MDE for Machine Learning-Enabled Software Systems: A Case Study and Comparison of MontiAnna & ML-Quadrat

Jörg Christian Kirchhof,
Evgeny Kusmenko, Jonas Ritz,
Bernhard Rumpe
Software Engineering
RWTH Aachen University
Germany
www.se-rwth.de

Armin Moin
School of Computation, Information
and Technology
Technical University of Munich,
University of Antwerp & Flanders
Make
Germany & Belgium
armin.moin@tum.de

Atta Badii
Department of Computer Science
University of Reading
United Kingdom
atta.badii@reading.ac.uk

Stephan Günnemann
School of Computation, Information
and Technology
Technical University of Munich &
Munich Data Science Institute
Germany
guennemann@in.tum.de

Moharram Challenger
Department of Computer Science
University of Antwerp
& Flanders Make
Belgium
moharram.challenger@uantwerpen.be

ABSTRACT

In this paper, we propose to adopt the MDE paradigm for the development of Machine Learning (ML)-enabled software systems with a focus on the Internet of Things (IoT) domain. We illustrate how two state-of-the-art open-source modeling tools, namely MontiAnna and ML-Quadrat can be used for this purpose as demonstrated through a case study. The case study illustrates using ML, in particular deep Artificial Neural Networks (ANNs), for automated image recognition of handwritten digits using the MNIST reference dataset, and integrating the machine learning components into an IoT-system. Subsequently, we conduct a functional comparison of the two frameworks, setting out an analysis base to include a broad range of design considerations, such as the problem domain, methods for the ML integration into larger systems, and supported ML methods, as well as topics of recent intense interest to the ML community, such as AutoML and MLOps. Accordingly, this paper is focused on elucidating the potential of the MDE approach in the ML domain. This supports the ML-engineer in developing the (ML/software) model rather than implementing the code, and additionally enforces reusability and modularity of the design through enabling the out-of-the-box integration of ML functionality as a component of the IoT or cyber-physical systems.

CCS CONCEPTS

• **Computing methodologies → Machine learning**; • **Software and its engineering → Design languages**.

KEYWORDS

model-driven engineering, artificial intelligence, domain specific modeling, machine learning, tools

1 INTRODUCTION

Model-Driven Engineering (MDE) aims to use models to support the development of engineered systems (such as software) at various stages, for example, through generating implementations from the models [38]. Further, ML is a branch of Artificial Intelligence (AI), currently of high impact, mostly due to the rise of deep learning technologies, to enable machines to *learn* through inference on data [27]. ML is used today in almost all areas and domains of software engineering. It is a popular choice for addressing problems that are difficult to overcome using traditional programming. In traditional programming, the developer has to specify the solution to the problem in an imperative or declarative manner. In contrast, ML is useful when the problem is highly data-intensive and the pattern space is at a large scale and too complex for human beings to analyze and solve directly. In such cases, inferences from observed data can be useful. However, ML is often not the best approach to follow in situations where the solution can be realized directly (i.e., in the traditional programming way) rather than through inference

[28]. In this work, we focus on smart software systems that are not rule-based (e.g., expert systems), but require inference from observed data, such as data collected though wireless sensors, also referred to as ambient intelligence, particularly in the context of smart environments, such as smart home, smart city, and smart mobility applications.

It transpires that creating the ML components for such software systems is often highly challenging for software developers. Even when using the high-level APIs of ML frameworks and libraries, such as Scikit-Learn [34], TensorFlow [13], Keras [9], or MXNet [8], developers need to have knowledge both about ML itself, and about the framework used in order to achieve satisfactory results. However, many tasks in the creation of ML pipelines, such as the addition of a convolutional layer to a deep neural network model, often follow a common set of steps, and differ essentially in the (hyper)parameters tuning phase.

Model-driven development promises to solve the above-mentioned problems in an efficient manner by raising the level of abstraction, and providing automation, for example, for code generation based on high-level specifications. Therefore, in addition to the above-mentioned ML libraries, a number of model-driven tools and associated domain-specific languages, such as MontiAnna [4, 23] and ML-Quadrat [30, 31] have been introduced in recent years. These frameworks utilize high-level specifications of ML models (e.g., deep learning networks) to generate code according to an underlying ML framework. By raising the level of abstraction, they liberate the developers from having to learn the specifics of the ML framework, and enable them to focus on the selection of the ML solutions they desire instead.

One application area where ML has proven useful is the IoT domain, which is suitable for the application of ML at several levels: On the one hand, IoT applications must be able to make inferences about their environment based on sensor data[1] (i.e., ambient situation assessment). Due to the data (stream) volume, velocity, and the complexity of the data, it can be difficult to derive knowledge about the environment from the data. This is where ML can help to identify patterns in the data, for example, to recognize a particular face in a photo, and consequently infer that a particular person is at a location. On the other hand, ML can also be used to make predictions about the future based on existing data. For example, patterns in the behavior of smart home residents can be detected to predict patterns of occupancy in rooms or spaces, and accordingly schedule cleaning tasks by a domestic robot cleaner.

