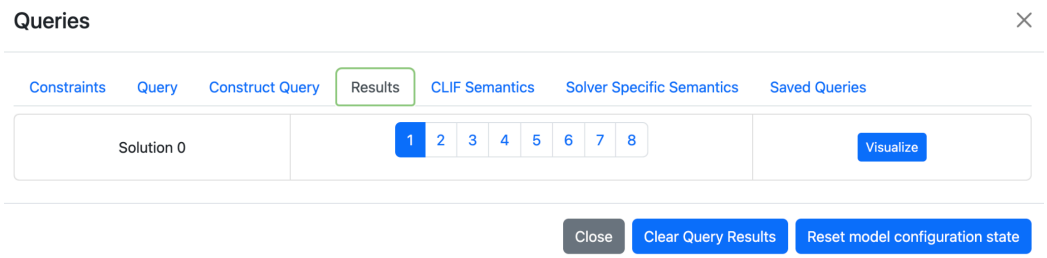


Table 3. Structure comparison of three different CNN models

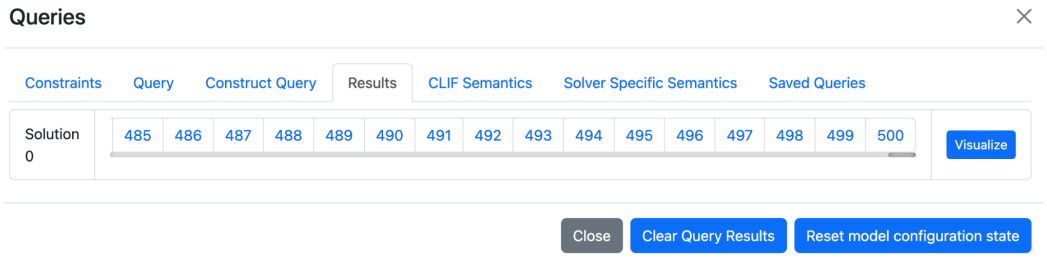
7.1.2. SEMANTIC VARIATIONS COMPARISON

For the comparison of the semantics, model 1 results in only 8 different variations, but models 2 and 3 result in more than 500 due to the combinatorial explosion of the hyper-parameters combined by the Dense and Convolutional Layers.

Model 1



Model 2



Model 3

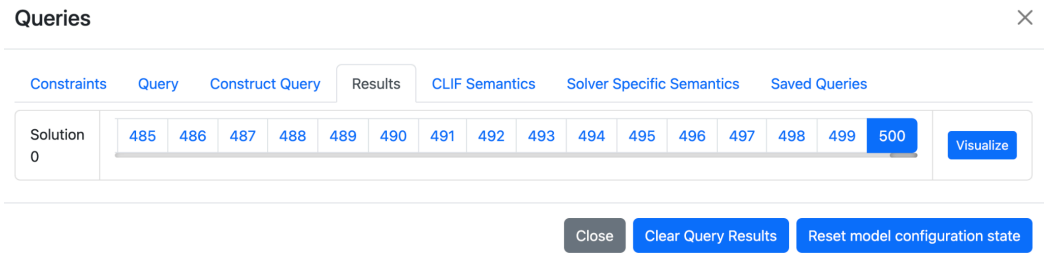


Table 4. Comparison of the variability of three models.

