[KNP+19] E. Kusmenko, S. Nickels, S. Pavlitskaya, B. Rumpe, T. Timmermanns:
Modeling and Training of Neural Processing Architectures.
In: Conference on Model Driven Engineering Languages and Systems (MODELS'19), pp. 283--293, IEEE, Munich, Sep. 2019.
www.se-rwth.de/publications/

Modeling and Training of Neural Processing Systems

Evgeny Kusmenko, Sebastian Nickels, Svetlana Pavlitskaya, Bernhard Rumpe, Thomas Timmermanns
Chair of Software Engineering, RWTH Aachen University, Aachen, Germany, kusmenko@se-rwth.de

***Abstract*—The field of deep learning has become more and more pervasive in the last years as we have seen varieties of problems being solved using neural processing techniques. Image analysis and detection, control, speech recognition, translation are only a few prominent examples tackled successfully by neural networks. Thereby, the discipline imposes a completely new problem solving paradigm requiring a rethinking of classical software development methods. The high demand for deep learning technology has led to a large amount of competing frameworks mostly having a Python interface - a quasi standard in the community. Although, existing tools often provide great flexibility and high performance, they still lack to deliver a completely domain oriented problem view. Furthermore, using neural networks as reusable building blocks with clear interfaces in productive systems is still a challenge. In this work we propose a domain specific modeling methodology tackling design, training, and integration of deep neural networks. Thereby, we distinguish between three main modeling concerns: architecture, training, and data. We integrate our methodology in a component-based modeling toolchain allowing one to employ and reuse neural networks in large software architectures.**

***Index Terms*—deep learning, neural networks, model-driven software engineering**

I. INTRODUCTION

Machine learning is becoming a more and more ubiquitous technology in complex systems. In particular, high-tech disciplines such as autonomous driving, speech recognition, and the like are powered by deep neural networks [1], [2]. The high demand for deep learning technology has led to a multitude of competing deep learning frameworks ranging from low-level tensor-processing libraries ideal for experimentation and research to high-level Application Programming Interfaces (APIs) targeting application oriented developers.

When talking about a deep learning framework, we distinguish between its backend delivering the actual functionality and its front-end providing access to this functionality for the framework user. On the one hand, today's deep learning frameworks compete on the backend offering more features, faster training, hardware parallelization support, etc. On the other hand, the importance of the front-end visible to the users may not be neglected. Practitioners, particularly in research and rapid prototyping, tend to prefer easy-to-use languages which lead to faster results. This is why languages such as Python dominate the field of machine learning over C++, although the latter is mostly superior in terms of performance and energy consumption. Instead of dealing with low-level constructs, a robotics expert wants a language representing his or her domain as naturally as possible. Efforts have been made to facilitate access to deep learning technology, e.g. by representing layered architectures as YAML or prototxt descriptions or by providing high-level Python interfaces such as Keras and Lasagne.

Solving problems using neural networks is fundamentally different from classical software development and we believe that neural processing systems are best approached using appropriate Domain Specific Modeling Languages (DSMLs). The first contribution of this paper comprises a platform-independent **declarative artificial neural network modeling framework** with an emphasis on modular and re-usable design. The second contribution is the seamless integration of the proposed framework into the Component & Connector (C&C) language family EmbeddedMontiArc resulting in a **holistic model-driven systems and software development tool for the design of modular automotive, robotics, and cyber-physical systems incorporating state-of-the art deep neural networks as standard black-box components**.

The proposed deep learning framework and further materials including functional examples such as a direct perception-based autonomous driving application [3] and a cifar-10 [4] classification example generated to executable MXNet or Caffe2-based C++ code are provided at the paper-accompanying website[†].

II. RUNNING EXAMPLE

To give the reader an idea of the problems we would like to address in this paper, we are going to introduce a simple but easy-to-follow example.

Consider the C&C model in Fig. 1. The so called *MNIST-Calculator* receives six images as its input, each representing a hand-written digit (MNIST is a handwritten digit dataset well-established in the machine leaning community and often used for benchmarking learning algorithms [5]). The purpose of our MNISTCalculator is to detect the correct digit in each image, to compose the first and the last three digits into two three-digit numbers and to output their sum.

Despite the simplicity of the task, the model reveals a series of problems we need to deal with when incorporating machine learning techniques, particularly, deep neural networks, into production software.

First, we need to be able to use neural networks as modules in well-established design methodologies and to seamlessly integrate them with classical software components. In this example we have embedded the `MNISTDetector` component, depicted in violet, into a C&C architecture where other

This work was supported by the Grant SPP1835 from DFG, the German Research Foundation.

[†]http://se-rwth.de/materials/deeplearning

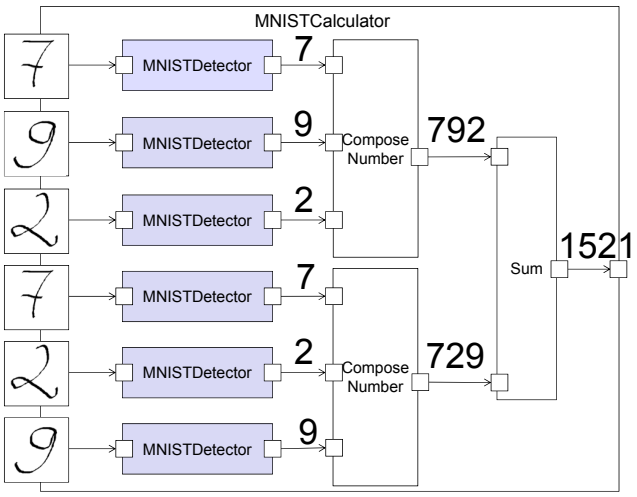

Fig. 1: C&C architecture of the MNIST Calculator example. Neural components (MNISTDetector) are highlighted in light violet.

components, depicted in white, are implemented as "standard code". This has two implications: we need to be able to package neural networks as self-contained components and, furthermore, neural networks need to offer interfaces compatible with classical component and port types, e.g. modeled using a C&C language such as Simulink [6], LabView [7], or EmbeddedMontiArc [8]. This means that we need to couple a C&C modeling language with a deep learning modeling framework.

Second, training neural networks is an expensive operation and should be performed only when really needed. A customer should not need to bother training an acquired software such as our `MNISTCalculator`. Instead, we as the software supplier want to deliver a well-trained out-of-the-box software package and, moreover, be able to hide the training process of the `MNISTDetector` network as our intellectual property. Consequently, the training process must be a compile-time issue at latest. This must be supported by a self-contained methodology supporting neural networks.

Third, as we can see in our example, neural networks can be employed at multiple positions of a software architecture. Thereby, the different neural network instances can have the same task (as it is the case in our example with each network detecting the digit represented by an image) or be responsible for completely different problems, e.g. one network dealing with image recognition and another for control in an autonomously driving vehicle. While the latter case is easy to handle, the former requires methodological support: in our example, all six `MNISTDetector` instances have the same functionality operating on different data. In many classical languages functionality reuse can be achieved at code level using instantiation. However, being a compile-time task as discussed above, network training must be orchestrated by an intelligent compiler toolchain. Such a toolchain must detect all occurrences of the `MNISTDetector` in our architecture and manage the training process and weight sharing automatically (the network should be trained only once). Moreover, if the neural network used in MNISTDetector is stateless, the compiler should create one flyweight instance of the network to represent all six `MNISTDetector` instances [9] in order to save loading six equal sets of parameters (it is not unusual for deep networks to have millions of such parameters).