In this paper, we study and compare two state-of-the-art model-driven ML tools, namely MontiAnna and ML-Quadrat, particularly from an IoT perspective, with a focus on their functional aspects. Accordingly, this paper makes the following contributions: i) It presents a case study to illustrate using both tools for modeling ML-enabled software. It also briefly studies each tool from an IoT perspective. ii) It conducts a functional comparison between the two model-driven tools. iii) It illustrates how model-driven engineering (MDE) can support different aspects of the development of machine learning-driven software.

---

[1]When we use the term *sensor data*, we also refer to data from highly complex sensors, such as cameras.
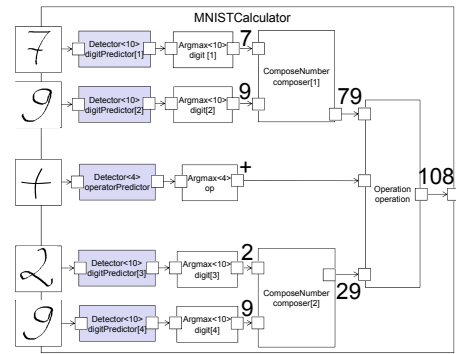

Figure 1: MNIST Calculator taking hand-written digits as inputs and outputting the sum according to the dataflow [22, 23]

As a common ground, we use the MNIST Calculator model from [22, 23], which is a 'hello world'-like example for the so-called *software 2.0* applications, consisting of ML and standard software components. The architecture of the MNIST Calculator is given in Figure 1.

The remainder of this paper is structured as follows. Section 2 introduces the MontiAnna and ML-Quadrat tools through a case study. Moreover, an in-depth comparison between MontiAnna and ML-Quadrat, which is mostly concentrated on their functional aspects is provided in Section 3. Further, we review the related work in Section 4. Finally, we conclude in Section 5.

2 TOOLS PRESENTATION AND CASE STUDY

In this section, we present both modeling tools and demonstrate how to use them through a case study for image recognition of handwritten digits. Hence, this section conforms to the first contribution mentioned in Section 1.

2.1 MontiAnna

MontiAnna [23] is a textual modeling framework and build system for the design of deep Artificial Neural Networks (ANNs). It is embedded into the component and connector-oriented modeling language family EmbeddedMontiArc [20, 25, 26], where the software functionality is encapsulated into components with interfaces defined by input and output ports. The communication of the components needs to be modeled explicitly by creating connectors between ports. A MontiAnna ANN is integrated into larger software architectures using the same component-based principle, which means each neural network is a component [24]. The component's ports are then mapped to the input and output layers of the network, respectively. The MontiAnna build system detects all ANNs in a software architecture and resolves trained model weights for these networks at build time. If no trained weights are available for a given network, or if the network model or the designated training data have changed, the build system trains the network automatically. The developer does not have to deal with the ML life cycle manually. An ML model in MontiAnna consists of a neural network architecture encapsulated into a component hull and a separate configuration model setting up the desired training scheme and

```
1   component Detector<Z(2:oo) classes = 10>{
2     ports in Z(0:255)^{1, 28, 28} data,
3     out Q(0:1)^{classes} softmax;
4
5     implementation CNN
6     {
7       def conv(channels, kernel=1, stride=1){
8         Convolution(kernel=(kernel,kernel),channels=channels) ->
9         Relu() ->
10        Pooling(pool_type="max", kernel=(2,2), stride=(stride,stride))
11      }
12
13      data ->
14      conv(kernel=5, channels=20, stride=2) ->
15      conv(kernel=5, channels=50, stride=2) ->
16      FullyConnected(units=500) ->
17      Relu() ->
18      Dropout() ->
19      FullyConnected(units=classes) ->
20      Softmax() ->
21      softmax; } }
```
MontiAnna

Figure 2: MNIST Detector CNN modeled in MontiAnna.

holding the hyperparameters. Furthermore, tag models can be used to assign versionable training data and/or existing weights to the ANNs.

MontiAnna provides a series of out-of-the-box training pipelines, for example, for supervised learning, several variants of reinforcement learning [14], generative adversarial networks (GANs), and variational autoencoders (VAEs). Further pipelines or pipeline components can be implemented as standard Python code and assembled using the component-based paradigm. To keep track of the available configuration options for the different pipelines, MontiAnna uses a modular schema system, enabling inheritance and combining of the configuration parameters. Figure 2 shows a MontiAnna deep learning component encapsulating a Convolutional Neural Network (CNN) for the detection of MNIST digits.

