A. Atouani, J. C. Kirchhof, E. Kusmenko, B. Rumpe:
Artifact and Reference Models for Generative Machine Learning Frameworks and Build Systems.
In: E. Tilevich, C. De Roover, editors, Proceedings of the 20th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE '21), Volume 20, ACM SIGPLAN, Oktober 2021
https://www.se-rwth.de/publications/

Artifact and Reference Models for Generative Machine Learning Frameworks and Build Systems

Abdallah Atouani
Software Engineering
RWTH Aachen University
Aachen, Germany
https://se-rwth.de

Jörg Christian Kirchhof
Software Engineering
RWTH Aachen University
Aachen, Germany
https://se-rwth.de

Evgeny Kusmenko
Software Engineering
RWTH Aachen University
Aachen, Germany
https://se-rwth.de

Bernhard Rumpe
Software Engineering
RWTH Aachen University
Aachen, Germany
https://se-rwth.de

Abstract

Machine learning is a discipline which has become ubiquitous in the last few years. While the research of machine learning algorithms is very active and continues to reveal astonishing possibilities on a regular basis, the wide usage of these algorithms is shifting the research focus to the integration, maintenance, and evolution of AI-driven systems. Although there is a variety of machine learning frameworks on the market, there is little support for process automation and DevOps in machine learning-driven projects. In this paper, we discuss how metamodels can support the development of deep learning frameworks and help deal with the steadily increasing variety of learning algorithms. In particular, we present a deep learning-oriented artifact model which serves as a foundation for build automation and data management in iterative, machine learning-driven development processes. Furthermore, we show how schema and reference models can be used to structure and maintain a versatile deep learning framework. Feasibility is demonstrated on several state-of-the-art examples from the domains of image and natural language processing as well as decision making and autonomous driving.

CCS Concepts: • **Computing methodologies** → **Machine learning**; • **Software and its engineering** → **Abstraction, modeling and modularity**; *Software development methods.*

GPCE '21, October 17–18, 2021, Chicago, IL, USA

ACM ISBN 978-1-4503-9112-2/21/10…$15.00
https://doi.org/10.1145/3486609.3487199

Keywords: machine learning, metamodeling, artificial intelligence, reference models, compiler, build systems, training, artifact models

ACM Reference Format:
Abdallah Atouani, Jörg Christian Kirchhof, Evgeny Kusmenko, and Bernhard Rumpe. 2021. Artifact and Reference Models for Generative Machine Learning Frameworks and Build Systems. In *Proceedings of the 20th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE '21), October 17–18, 2021, Chicago, IL, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3486609.3487199

1 Introduction

Machine learning models such as deep neural networks have proven to be applicable to a variety of problems in a multitude of domains in the last decade and the boundaries are pushed further on a daily basis. Such methods are used for both implementing application functionality, *e.g.,* facial recognition, and supporting the development process itself, *e.g.,* through automatic model repair [2, 3] or understanding the semantics of code [32, 46].

The widespread usage of machine learning in more and more software systems leads to the necessity of processes, frameworks, and tools supporting software architects and developers to cope with the abundance of available algorithms and variants thereof, the integration of machine learning components into large systems, as well as their continuous evolution.

It turns out that from a software engineering point of view, the development of machine learning-based software is in several aspects more intricate than classical software development and poses more challenges for engineers and tools. For instance, the modularization and encapsulation of machine learning models can prove difficult due to a variety of reasons [42]. Furthermore, machine learning-driven software development introduces a multitude of new types of artifacts, which are not present in conventional software systems.

In a machine learning-driven project, in addition to a standard code base written in a general purpose programming language (GPL) or a domain-specific language (DSL), the developers need to deal with a variety of artifact types including but not limited to machine learning models, *e.g.*, neural network descriptions, training data, training configurations, the learned parameters (the trained model, *e.g.*, the weights of a neural network), training result metadata holding information on the quality of the trained network, etc.

To implement an efficient machine learning development process, the developers and the tools need to be aware of the artifact heterogeneity, understand the relationships between the artifact types and organize them efficiently.

The first research question of this paper is therefore:

(RQ1) Which constituents and relationships are present in machine learning-based software a build system needs to be aware of in order to accelerate and automate its development?

To this end, we contribute a **machine learning artifact model** making the types and relationships of artifacts present in machine learning-based projects explicit, delivering a foundation for the automation of machine learning-oriented development processes. An extensive evaluation, which is based on a reference implementation, shows how the machine learning-oriented artifact model can be used as a basis for the automation of a data-driven development process serving as the core of the required build infrastructure.

Nowadays a zoo of machine learning frameworks exists. These frameworks vary in different dimensions such as the offered level of abstraction, the target audience (education vs. research vs. production), but also the specialization, *e.g.*, supervised deep learning, reinforcement learning, generative adversarial networks (GANs), etc. This leads us to our second research question:

(RQ2) How can a high level of variability regarding the training and building of machine learning components be enabled in a single generative framework?

A goal of our research is the unification of a broad range of deep learning algorithms in a single powerful, yet maintainable generative framework for applied machine learning with a slim, user-friendly application programming interface (API). Therefore, the second main contribution of this paper is a **reference modeling approach to deep learning framework design**. This approach uses component-based role modeling and configuration schemas to define training pipelines explicitly, thereby enhancing the maintainability and extensibility of deep learning frameworks. The reference models are used to define the configuration languages for the training pipelines they describe and can therefore be seen as the framework's metamodels.

In summary, the contributions of this paper are:

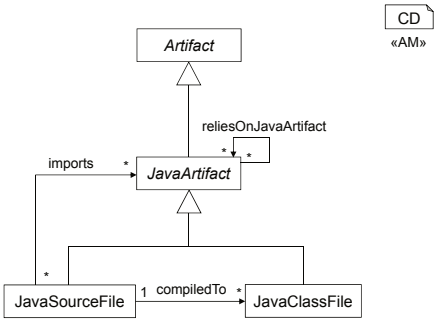


Figure 1. Artifact model for Java (taken from [15]).

- a machine learning artifact model making explicit the types and relationships of artifacts present in deep learning-based projects
- a reference modeling approach to deep learning framework design
- an evaluation featuring a Maven-based, automated build infrastructure and three examples making use of the artifact and the reference models.

The remainder of this paper is structured as follows. In section 2 we provide the required preliminaries as well as related work. In section 3 we introduce our artifact model for machine learning-based software. In section 4, we discuss how training reference models can be employed to keep the complexity of continuously evolving machine learning frameworks under control. An evaluation is given in section 5. Finally, we conclude our work in section 6.

2 Preliminaries and Related Work

Since our work is based on both advances in machine learning as well as model driven software engineering (MDSE), in particular artifact modeling, our related work section is subdivided in multiple parts accordingly.

Artifact Models. According to Hillemacher et al. [21], "an artifact is an individually storable and uniquely named unit serving a specific purpose in the context of a development process". Artifact models are used to structure file types, *e.g.*, in a model-driven development (MDD) project. They describe the structure of the (possibly generated) development artifacts and the relations between them using class diagram (CD) syntax and can serve as a basis for tool construction. Figure 1 shows an exemplary artifact model for Java projects. Java artifacts may rely on other Java artifacts and are either source files or the compiled class files that are compiled from the source files. The source files may also import other Java artifacts

As MDD projects may contain various different file types, artifact models help to tame the complexity introduced by this [8, 15, 16]. Especially, artifact models can also help tools to understand artifacts and their relations. Such tools can

TensorForce

```
from tensorforce.agents import DQNAgent
from tensorforce.execution import Runner
from tensorforce.contrib.openai_gym import OpenAIGym
q_network = [{'type': 'dense', 'size': 64, 'activation': 'tanh'},
             {'type': 'dense', 'size': 64, 'activation': 'tanh'},]
agent = DQNAgent(states={'type': 'float', 'shape': (4,)},
          actions={'type': 'int', 'shape': (2,)},
          network=q_network, memory={'type': 'replay', 'capacity':
                                                          10000},
          step_optimizer={'type': 'adam', 'learning_rate': 0.001},
           states_preprocessing={'type': 'divide', 'scale': 10})
environment = OpenAIGym('CartPole-v0')
runner = Runner(agent=agent, environment=environment)
runner.run(episodes=1000)
runner.close()
```

Figure 2. Setting up and training a Deep Q-Network (DQN) agent using Tensorforce.

then support developers. For example, Integrated Development Environments (IDEs) need to understand the files used by a project. Therefore, IDEs providing extensive support for a specific language are often tailor-made for this very language and its corresponding artifacts. One of the most widely used IDEs for Java, IntelliJ, is also offered in different variations for, *e.g.*, C++ (CLion), Python (PyCharm), and many other languages. Plugins and modes within the IDE such as the *Scientific Mode* of PyCharm[1] already enable IDEs to adapt to specific domains. In contrast to those handwritten extensions, artifact models foster the development of IDEs, or more generally speaking tools, which can adapt to the domains in which they are used and the domain-specific artifacts.