III. REQUIREMENTS

Based on experience and pains we endured in a series of projects involving deep learning techniques, we have elicited a set of requirements for a model-driven development methodology of artificial intelligence (AI) systems in the embedded and Cyber-Physical System (CPS) domains. These requirements will help us assess the existing frameworks in section IV and are the basis for our solution presented in Sections V, VI and VII. We distinguish between requirements for the deep learning **l**anguage(s) and the integration **m**ethodology, denoted as **RL** and **RM**, respectively.

(RL1) Appropriate domain representation: we expect from a deep learning language or framework that it has an intuitive representation of the domain concepts and is aware of their semantics. Central concepts of the majority of today's network architectures are *neuron layers* and *layer connections*. Such concepts should be offered by a deep learning languages as first-level citizens and the developer should not be *obliged* to construct them from lower level concepts.

(RL2) Domain specific pragmatics: Information on the network structure should not be required to be made explicit in unambiguous contexts. The framework should rather be able to infer all missing information. Particularly, we expect syntactic sugar facilitating *layer stacking* and *layer independence*. Thereby, layer stacking denotes a repetition of the same layer which is a current pattern in neural processing. By layer independence we mean that a layer should be instantiated independently of all other layers. For instance, in many cases the number of neurons in a layer depends on the output of the preceding layer. Although, this information can be inferred automatically, many frameworks require it as explicit parameters. Making the layer size explicit, however, is a problem, as it needs to be updated manually whenever a change occurs in one of the preceding layers.

(RL3) Reusability and modularity: Structural patterns can be easily found in all kinds of neural networks and therefore require an adequate modularity concept. In particular, we expect means of *layer composition* enabling the developer to group several layers to a new custom layer; *network parameterization* for the adaptation of single layers but also of whole networks to a specific problem without having to change their internal structure.

(RL4) Separation of concerns: Deep network engineering consists of three major concerns, namely the *architecture definition*, where the structure of the network is defined; *network training*, where the neuron weights are optimized based on a given training set and which can include validation and model-selection; the intended *network execution* in the final system. A deep learning framework must separate these

concerns clearly, e.g. the structural model must not contain any information on the training as the right combination is highly-dependent on the application.

(RM1) Neural component integrity: to be able to integrate neural networks in complex robotics software, a neural network needs to exhibit a standardized interface, possibly accessible through a middleware such as Robot Operating System (ROS) [10]. The interface should enable easy integration with other components as well as integrity checks, e.g. making sure that the neural network is compatible with the image format provided by an upstream camera.

(RM2) AI awareness: the engineering methodology must be aware of the neural network components present in the system under development. It should know where to find training data and when to train or re-train individual networks. Unnecessary training needs to be avoided as far as possible.

(RM3) Platform independence: An AI framework should be compatible with a variety of platforms to increase reusability of neural networks and to broaden the spectrum of available features. Since the domain is highly active, new algorithms and features are added to the toolset of an AI engineer on a daily basis. However, mostly new features are first implemented in a small number of frameworks. The ability to exchange the AI backend without considerable effort is therefore crucial.

IV. RELATED WORK

The steadily rising interest in neural signal processing has led to a multitude of competing deep learning languages and frameworks. In this section we want to discuss a selection of available approaches to highlight the main differences as well as their advantages and drawbacks. Table I summarizes the most important facts. Note that as we are mainly interested in the frameworks' frontends, we refer to relevant literature for performance benchmarks [11].

a) Theano: Theano [12] has a Python API but all computationally expensive functions are implemented in C++ and CUDA. Despite being mainly developed and used for deep learning, it is rather a powerful general purpose framework for the manipulation of symbolic mathematical expressions. Theano represents math expressions as static computation graphs and can compute gradients of mathematical expressions through symbolic differentiation. A computation graph in Theano is a bipartite directed acyclic graph (DAG). The nodes of the DAG are either "variable" or "apply" nodes representing data and mathematical operations, respectively. Variable nodes have a static type (`float32`, `int64`, etc.) and usually appear as tensors. Variable nodes can be graph inputs, graph outputs and intermediate values. During the execution of a graph, intermediate and output values are computed from provided input values. The mathematical expressions in Theano can be written in a syntax similar to the Python library numpy [13]. A consequence of not being tailored to the deep learning domain is that Theano lacks explicit domain concepts violating **(RL1)** and **(RL2)**. Neural networks have to be constructed from low level operations which can be cumbersome and error-prone. Debugging static computation graphs is difficult as Python error messages are related to the instruction executing the graph but are unaware of the construction code actually leading to the problem. The lack of a C/C++ front-end as well as the absence of deployment libraries make Theano rather unsuitable for production systems.

b) Torch and PyTorch: Torch is a versatile numerical computing framework and machine learning library with a Lua front-end [14], [15]. It runs on the LuaJIT compiler but has a C implementation under the hood. The framework uses dynamic computation graphs offering several advantages over their static counterparts used in Theano. Torch builds and rebuilds these graphs while they are executed. Layers of a network can be executed line by line eliminating the discrepancy between construction and execution which we observed in Theano. Dynamic graphs enable standard imperative statements and control structures during the construction and execution of the network. The network architecture can be changed at runtime. The main disadvantage of dynamic computation graphs, however, is that they cannot be optimized in the same way as static graphs. Torch has a neural network library (nn) able to build neural networks as arbitrary acyclic computation graphs supporting automatic differentiation. A network has a forward function computing the output for a given input and a backward function calculating the gradient for each parameter. A network can either be built sequentially by adding one layer after another, or in a functional way by setting the input of each layer explicitly. PyTorch is a Python interface to Torch making it available for Python users.

c) TensorFlow: TensorFlow [16] follows the same general approach as Theano using static computation graphs allowing for symbolic differentiation of arbitrary math expressions. Therefore, it exhibits the same disadvantages as Theano: bad error messages and a complex API. To overcome these problems, TensorFlow is often used in conjunction with the higher level library Keras [17] featuring a more domain oriented, well-readable representation of layered networks. Extensions such as the eager execution environment enable the execution of operations in an imperative define-by-run style similar to Torch.

d) Caffe & Caffe2: Caffe was developed by the Berkeley Vision and Learning Center (BVLC) as a C++ framework [18]. It focuses on the efficient implementation of Convolutional Neural Networks (CNNs) and an easy out-of-the-shelf deployment of pretrained models. Caffe was one of the most popular deep learning frameworks especially in the industry due to the fact that it was the first framework which offered an easy way to share network architectures and trained state-of-the-art networks. Caffe provides interfaces for C++, Python, Matlab, as well as command line tools for training and prediction. The network is defined as a prototxt file. Thereby, four different model types are used, leveraging the separation of concerns principle, cf. **(RL4)**: a network description for training, the network architecture for deployment, the hyperparameter configuration, as well as the stored dataset mean which is computed automatically by Caffe for the sake of dataset normalization. The training data can be stored in a

high-performance database such as LevelDB [19], LMDB, or HDF5 [20]. Caffe uses two descriptions for the same network because the model for training contains additional information like the name of the dataset, the initialization parameters of each layer and the used loss function. Caffe focuses on the domain as required by **(RL1)**, but in contrast to Theano and Torch it does not provide low-level math operations for network construction. Thus, it is not possible to create new loss functions or new layers without extending Caffe itself. One of the main drawbacks is the verbosity of the prototxt data format lacking to fulfill **(RL2)** and **(RL3)**, making the construction of large networks such as the ResNet [21] cumbersome and difficult to read.