In the context of the IoT, MontiAnna can be used as part of MontiThings, which is a DSL-based framework [21] to specify the behavior of components in IoT applications. For example, even if edge devices in smart home applications may indeed have insufficient computational resources to perform full speech recognition themselves, many of them could detect a key phrase (e.g., the wake up commands "Hey Siri" or "OK Google") to know when to start recording. Using MontiAnna, models for such simplified speech recognition can be trained and integrated into IoT applications.

Figure 2 shows a MontiAnna component definition specifying the ANN architecture for the MNIST detector components of the MNIST calculator architecture of Figure 1. The component is named Detector in line 1 in Figure 2. Furthermore, it is defined as a generic component with the generic interface parameter classes. This supports the usability of the network architecture for similar problems with a different number of classes (with appropriate re-training). The interface is defined in lines 1-2 in Figure 2, streaming the input data for 28x28 matrices within the range of 0 and 255 for grayscale images. The output is a softmax vector with the dimensionality defined by the classes parameter.

The layer-specific ANN architecture is defined in the implementation block (see lines 13-21). The input and output ports' names are used as the input and output of the network (see lines 13 and 21, respectively). The network layers are instantiated by using library layers such as Convolution, FullyConnected, etc. Reoccurring patterns are grouped to new layer classes in the def block in lines

6-11. Should the user be uncertain as to the how to select the appropriate architecture, wild card layers could be included; whereby the framework includes network layers iteratively based on a heuristic, e.g. AdaNet [11], compares the results of each iteration, and eventually returns the best ANN model found. The same network architecture can be re-used to learn an operator detector for the MNIST calculator by providing the needed training examples and changing the class parameter to the number of supported operators, for example, +, -, ×, ÷. The machine learning components can then be interconnected with other components using connectors as was shown in [23, 26].

2.2 ML-Quadrat

ML-Quadrat is a modeling tool for creating ML-enabled IoT services. It offers full code generation out of software models and comes with a desktop version, as well as a web-based version. It is based on the Eclipse Modeling Framework (EMF) [12] and the Xtext framework [39]. It has a text-based model editor, which uses Xtext and offers typical IDE features, such as syntax highlighting and auto-completion, as well as a tree or form-based model editor, which is based on the EMF tree model editor. ML-Quadrat generates Python and Java code. The Python code is responsible for the ML components and uses the APIs of Scikit-Learn or Keras with the TensorFlow backend. The generated Python and Java codes are seamlessly integrated. In addition, ML-Quadrat inherited the DSL of the modeling tool ThingML [16, 37], including its code generation framework. Thus, it generates code for a range of programming languages and platforms, for example, C code for the Arduino and POSIX platforms.

A software model in ML-Quadrat consists of a structural part, a behavioral part, an ML component, as well as a configuration part. The latter is absent in the case of Platform-Independent Models (PIMs). However, for code generation, Platform-Specific Models (PSMs), which include configurations, are needed [32]. Note that *configurations* in ML-Quadrat are concerned with the entire software model, thus these are different from the *configurations* in MontiAnna. The structural part specifies the *things* in the system, as well as their *ports*, *messages*, *parameters*, *properties* (i.e., local variables), etc. A *Thing* in ML-Quadrat can be seen as an *actor* or *agent* that is connected to the IoT. *Things* can communicate with each other asynchronously through message-passing via their *ports*. Furthermore, each *thing* must have a behavioral part, which is a Finite-State Machine (FSM), also known as a state machine, state diagram, or state chart. The semantics are adopted from the UML 2 standard [1]. It is this part of the software model that deploys the ML model predictions to make the software model *smart*. For instance, the predictions of the ML model may affect the state transitions.

The ML part of the software model typically comprises the specific ML method that should be deployed, for instance, the model architecture family (e.g., the Multi-Layer Perceptron ANN), the hyperparameters, such as the choice of the optimizer or learning algorithm for training the ML model, the number of hidden layers, the layer sizes, and the choice of the activation functions in each layer in the case of ANNs. If a hyperparameter is absent in the software model, its default value in the respective ML library or framework (e.g., Keras/TensorFlow) will be assumed. In the case that an ML

```
1   data_analytics daml1
2     @dalib "keras-tensorflow"                          ML²
3     {
4       labels ON
5       features image_pixels, digit_prediction
6       prediction_results digit_prediction
7       dataset "data/my_mnist.csv"
8       model_algorithm nn_multilayer_perceptron my_nn_mlp(
9         hidden_layer_sizes (128),
10        hidden_layers_activation_functions (relu),
11        optimizer adam, learning_rate_init "0.001",
12        loss SparseCategoricalCrossentropy
13      )
14      training_results "data/training_daml1.txt"
15  }
```

Figure 3: MNIST detector MLP ANN modeled in ML-Quadrat.