Similarly to artifact models, *megamodels* provide a global overview of artifacts within MDD projects by making statements about their relations [5, 15, 19]. Broy presents an artifact model for linking artifacts in different views of the same system, *e.g.*, requirements and architectural views [7]. [12] and [4] address the problem of linking requirements to artifacts. However, to the best of our knowledge, no prior work examined artifact models in the context of machine learning and its artifacts in software engineering processes. Creating reusable machine learning components has been examined, *e.g.*, in [18], but without putting them in the context of a clear artifact model. Machine learning has also been applied to classify metamodels in repositories with a large number of metamodels [38, 39].

Machine Learning Frameworks. Today a multitude of deep learning frameworks exists targeting different audiences. Low-level frameworks such as TensorFlow [1] provide a whole lot of flexibility and are well suited for experimentation and research. High level pendants or APIs such as Keras [17] on the other hand focus on usability and enable rapid prototyping of neural network models. Comparisons of different GPL and model-based approaches can be found in [28, 29].

[1]https://www.jetbrains.com/help/pycharm/matplotlib-support.html

gin

```
DQNAgent.gamma = 0.99
DQNAgent.update_period = 4
tf.train.RMSPropOptimizer.learning_rate = 0.00025
tf.train.RMSPropOptimizer.decay = 0.95
atari_lib.create_atari_environment.game_name = 'Pong'
create_agent.agent_name = 'dqn'
Runner.num_iterations = 200
Runner.training_steps = 250000
WrappedReplayBuffer.replay_capacity = 1000000
WrappedReplayBuffer.batch_size = 32
```

Figure 3. Excerpt of a Google Dopamine gin configuration to train a DQN agent in the Atari Pong environment.

A particular trait of high-level deep learning frameworks is that they aim to hide the complexity of the training procedure from the machine learning developer. The algorithms for supervised training are provided out of the box, but require a proper hyperparameterization. Inherently different training methods either need to be crafted manually or are provided by dedicated frameworks having their own hyperparameters to be set. For instance, reinforcement learning is offered by specialized frameworks such as Tensorforce [27] or Google Dopamine [9]. As can be seen in Figures 2 and 3, setting up a DQN based reinforcement training [37] can be achieved with a few lines of code in both frameworks, which are mostly dedicated to setting the required hyperparameters. While it might seem convenient to work with a tailored framework for each training approach when there is only a small number of such approaches, the situation becomes more and more confusing with the rising number of learning methods and the lack of a common ground, *e.g.*, a common parameterization facility. For this reason, the aim of section 4 is to create an extensible and maintainable metamodeling approach for a universal machine learning modeling framework.

Integration of Machine Learning Components. While the aforementioned deep learning frameworks such as TensorFlow, Tensorforce, and Dopamine focus on the algorithmic part of machine learning, there is much less work available on the integration of machine learning models as building blocks into complex software development processes, which is the aim of this paper.

Working with data and applying machine learning can become complex and cluttered. Therefore, tools are needed to help manage and automate the development process of data-driven projects. An example of such a tool is the open-source platform MLflow [48]. MLflow's primary functions are tracking experiments, packaging ML code such that it can be reused or transferred to production, managing and deploying models from a variety of ML libraries and providing a central model store[2]. The functionalities of MLflow are independent of the used machine learning framework. They can be used directly in the code to train and build a machine learning model through a Python, Java, R, or REST API.

[2]https://mlflow.org/docs/latest/index.html

ML Metadata (MLMD) [23] is a library for recording and retrieving metadata when running machine learning pipelines[3]. While MLflow focuses on models, MLMD analyzes machine learning pipelines and looks at the interconnections between pipeline components, *e.g.,* which hyperparameters were used to train the model. Similarly to MLflow, MLMD provides a Python library to use the functionalities in the training code directly. Furthermore, it is independent of the used machine learning framework. TensorFlow Extended (TFX) is a machine learning platform that uses MLMD, but only supports TensorFlow.

While MLflow and MLMD provide means to work with machine learning artifacts and keep track of experiments, the goal of our paper is to derive an explicit artifact model from a series of machine learning-based projects featuring convolutional neural networks (CNNs), reinforcement learning, GANs, etc. enabling us to build tools for automated data-driven software development and the integration of machine learning components into large software systems.

EmbeddedMontiArc Deep Learning. For the proof-of-concept and evaluation of the theoretical foundations, we are going to build upon the textual EmbeddedMontiArc Deep Learning (EMADL) modeling framework, a research platform for component-based engineering of machine learning-driven software [28]. EMADL is a generative framework and lifecycle management system that comes with a wide range of tools including parsers and code generators. The structure of an EMADL program is defined by the means of hierarchically decomposable components and the interconnection thereof [30, 31]. The interface of a component is given by a set of typed input and output ports. For the implementation of the behavior of a component, if it cannot be expressed as a network of subcomponents, the modeler can choose between different approaches.

First, a GPL can be employed to implement the mapping of the input to the output ports in a conventional way. EMADL offers its own, math-oriented MATLAB-like scripting language for this purpose, but variants such as MontiThings Deep Learning (MTDL) for the Internet of Things (IoT) domain allow a direct integration of C++ code, as well. Second, the behavior of a component can be defined as a neural network using MontiAnna, the deep learning DSL of EMADL. In MontiAnna, a neural net is represented as a directed acyclic graph (DAG) of neuron layers and trained by the compiler when the model is processed based on a training configuration file. MontiAnna has been shown to be applicable to a wide range of deep learning problems including CNNs, reinforcement learning, GANs, as well as state-of-the-art language modeling networks such as BERT and GPT [14, 28, 29][4]. Depending on the user's choice, the code generated from MontiAnna models is based on MXNet Gluon (Microsoft/Amazon), Tensorflow (Google), or Caffe2. For this reason and due to its openness and extensibility it is well-suited as a basis for our research.

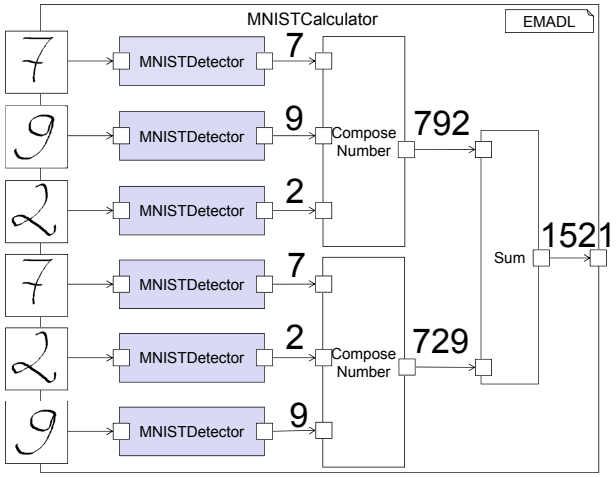


Figure 4. The MNIST calculator is the hello world example of a machine learning-based system, which consists of both machine learning components and classically written software [28].