7.1.3. NOTEBOOK GENERATION COMPARISON

For the Notebook generation comparison, Model 1 shows the Input and Output layer with an additional Flatten layer added in the code generation to reduce dimensionality; Model 2 shows the additional Dense layer with its hyper-parameters “units” and “activation”; and Model 3 besides showing the Dense layer shows the Conv2D with its hyper-parameters “filters”, “kernel_size” and “activation”.

```
import keras
from keras import models
from keras.layers import Input, Flatten, Dense
from tensorflow.keras.utils import to_categorical

# Load the CIFAR-10 dataset
(x_train, y_train), (x_test, y_test) = keras.datasets.cifar10.load_data()

num_classes = 10
input_shape = (32, 32, 3)
x_train = x_train.astype('float32') / 255.0
x_test = x_test.astype('float32') / 255.0

# Convert labels to one-hot encoded vectors
y_train = to_categorical(y_train, num_classes)
y_test = to_categorical(y_test, num_classes)

print("x_train shape:", x_train.shape)
print(x_train.shape[0], "train samples")
print(x_test.shape[0], "test samples")

model = models.Sequential([
    Input(shape=input_shape),
    Flatten(),
    Dense(num_classes, activation='sigmoid')
])
model.compile(optimizer='Adam', loss="categorical_crossentropy", metrics=["accuracy"])

model.summary()
batch_size = 128

model.fit(x_train, y_train, batch_size=batch_size, epochs=3, validation_split=0.1)

score = model.evaluate(x_test, y_test, verbose=0)
print("Test loss:", score[0])
print("Test accuracy:", score[1])
```

Figure 21. Generated source code of model 1

```python
import keras
from keras import models
from keras.layers import Input, Dense, Flatten
from tensorflow.keras.utils import to_categorical

# Load the CIFAR-10 dataset
(x_train, y_train), (x_test, y_test) = keras.datasets.cifar10.load_data()

num_classes = 10
input_shape = (32, 32, 3)
x_train = x_train.astype('float32') / 255.0
x_test = x_test.astype('float32') / 255.0

# Convert labels to one-hot encoded vectors
y_train = to_categorical(y_train, num_classes)
y_test = to_categorical(y_test, num_classes)

print("x_train shape:", x_train.shape)
print(x_train.shape[0], "train samples")
print(x_test.shape[0], "test samples")

model = models.Sequential([
    Input(shape=input_shape),
    Dense(units=10, activation='selu'),
    Flatten(),
    Dense(num_classes, activation='softmax')
])
model.compile(optimizer='SGD', loss="categorical_crossentropy", metrics=["accuracy"])

model.summary()
batch_size = 128

model.fit(x_train, y_train, batch_size=batch_size, epochs=3, validation_split=0.1)

score = model.evaluate(x_test, y_test, verbose=0)
print("Test loss:", score[0])
print("Test accuracy:", score[1])
```

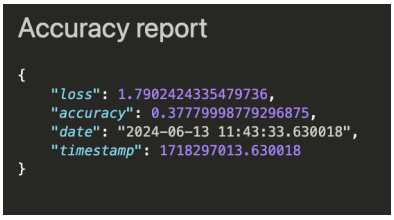
Figure 22. Generated Notebook source code of model 2

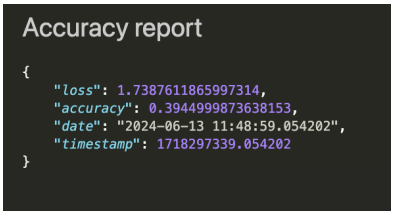
```python
import keras
from keras import models
from keras.layers import Input, Dense, Conv2D, Flatten
from tensorflow.keras.utils import to_categorical

# Load the CIFAR-10 dataset
(x_train, y_train), (x_test, y_test) = keras.datasets.cifar10.load_data()

num_classes = 10
input_shape = (32, 32, 3)
x_train = x_train.astype('float32') / 255.0
x_test = x_test.astype('float32') / 255.0

# Convert labels to one-hot encoded vectors
y_train = to_categorical(y_train, num_classes)
y_test = to_categorical(y_test, num_classes)

print("x_train shape:", x_train.shape)
print(x_train.shape[0], "train samples")
print(x_test.shape[0], "test samples")

model = models.Sequential([
    Input(shape=input_shape),
    Dense(units=10, activation='selu'),
    Conv2D(filters=8, kernel_size=(1, 1), activation='relu'),
    Flatten(),
    Dense(num_classes, activation='sigmoid')
])
model.compile(optimizer='SGD', loss="categorical_crossentropy", metrics=["accuracy"])

model.summary()
batch_size = 128

model.fit(x_train, y_train, batch_size=batch_size, epochs=3, validation_split=0.1)

score = model.evaluate(x_test, y_test, verbose=0)
print("Test loss:", score[0])
print("Test accuracy:", score[1])
```

Figure 23. Generated Notebook source code of model 3

7.1.4. REPORT COMPARISON

For the report comparison, Model 1 shows an accuracy of ~37% and a loss of ~179%; Model 2, since it contains an additional Dense layer increases accuracy and reduces loss, with an accuracy of ~39% and a loss of ~173%; and the Model 7, which includes a Convolutional neural network, also increases accuracy and reduces loss, showing an accuracy of ~40% and a loss of ~172%.