The successor, Caffe2, has abandoned the prototxt format and relies on a Python interface. Furthermore, it refrains from the layer-based modeling of Caffe and introduces the concept of operators. The latter can represent complete layers or low-level math operations making Caffe2 more of a general purpose framework similar to Theano and Torch.

e) MXNet / Gluon: Similar to Theano and TensorFlow, MXNet [22] supports static computation graphs and automatic differentiation on symbolic functions. Additionally, the *autograd* package enables automatic differentiation on *NDArray* operation graphs using the define-by-run principle. MXNet is available on all major operating systems and offers APIs for multiple languages. It is possible to define custom loss functions with symbolic expressions in MXNet, but widely used ones are predefined and combined with the prediction output in special optimized output layers. Thus, the model looks the same for training and prediction. MXNet stores the constructed and the trained network in two separate files: the first one is a JSON description of the network architecture, while the second one contains the weights of the model in a binary format. For deployment MXNet offers a C++ based prediction API providing functions to load an already trained model and to apply it to a given input. MXNet offers the ability to amalgamate the complete prediction library into a single file including the necessary dependencies, which facilitates the deployment in mobile and embedded systems. MXNet comes up with two interfaces, the rather low-level Symbol API and the newer Gluon frontend. The Symbol API can only be used to build static computation graphs leading to the same disadvantages as Theano and TensorFlow: obscure error messages and a complex API. The newer Gluon API can be used for both dynamic and static computation graphs without having to write the same code twice. A user can debug the network with its dynamic computation graph and hybridize it into a static computation graph to speed up the execution and export it later.

f) Keras: Keras [17] is a high level Python library focusing more on deep learning concepts than the frameworks listed above at the cost of flexibility. It does not provide an own backend but can use Theano, TensorFlow, CNTK [23], or MXNet instead, thereby fulfilling **(RM3)**. Due to its domain orientation and simplicity, Keras is one of the most popular deep learning frameworks today. A network can be constructed either as a sequential model by listing all layers of the network or in a functional way, where the output of a model can be passed as an argument to other layers. Keras saves the architecture of a network either in a JSON or YAML file. Weights are stored in an HDF5 database.

g) Matlab Neural Network Toolbox: Mathworks Matlab is a matrix-based language focusing on engineering domains [6] and providing deep learning functionality through its Neural Network Toolbox [24]. In Matlab a network is constructed by listing layers sequentially in a vector. This list is then transformed into a graph using the `layergraph` function. The Neural Network Toolbox constructs a neural network as a static directed acyclic graph of layers. To create non-sequential architectures, additional layers can be added to the graph with the `addLayers` function. Initially, added layers are not connected to any node of the graph. New edges in the DAG have to be explicitly created by calling the `connectLayers` function, which expects the graph itself as well as the source and target layers as its arguments. A complete network can then be trained with the `trainNetwork` function by providing the data and training options.

The toolbox supports all kinds of feed-forward and certain types of recurrent or *dynamic* neural networks. It also offers pretrained state-of-the-art networks for easy reuse. For training, it is possible to use the *GPU-coder* of Matlab to run the computationally expensive process on one or multiple GPUs. However, the Neural Network Toolbox does not offer symbolic differentiation. To define a custom layer or loss function, it is necessary to implement both the *forward* and the *backward* functions for the gradient. Neural networks can be encapsulated into Simulink [6] components making them suitable for component-based system design according to **(RM1)**. However, the neural network code can be mixed with other unrelated code leading to dirty components. Hence, there is no AI awareness as required by **(RM2)**, i.e. the Simulink architecture does not know how and when to train its networks automatically. This needs to be declared by the developer explicitly.

Although, the presented frameworks cover our requirements at least partially, the realizations tend to be too complex and require a lot of boilerplate code. Lacking *separation of concerns* leads to network descriptions being mixed with training parameters and file system access code for loading weights. *Layer stacking* and *composition* can mostly be realized using the host language, e.g. Python. The developer needs to take care of the neural networks present in a large system and to decide when to train and to retrain them. The most domain specific deep learning frontend is provided by Caffe. However, it suffers from a lack of modularity and reusability concepts leading to massive amounts of code needed to define practical network architectures.

For an objective comparison, we gathered implementations of the well-known ResNet152 CNN for image classification at our additional materials website[†]. The ResNet152 is a representative of the family of residual networks tackling the issue of vanishing gradients during training of very deep

TABLE I: Comparison of established deep learning frameworks and languages, √: yes, P: partially, -: no, *: depending on chosen backend, **: provided by extensions.

Deep learning framework	C&C integration	Type-safe component interface	Directed acyclic graphs	Low-level operators	Layer composition	Layer stacking	Layer independence	Parameterization	Interfaces	Static/Symbolic computation	Dynamic/Imperative computation	General purpose framework	Mobile deployment
MontiAnna (this paper)	√	√	√	-	√	√	√	√	MontiAnna	*	*	-	*
Theano	-	-	√	√	√	√	√	√	Python	√	-	√	-
Torch	-	-	√	√	√	√	-	√	Lua, C	-	√	√	-
PyTorch	-	-	√	√	√	√	-	√	Python	-	√	√	-
Caffe	P	-	√	-	-	-	√	-	Command Line, C++, Matlab, Python	√	-	-	-
Caffe2	-	-	√	√	√	√	-	√	Python, C++	√	-	√	√
TensorFlow	-	-	√	√	√	√	√	√	Python, C++	√	-**	√	√
MXNet	P	-	√	√	√	√	√	√	Python, C++, R, Julia, Matlab, Go, Scala, Perl, JavaScript	√	√	√	√
Keras	-	-	√	-	√	√	√	√	Python	√	*	-	*
Matlab NNT	√	-	√	-	P	P	√	√	Matlab	√	-	-	-

architectures by introducing residual blocks [21]. Most frameworks require several hundreds of lines of code to model ResNet152. In the following chapter we introduce a deep learning language family enabling us to express the very same network architecture in no more than 33 lines of code.

V. MontiAnna

In this paper we introduce a holistic deep learning modeling framework called MontiAnna. An overview of the MontiAnna framework is given in Fig. 2.