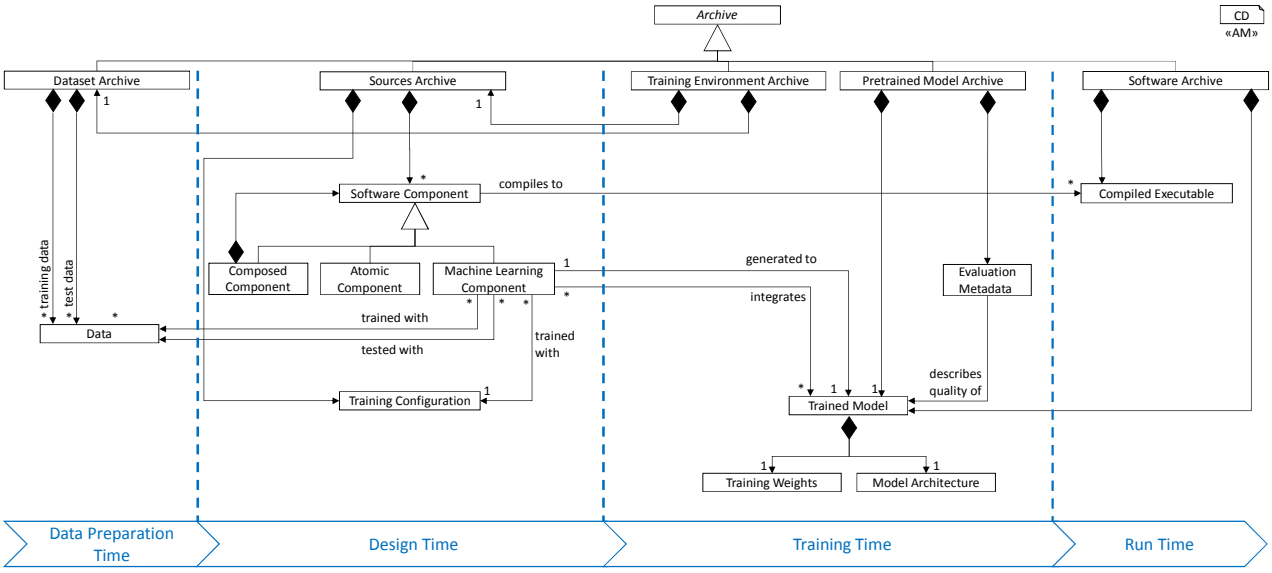
A simple hello world style example of an EMADL architecture is given in Figure 4. The purpose of this architecture is to read six handwritten digits, compose them into two three-digit numbers and, finally, compute the sum of these two numbers. To detect the digits provided as small images, each of the image inputs is forwarded to an `MNISTDetector` component. This component type is implemented as a CNN and is trained based on the widely used MNIST dataset [34]. In EMADL, neural networks are first level citizens, *i.e.,* the compiler knows that the `MNISTDetector` components need to be trained and handles the training during compile-time based on a declarative hyperparameter configuration file (similar to Dopamine gin files). What is more, knowing that the `MNISTDetector` instances are of the same type, have the same training data, and are even stateless, the compiler performs training only once and instantiates just one flyweight instance [13].

3 Machine Learning Artifact Model

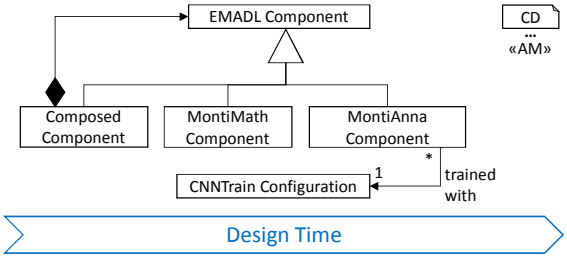
In this section we present a machine learning-oriented artifact model, which we are going to use as a basis for tool development. Thereby we demonstrate that it can be integrated in a machine learning-oriented software development process definition later on. The central concepts of the artifact model are given as a class diagram in Figure 5(a). We take a component-based perspective aiming for a platform- and paradigm-independent theoretical framework. In this sense, a component can be any constituent of a software system,

[3] https://www.tensorflow.org/tfx/guide/mlmd

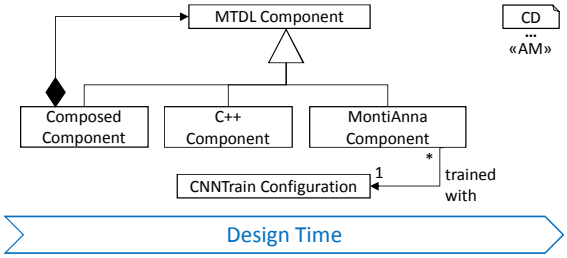
[4] Models and CI pipelines including code generation, training, and testing are available under (hidden for double-blind review) .



(a) The machine learning-oriented artifact model for software engineering.



(b) The machine learning-oriented artifact model adapted for EMADL (EMADL Component, MontiMath Component, and MontiAnna Component replace Software Component, Atomic Component, and Machine Learning Component, respectively, in Figure 5(a)).



(c) The machine learning-oriented artifact model adapted for MTDL (MTDL Component, C++ Component, and MontiAnna Component replace Software Component, Atomic Component, and Machine Learning Component, respectively, in Figure 5(a)).

Figure 5. The machine learning-oriented artifact model and its adaptions to the languages EmbeddedMontiArc (EMA) and MTDL.

such as a class, a function, a module, etc. The only distinction we make is between "classical" software components including everything except machine learning models on the one hand side and components representing such trainable models on the other hand. The former are represented by the artifact type `ComposedComponent`, *i.e.,* components that may contain other components, and `AtomicComponent`, *i.e.,* components that do not contain other components. The atomic components serve us as a hook for other artifact models such as the Java artifact model given in Figure 1. For instance, the `AtomicComponent` can be represented by a `JavaArtifact` of Figure 1. Figure 5(b) and Figure 5(c) show how this part of the artifact model can be applied to the EMADL and MTDL languages. While EMADL uses MontiMath to describe the behavior of atomic components, MontiThings uses C++ [25]. For machine learning, EMADL and MTDL use MontiAnna components. Both share the common CNNTrain configuration language for hyperparameter specification.

As suggested by [28] we aim to achieve a clear encapsulation of machine learning models, thereby enforcing the high cohesion principle. A machine learning component consists only of the machine learning model it encapsulates, *e.g.,* a neural network, and is by no means intertwined with other parts of the software such as data loading, preprocessing, or business logics. This enables us to view machine learning components as first level citizens in a development methodology, giving us the possibility to treat such components in a specific way.

Each machine learning component is trained and tested with data artifacts. Depending on the form in which data is stored, it can consist of one or more artifacts. For instance, data can be provided as a single HDF5 database or a dedicated

file for each data point. Training and testing of machine learning components should be carried out on different data, although our artifact model does not enforce this.

By means of training data and a training configuration, a machine learning component can be generated to a trained model. In the case of deep learning, this model comprises both the neural network architecture as well as the training weights. During the evolution of a system, different versions of a trained model may have different levels of quality. Therefore, we need to store evaluation metadata describing the quality of each such model. The evaluation is usually created using the test data of a machine learning component. Persisting the evaluation results in evaluation metadata files enables developers to compare different versions of the model with each other during model evolution.

In addition, pretrained networks can be embedded into a neural network model as "layers". This is helpful if certain parts of the modeled network are already available as pretrained building blocks. Assume that we'd like to train a Natural Language Processing (NLP) downstream task on top of a language model such as GPT-3 [6] or BERT [11]. Of course, we don't want to carry out a time-consuming training of a complex language model. Instead, we can reuse a pretrained implementation and connect it to additional neuron layers for the downstream functionality. Thus only a small number of parameters on top of the reused model needs to be trained by means of a special data set.

After training the machine learning models and developing the classical software components, the system can be compiled to a set of executables. We propose the usage of five archive types to package machine learning-based projects conveniently and to leverage artifact reusability.

SAR : The software archive contains the final, usable software including compiled files as well as fully trained machine learning models. Thereby, it has a similar purpose as the standard executable Java Archive (JAR).

SCAR : The source code archive contains the plain source code including both standard as well as machine learning components. However, there are no trained parameters for the machine learning models. This archive type is similar to a sources JAR and is useful as documentation or as a basis for experimentation and further development. For interpreted languages like Python, the only difference between the source code archive and the software archive is the absence of the trained models in the former.

DAR : The dataset archive contains training data to be used in training by machine learning modules. The contained data can be organized as training, test, and, evaluation data. Dataset archives facilitate the reuse and evolution of training data.

PAR : The pretrained model archive contains a trained machine learning model consisting of the learned parameters, *e.g.,* a neuron network's weights as well as further information needed to use the parameters, *e.g.,* a neural network's structure.

TEAR : The training environment archive unifies the source code archive and the dataset archive into a single container. Such an archive can be used out-of-the-box to train or extend a machine learning system without having to depend on an external dataset archive.

The TEAR contains both the handwritten artifacts form the SCAR and the data from the DAR. Thus, it contains all information necessary to create the trained models. By storing the trained models in a separate PAR, we enable their reuse across different projects. Using the PAR and the SCAR, the final SAR containing the executable artifacts can be created and shipped or deployed.