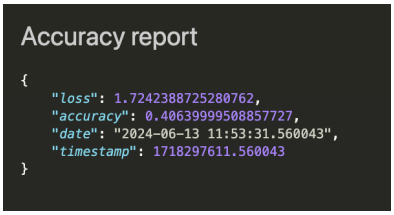


Table 5. Accuracy report comparison of three different models

For the top 5 reports obtained through Pareto optimization, the last model appears at the top due to its lower loss and higher accuracy than the other two.

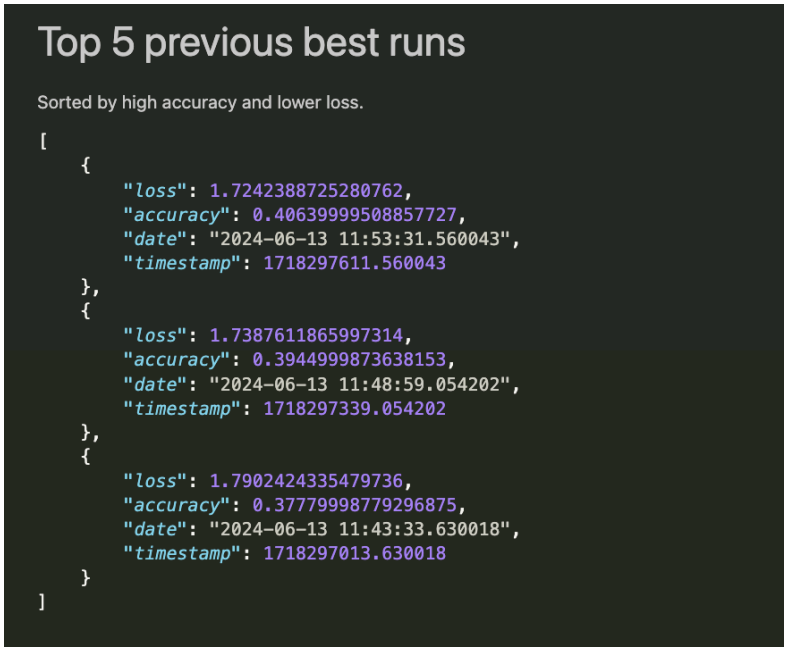


Figure 24. Top 5 reports of the previous 3 models.

After generating these three models, the approach can provide significant convenience when designing CNN architectures through a visual drag-n-drop modeler, insights about the number of possible architectural variations, auto-generated source code with a click of a button, and report comparison. This results in a very similar experience as the Computer-Aided Design (CAD) [27] and the low-code/no-code [28] tools widely used in the industry.

7.2. LANGUAGE SYNTAX AND SEMANTICS VALIDATION

The Abstract and Concrete Syntaxes were based on an existing VariaMos Language Model named “Feature Model with Attributes” which provides a versatile starting point for creating a new language since it provides elements that are displayed as properties when added to the playground, showed all the types of connections language elements can use, and included a working implementation of the language semantics in CLIF [23].