A. Modeling Languages

First, we identify three concerns which can and should be modeled independently to ensure a high degree of re-usability and maintainability keeping the models clean and concise.

a) Network architecture: The first thing coming to mind when designing a neural network is its actual architecture consisting of neurons, mostly organized as layers, and connections between the neurons defining the data flows. Moreover, we need to be able to assign specific tasks to some neuron layers, e.g. making them use a particular activation function or share weights according to some appropriate neural patterns, as is often done in image recognition. We will cover modeling network architectures in section VI in detail.

b) Network training: Having even the best network architecture for a specific task is meaningless if it is not trained appropriately. Network training is a composition of optimization algorithms which needs to be modeled and parameterized itself. Defining the training separated from the network architecture keeps the models concise and exchangeable. The developer can adapt the training procedure without touching the architecture or combine existing architectures and training models without changing the models at all.

c) Dataset model: Finally, to train a network the compiler requires information about where to find the training data and how to load it. There are various ways to store and read training datasets, some prominent technologies being high performance databases such as HDF5, LevelDB, and LMDB [25], [26]. Furthermore, the dataset needs to be subdivided into training and test data; we might want to use only parts of the data or even skip training if the dataset has already been learned. Loading a dataset or switching from one database type to another requires specific boilerplate code to be written by a deep learning engineer in languages like Python. In our generative framework, we concentrate information regarding the datasets in a declarative dataset model. This model is not to be confused with a *data model* capturing types and their relations, e.g. as a UML class diagram.

B. Model composition

Although the three modeling sub-domains are orthogonal, only together they represent a neural processing system. Therefore, we need to merge them into a single composed model before we can generate the actual application code. The resulting *composed model* serves as a basis for consistency checks, but also complex context conditions. For instance, we need to check whether the data contained in the specified dataset fits the given network architecture.

The training model must fit the architecture as well: consider the loss function we want to use during training in order to evaluate how well our network performs. Being a training aspect, the loss function is defined in the training model. However, certain loss functions are designed for specific kinds of network output. For instance, cross-entropy, an information theoretic loss function, is designed to measure the distance between two probability distribution. It is therefore mainly used in classification networks where each of the possible

classes is assigned a certain probability. This means that the network output must be a valid probability vector summing to one. This is usually ensured by applying a specific scaling function, e.g. the *softmax* function to the network output, by adding a softmax layer as the last layer of the network architecture. Hence, we have an inter-model check verifying that the training algorithm used makes sense when paired with the underlying architecture.

The composed model is a data structure which is assembled from the abstract syntax trees (ASTs) and symbol tables of the respective (parsed) partial models providing an infrastructure to navigate from the elements of one model to those of the others, e.g. from the AST node representing the training algorithm to the symbol representing the output layer of the architecture. This is realized using the language aggregation mechanism of the MontiCore language workbench [27]. Once the composed model is assembled and validated, it is passed to the actual code generator.

C. Generated Artifacts

The intention of this work is a better modeling language support for deep learning engineers. Instead of coming up with an own backend, we decided to take the generative approach transforming our (composed) models to the well-established frameworks discussed in section IV. The compiler layer has a clear and abstract API making the concrete compiler easily exchangeable. This enables us to steadily extend our backend support (currently we have implementations targeting the MXNet Symbol API, Gluon, TensorFlow and Caffe2). The task of the code generator comprises further checks of the composed model, ensuring that the chosen backend supports all required functionality. This is particularly convenient if backends are maintained with different effort or if a backend supports exclusive functionality which cannot be provided in other backends. For instance, Google has recently released an implementation of MorphNet, a regularization technique which can be used to shrink neural networks to fulfill resource constraints, based on the TensorFlow framework [28], [29]. To add MorphNet support to our framework, we first need to make it available in the syntax, i.e. at modeling level. Second, we need to adapt the code generators to map models using MorphNet to executable code. This functionality however can only be provided by a TensorFlow backend, i.e. through a TensorFlow generator. This means in turn that the modeling languages are completely decoupled from the compiler in terms of features.

Once, it is ensured that the composed model can be mapped to the solution space of the chosen backend, model instantiation and training artifacts are generated. As we aim at a fully model-driven solution, our goal is to avoid any hand-written general purpose language (GPL) code. Therefore, the compiler generates three main types of artifacts, which we are going to discuss in the following paragraphs. Additionally, the generator produces CMake build files to facilitate integration and assembly of the generated software.

a) Network Creator: The so called `NetworkCreator` is an intermediate product, which contains the program creating the network of the modeled neural architecture using the chosen backend framework, e.g. MXNet.

b) Network Trainer: Once the modeled network is created by the `NetworkCreator`, it is trained by another generated program: the `NetworkTrainer`. The variability of this artifact is mainly governed by the training and the dataset model introduced above. It receives the network object created by the `NetworkCreator` and performs the training. The result is a serialized representation of the trained network, which can be loaded in an application to be used for its actual purpose, e.g. prediction.

However, network training is not always necessary. After each training, the framework stores a meta-data file including the creation date and the hash value of the database used in the repository of the respective model. Whenever generation of the model is requested, the generator checks if a trained network artifact is available. If yes, it checks whether the database used has changed by comparing its actual meta-data with the stored meta-data file. If this is not the case and the model artifacts have not changed either, the generation of the `NetworkCreator` and `NetworkTrainer` are skipped and the network API reuses the old serialized network. This is a basis for regarding neural network artifacts as versionable archives, which can be deployed and reused as dependencies in build management systems such as Maven.

c) Network API: Eventually, we would like to integrate the obtained neural network into a software architecture as a module or a library, not worrying about its internal structure or about how it was trained. The network API artifacts are generated C++ code consisting of an execution interface as well as an implementation to load and run the trained network.

Interestingly, although our generator toolchain is supposed to produce compiled native code, it is still easier to create and train the networks in Python. Python APIs are often much better documented and in some cases provide more functionality than their C++ counterparts. Hence, for our supported backends, network creation and training are performed in Python while the network API is a C++ artifact, which is embedded directly into the target architecture such as the `MNISTCalculator`.

VI. Neural Architecture Modeling

A. Convolutional neural networks

We introduce the main concepts of our neural architecture modeling language using CNNs, a class of neural networks widely-used for a variety of image processing problems [30]. CNNs are relatively easy to understand, as they mostly have a simple feed-forward, layer-based structure.

Consider our hands-on example modeling the ResNet152 [21] in Fig. 3. The header of the network is defined in L.1 using the `architecture` keyword followed by the network's name and a list of parameters, each including a default value. The network can be adapted easily to alternative color spaces or input sizes by changing these parameters.

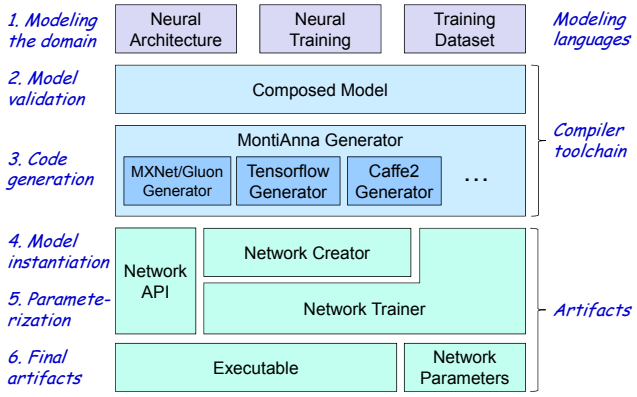

Fig. 2: Layers of the MontiAnna framework representing the modeling languages, compiler toolchain, and the resulting artifacts.