Overall these archive types enable developers to package and, hence, reuse related artifacts throughout different development stages in machine learning-based projects. The compilation of classical software components can usually be done in continuous integration (CI) pipelines in a reasonable amount of time. Incremental builds can accelerate the process. In contrast, the training of machine learning models often requires not only a non-negligible amount of time but also expensive computing resources. It is thus highly desirable to avoid model retraining with the same data and to reuse training results as dependencies instead.

4 Training Metamodeling

As can be seen in the artifact model presented in the previous section, a central development stage of a machine learning-based system is its training preceding the final build of the system and delivering important artifacts for the latter. While training cannot be automated completely since its application-tailored setup is crucial for the quality of the developed system, high-level frameworks tend to hide the implementation details from the developer and only require a hyperparameter configuration (as was shown for TensorForce and Google Dopamine in the related work section). Such an approach allows us to regard the training phase as part of a highly configurable compilation process.

A machine learning-aware language and its compiler such as EMADL are able to detect the trainable machine learning components of the system such as neural networks and prepend a training phase if needed. Now, the difficulty is that, as was also pointed out in the related work section, different training approaches require their own training pipelines and individual parameter sets which leads to specialized frameworks as in the case of reinforcement learning. The multitude of solutions can be confusing and inconvenient. A goal of this section is to provide the foundations for an integrated framework offering different training approaches hidden in

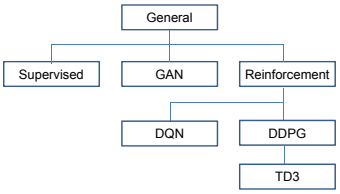


Figure 6. A taxonomy excerpt of training pipelines supported by MontiAnna.

the compiler and parameterizable meaningfully through a single interface, a universal training configuration language.

To achieve this, we introduce the notion of training pipelines in the EMADL compiler. Each training pipeline represents a class of similar training algorithms and can be tuned by pipeline-specific hyperparameters. An excerpt of the training pipelines of EMADL is depicted in Figure 6. It is mainly subdivided into supervised, reinforcement, and generative adversarial learning. Children nodes inherit the configuration parameters from their parents, *e.g.,* general parameters are available for all three subtypes. To realize such a highly configurable multi-pipeline training framework, we use two modeling techniques, which we are going to discuss in the next paragraphs. First, we use schema models enabling us to define sets of hyperparameters for the different pipelines along with their respective types. Second, we use reference models capturing the basic architectures of each training pipeline using the dataflow-centric component-and-connector (C&C) paradigm. The reference models implicitly define further configuration parameters required by the respective pipelines, but also can be used to generate the structural pipeline code.

Schema-Based Training Definition. The schema and reference model-backed training pipeline configuration language we are developing here is syntax-wise compatible with the original EMADL training configuration language presented in [28, 29]. It can be used to set simple numeric, string-typed, or enum-typed as well as complex (nested) parameters, cf. examples in Figures 8(b) and 8(c). The syntax for setting a parameter is simply the parameter name followed by a colon and the parameter value. While the language is very similar to JSON, a major difference is that complex parameter values are similar to clabjects known from object-oriented multi-level modeling [20] as they have traits of values and types at the same time. For instance, if we need to set the optimizer for our training, we can do this by setting the `optimizer` parameter to an optimizer type such as `sgd` or `adam` representing the widely used stochastic gradient descent (SGD) and Adam [24] algorithm, respectively. Depending on the choice, different sets of subparameters become available, making `sgd` and `adam` not only values, but also types (cf. L.9-13 in Figure 8(b)). For instance, while `sgd` defines the `momentum` parameter in addition to

```
Schema
1  schema General {
2    learning_method: schema //supervised, reinforcement, gan
3    context: enum { CPU, GPU }
4    optimizer: opt
5    define interface opt {
6      learning_rate_minimum: Q
7      learning_rate: Q
8      weight_decay: Q
9      learning_rate_decay: Q
10   }
11   define sgd extends optimizer {
12     momentum: Q
13 } … }
```

```
Schema
1  schema reinforcement {
2    rl_algorithm: schema //dqn, ddpg
3    num_episodes: N
4    target_score: Q
5    …
6  }
```

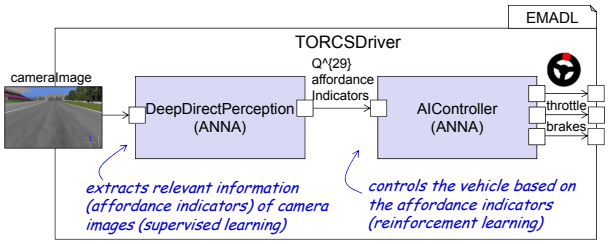
Figure 7. Excerpt of the General and Reinforcement schemas defining the training configuration language together.

core optimizer parameters such as the learning rate, `adam` requires the exponential decay rates `beta1` and `beta2` for the moment estimates as its further parameters.

To constrain and manage the parameters of the configuration language, we introduce an appropriate schema language. Schema languages are a widely used principle for the definition of configuration and data description languages. Some prominent examples are the XML Schema Definition (XSD) language for the definition of XML-based languages as well as JSON Schema. Furthermore, configuration object structures can be constrained using class diagrams and the Object Constraint Language (OCL) [41]. For our purposes we use our own schema language supporting the abstract EMADL type system, cf. Figure 7. This enables us to constrain the entries of the MontiAnna configuration language using its original type system featuring the abstract types `N`, `Z`, `Q`, `C`, and `B`[5] as well as derivations and multidimensional extensions thereof [30]. Figure 7 shows an excerpt of the general schema, applicable to all training configurations, as well as a snippet of the reinforcement learning-specific schema. The basic syntax is straightforward and similar to the one of the configuration language: to add a parameter to the schema, we need to specify its name followed by a colon and the desired type. If the type is not atomic, it might need to be defined first. This is done for the optimizer interface `opt` and the `sgd` type in L.5-13 of the general schema.

To keep the schema modular and extensible, we use a schema linking mechanism enabling us to activate schema modules in a concrete configuration model by setting special variables to values referencing these modules. Consider the parameter `learning_method` in L.2 of the general schema. Its type is set to the special value `schema`. This means that

[5]These types are based on the mathematical notation for sets and represent positive integers, integers, rationals, Gaussian rationals, and Booleans, respectively.



(a) C&C model of an autonomous driving system for the TORCS simulator incorporating two neural networks.

```
configuration DeepDirectPerception {
  context : gpu
  learning_method : supervised
  num_epoch : 100,
  batch_size : 64,
  eval_metric : mse,
  context : gpu,
  normalize : true,
  optimizer : sgd {
    learning_rate : 0.01
    learning_rate_decay : 0.9
    step_size : 8000
} }
```

CNNTrain

The deep direct perception CNN is trained using the supervised learning pipeline

(b) Excerpt of the training configuration model for the supervised learning-based DeepDirectPerception network based on [29].

general parameter available for all learning methods

EMA components only used during training play specific roles

```
configuration TorcsActor {
  context : gpu
  learning_method : reinforcement
  rl_algorithm: ddpg-algorithm
  critic: torcs.agent.network.torcsCritic
  reward: torcs.agent.network.Reward
  environment : ros_interface {
    state_topic : "/torcs/state"
    terminal_state_topic : "/torcs/terminal"
    action_topic : "/torcs/step"
    reset_topic : "/torcs/reset"
  }
  actor_optimizer: adam {
    learning_rate: 0.0001 }
  critic_optimizer: adam {
    learning_rate: 0.001 }
…}
```

CNNTrain

parameters for schema instantiation

environment is integrated using a ROS adapter (each port is mapped to a topic)

general neural network parameters are available for all networks of the reference model using the role prefix

(c) Excerpt of the training configuration model for the reinforcement learning based AIController based on [14].