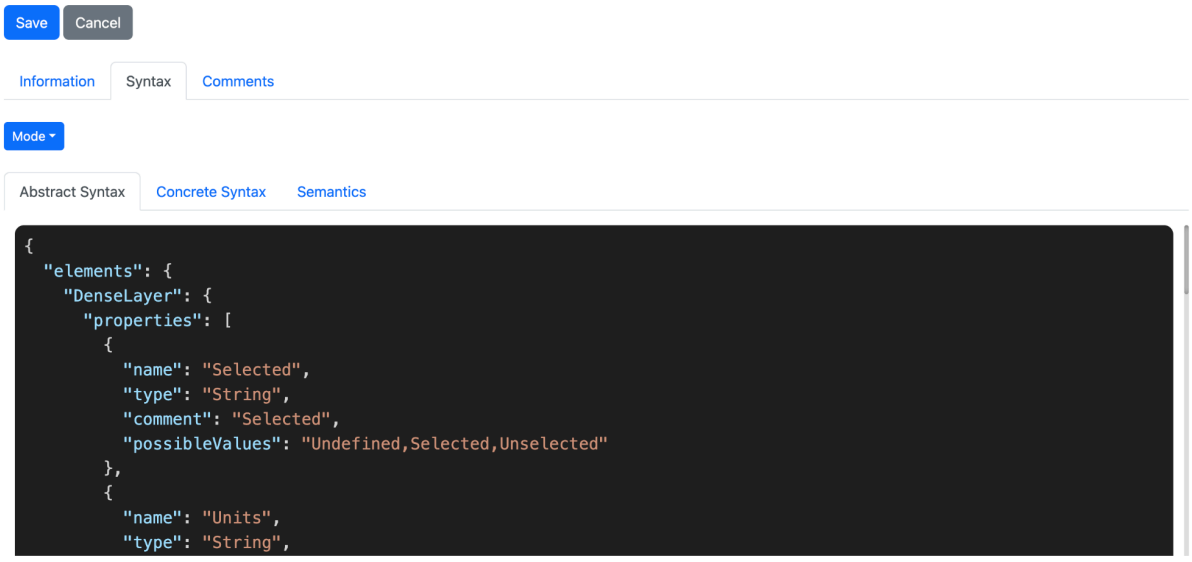


Figure 25. VariaMos abstract syntax shown in the VariaMos Language Manager

To test the Abstract Syntax, the VariaMos Language Manager module was used to load such a syntax, making it available to use on the VariaMos playground. Some iterations were needed until the desired number of attributes, elements, and restrictions were completely defined.

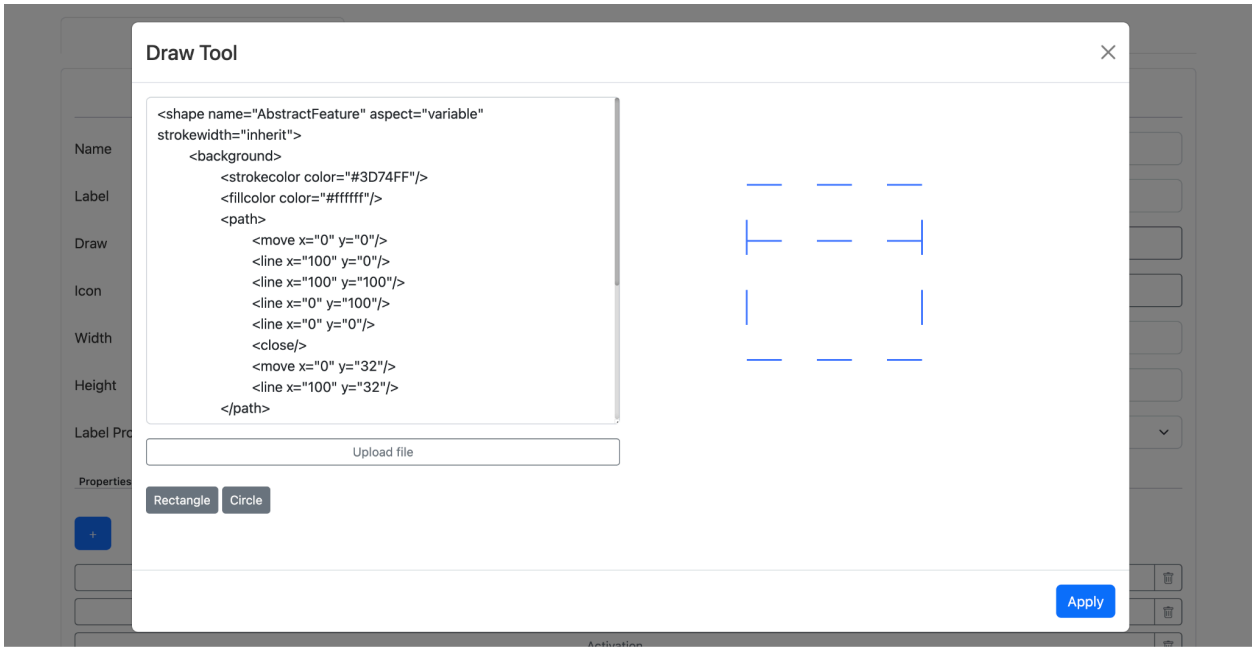


Figure 26. VariaMos concrete syntax in the VariaMos Language Manager Visual Editor