L.2 and L.3 define a strictly typed interface, i.e. the input and the output of the network, respectively, facilitating its integration into a software architecture later on. We employ the abstract, math oriented type system presented in [31] featuring the primitive types `Z`, `Q`, `C`, `B` to represent integers, rational, complex, and Boolean values, respectively. The AI engineer does not need to care about the actual implementation of the data types which, by the way, may differ depending on the deep learning backend, the system architecture, or the operating system. The language automatically chooses an appropriate mapping for the developer and deals with numerical inaccuracies. To further restrict the type, the developer can define a *range* as well as the *dimensionality*. Our `image` input is declared as a third order tensor with the dimensions `channels`, `height`, and `width` inspired by LaTeX syntax. Furthermore, each entry of the tensor is restricted to the values between 0 and 255 as is typical for color values.

The network architecture is constructed in L.4-18 from *neuron layers*, which, according to **(RL1)**, are the main building blocks or the so called first-level citizens in our language. MontiAnna differentiates between two kinds of layer types: predefined and custom layer types. Predefined layer types are atomic elements provided by the language. In our example you can find the predefined layer types `Pooling`, `Convolution`, `FullyConnected`, etc., which are indispensable in the CNN domain. On the other hand, custom layer types can be defined using the `def` keyword enabling the composition and reuse of networks or parts thereof, cf. **(RL3)**.

The network is assembled using connect operators defining the data flow between the layers. The architecture can be regarded as a single expression which is a major difference compared to the frameworks discussed in section IV. Most frameworks either create connections by listing layers in a sequential order (e.g. Keras, Matlab) or create them explicitly by declaring names (e.g. Matlab, Caffe) or variables (e.g. Keras, TensorFlow, MXNet) as inputs. We employ two data flow operators, namely the *serial* connection `->` and the *parallelization* operator `|` to assemble the neural network. Using a layer type name as an operand in the network expression creates an *anonymous* layer instance of this layer type.

In a serial connection `a -> b` the output stream of the left operand is fed into the right operand as its input. All purely sequential architectures, e.g. the LeNet [32], can be constructed using the serial connection operator exclusively. Note that it is also the most frequent connect operator in our ResNet152 example.

The parallelization operator `|` splits the operands into separate groups having the same input but not connected to each other. This operator can be applied successively to create multiple parallel groups and a group can also be empty to implement a *skip connection*. The parallelization operator has a lower precedence than the serial connection. Therefore, it is necessary to use parenthesis to be able to combine the outputs of these groups. Let m be a layer type, then the expression $(m()|m()\text{->}m()|m())\text{->}$ creates three parallel groups which are combined in a *parallelization block*. The language restricts the number of outputs of each group to be zero or one. The number of output streams of a group can only be zero in the unusual case where the network has multiple output ports and one of those ports is used in the parallelization block. The output of the whole parallelization block is the combined list of the outputs of each group. This list of data streams can then be reduced to a single stream either by using one of the two available merge layers `Concatenate` and `Add` or the selection layer. The selection layer is denoted by `[index]` and selects the stream corresponding to the given index from the stream list.

Since it is common for deep nets to repeat the same layer multiple times, we introduce the concept of *structural arguments* to leverage **(RL2)**, allowing one to pass a connect operator as an argument to any layer with an integer assigned to it denoting the number of repetitions for this layer. In L.10 the `resLayer` is repeated seven times by passing `-> = 7` to its interface. Similar to the sequential operator, the parallelization operator can be used as a structural argument, as well. Structural arguments contribute heavily to the compactness and the readability of the network.

To make our ResNet152 model modular and easy to read we define the two custom layers `conv` and `resLayer` in L.20-23 and L.24-33, respectively. Note how `resLayer` applies the parallelization operator in L.29 to realize residual connections in an elegant way. A further structural argument available in MontiAnna and used in the two custom layers is the *conditional argument* `?`, cf. L.23 and L.30. If set to `false`, the layer is removed from the network. In our example, it is used to skip the `Relu` layer in the third and fourth instance of the `conv` layer in the `resLayer` definition. Furthermore, it is a convenient way to express variability in a neural network. This is particularly convenient for automated experimentation when a large number of different layer combinations needs to be tested. Note that to comply with **(RL2)** all layers in our model are independent of each other, i.e. the developer does not have to specify the dimensions for each layer. The framework computes them automatically.

```
1  architecture ResNet152(N1 channels=3, N1 height=224,
       ↪ N1 width=224, N1 classes=1000){
2  def input Z(0:255)^{channels, height, width} image;
3  def output Q(0:1)^{classes} predictions;
4  image ->
5  conv(kernel=7, channels=64, stride=2) ->
6  Pooling(pool_type="max", kernel=(3,3), stride=(2,2))
       ↪ ->
7  resLayer(channels=64, addSkipConv=true) ->
8  resLayer(channels=64, ->=2) ->
9  resLayer(channels=128, stride=2, addSkipConv=true) ->
10 resLayer(channels=128, ->=7) ->
11 resLayer(channels=256, stride=2, addSkipConv=true) ->
12 resLayer(channels=256, ->=35) ->
13 resLayer(channels=512, stride=2, addSkipConv=true) ->
14 resLayer(channels=512, ->=2) ->
15 GlobalPooling(pool_type="avg") ->
16 FullyConnected(units=classes) ->
17 Softmax() ->
18 predictions;
19
20 def conv(channels, kernel=1, stride=1, act=true){
21 Convolution(kernel=(kernel,kernel),channels=channels,
       ↪ stride=(stride,stride)) ->
22 BatchNorm() ->
23 Relu(?=act);}
24 def resLayer(channels, stride=1, addSkipConv=false){
25 (
26 conv(kernel=1, channels=channels, stride=stride) ->
27 conv(kernel=3, channels=channels) ->
28 conv(kernel=1, channels=4*channels, act=false)
29 |
30 conv(channels=4*channels, stride=stride, act=false, ?
       ↪ = addSkipConv)
31 ) ->
32 Add() ->
33 Relu(); }}
```

Fig. 3: MontiAnna architecture model of the ResNet152 architecture

A further means of layer stacking contributing to model compactness and readability and supporting **(RL2)** is provided by *argument sequences* (not used in our ResNet152 example). Argument sequences can be passed in square brackets instead of regular layer arguments to declare that a layer should be repeated for each value in the sequence. Thereby, the data flow operators connecting the resulting layer stack are part of this sequence, e.g. a) `[1->2->3->4]`, b) `[true | false]`, c) `[1 | 3->2]`, d) `[ |2->3]`, e) `[1->..->4]`. Note that e) is defined as a range and is semantically equivalent to a). A parallel group can be empty as shown in d), realizing a skip connection. Argument sequences can be combined with single valued arguments. The latter are equivalent to an argument sequence containing the same value for each layer in the stack.