Figure 8. Graphical TORCS driver model and the corresponding training configurations.

in a configuration model conforming to this schema, the developer can set `learning_method` to an identifier of another schema. Thereby, the referenced schema name is resolved and, if available, activated, i.e. the effective schema for the configuration file is extended by the definition of the referenced schema module. In this concrete example, if `learning_method: reinforcement`, the `reinforcement` schema given in the bottom listing of Figure 7 is activated and its parameters can be used in the configuration model, cf. Figure 8(c). Hence, appending new learning methods and variants to the tree in Figure 6 and adding the corresponding configuration options to the training configuration language can be realized using schema modularization. Old schemas do not need to be changed when new subschemas are added, which

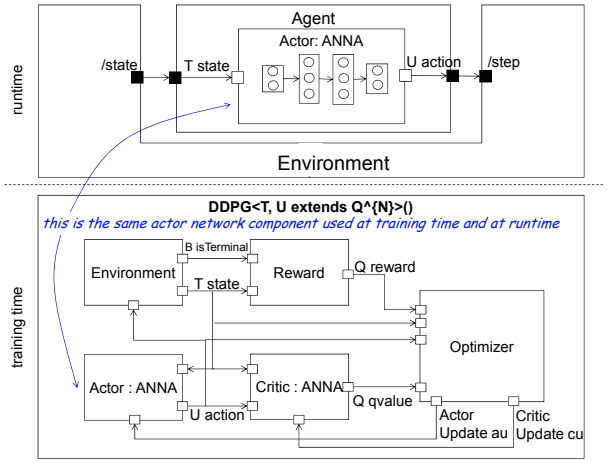


Figure 9. A simple runtime architecture using a reinforcement learning agent (top) and the role metamodel (or training reference model) of the DDPG training algorithm used for this agent's training (bottom).

makes the approach highly extensible. Similarly, schema inheritance can be applied.

The schema definition is not only used to check the validity of training configurations, but also to generate structural code for the corresponding training pipelines. For instance, to extend the EMADL compiler by a reinforcement learning pipeline, a schema defining the additional configuration parameters is modeled (cf. bottom of Figure 7). Then this schema is fed into a pipeline generator, which creates Python code providing stubs for basic pipeline functions as well as access to the configuration parameters according to the new schema. The developer of the reinforcement learning pipeline needs to implement the stubs of the generated pipeline skeleton and can eventually integrate it into the toolchain. A reinforcement learning model developer, i.e. the pipeline user, can then use this pipeline by creating configuration models conforming to the general and reinforcement learning schemas by setting `learning_method: reinforcement`.

Role-based Metamodeling. While our schema modeling framework is a simple yet powerful means to define a training configuration language, it lacks context-sensitivity and cannot be used to model the interactions of different training pipeline constituents. For this reason, we provide training reference models capturing the required components, types, and dataflows explicitly. We reuse the C&C architecture modeling language of EMADL and introduce the notion of component roles enabling us to incorporate component instances into an ensemble of specialized components in the scope of a training pipeline.

While role modeling has been mostly studied for object-oriented structures [22, 26, 44], we are going to transfer the

main concepts to C&C modeling in this work. A component role consists of a role interface and the definition of its relationships to other roles in specific contexts. In contrast to object-oriented modeling, we abstain from dynamic properties of roles such as the ability of acquiring and abandoning roles dynamically at runtime. Instead we are going to use predefined roles for the usage in different training pipelines, *e.g.,* reinforcement and GAN [36] learning. For instance, in Deep Deterministic Policy Gradient (DDPG) learning [35], an actor-critic reinforcement learning method for continuous action spaces, we need components playing the actor, the critic, and the reward role at training time. The training time reference architecture for the DDPG training pipeline of MontiAnna is depicted in the lower part of Figure 9. The environment is exchangeable between the training and the runtime phases. Critic and reward components on the other hand, become obsolete and are thrown away after training time. The reference training architecture is fixed for a given training pipeline and defines the roles required for the algorithm to function as subcomponents.

The port types in the role definitions are kept generic and are adapted to the application. Training time reference models make the training time architecture of a machine learning system explicit and fulfill two key purposes: (partial) definition of the configuration language schema and training code generation. They can hence be regarded as metamodels spawning new configuration languages.

We use reference models to extend the schemas of the training configuration language. For instance, the reference model at the bottom of Figure 9 extends the corresponding DDPG schema. For each role (i.e. subcomponent) defined in the reference architecture, the configuration language will allow a corresponding entry and, if applicable, nested entries for the role component parameters. The value assigned to such an entry can be a reference to the implementing component with the corresponding interface, *e.g.,* a Java class or, in our case, an EMADL or an MTDL component. For instance, if the DDPG training pipeline is used in a project, the reward component can be set in the configuration model as `reward: MyRewardComponent`, where the `reward` parameter stems from the corresponding component (or role) of the reference model.

As an alternative, to facilitate the integration of external software, instead of providing a concrete implementation, the role can be assigned a data source. For our implementation we integrated the Robot Operating System (ROS) middleware, which supports the public/subscribe communication pattern [40]. If the a role is played by an external source, we need to map each of the role's ports to a ROS topic, cf. Figure 8(c).

Training reference architectures are conceptual models. They define dataflows, types, and component roles. However, they do not contain any information on the concrete behavior of the trainer. For this reason, in contrast to the fully generative EMADL runtime models, they are only used to generate stubs, dataflows, and role component access functions for the training pipeline developer (which is also the case for schema models) as well as to check a concrete project using the pipeline for consistency, *e.g.,* whether the state types of the actor and the critic components provided by the training pipeline user are compatible. The concrete training procedure needs to be implemented by the framework or training pipeline developer. To do so the framework developer needs to fill the generated stubs with behavior code, *e.g.,* the computation of the neural network's weight updates. Therefore, the generated code provides access to the role components defined in the reference model and their ports. Furthermore, it provides functions to trigger the execution of these components. The combination of code generation and handwritten implementation is a deliberate design decision granting the framework developer all the freedom she needs while ensuring consistency and a clear framework structure.

We abstain from showing reference models for further training pipelines due to space limitations. Some reinforcement learning pipelines do not use a critic network, which leads to a simpler reference model. The GAN pipeline features a generator and a discriminator role. The generator network generates data and passes it to the discriminator network, which tries to determine whether the provided input is fake or not. Similar to the critic network of the reinforcement learning pipeline, the discriminator is only used during training, i.e. it is only present in the training reference architecture, but not in the runtime model.

5 Evaluation

To demonstrate the applicability of our concepts to automated build systems, we integrate them into the Maven build infrastructure. As a technical foundation of a development process based on our artifact model we developed a Maven plugin, which is capable of dealing with the proposed archive types and the development artifacts contained therein (another build platform such as Gradle is conceivable, as well). It is a basis for machine learning-aware DevOps and enables us to implement continuous integration / continous training (CI/CT) pipelines for our projects, where continuous training and retraining contribute to a steady evolution of machine learning-based software.

The Maven plugin supports the following goals: Similar to the standard `install` and `deploy` goals provided for Java projects by Maven by default, our plugin provides `install` and `deploy` goals for all kinds of archives of our artifact model. The `install` goal only installs the artifacts locally, whereas the `deploy` goal loads the artifacts into a remote repository making them available for usage in other projects. Following our artifact model from Figure 5(a), there are Maven goals for the following archive types: dataset, sources,