method is available in both Scikit-Learn and Keras, the practitioner may explicitly select the one to be used, or the system can decide on its own. Moreover, the practitioner may bring a pre-trained ML model, which has any arbitrary architecture and has been trained with any arbitrary algorithm and connect it to the software model. This brings some flexibility since the options for ML methods will not be limited to the ones that are supported out of the box. Figure 3 illustrates the ML part of the textual model instance in ML-Quadrat that models a software service for automated handwritten digit recognition based on the MNIST reference dataset using an MLP. The DSL keywords are highlighted in blue. The annotation *da_lib* enables the practitioner to select the target ML library. For instance, we support the APIs of both Scikit-Learn and Keras (with the TensorFlow backend) for the MLP ANN method. Moreover, the *labels* keyword specifies whether the data are class labeled (i.e., supervised learning is applicable) or not. The possible options here are *ON*, *OFF*, and *SEMI* for supervised, unsupervised, and semi-supervised learning, respectively. Moreover, the ML features are listed after the *features* keyword. Further, the *prediction_results* keyword can specify where the future predictions of the ML model should be stored once the training is accomplished. Additionally, the *dataset* keyword is used to introduce the path of the Comma-Separated Values (CSV) file that contains the training data on the file system. The core of this part of the software model is the *model_algorithm* specification, which models the ML method that should be created and deployed. In this case study, we train an MLP ANN. The hyperparameters are mostly selected from the respective ML libraries, such as Scikit-Learn. If a hyperparameter (e.g., optimizer) is missing, the default choice of the respective library will be selected automatically. In this example, only one hidden layer with the size of 128 will be created. For instance, if another hidden layer of size 64 had been desired, we would have (128, 64) instead of (128) for the *hidden_layer_sizes* hyperparameter. Likewise, the activation function of each hidden layer needs to be specified through the *hidden_layers_activation_functions* hyperparameter respectively. Furthermore, the rest of the shown hyperparameters specify the optimizer (in this case Adam), the initial learning rate, as well as the choice of the loss function. Finally, the training log will be stored in the stated text file whose path is provided through the *training_results* keyword.

Figure 4 depicts the overall architecture of the IoT service that deploys the mentioned ML components in order to offer automated handwritten digit recognition. This service comprises three *things*, which are represented by the blue rectangles: i) An end-device (such as a smartphone or tablet); ii) A camera; iii) A server for carrying
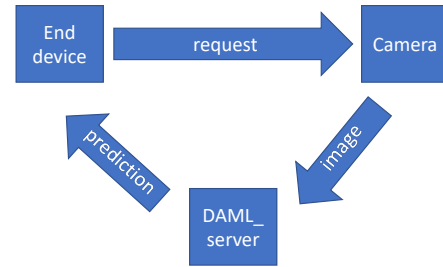
Figure 4: Architecture of the generated service via ML-Quadrat for handwritten digit recognition.

```
1   statechart DAML_server_behavior init Preprocess {
2     on entry print "DAML server started!\n"        ML²
3     state Preprocess {
4       on entry do
5         da_preprocess daml1
6       end
7       transition -> Train
8     }
9     state Train {
10      on entry do
11        da_train daml1
12      end
13      transition -> Ready
14    }
15    state Ready {
16      transition -> Predict
17      event e: image_recognition_service?image
18      action do
19        image_pixels = e.pixels
20      end
21    }
22    state Predict {
23      on entry do
24        da_predict daml1(digit_prediction)
25        image_recognition_service!prediction(digit_prediction)
26      end
27      transition -> Ready
28    }
29  }
```

Figure 5: The behavioral model of DAML_server in ML-Quadrat.

out the Data Analytics and ML (DAML) tasks, called DAML_server. The ML component shown in Figure 3 belongs to the latter.

Finally, the behavioral model of the DAML_server *thing*, which deploys the above-mentioned ML component, as shown in Figure 3, is presented in Figure 5. Initially, the data pre-processing is conducted. Then, the ML model is trained. Next, the system will switch to the ready state, which is the standby state. Once an image is received on the *image_recognition_service* port, its pixel intensities will be provided to the ML model to recognize the digit. Afterwards, the system reverts to the ready state, thus standing by again. Note that the *da_preprocess*, *da_train*, and *da_predict actions* lead to the execution of the data pre-processing (i.e., data preparation), ML model training, and prediction tasks of the ML pipeline. Lastly, the question mark and the exclamation mark, which are used in the statechart in Figure 5 check for receiving a particular message on a specific port, and result in sending a particular message on a specific port, respectively.

3 COMPARISON

Table 1 summarizes a comparison of MontiAnna and ML-Quadrat with respect to several aspects of their capabilities. Below, we elaborate on these by reference to the respective rows. This section

constitutes the second and the third contributions mentioned in Section 1, as on the one hand it shows a functional comparison of the two model-driven tools, while on the other hand it elucidates the broad range of possible support through MDE in the development of ML-driven software.