B. Recurrent architectures

If we think of a layered neural network as a graph where each layer is a node and the edges represent the directed data flows between the layers, our architecture modeling language is DAG complete, i.e. it can express any directed graph having no paths starting and ending at the same node. This is sufficient to cover most practical neural network architectures as long as they have no recurrent structures.

However, as soon as we need our neural network to deal with data sequences such as speech or text, we have to introduce a new type of neurons able to memorize the past. Some prominent recurrent cell-types are the so called long short-term memory (LSTM) and gated recurrent unit (GRU) cells [33], [34]. These cells differ from standard neurons as used in Fig. 3 by an inner state which is maintained and adapted over multiple processing steps as well as a more complex interface. The latter has two reasons: first, the neuron is more complex and we might need to access different parts, e.g. its inner state. Second a network architecture based on recurrent units has temporal dependencies. For instance, we might want neuron A to receive the output of neuron B with a delay of two time steps.

To illustrate the main recurrent neural network (RNN) concepts, we analyze the basic machine translation encoder-decoder model based on [35]. Given a sequence in the source language, we want to generate the best fitting sequence in the target language. The model has two LSTMs, one of which is the encoder, while the other one acts as decoder. The encoder processes the source sequence to create an intermediate representation. The decoder then uses this intermediate representation to generate the target sequence word by word. To connect both LSTMs, the decoder's inner state is initialized with the last encoder's inner state. In Fig. 4 the *unrolled* network is depicted, i.e. the LSTM cells are replicated for each timestep. Each replication is represented by a rectangle with the corresponding time included for the decoder part. We use start and end of sequence symbols (<SOS> and <EOS>) to let the LSTMs know where the start and the end of the sequences are. In our example we input the source sequence "Ich bin" in German to obtain the target sequence "I am" in English.

To be able to model recurrent networks we need to introduce new modeling constructs. The model in Fig. 5 is an implementation of the previously described machine translation model in MontiAnna. While the header remains structurally unchanged, we encounter the new *layer* keyword in L.5 and L.11. In our CNN example, layer instances are anonymous, as each layer instance is used exactly once. However, when working with RNNs, we might need to access a layer's state, input, or output multiple times. Using the `layer` keyword, followed by the layer type (`LSTM` in this example) and a name, defines a named layer instance without adding it to the network. The name can then be used instead of the layer type to add the instance to a network expression, cf. L.9 and L.14 (note that in L.14 we access a specific part of the neuron, namely the neuron's inner state, using the dot operator).

Note that in contrast to Fig. 3, where the whole network is defined as one single expression, we have four network expressions in our RNN model, each terminated by a semicolon. The first subnetwork defined in L.7-9 is the encoder part. The internal fixed-size representation of the source sentence is saved in the inner state of the encoder LSTM, cf. L.8-9. Later, it is input into the inner state of the decoder LSTM in L.14.

The most interesting subnetwork is given in L.16-23. It is preceded by a `timed<t>` modifier meaning that this subnetwork has temporal interdependencies based on the time variable `t`. The time variable does not carry an actual value, but is used to describe temporal relationships. Inputs, outputs, and layer attributes can now be accessed with a temporal

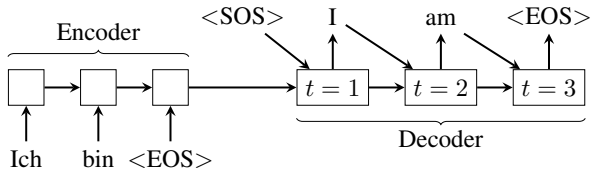

Fig. 4: RNN encoder-decoder model example

```
1  architecture RNNencdec<N1 max_length=50, N1
       ↪ vocabulary_size=30000, N1 hidden_size=1000>{
2    def input Q^{max_length, vocabulary_size} source;
3    def output Q^{max_length, vocabulary_size} target;
4
5    layer LSTM(units=hidden_size) encoder;
6
7    source ->
8    Embedding(units=hidden_size) ->
9    encoder;
10
11   layer LSTM(units=hidden_size) decoder;
12
13   1 -> OneHot() -> target[0];
14   encoder.state -> decoder.state;
15
16   timed<t> GreedySearch(size=max_length) {
17       target[t-1] ->
18       Embedding(units=hidden_size)
19       decoder ->
20       FullyConnected(units=vocabulary_size) ->
21       Softmax() ->
22       target[t]
23   }; }
```

Fig. 5: RNN encoder-decoder model implementation

argument, cf. L.17 and L.22. Here, we model the network output of the *last* timestep, denoted by `target[t-1]`, to be input into the `Embedding` layer. The temporal argument enables a concise modeling of temporal interdependencies, which the developer needs to take care of manually in classical GPL-based deep learning frameworks.

Now, if the output of the network depends on the past of `target`, how can we compute it for the *first* timestep? This is solved using subnetworks defining invariants such as the one given in L.13. Here, `target[0]` is initialized statically with a so called one-hot vector representing the <SOS> symbol. The final network architecture is assembled from the partial network expressions starting from the input layer (`source`) based on the respective interfacing layers.

C. The training model

So far, we have been looking at the neural architecture modeling language. However, as mentioned in section V, the best architecture is useless if not trained adequately. To comply with the separation of concerns requirement **(RL4)**, we provide a dedicated training language allowing the developer to define the training hyper-parameters, set the error function, etc. In contrast to JSON and similar data formats, our training language only allows a predefined set of parameters and checks their data types and integrity. When a configuration model is applied to a network model, the compiler checks, whether the two fit together using the composed model (cf. section V). For instance, a network with an unlimited output cannot be trained with a cross-entropy error function designed

```
1  training MNISTDetector{
2    num_epoch:1,
3    batch_size:64,
4    normalize:true,
5    optimizer:adam{
6      learning_rate:0.01,
7      learning_rate_decay:0.8,
8      weight_decay:0.01 }}
```

Fig. 6: Training model for the MNIST example

to compare probability distributions. The training file used to train our `MNISTDetector` network is depicted in Fig. 6. It fixes the number of epochs to be trained as well as the batch size to process. Furthermore, it specifies the optimization algorithm to be set to Adam [36], which in turn requires its own nested hyper-parameters: the learning rate, its decay, and the weight decay. Other optimizers might require a completely different or an overlapping set of hyper-parameters.

VII. Integration of Neural Modules

Until now we have understood how to model and train a single neural network. However, for a software or a systems engineer, a neural network is just a building block in a large architecture as we have seen in our MNISTCalculator example. The design of complex systems requires elaborate engineering means dealing with their complexity, but also supporting product line engineering, evolution, as well as thorough verification and testing. Therefore, we need to embed the presented framework into a software engineering methodology.