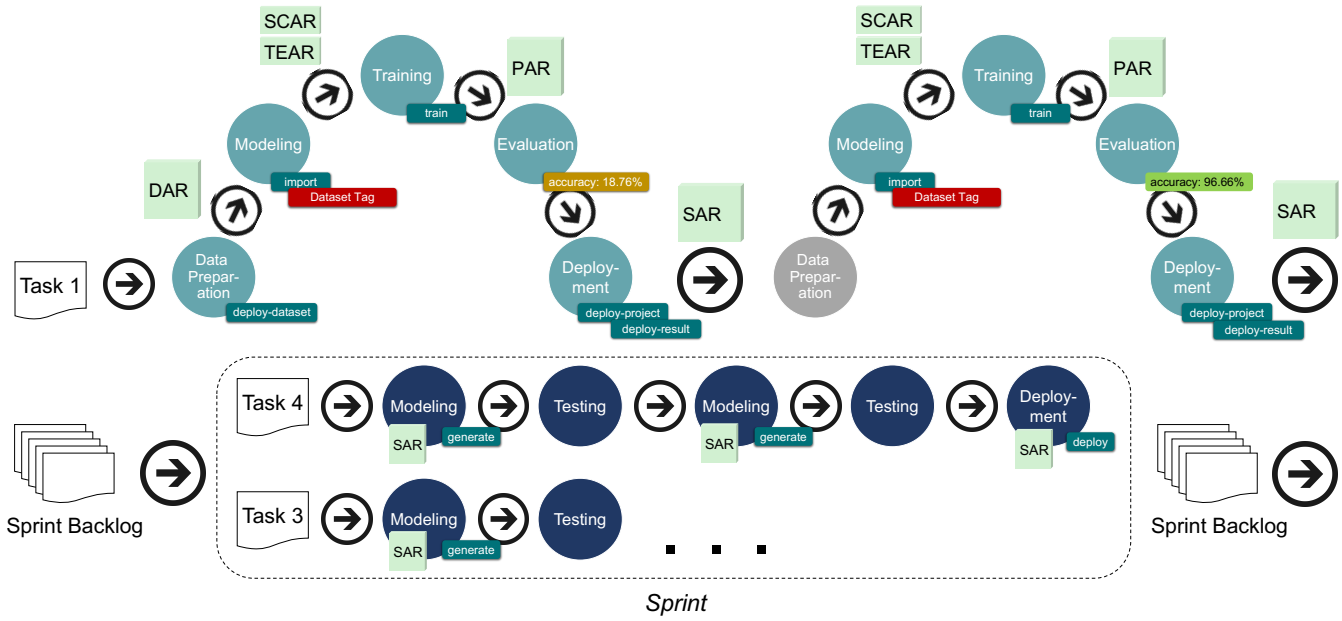


Figure 10. Iterative process of developing the MNIST calculator application. Circles represent process steps, while boxes denote archive types produced. Furthermore, some process steps are tagged with the Maven goals invoked. Each iteration ends with a SAR containing the potentially shippable product increment. In the second iteration the data preparation phase is skipped due to reusing the DAR from the first iteration.

training environment, pretrained models, (compiled) software. Additionally, the `train` goal can be used to train the machine learning components of a project.

Our approach avoids mixing model and training code with the functionality of the machine learning management tool; the functionalities of our Maven approach can be used in a declarative way. Thus, neural network architectures and training configurations are clearly separated from the Maven-based build infrastructure, there is no scattering and tangling of concerns.

Application examples shown in the following paragraphs have been chosen to highlight different aspects of our contributions. The *evolution example* shows that our artifact model covers artifact types necessary to enable a high level of reuse on system level and to avoid redundant retraining when integrated into a build and dependency system. The *neural network composition* example shows how the artifact model supports us on neural network level enabling us to reuse pretrained network parts and extend them by new layers. Hence, the first two examples cover artifact usage on system and on network levels thereby showing that our artifact model successfully answers RQ1. Finally, the third example demonstrates how a system incorporating different training algorithms can be implemented using a single framework based on training reference models. Hence, it is shown that variability, extensibility, and configurability of machine learning frameworks as demanded by RQ2 can be tackled using explicit training time models, which can be used to derive a universal configuration schema and generate the structure of training pipelines.

Evolution Example. In this illustrative case study, a team develops a calculator that can work with handwritten input, which was already mentioned to introduce EMADL in section 2. More specifically, two three-digit numbers should be summed. The architecture for this project is shown in Figure 4. The `MNISTDetector` component is a machine learning component responsible for converting handwritten digits into digital numbers. A requirement states that the recognition rate must be above 95%. The `ComposeNumber` component takes three single digits and joins them into a single three-digit number. Two such numbers are produced and then summed up by the `Sum` component.

Figure 10 shows the iterative process used to develop this application. First, a team of data scientists prepares the training and test data. They decide to use the MNIST dataset containing 70000 labeled handwritten digits and transform it into HDF5 databases for the training and test data. This dataset is packaged as a DAR according to our artifact model using the Maven plugin and deployed in a Gitlab package repository. In the modeling step, the architecture of the machine learning architecture is designed. The team chooses a LeNet architecture [33]. Using the training data and the architecture, the models can be trained in the third step of the process. This is done using MxNet and Gluon as deep learning backend generated from EMADL models. The trained

models are evaluated using the test data in the fourth step of the process. Lastly, the system can be deployed.

Unfortunately, the developed component does not reach the desired accuracy of 95 % during the first iteration. In the next iteration the team wants to improve the accuracy. As the data has already been prepared during the first iteration of the process, the development team can skip the first step in the second iteration. Instead, the database packaged as a DAR can be downloaded using the Maven plugin as a dependency from our Gitlab package registry. The team changes the machine learning architecture by adding Relu and dropout layers. Furthermore, the team adjusts the learning rate and step size of the training configuration. Then, the models are trained again. The evaluation phase reveals an improvement to an accuracy of 96.66 %. Thereby, we reach the desired 95 % accuracy. Final PAR and SAR archives are deployed. The trained machine learning models can be reused without retraining in future iterations and other projects.

Neural Network Composition. The field of neural NLP has been drawing more and more attention over the last view years. In particular, recent developments by OpenAI and its GPT-3 network have shown astonishing results in automated text production, translation, and other processing tasks [6]. The GPT networks but also several other successful language models such as BERT [11] are based on the Transformer architecture [45]. Transformers refrain from using classical recurrent neural network (RNN) or long short-term memory (LSTM) neurons and apply the attention mechanism instead to analyze dependencies between different parts of a sequence. Another important trait is the typical modular encoder-decoder structure, which is also important for our evaluation. A sequence, *e.g.,* a natural language sentence or text, is input into the encoder whose task is to analyze the sequence and to find an internal representation.

The decoder on the other hand takes the encoder's results and uses them to produce the final network output. The latter might be another sequence, *e.g.,* the same sentence in another language, but also a classification result, *e.g.,* identifying specific text properties. A generic language model such as BERT can be fine-tuned for specific applications or even combined with an application-specific output network, *e.g.,* a classification network.

The training of a generic language model requires masses of natural language data and is computationally very expensive. The application-specific output network on the other hand can be trained much easier depending on the application. Consequently, we need means for neural network reuse and composition enabling us to design new NLP applications rapidly. In particular, we want to skip the data preparation, design, and training time of the language model and concentrate on the design and training of the application.

Consider the neural network definition in Figure 11(a). The goal of this neural network is to indicate the sentiment of textual input. The in and out ports of the network define the component interface in EMADL syntax in L.2-5. The actual network structure is captured in MontiAnna syntax [28] as a sequence of neuron layers in L.8-14. The `LoadNetwork` layer in L.9 receiving the component's input is a special layer loading a pretrained network according to our artifact model (machine learning component integrates trained model). It enables us to embed arbitrary pretrained networks into a network model. In our example we load the BERT-small model bert_12_768_12 trained on the book_corpus_wiki_en_uncased dataset provided by GluonNLP[6]. The `LoadNetwork` layer receives the name of the network to be included as a parameter. The compiler then tries to resolve it in the project scope. This is similar to importing and instantiating external classes in Java. To automate the embedding, we packaged the pretrained network as a PAR archive using our Maven plugin and deployed it in the package registry of our Gitlab instance, thus making it available for use by other projects compatible to our artifact model as a Maven dependency. Figure 11(b) shows a Maven dependency snippet a user needs to include into the Maven pom.xml file in order to obtain the pretrained network from our archive repository. Then the network is installed in the local Maven repository and can be used in our project.

The `LoadNetwork` layer is followed by a simple application-specific classification network consisting of two fully connected layers (also referred to as dense layers in some frameworks) having 768 and 2 neurons, respectively. As the training data for the fine-tuning of the sentiment analysis application we use the Stanford Sentiment Treebank v2 (SST2) dataset [43]. To enable the integration of the dataset into our Maven-based lifecycle, we package it as a DAR and deploy it in our Gitlab package registry as we did for the MNIST data. Now it can be used as a Maven dependency, as well.

Reference Model-Based Training Configuration. To show the benefits of the reference model-backed training configuration, we are going to discuss a multi-network architecture from the autonomous driving domain consisting of two neural networks aiming to control a racing vehicle in the TORCS simulator [47]. The system receives camera images as input and is expected to output actuation commands for the steering wheel, throttle, and braking pedal in order to finish the race as quickly as possible without crashing or leaving the track.