Problem domain and Integration: The first aspect to be compared in examining the two modeling tools is the specific problem domain on which they focus. MontiAnna offers the possibility to create standalone ML applications, while the ML functionality can also be used in the IoT domain by using MontiThings [21]. Therefore, the ML functionality may be encapsulated into a component and re-used in the EmbeddedMontiArc framework for integration into the full system, or re-used in MontiThings for integration into IoT services. By contrast, ML-Quadrat offers an integrated modeling language, which covers both IoT and ML. In fact, this provides for a one-stop-shop for ML-enabled heterogeneous and distributed services for the IoT. This is realized by the integration of the APIs of the ML libraries and frameworks Scikit-Learn and Keras into the prior work ThingML [16, 37] both at the meta-model level and at the level of code generators.

Modeling methodology and ML methods: In addition, the tools differ concerning the modeling methodology and the corresponding ML methods. MontiAnna is concerned with a particular family of ML models, namely deep ANNs. Accordingly, MontiAnna has its dedicated modeling language for ANNs , called `CNNArchLang`, which provides for layer-specific modeling of ANNs with a broad range of supported layers, as shown in section 2.1. The network architecture is automatically validated, such that - among other checks - it is ascertained that the output dimensions and input dimensions of consecutive layers match. In MontiAnna, the networks can be trained through a supervised learning approach, a reinforcement learning attempt, and unsupervised learning approaches, such as GANs or VAEs. Moreover, there is a strict differentiation between the network architecture and the hyperparameters. A dedicated language called `ConfLang` was developed, which has a JSON-like syntax for configuring a broad range of hyperparameters. These include the number of epochs to be trained, the batch size, and even a nested configuration establishing the optimizer and additionally determining its parameters such as the learning rate. Distinguishing between the network architecture modeling language and a language for hyperparameters builds a clear separation of concerns and avoids mixing up domains. In contrast, ML-Quadrat offers a single DSL for modeling the entire software service or application, including the ML model architecture, the hyperparameters for ML model training, as well as other elements (e.g., IoT components). ML-Quadrat supports two ways of deploying machine learning methods. The practitioner may either select an ML method that is supported out of the box (e.g., MLP ANN, or decision trees), or use a pre-trained ML model. In the latter case, which is called the *black-box* ML mode (given that the ML model is dealt with as a black-box by the software model), the ML model may possess an arbitrary architecture in the supported libraries and could be trained using any arbitrary learning algorithm, method, and techniques, which are supported in the libraries. Hence, the practitioner's options are not limited to the out-of-the-box ML methods. These include linear methods for classification and regression (i.e., logistic regression and linear regression), Naïve Bayes with various kernels

(e.g., Gaussian and Bernoulli), Decision Trees, Random Forests, and MLP ANNs for supervised ML. In addition, K-Means, Mini-Batch K-Means, DB-SCAN, Spectral Clustering, and Gaussian Mixture Model are enabled for unsupervised ML. Last but not least, in the case of semi-supervised learning (i.e., partially labeled data), the Self-Training, Label Propagation, and Label Spreading methods can be deployed.

Target ML libraries & frameworks and ML pipelines: When working with MontiAnna, the ML engineer can configure a *backend*, which is an ML library with a Python interface, to generate the code resulting from the model. So far, Caffe2, TensorFlow, Pytorch, and MxNet/Gluon are supported. This flexibility enables easy benchmarking between different backends and paves the way for experimentation with functionalities only supported in certain backends. The generated code concerning the network architecture and the training procedure is written in Python, while the execution can be done in both Python and C++, as C++ is the most common language in the target domain. However, ML-Quadrat supports the Python ML libraries Scikit-Learn, and Keras (with the TensorFlow backend). Besides the ML component, the rest of the IoT services modeled in ML-Quadrat, may be generated for a range of target IoT platforms, programming languages, and APIs. The choices include, but are not limited to C code for POSIX and Arduino, as well as Java, Javascript, and Go. If Java is desired, the generated Java code will be seamlessly integrated with the generated Python code for the ML component. To this aim, the Java and Python code generator that is offered by ML-Quadrat can be used.