Due to its architecture-centric, data-flow oriented divide and conquer approach, C&C modeling has been an established paradigm in engineering domains such as automotive, automation, and avionics for many years. Our aim is a methodology supporting the integration of neural networks as standardized components in C&C models. This is partially achieved by Matlab/Simulink by including neural network code into the Matlab implementation of custom Simulink components. However, there is no clear notion of what a neural network in Matlab is. Hence, it might be contaminated by code related to other tasks. This leads to bad encapsulation of neural networks and impurified interfaces. An emerging problem is that the superordinate Simulink model is not aware of the presence of the neural network and has thus no means to deal with it accordingly, e.g. to retrain it automatically.

To achieve a higher level of AI awareness in the system architecture, cf. **(RM2)**, we embed MontiAnna into the textual C&C language family EmbeddedMontiArc [8], [31], [37]. An architecture like the one of our MNISTCalculator example depicted in Fig. 1 can be defined in EmbeddedMontiArc textually.

The behavior of atomic EmbeddedMontiArc components, i.e. components which are not decomposed into smaller subcomponents, is usually defined in MontiMath, a matrix-based imperative language with a strict and static type-system. MontiMath is designed to implement math-heavy algorithms of an intelligent system such as planners, controller, etc. However, it is not well-suited to describe and train deep neural network architectures. We close this gap by adding MontiAnna as a

second, *alternative* behavior language for EmbeddedMontiArc components by means of the language composition features of the MontiCore language workbench [27]. For the implementation of an atomic EmbeddedMontiArc component, we allow the developer to choose between MontiMath and MontiAnna. Hence, MontiAnna neural networks cannot be intertwined with unrelated general purpose operations enabling a clean neural network encapsulation as required by **(RM1)** (in contrast to Matlab/Simulink, where a neural network is ordinary code).

The resulting composed language family is aware of the deep learning components present in a C&C architecture, since the EmbeddedMontiArc compiler can check the implementation language of each component. Hence, it is able to govern the life-cycle of deep learning components in accordance with **(RM2)**. This facilitates the integration of tasks such as training and cross-validation into a model-driven development process, e.g. SMARDT [38], using the code generator toolchain introduced in section V.

This brings us back to the *dataset model* which we have introduced in section V without discussing it in detail. It turns out that such a model is most powerful if attached to a whole system architecture, i.e. a C&C model, instead of a single neural network. The model contains a set of entries, each of which holds either a component *type* or a component *instance* name and the path to the corresponding training and test data. Optionally, the database type can be specified (otherwise, the backend generator chooses its default, e.g. LMDB for Caffe2 and HDF5 for MXNet). We model this information using a tagging language, which enables us to append additional information to model elements of another modeling language. The same technique was employed to enrich EmbeddedMontiArc ports with middleware information or to manifest extra-functional properties [39], [40].

If an entry contains the name of a concrete component instance, we have a one-to-one mapping of a training data set to a network. However, if the entry is related to a component *type*, the compiler will use the given training data to train a network once and to copy it to all component instances of the given type in our EmbeddedMontiArc architecture, similar to a prototype pattern [9]. If the network is stateless, one central flyweight network instance is created and reused by *all* components of the corresponding type. This comes in handy in our MNISTCalculator example. We have a model containing six digit detectors. However, as all six detectors are instances of the same type and, what is more, the convolutional neural network is stateless, the compiler performs network training only once and instantiates only one network at runtime. Furthermore, as already mentioned, a training is only performed during the compilation process if neither the data referenced by the dataset model nor the MontiAnna models have changed.

VIII. Discussion

As the main contribution, we have presented a novel holistic modeling framework integrating artificial neural networks as components into software architectures and allowing software architects to deal with these networks as standard components. Furthermore, the stand-alone deep learning DSML shifts the focus from GPL elements to domain concepts. Splitting the three concerns network architecture, training, and dataset is also a novel approach enhancing the modularity and reusability of deep learning code. Such a separation of concerns needs to be enforce actively by the development team if a GPL is used.

The syntax of the proposed network architecture description language resembles the layer-based structure of networks written in high-level frameworks such as Keras. However, introducing a declarative facility for connecting neuron states of different points in time facilitates the handling of recurrent network architectures. In today's frameworks the developer needs to keep track of temporally changing neuron states manually if she needs to reuse them in later steps.

A component-based dataset model lets the compiler integrate the training process into the compilation procedure. Expensive re-training is skipped if training artifacts haven't changed. Learned parameters can be deployed in an artifact repository and reused as dependencies in other projects.

A further advantage of integrating neural network design into a C&C methodology is an arsenal of available tools [41]. Examples include component-based product lines engineering supporting the creation and management of variants of intelligent systems, model evolution and the execution of backward-compatibility checks when the AI platform is updated [42] as well as the definition of extra-functional properties [40].

To demonstrate the presented methodology in a more technical context, we developed a direct perception based autonomous vehicle [3] as a C&C architecture in EmbeddedMontiArc. The deep net predicting the affordance indicators was defined and trained in MontiAnna while MontiMath was used for the controller and filter design. The model was enriched by ROS tags in order to automatically generate ROS middleware code enabling us to integrate the system with Open Racing Car Simulator (TORCS) [43]. The complete model, a video tutorial demonstrating the generative process, and the final result are available at our website[†].

IX. Conclusion and Future Work

Despite the growing interest in deep learning, our analysis showed that the domain lacks mature language support and methodologies to integrate neural networks in large productive systems. We tackle these issues by proposing a deep learning modeling language family focusing on the main domain concepts while providing means for modularity and extensibility. The solution covers a variety of architectural neural network styles and provides a C++ generator supporting multiple deep learning backends. Furthermore, the framework is embedded into a C&C architecture description language allowing software architects to seamlessly integrate deep learning technology into large systems. Ongoing and future work includes the development of further backend code generators to support more platforms, the steady addition of latest developments, but also the integration of analysis tools. The latter must provide backend-independent quality assessment mechanisms for the experimentation phase.

REFERENCES

[1] Brody Huval, Tao Wang, Sameep Tandon, Jeff Kiske, Will Song, Joel Pazhayampallil, Mykhaylo Andriluka, Pranav Rajpurkar, Toki Migimatsu, Royce Cheng-Yue, et al. An empirical evaluation of deep learning on highway driving. *arXiv preprint arXiv:1504.01716*, 2015.

[2] Alex Graves, Abdel-rahman Mohamed, and Geoffrey Hinton. Speech recognition with deep recurrent neural networks. In *2013 IEEE international conference on acoustics, speech and signal processing*, pages 6645–6649. IEEE, 2013.

[3] Chenyi Chen, Ari Seff, Alain Kornhauser, and Jianxiong Xiao. Deepdriving: Learning affordance for direct perception in autonomous driving. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 2722–2730, 2015.

[4] Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. The cifar-10 dataset. *online: http://www. cs. toronto. edu/kriz/cifar. html*, 2014.

[5] Yann LeCun. The mnist database of handwritten digits. *http://yann. lecun. com/exdb/mnist/*.

[6] Mathworks Inc. Simulink User's Guide. Technical Report R2019a, MATLAB & SIMULINK, 2019.

[7] National Instruments. BridgeView and LabView: G Programming Reference Manual. Technical Report 321296B-01, 1998.