The first neural network aims to extract affordance indicators from camera images following the direct perception approach [10]. We reuse the training data provided by the authors of the original paper for a supervised training. Affordance indicators resemble sensor signals including distances to other objects, orientation relative to the track, and the like, which can be estimated from visual inputs as is also done by humans. These affordance indicators can be fed into a

[6]https://nlp.gluon.ai/model_zoo/bert/index.html

EMADL

```
component Network<Z(1:oo) n = 128> {
  ports in Z(0:oo)^{n} data_0,
        in Z(0:oo)^{n} data_1,
        in Z(0:oo)^{1} data_2,
        out Q(0:1)^{2} softmax;

  implementation CNN {
    (data_0 | data_1 | data_2) ->
    LoadNetwork(network="tag:bert_base", numInputs=3,
                                         outputShape=(128, 768) ) ->
    FullyConnected(units=768) ->
    FullyConnected(units=2) ->
    Softmax() ->
    softmax; } }
```

(a) BERT-based sentiment analyzer network including the BERT base as a dependency using a LoadNetwork layer.

Maven

```
<dependency>
  <groupId>org.models.conference</groupId>
  <artifactId>bert-base</artifactId>
  <version>1</version>
  <classifier>pretrained</classifier>
</dependency>
```

(b) Pretrained network dependency in Maven.

Figure 11. Sentiment analyzer project using a pretrained BERT language model as a Maven dependency

controller to compute a driving behavior as suggested in the aforementioned paper and as was modeled using EMADL in [29].

Instead of writing the controller manually, it can be implemented as a neural network, as well. In [14], reinforcement learning was employed to train an agent controlling a TORCS vehicle using sensor inputs. In our evaluation we combine the two networks trained using different pipelines (supervised and reinforcement) to get rid of manually written parts. The resulting C&C architecture is depicted in Figure 8(a). Due to space limitations, we abstain from including the concrete neural network models. Excerpts of the training time configurations are given in Figure 8(b) and Figure 8(c) for the `DeepDirectPerception` and the `AIController` components, respectively. The well-formedness of the supervised training configuration is checked against the supervised learning schema. Again, to facilitate the automated integration of the required training data we first package and deploy it as a DAR archive.

The training configuration of the `AIController` is checked against the `reinforcement_learning` and the `ddpg_algorithm` schemas as well as the DDPG reference model, cf. Figure 9. In particular, concrete components need to be bound to the roles *critic*, *reward*, and *environment*. The first two are assigned in L.5-6 of Figure 8(c). However, we do not want to provide an environment as a self-written component. In most applications, simulation software is very complex and it is desirable to reuse it. To enable the integration of third-party software according to our reference model, instead of providing a simulation component, we map all of its ports to ROS topics, cf. L.7-12 of Figure 8(c). This way, our training framework is compatible not only with TORCS, but with any simulator supporting ROS. Adapters for other middleware solutions such as Message Queuing Telemetry Transport (MQTT) can be integrated in a similar way.

Recall that the generic port types of the reference model are adapted to the application at compile-time. The generic type parameters of the roles are bound by providing a concrete component for the corresponding role as is done for the actor role in Figure 8(a). Hence, the state input type `T` of the DDPG reference model becomes the 29-dimensional affordance vector and the action type `U` is bound to a three-dimensional vector type containing the steering, braking, and acceleration of the vehicle. Manually implemented context conditions checking types and compatibility of training time components become obsolete. The reinforcement learning related components including the actor, the critic, and the reward only need to be checked against the reference model of Figure 9. Furthermore, the port types of the runtime and the training time actor must match.

6 Conclusion

Our conclusion is that the unceasingly increasing complexity and variety of machine learning frameworks can be coped with by means of appropriate metamodeling techniques. We presented an artifact model which is an important foundation for the development of machine learning-oriented tooling and frameworks, enabling reusability and evolution of all constituents in a machine learning-driven project. By introducing the training-time as an explicit step of the build process, we enable individual training and prevent unnecessary retraining of machine learning components such as deep neural networks. What is more, training reference models coupled with configuration schemas can serve as a strong basis for deep learning frameworks, drastically enhancing their maintainability and extensibility.

Acknowledgements

Funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under the grant SPP1835 and Germany's Excellence Strategy – EXC 2023 Internet of Production - 390621612. Website: https://www.iop.rwth-aachen.de

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*. 265–283.

[2] Angela Barriga, Adrian Rutle, and Rogardt Heldal. 2019. Personalized and Automatic Model Repairing using Reinforcement Learning. In *ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*. 175–181.

[3] Angela Barriga, Adrian Rutle, and Rogardt Heldal. 2020. Improving Model Repair through Experience Sharing. *Journal of Object Technology* 19, 2 (July 2020), 13:1–21.

[4] M. Berkovich, S. Esch, C. Mauro, J. Leimeister, and H. Krcmar. 2011. Towards an Artifact Model for Requirements to IT-enabled Product Service Systems. In *Wirtschaftsinformatik*.

[5] Jean Bézivin, Frédéric Jouault, and Patrick Valduriez. 2004. On the Need for Megamodels. In *Proceedings of the OOPSLA/GPCE: Best Practices for Model-Driven Software Development workshop, 19th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications,(2004)*. Vancouver, Canada. https://hal.archives-ouvertes.fr/hal-01222947

[6] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165* (2020).

[7] Manfred Broy. 2018. A logical approach to systems engineering artifacts: semantic relationships and dependencies beyond traceability—from requirements to functional and architectural views. *Software & Systems Modeling* 17, 2 (2018), 365–393. https://doi.org/10.1007/s10270-017-0619-4

[8] Arvid Butting, Timo Greifenberg, Bernhard Rumpe, and Andreas Wortmann. 2018. On the Need for Artifact Models in Model-Driven Systems Engineering Projects. In *Software Technologies: Applications and Foundations (LNCS 10748)*, Martina Seidl and Steffen Zschaler (Eds.). Springer, 146–153. http://www.se-rwth.de/publications/On-the-Need-for-Artifact-Models-in-Model-Driven-Systems-Engineering-Projects.pdf

[9] Pablo Samuel Castro, Subhodeep Moitra, Carles Gelada, Saurabh Kumar, and Marc G Bellemare. 2018. Dopamine: A research framework for deep reinforcement learning. *arXiv preprint arXiv:1812.06110* (2018).

[10] Chenyi Chen, Ari Seff, Alain Kornhauser, and Jianxiong Xiao. 2015. DeepDriving: Learning affordance for direct perception in autonomous driving. In *Proceedings of the IEEE international conference on computer vision*. 2722–2730.

[11] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).

[12] D. Fernández and Birgit Penzenstadler. 2014. Artefact-based requirements engineering: the AMDiRE approach. *Requirements Engineering* 20 (2014), 405–434.

[13] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, and Design Patterns. 1995. Elements of Reusable Object-Oriented Software. *Design Patterns. massachusetts: Addison-Wesley Publishing Company* (1995).

[14] Nicola Gatto, Evgeny Kusmenko, and Bernhard Rumpe. 2019. Modeling Deep Reinforcement Learning Based Architectures for Cyber-Physical Systems. In *Proceedings of MODELS 2019. Workshop MDE Intelligence* (Munich), Loli Burgueño, Alexander Pretschner, Sebastian Voss, Michel Chaudron, Jörg Kienzle, Markus Völter, Sébastien Gérard, Mansooreh Zahedi, Erwan Bousse, Arend Rensink, Fiona Polack, Gregor Engels, and Gerti Kappel (Eds.). 196–202. http://www.se-rwth.de/publications/Modeling-Deep-Reinforcement-Learning-based-Architectures-for-Cyber-Physical-Systems.pdf

[15] Timo Greifenberg. 2019. *Artefaktbasierte Analyse modellgetriebener Softwareentwicklungsprojekte*. Shaker Verlag. http://www.se-rwth.de/phdtheses/Diss-Greifenberg-Artefaktbasierte-Analyse-modellgetriebener-Softwareentwicklungsprojekte.pdf

[16] Timo Greifenberg, Steffen Hillemacher, and Katrin Hölldobler. 2020. *Applied Artifact-Based Analysis for Architecture Consistency Checking*. Springer, 61–85. http://www.se-rwth.de/publications/Applied-Artifact-Based-Analysis-for-Architecture-Consistency-Checking.pdf

[17] Antonio Gulli and Sujit Pal. 2017. *Deep learning with Keras*. Packt Publishing Ltd.

[18] Thomas Hartmann, Assaad Moawad, Francois Fouquet, and Yves Le Traon. 2019. The next evolution of MDE: a seamless integration of machine learning into domain modeling. *Software & Systems Modeling* 18, 2 (2019), 1285–1304.