ML pipelines: Typically, ML problems are tackled using an ML pipeline (i.e., workflow). This pipeline can vary depending on the problem. However, it often incorporates some kind of preprocessing of the data (i.e., data preparation), a feature engineering step (usually for non-deep-learning approaches), an ML model training process (i.e., *learning* the optimized parameters in the case of parametric ML models), and an evaluation, which ensures that no *over-fitting* occurs. The pre-processing phase can consist of a change in the color space for images, data cleaning, imputation of missing values, normalization, standardization, and stratified sampling. In the case of End-to-End ML, the ML model itself includes all the pipeline implicitly and does everything self-sufficiently. Thus, when the input data are fed into the trained ML model, it knows all the necessary steps to deliver the final result (e.g., prediction). Modeling a pipeline implies supporting the practitioner at two levels: i) putting together components to create a pipeline; ii) creating the realization of the pipeline components. In MontiAnna, pipelines can be constructed using an established Component and Connector Language, called EmbeddedMontiArc. Components can be created and connected via ports. The realization of the components can take place via the following alternative routes: i) through selection out-of-the-box, for example, a data cleaning procedure provided by the framework; ii) the generation of the neural network and training procedure using the aforementioned capabilities of MontiAnna; iii) be handcrafted by the practitioner to suit the generated interfaces as derived from the components and their ports. In contrast, ML-Quadrat enables data pre-processing through the *da_preprocess* action of the DSL (see Figure 5 in Section 2). Currently, this mostly comprises of feature scaling (i.e., standardization and normalization

of numeric values), as well as label encoding (i.e., one-hot encoding) for categorical labels. The practitioner may either manually adapt the generated Python script for pre-processing or adapt the model-to-code transformation accordingly.

Modularity and Compatibility: Besides the modularity offered through the exchangeability of the pipeline components in MontiAnna, pre-trained networks can be loaded and imported as network layers. They are then trained together with the network architecture modeled around this layer. To guarantee framework interoperability, both export and import are implemented in the ONNX standard format. Furthermore, modularity in the context of ML-Quadrat is provided through the possibility of importing pre-trained ML models of various architectures from the supported ML libraries and (de)serialization of these models, for example, as Python Pickle objects for Scikit-Learn or HDF5 in the case of Keras.

AutoML AutoML is a rising ML subtopic. In MontiAnna, the integration of AdaNet serves as the first step toward AutoML by aiming at conducting a Neural Architecture Search (NAS) for any problem. This approach seeks to create an ANN through an additive growing ensemble by putting together sub-networks. Moreover, with the help of the explained concept of pipelining, AutoML can easily be enabled. Therefore, MontiAnna possesses two levels of AutoML support: i) Automatically optimizing the parameters that define a component, for example, the hyperparameters of a training configuration. ii) At a higher level, complete components can be exchanged, if the exchanged component is taken from a set with suitable interfaces. Additionally, ML-Quadrat supports AutoML at the following two levels. First, it offers rule-based support by checking certain constraints based on the API documentation of the respective backend libraries, and ML domain knowledge. For instance, if a hyperparameter has been set outside the permitted or recommended range, the practitioner can be warned about this. Also, in certain cases, such as scaling numeric data in the data pre-processing of ANNs, or avoiding data shuffling and cross-validation in the case of sequential (e.g., time series) data, decisions will be made and enforced automatically should the AutoML mode be enabled. Second, for certain ML methods, automated ML model architecture/type selection, as well as automated hyperparameter optimization using Bayesian Optimization through the Hyperopt library can be offered. For the latter, the practitioner needs to use the standalone open-source tool AutoNIALM [5], which was designed for a particular use case, namely energy disaggregation, but can be adapted and deployed for other problems as well.

Re-training: In MontiAnna, the training procedure is executed only if the input data or the model have changed. Otherwise, the trained model remains the same as compared to the previous training run. Automated re-training is also implemented in the MontiAnna framework: When an extension of an existing dataset is deployed, an event is triggered that initiates the re-training process of the model. The new training process starts, where the last training process ended and takes over the learned parameter, such as the learning rate that was automatically adapted in the previous run. By contrast, in ML-Quadrat, the practitioner may deploy a timer in the software model to re-train the ML models periodically, or it can occur in an event-based manner following the adopted event-driven programming paradigm (e.g., upon the receipt of a

particular message type on a specific port of the *thing* (i.e., the agent), which contains the respective ML component).

Generated Artifacts and Artifact Management: In MontiAnna, the generated artifacts comprise the source code for the creation of the system, the source code for the training of the ANN model, and either the out-of-the-box functionality for the pipeline components or pre-generated interfaces for the user to realize the pipeline component manually. When bundled as a package, these artifacts constitute the source code archive. Other archives being created by the framework are the ANN model archive, which includes the weights of the trained model, and the dataset archive, which contains the dataset associated with a connection to the ANN model which it was trained with. These packages can be managed with Apache Maven. Maven goals exist for the deployment of the archives as well as for the installation of the archives to the local machine. Similarly, ML-Quadrat generates all the artifacts of the software solution automatically out of the software model, which is designed by the practitioner. These include the entire source code, ML models (ANNs or other ML model families), as well as the build and run scripts. The generated source code is seamlessly integrated with the generated ML models and can train, deploy, and re-train them automatically as required. In the case of Java (and Python), this includes code generation, where the Python scripts are in charge of ML, and the Java code is responsible for the rest of the IoT service functionality; here, Apache Maven is deployed (similar to MontiAnna) for artifact and life-cycle management of the generated software solution. In this case, an executable JAR that contains all the dependencies of the generated IoT service will be produced, and can be used conveniently by the operator or end-user to deploy and run the IoT service.