Similarly, with the Concrete Syntax, the VariaMos Language Manager offered a way to visualize the language elements; some of the aspects of the CNN layers were taken for the "Feature Model with Attributes" language. Additionally, the VariaMos Language Manager offered a way to update variables like styles, positions, and sizes, it allowed for bigger customization, resulting in the red outline colors used in CNNArchitecture and the more subtle blue dotted outline of the Dense and Convolutional layers as shown in Figure 25.

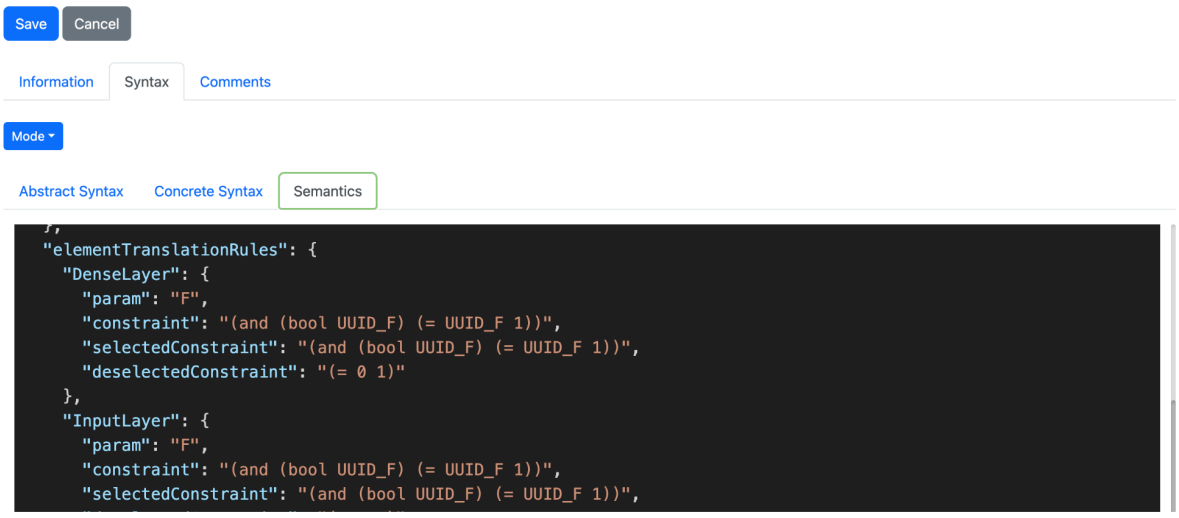


Figure 27. VariaMos semantics in the VariaMos Language Manager

The semantics used were also based on the “Feature Model with Attributes” language since its implementation offered attribute variability but only to attributes added after the model was used in the playground. To support native attributes, variability modifications to the Semantic Translator had to be made, extending the support of property variability to all the VariaMos languages.

7.3. TESTING THE ELEMENTS OF THE CNN MODELS

To verify that the CNN variability model and the services surrounding it will work as expected, a series of behavior-driven-development scenarios based on the industry-wide Given-When-Then structure [26] were used to ensure the objectives stated on this document and the language element attributes worked as expected.

The scenarios listed below were done in different system components and passed successfully the testing of the language elements.

Scenario: Executing Notebooks

- Given that the user is at the playground and has created a CNN model, the user selects Tools > Generate Report, and then the system should run the notebook.
- Given that “Generate Report” was selected. When the server is active, the system should receive the project data in a JSON format.
- Given that “Generate Report” was selected. When Notebook runs, the system adds a report based on the last Notebook run.
- Given that “Generate Report” was selected. When Notebook is generated, the Input Layer should contain the input shape of the date input.
- Given that “Generate Report” was selected. When Notebook is generated, the Output Layer should contain the output shape of the expected output.