[19] Regina Hebig, Andreas Seibel, and Holger Giese. 2011. On the Unification of Megamodels. In *Proceedings of the 4th International Workshop on Multi-Paradigm Modeling (MPM 2010) (Electronic Communications of the EASST, Vol. 42)*, Vasco Amaral, Hans Vangheluwe, Cécile Hardebolle, Laszlo Lengyel, Tiziana Magaria, Julia Padberg, and Gabriele Taentzer (Eds.). http://journal.ub.tu-berlin.de/eceasst/article/view/704/713

[20] Brian Henderson-Sellers and Cesar Gonzalez-Perez. 2005. The rationale of powertype-based metamodelling to underpin software development methodologies. In *Conferences in Research and Practice in Information Technology Series*.

[21] Steffen Hillemacher, Nicolas Jäckel, Christopher Kugler, Philipp Orth, David Schmalzing, and Louis Wachtmeister. 2021. *Artifact-Based Analysis for the Development of Collaborative Embedded Systems*. Springer, 315–331. http://www.se-rwth.de/publications/Artifact-Based-Analysis-for-the-Development-of-Collaborative-Embedded-Systems.pdf

[22] Robert Hirschfeld, Pascal Costanza, and Oscar Marius Nierstrasz. 2008. Context-oriented programming. *Journal of Object technology* 7, 3 (2008), 125–151.

[23] Abhijit Karmarkar, Ahmet Altay, Aleksandr Zaks, Neoklis Polyzotis, Anusha Ramesh, Ben Mathes, Gautam Vasudevan, Irene Giannoumis, Jarek Wilkiewicz, Jiri Simsa, et al. 2020. Towards ML Engineering: A Brief History Of TensorFlow Extended (TFX). *arXiv preprint arXiv:2010.02013* (2020).

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

[25] Jörg Christian Kirchhof, Bernhard Rumpe, David Schmalzing, and Andreas Wortmann. 2022. MontiThings: Model-driven Development and Deployment of Reliable IoT Applications. *Journal of Systems and Software* 183 (January 2022), 111087. https://doi.org/10.1016/j.jss.2021.111087

[26] Thomas Kühn, Max Leuthäuser, Sebastian Götz, Christoph Seidl, and Uwe Aßmann. 2014. A metamodel family for role-based modeling and programming languages. In *International Conference on Software Language Engineering*. Springer, 141–160.

[27] Alexander Kuhnle, Michael Schaarschmidt, and Kai Fricke. 2017. Tensorforce: a TensorFlow library for applied reinforcement learning. Web page. https://github.com/tensorforce/tensorforce

[28] Evgeny Kusmenko, Sebastian Nickels, Svetlana Pavlitskaya, Bernhard Rumpe, and Thomas Timmermanns. 2019. Modeling and Training of Neural Processing Systems. In *Conference on Model Driven Engineering Languages and Systems (MODELS'19)* (Munich), Marouane Kessentini, Tao Yue, Alexander Pretschner, Sebastian Voss, and Loli Burgueño (Eds.). IEEE, 283–293. http://www.se-rwth.de/publications/Modeling-and-Training-of-Neural-Processing-Systems.pdf

[29] Evgeny Kusmenko, Svetlana Pavlitskaya, Bernhard Rumpe, and Sebastian Stüber. 2019. On the Engineering of AI-Powered Systems. In *ASE'19. Software Engineering Intelligence Workshop (SEI'19)* (San Diego, California, USA), Lisa O'Conner (Ed.). IEEE, 126–133. http://www.se-rwth.de/publications/On-the-Engineering-of-AI-Powered-Systems.pdf

[30] Evgeny Kusmenko, Alexander Roth, Bernhard Rumpe, and Michael von Wenckstern. 2017. Modeling Architectures of Cyber-Physical Systems. In *European Conference on Modelling Foundations and Applications (ECMFA'17)* (Marburg) *(LNCS 10376)*. Springer, 34–50. http://www.se-rwth.de/publications/Modeling-Architectures-of-Cyber-Physical-Systems.pdf

[31] Evgeny Kusmenko, Bernhard Rumpe, Sascha Schneiders, and Michael von Wenckstern. 2018. Highly-Optimizing and Multi-Target Compiler for Embedded System Models: C++ Compiler Toolchain for the Component and Connector Language EmbeddedMontiArc. In *Conference on Model Driven Engineering Languages and Systems (MODELS'18)* (Copenhagen). ACM, 447 –

457. http://www.se-rwth.de/publications/Highly-Optimizing-and-Multi-Target-Compiler-for-Embedded-System-Models.pdf

[32] Jeremy Lacomis, Pengcheng Yin, Edward Schwartz, Miltiadis Allamanis, Claire Le Goues, Graham Neubig, and Bogdan Vasilescu. 2019. DIRE: A Neural Approach to Decompiled Identifier Naming. In *34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 628–639.

[33] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. 1998. Gradient-based learning applied to document recognition. *Proc. IEEE* 86, 11 (1998), 2278–2324.

[34] Yann LeCun, Corinna Cortes, and CJ Burges. 2010. MNIST handwritten digit database. *ATT Labs [Online]. Available: http://yann.lecun.com/exdb/mnist* 2 (2010).

[35] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. 2015. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971* (2015).

[36] Mehdi Mirza and Simon Osindero. 2014. Conditional generative adversarial nets. *arXiv preprint arXiv:1411.1784* (2014).

[37] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. 2015. Human-level control through deep reinforcement learning. *nature* 518, 7540 (2015), 529–533.

[38] Phuong T. Nguyen, Juri Di Rocco, Davide Di Ruscio, Alfonso Pierantonio, and Ludovico Iovino. 2019. Automated Classification of Metamodel Repositories: A Machine Learning Approach. In *ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems (MODELS)*. 272–282. https://doi.org/10.1109/MODELS.2019.00011

[39] Phuong T. Nguyen, Davide Di Ruscio, Alfonso Pierantonio, Juri Di Rocco, and Ludovico Iovino. 2021. Convolutional neural networks for enhanced classification mechanisms of metamodels. *Journal of Systems and Software* 172 (2021), 110860.

[40] Morgan Quigley, Brian Gerkey, Ken Conley, Josh Faust, Tully Foote, Jeremy Leibs, Eric Berger, Rob Wheeler, and Andrew Ng. 2009. ROS: an open-source Robot Operating System. In *Proc. of the IEEE Intl. Conf. on Robotics and Automation (ICRA) Workshop on Open Source Robotics*.

[41] Mark Richters and Martin Gogolla. 2000. Validating UML models and OCL constraints. In *International Conference on the Unified Modeling Language*. Springer, 265–277.

[42] David Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips, Dietmar Ebner, Vinay Chaudhary, Michael Young, Jean-Francois Crespo, and Dan Dennison. 2015. Hidden technical debt in machine learning systems. *Advances in neural information processing systems* 28 (2015), 2503–2511.

[43] Richard Socher, Alex Perelygin, Jean Wu, Jason Chuang, Christopher D Manning, Andrew Y Ng, and Christopher Potts. 2013. Recursive deep models for semantic compositionality over a sentiment treebank. In *Proceedings of the 2013 conference on empirical methods in natural language processing*. 1631–1642.

[44] Friedrich Steimann. 2000. On the representation of roles in object-oriented and conceptual modelling. *Data & Knowledge Engineering* 35, 1 (2000), 83–106.

[45] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *arXiv preprint arXiv:1706.03762* (2017).

[46] Yao Wan, Jingdong Shu, Yulei Sui, Guandong Xu, Zhou Zhao, Jian Wu, and Philip S. Yu. 2019. Multi-Modal Attention Network Learning for Semantic Source Code Retrieval. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering* (San Diego, California) *(ASE '19)*. IEEE Press, 13–25. https://doi.org/10.1109/ASE.2019.00012

[47] Bernhard Wymann, Eric Espié, Christophe Guionneau, Christos Dimitrakakis, Rémi Coulom, and Andrew Sumner. 2000. TORCS, the open racing car simulator. *Software available at http://torcs. sourceforge. net* 4, 6 (2000), 2.

[48] Matei Zaharia, Andrew Chen, Aaron Davidson, Ali Ghodsi, Sue Ann Hong, Andy Konwinski, Siddharth Murching, Tomas Nykodym, Paul Ogilvie, Mani Parkhe, et al. 2018. Accelerating the Machine Learning Lifecycle with MLflow. *IEEE Data Eng. Bull.* 41, 4 (2018), 39–45.