4 RELATED WORK

Various prior works in the literature addressed the topic of deploying high-level specifications, abstractions, and visual programming, to improve AI (in particular ML) engineering. In the following, we briefly review some of them them. We are particularly interested in those which deployed the MDE paradigm.

First, high-level APIs concerning ML were provided through the ML frameworks, such as Scikit-Learn [34], TensorFlow [13], Torch [10] and Keras [9]. More of these frameworks are presented in [23]. These frameworks come with methods to build, train, and evaluate neural networks as well as other ML models. Usually, they are implemented in C or C++ and accessed via a Python or C++ API. Although they are very comprehensive, the ML engineer has to implement their solutions with a general-purpose language, thus being obliged to learn it for each and every platform that is needed beforehand. Second, ML workflow designers and workbenches, such as KNIME [6], WEKA [15], and RapidMiner [35] aimed for supporting a more efficient ML practice. In addition, visual programming for ML was enabled through a number of tools, such as TensorBoard [36]. Further, Infer.Net [7, 29] proposed MDE for ML. However, they were focused on probabilistic programming, thus using Probabilistic Graphical Models (PGMs) as software models for producing the entire software source code in C# out of them. Moreover, GreyCat [17–19] seamlessly integrated ML into domain models. Their work was similar to ML-Quadrat, but only targeted Java, Javascript,

Table 1: MontiAnna vs. ML-Quadrat

	Properties	MontiAnna	ML-Quadrat
1	Problem domain	ML self-contained (or together with MontiThings for the IoT)	ML-enabled IoT services
2	Integration	Component encapsulation in EmbeddedMontiArc / MontiThings	Encapsulation in ThingML
3	Modeling methodology	Multiple DSLs	Single DSL
4	ML methods	Various deep ANNs with supervised, unsupervised and reinforcement learning	Various supervised, unsupervised, and semi-supervised ML approaches
5	Target ML libraries & frameworks	MXNet Gluon, TensorFlow, PyTorch, Caffe2	Scikit-Learn and Keras (with the TensorFlow backend)
6	Target languages for code generation	Python (training and runtime), C++ (runtime)	Python (for ML), Java, C, Javascript, and Go
7	ML pipelines	Modeled via the DSL; functionality out of the box or implemented by user	Out-of-the-box
8	Modularity and Compatibility	Pre-trained networks can be loaded as network layers and ONNX [33]	Any pre-trained ML model in Scikit-Learn or Keras may be imported/plugged in
9	AutoML	Neural Architecture Search with AdaNet, hyper-parameter optimization and component-based AutoML planned	Rule-based and through Bayesian Optimization (Hyperopt).
10	Re-training	Re-training only if data or model have changed, event-based automated re-training	Event-based automated re-training, for example, when new data arrive, or timer-based.
11	Generated artifacts	Full source code containing ANNs (from `CNNArchLang` models) and ML model training scripts (from `ConfLang`), complete source code and dataflows or only interfaces (from Pipeline Model created with `EmbeddedMontiArc` and `ConfLang`)	ML models, full source code (including the Python scripts for pre-processing, training, and prediction), build and run scripts.
12	Artifact management	Maven-based artifact re-use (source code, trained models, datasets are packaged independently)	Apache Maven projects generated for Java

and Typescript for code generation. Thus, it was not suitable for typical resource-constrained IoT platforms. Another MDE solution for the deep learning domain used in practice is ML.NET, which was developed by Microsoft. The framework promises *"authoring production-grade machine learning pipelines [...]"* [2]. Based on the .NET platform, developers can create pipelines as Directed Acyclic Graphs (DAG) with out-of-the-box functionalities that are easy to share efficiently through an abstraction called DataView. However, the model customizability is limited to the parameters. Furthermore, IBM SPSS Neural Networks [3] offers the possibility to integrate neural networks to IBM SPSS, which is common a software for statistics and data analytics. This approach can simultaneously be seen as a reduction of the manual expense to the minimum and a realization of AutoML. Instead of designing the neural network architecture manually, it is seen as a non-configurable black box simply automatically created based on the data. As IBM SPSS is statistical software, it does not support convolutional layers for images or graph convolutional layers, but the fully-connected architecture model aims at, for instance, solving classical statistical tasks using neural networks. These can thereby serve as a surrogate for linear regression. Finally, Azure Machine Learning is a tool for ML that was developed by Microsoft. It promises end-to-end support for the complete lifecycle of the ML application and is integrated into the cloud environment offered by Azure. Training data can be stored using the Azure Blob storage, and ML models can be trained based on this data using clusters provided by Microsoft. Via the web interface, the developer can create a pipeline in a C&C-like manner, but only with predefined components. AutoML techniques are also supported; for example, if data is uploaded in a CSV format, the user can specify which parts of the file's content serve as a feature and which values are to be predicted by the model. Attached to this are a fully automated construction and the training of a network.