- Given that “Generate Report” was selected. When Notebook is generated, the Output Layer should always contain a Flatten layer before it to reduce the dimensionality before reaching the output.

Scenario: Adding Layers to the VariaMos playground

- Given that the user is at the VariaMos playground, the user drags a CNNArchitecture element to the playground, Then the user should see a red outlined element in the playground.
- Given that the user is at the VariaMos playground, the user drags an InputLayer element to the playground, Then the user should see a green outlined element in the playground.
- Given that the user is at the VariaMos playground, the user drags an OutputLayer element to the playground, Then the user should see a yellow outlined element in the playground.
- Given that the user is at the VariaMos playground, the user drags a DenseLayer element to the playground, Then the user should see a blue dotted outline element in the playground.
- Given that the user is at the VariaMos playground, the user drags a DropoutLayer element to the playground, Then the user should see a blue dotted outline element in the playground.
- Given that the user is at the VariaMos playground, the user drags a FlattenLayer element to the playground, Then the user should see a blue dotted outline element in the playground.
- Given that the user is at the VariaMos playground, the user drags a PoolingLayer element to the playground, Then the user should see a blue dotted outline element in the playground.
- Given that the user is at the VariaMos playground, the user drags a ConvolutionalLayer element to the playground, Then the user should see a blue dotted outline element in the playground.

Scenario: Connecting Layers

- Given that the user has layers placed in the playground, the user connects the CNNArchitecture element with the InputLayer, Then the user should be able to do it.

- Given that the user has layers placed in the playground, the user connects the CNNArchitecture element with the OutputLayer, Then the user should see an error message.
- Given that the user has layers placed in the playground, the user connects the CNNArchitecture element with the DenseLayer, Then the user should see an error message.
- Given that the user has layers placed in the playground, the user connects the CNNArchitecture element with the DropoutLayer, Then the user should see an error message.
- Given that the user has layers placed in the playground, the user connects the CNNArchitecture element with the FlattenLayer, Then the user should see an error message.
- Given that the user has layers placed in the playground, the user connects the CNNArchitecture element with the PoolingLayer, Then the user should see an error message.
- Given that the user has layers placed in the playground, the user connects the CNNArchitecture element with the ConvolutionalLayer, Then the user should see an error message.

Scenario: Generating Report

- Given that the user is at the playground and has created a CNN model, the user selects Tools > Generate Report, Then the system should trigger a download of the Notebook.
- Given that I've downloaded the generated Notebook, the user runs the Notebook in Google Colab, Then the notebook should run without errors.
- Given that I've downloaded the generated Notebook, the user runs the Notebook in Visual Studio code, Then the notebook should run without errors.
- Given that I've made proper changes to Notebook, the user runs the Notebook in Visual Studio code, Then the notebook should run without errors.
- Given that I've downloaded the generated Notebook, the user opens the Notebook, Then the user should see a top 5 report at the top of the notebook.

- Given that I've downloaded the generated Notebook, the user opens the Notebook, Then the user should see the current report results at the bottom of the notebook.

Scenario: Generating Variations

- Given that the user is using the semantic translator, and has string attributes on an element, the system should generate variations on such attributes.
- Given that the user is using the semantic translator, the user generates 10 variations, Then the user should be able to select and see any of the variations.
- Given that the user is using the semantic translator, the user generates 50 variations, Then the user should be able to select and see any of the variations.
- Given that the user is using the semantic translator, the user generates 100 variations, Then the user should be able to select and see any of the variations.
- Given that the user is using the semantic translator, the user generates 400 variations, Then the user should be able to select and see any of the variations.

More specific versions of these tests were added as unit tests inside the internal AGS modules and ran continuously through the making of this approach to (i) guarantee their functionality and (ii) reduce at minimum the regression changes.