[8] Evgeny Kusmenko, Bernhard Rumpe, Sascha Schneiders, and Michael von Wenckstern. Highly-Optimizing and Multi-Target Compiler for Embedded System Models: C++ Compiler Toolchain for the Component and Connector Language EmbeddedMontiArc. In *MoDELS'18*, 2018.

[9] Erich Gamma. *Design patterns : elements of reusable object-oriented software*. Addison-Wesley, Reading, Mass, 1995.

[10] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y Ng. ROS: an Open-Source Robot Operating System. In *ICRA workshop on open source software*, volume 3, page 5. Kobe, Japan, 2009.

[11] Shaohuai Shi, Qiang Wang, Pengfei Xu, and Xiaowen Chu. Benchmarking state-of-the-art deep learning software tools. In *Cloud Computing and Big Data (CCBD), 2016 7th International Conference on*, pages 99–104. IEEE, 2016.

[12] Rami Al-Rfou, Guillaume Alain, Amjad Almahairi, Christof Angermueller, Dzmitry Bahdanau, Nicolas Ballas, Frédéric Bastien, Justin Bayer, Anatoly Belikov, Alexander Belopolsky, et al. Theano: A python framework for fast computation of mathematical expressions. *arXiv preprint*, 2016.

[13] Stéfan van der Walt, S Chris Colbert, and Gael Varoquaux. The numpy array: a structure for efficient numerical computation. *Computing in Science & Engineering*, 13(2):22–30, 2011.

[14] Ronan Collobert, Koray Kavukcuoglu, and Clément Farabet. Torch7: A matlab-like environment for machine learning. In *BigLearn, NIPS workshop*, number EPFL-CONF-192376, 2011.

[15] Ronan Collobert, Samy Bengio, and Johnny Mariéthoz. Torch: a modular machine learning software library. Technical report, Idiap, 2002.

[16] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: a system for large-scale machine learning. In *OSDI*, volume 16, pages 265–283, 2016.

[17] François Chollet et al. Keras: Deep learning library for theano and tensorflow. *URL: https://keras. io/k*, 7:8, 2015.

[18] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM international conference on Multimedia*, pages 675–678. ACM, 2014.

[19] Sanjay Ghemawat and Jeff Dean. Leveldb. *URL: https://github. com/google/leveldb, http://leveldb. org*, 2011.

[20] Mike Folk, Gerd Heber, Quincey Koziol, Elena Pourmal, and Dana Robinson. An overview of the hdf5 technology suite and its applications. In *Proceedings of the EDBT/ICDT 2011 Workshop on Array Databases*, pages 36–47. ACM, 2011.

[21] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[22] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274*, 2015.

[23] Frank Seide and Amit Agarwal. Cntk: Microsoft's open-source deep-learning toolkit. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 2135–2135. ACM, 2016.

[24] Howard Demuth and Mark Beale. Matlab neural network toolbox users guide version 6. the mathworks inc. 2009.

[25] Mike Folk, Gerd Heber, Quincey Koziol, Elena Pourmal, and Dana Robinson. An Overview of the HDF5 Technology Suite and Its Applications. In *Proceedings of the EDBT/ICDT 2011 Workshop on Array Databases*, AD '11, 2011.

[26] Jeff Dean Sanjay Ghemawat. LevelDB, 2012.

[27] Katrin Hölldobler and Bernhard Rumpe. *MontiCore 5 Language Workbench Edition 2017*. Aachener Informatik-Berichte, Software Engineering, Band 32. Shaker Verlag, December 2017.

[28] Ariel Gordon, Elad Eban, Ofir Nachum, Bo Chen, Hao Wu, Tien-Ju Yang, and Edward Choi. MorphNet: Fast & simple resource-constrained structure learning of deep networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1586–1595, 2018.

[29] MorphNet on Github. https://github.com/google-research/morph-net. Accessed: 2019-04-26.

[30] Yann LeCun, Koray Kavukcuoglu, and Clément Farabet. Convolutional networks and applications in vision. In *Circuits and Systems (ISCAS), Proceedings of 2010 IEEE International Symposium on*, 2010.

[31] Evgeny Kusmenko, Alexander Roth, Bernhard Rumpe, and Michael von Wenckstern. Modeling Architectures of Cyber-Physical Systems. In *European Conference on Modelling Foundations and Applications (ECMFA'17)*, LNCS 10376, pages 34–50. Springer, July 2017.

[32] Yann LeCun, Bernhard Boser, John S Denker, Donnie Henderson, Richard E Howard, Wayne Hubbard, and Lawrence D Jackel. Backpropagation applied to handwritten zip code recognition. *Neural computation*, 1(4):541–551, 1989.

[33] Felix A Gers, Jürgen Schmidhuber, and Fred Cummins. Learning to forget: Continual prediction with lstm. 1999.

[34] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

[35] Kyunghyun Cho, Bart van Merriënboer, Dzmitry Bahdanau, and Yoshua Bengio. On the properties of neural machine translation: Encoder–decoder approaches. *Syntax, Semantics and Structure in Statistical Translation*, page 103, 2014.

[36] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[37] Evgeny Kusmenko, Jean-Marc Ronck, Bernhard Rumpe, and Michael von Wenckstern. EmbeddedMontiArc: Textual modeling alternative to Simulink. In *EXE at MODELS*, 2018.

[38] Steffen Hillemacher, Stefan Kriebel, Evgeny Kusmenko, Mike Lorang, Bernhard Rumpe, Albi Sema, Georg Strobl, and Michael von Wenckstern. Model-Based Development of Self-Adaptive Autonomous Vehicles using the SMARDT Methodology. In *MODELSWARD'18*, pages 163 – 178. SciTePress, January 2018.

[39] Alexander Hellwig, Stefan Kriebel, Evgeny Kusmenko, and Bernhard Rumpe. Component-based Integration of Interconnected Vehicle Architectures. In *30th Intelligent Vehicles Symposium (IV'19). Workshop on Cooperative Interactive Vehicles*, pages 146–151. IEEE, June 2019.

[40] Shahar Maoz, Ferdinand Mehlan, Jan Oliver Ringert, Bernhard Rumpe, and Michael von Wenckstern. OCL Framework to Verify Extra-Functional Properties in Component and Connector Models. In *Proceedings of MODELS 2017. Workshop EXE*, CEUR 2019, September 2017.

[41] Vincent Bertram, Shahar Maoz, Jan Oliver Ringert, Bernhard Rumpe, and Michael von Wenckstern. Component and Connector Views in Practice: An Experience Report. In *Conference on Model Driven Engineering Languages and Systems (MODELS'17)*, pages 167–177. IEEE, September 2017.

[42] Bernhard Rumpe, Christoph Schulze, Michael von Wenckstern, Jan Oliver Ringert, and Peter Manhart. Behavioral Compatibility of Simulink Models for Product Line Maintenance and Evolution. In *SPLC*, 2015.

[43] Bernhard Wymann, Eric Espié, Christophe Guionneau, Christos Dimitrakakis, Remi Coulom, and Andrew Sumner. TORCS, the open racing car simulator, 2013.