CHAPTER 8. CONCLUSION AND FUTURE WORK

Regarding the objectives that were accomplished in this document, the first objective, “Design a variability modeling language for CNN architectures that can be used to do random DNN hyper-parameter generation,” was achieved in Chapter 5 from Section 5.1 to Section 5.4. This was accomplished by the creation of a VariaMos language specifically designed for CNN architectures.

The second objective is to “Design and implement a solution to generate and execute CNN architectural code to provide metrics based on the CNN architectures generated to help the hyper-parameter search." This objective can be seen in Chapter 5 in Section 5.5 about Notebooks and reports. This objective was achieved by providing the **CNN variability modeling language** with a web service able to generate Jupyter Notebooks, run them, and create comparative reports from them.

For the third and last objective, “Provide a state-of-the-art of the current progress on the automatic design of CNN architectures and hyper-parameter search.” The accomplishment of this objective can be seen in Chapter 3 and is related to all the literature reviews done in the document.

Automatic Design of Convolutional Neural Networks is an area inside the fields of Machine Learning and Artificial Neural Networks that can offer a lot of opportunities to do work that can provide advances to the current state-of-the-art using different approaches. This area allows the combination of all sorts of fields, like genetic algorithms, statistical methods, and even Software Product Lines. This is done to solve specific problems derived from the manual work of DNN architecture design and to reduce the effort it takes to create optimal models by letting the machines deal with such iterations.

Automation is a process that has taken the human race far since the Industrial Revolution and has been a motivator to remove menial work and replace it with a meaningful one. The work done in this document reflects that philosophy by removing menial machine learning tasks that only a few have the experience to perform for meaningful results that many can access. With the continual progress in different areas of machine learning, neural networks, and generative

models, it is possible to see that the work done in this document can be improved many-fold with the inclusion of such tools, helping reduce the time it takes to create better models and keep democratizing access to them.

8.1. FUTURE WORK

The creation of this approach allowed running convolutional neural networks and delivering reports that can be compared with previous results, however, all of the networks created with this model are sequential. Future work could allow the creation of interconnected CNNs like state-of-the-art models, Generative Adversarial Networks, and quite possibly models like Transformers, which take more complicated connection paths than sequential neural networks. Additionally, including templates that the architect can use without needing to construct a model could further improve the current approach's usability.

REFERENCES

1. Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," Nature, vol. 521, no. 7553, pp. 436-444, 2015.
2. J. Schmidhuber, U. Meier, and D. Ciresan, "Multi-column deep neural networks for image classification," in *2012 IEEE Conference on Computer Vision and Pattern Recognition*, 2012, pp. 3642-3649.
3. A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, 2012, pp. 25.
4. R. M. Cichy, A. Khosla, D. Pantazis, A. Torralba, and A. Oliva, "Deep neural networks predict hierarchical spatiotemporal cortical dynamics of human visual object recognition," *arXiv preprint arXiv:1601.02970*, 2016.
5. J. Patterson and A. Gibson, *Deep learning: A practitioner's approach*. "O'Reilly Media, Inc.", 2017.
6. L. Alzubaidi, J. Zhang, A. J. Humaidi, A. Al-Dujaili, Y. Duan, O. Al-Shamma, et al., "Review of deep learning: concepts, CNN architectures, challenges, applications, future directions," *Journal of Big Data*, vol. 8, pp. 1-74, 2021.
7. J. Bergstra and Y. Bengio, "Random search for hyper-parameter optimization," *Journal of Machine Learning Research*, vol. 13, no. 2, 2012.
8. Z. Zhong, W. Wu, J. Shao, and C. L. Liu, "Practical block-wise neural network architecture generation," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2018, pp. 2423-2432.
9. E. A. de Oliveira Junior, I. M. de Souza Gimenes, E. H. M. Huzita, and J. C. Maldonado, "A variability management process for software product lines," in *CASCON*, 2005, pp. 225-241.
10. R. Mazo, C. Salinesi, and D. Diaz, "VariaMos: a tool for product line driven systems engineering with a constraint-based approach," in *24th International Conference on Advanced Information Systems Engineering (CAiSE Forum'12)*, 2012.

11. S. Li, Y. Sun, G. G. Yen, and M. Zhang, "Automatic design of convolutional neural network architectures under resource constraints," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 34, no. 8, pp. 3832-3846, 2021.
12. Y. Chen, K. Zhu, L. Zhu, X. He, P. Ghamisi, and J. A. Benediktsson, "Automatic design of convolutional neural network for hyperspectral image classification," *IEEE Transactions on Geoscience and Remote Sensing*, vol. 57, no. 9, pp. 7048-7066, 2019.
13. B. R. Boozarjomehry and W. Y. Svrcek, "Automatic design of neural network structures," *Computers & Chemical Engineering*, pp. 1075-1088, 2001.
14. T. Mariyama, K. Fukushima, and W. Matsumoto, "Automatic design of neural network structures using ais," in *Neural Information Processing: 23rd International Conference, ICONIP 2016, Kyoto, Japan, October 16–21, 2016, Proceedings, Part II 23*, Springer International Publishing, 2016, pp. 280-287.
15. S. Ghamizi, M. Cordy, M. Papadakis, and Y. L. Traon, "Automated search for configurations of deep neural network architectures," *arXiv preprint arXiv:1904.04612*, 2019.
16. J. Mellor, J. Turner, A. Storkey, and E. J. Crowley, "Neural architecture search without training," in International Conference on Machine Learning, pp. 7588-7598, PMLR, 2021.
17. B. Lu, J. Yang, L. Y. Chen, and S. Ren, "Automating deep neural network model selection for edge inference," in 2019 IEEE First International Conference on Cognitive Machine Intelligence (CogMI), pp. 184-193, IEEE, 2019.
18. F. Assunção, J. Correia, R. Conceição, M. J. M. Pimenta, B. Tomé, N. Lourenço, and P. Machado, "Automatic design of artificial neural networks for gamma-ray detection," IEEE Access, vol. 7, pp. 110531-110540, 2019.
19. F. Assunção, N. Lourenço, P. Machado, and B. Ribeiro, "Fast-DENSER++: Evolving fully-trained deep artificial neural networks," arXiv preprint arXiv:1905.02969, 2019.
20. F. Assunção, N. Lourenço, P. Machado, and B. Ribeiro, "DENSER: deep evolutionary network structured representation," Genetic Programming and Evolvable Machines, vol. 20, pp. 5-35, 2019.
21. A. Gulli and S. Pal, Deep Learning with Keras. Packt Publishing Ltd, 2017.

22. K. Simonyan and A. Zisserman, "Very Deep Convolutional Networks for Large-Scale Image Recognition," arXiv preprint arXiv:1409.1556, 2014.
23. C. Restrepo, J. Robin, and R. Mazo, "Generating Constraint Programs for Variability Model Reasoning: A DSL and Solver-Agnostic Approach", in Proceedings of the 22nd ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences, pp. 138-152, 2023.
24. K. Peffers, T. Tuunanen, M. A. Rothenberger, and S. Chatterjee, "A design science research methodology for information systems research," Journal of Management Information Systems, vol. 24, no. 3, pp. 45-77, 2007.
25. V. Potluri, S. Singanamalla, N. Tieanklin, and J. Mankoff, "Notably Inaccessible—Data-Driven Understanding of Data Science Notebook (In) Accessibility," in Proceedings of the 25th International ACM SIGACCESS Conference on Computers and Accessibility, pp. 1-19, 2023.
26. S. Rodriguez, J. Thangarajah, M. Winikoff, and D. Singh, "Testing Requirements via User and System Stories in Agent Systems," in Proceedings of the 21st International Conference on Autonomous Agents and Multiagent Systems, pp. 1119-1127, 2022.
27. B. F. Robertson and D. F. Radcliffe, "Impact of CAD tools on creative problem-solving in engineering design," Computer-Aided Design, vol. 41, no. 3, pp. 136-146, 2009.
28. T. Beranic, P. Rek, and M. Hericko, "Adoption and usability of low-code/no-code development tools," in Central European Conference on Information and Intelligent Systems, pp. 97-103, Faculty of Organization and Informatics Varazdin, 2020.