5 CONCLUSION

In this work, we have compared two open-source MDE tools for ML-enabled software systems, namely MontiAnna [4, 23] and ML-Quadrat [30, 31]. First, we have conducted a case study using both tools to introduce them, as well as the concept of modeling-driven engineering of ML software. Thereafter, we have compared the two from a functional perspective. To the best of our knowledge, this work represents the first such study focused on comparative analysis of MDE4ML in the context of IoT.

While MontiAnna has a strong focus on the development of ANNs and the integration of ML functionality in IoT systems via MontiThings, ML-Quadrat supports the use of other ML methods

besides ANNs, and seamlessly integrates the ML functionality and the rest of the smart IoT services. However, ML-Quadrat is limited in terms of out-of-the-box and modular support for advanced ANN architectures. Conceptually, both approaches are based on the idea of incorporating an ML model as a component in a larger system and enabling the code to be generated, although the frameworks and languages generated may differ.

From the functional comparison conducted in this work, we can draw insights on the way model-driven engineering supports different aspects of the development of ML-driven software. The model-driven approach relieves the ML engineer from the burdens of solving the task in the framework-specific implementation, thus shifting the focus towards the quintessence of the development process. Thereby, it helps with the development of ML software as well as with its integration into larger scale software systems, such as IoT applications, through a simple specification of the key parameters without loss in flexibility.

Both tools are research prototypes, which are still under development. However, they are provided as open-source software, and promote open standards (e.g., ONNX compatibility in the case of MontiAnna, as well as interoperability with Scikit-Learn and Keras/TensorFlow in the case of ML-Quadrat). In this way, we expect synergies and network effects in the software engineering and machine learning communities leading to a rapid adoption and extension of MDE tools for machine learning in academia, as well as their exploitation and adoption in the industry.

REFERENCES

[1] 2017. *Unified Modeling Language.* Standard. Object Management Group (OMG). https://www.omg.org/spec/UML/2.5.1/About-UML/
[2] Zeeshan et al. Ahmed. 2019. Machine learning at Microsoft with ML. NET. In *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining.* 2448–2458.
[3] A Al-Imam. 2019. A Gateway Towards Machine Learning: Predictive Analytics and Neural Networks in IBM-SPSS (SPSS v. 24). *Retrieved January* 3 (2019), 2019.
[4] Abdallah Atouani, Jörg Christian Kirchhof, Evgeny Kusmenko, and Bernhard Rumpe. 2021. Artifact and Reference Models for Generative Machine Learning Frameworks and Build Systems. In *GPCE'21.* 55–68.
[5] autonialm 2018. Ukrit Wattanavaekin's Master's Thesis Source Code - Automated Data Analytics for Motifs and Discords Mining. https://github.com/ukritw/autonialm. Accessed: 2022-03-24.
[6] Michael R. Berthold, Nicolas Cebron, Fabian Dill, Thomas R. Gabriel, Tobias Kötter, Thorsten Meinl, Peter Ohl, Kilian Thiel, and Bernd Wiswedel. 2009. KNIME - the Konstanz Information Miner: Version 2.0 and Beyond. *SIGKDD Explor. Newsl.* 11, 1 (Nov. 2009), 26–31. https://doi.org/10.1145/1656274.1656280
[7] Christopher M. Bishop. 2013. Model-Based Machine Learning. *Philosophical Transactions of the Royal Society A* 371, 1984 (February 2013), 1–17. https://doi.org/10.1098/rsta.2012.0222
[8] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. 2015. MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems. arXiv:1512.01274 [cs.DC]
[9] François Chollet et al. 2015. Keras. https://keras.io.
[10] Ronan Collobert, Koray Kavukcuoglu, and Clément Farabet. 2011. Torch7: A Matlab-like Environment for Machine Learning. In *NIPS 2011.*
[11] Corinna Cortes, Xavier Gonzalvo, Vitaly Kuznetsov, Mehryar Mohri, and Scott Yang. 2017. Adanet: Adaptive structural learning of artificial neural networks. In *International conference on machine learning.* PMLR, 874–883.
[12] EMF (publication date not applicable). Eclipse Modeling Framework (EMF). https://www.eclipse.org/modeling/emf/. Accessed: 2022-03-23.
[13] Martín Abadi et al. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. http://tensorflow.org/ Software available from tensorflow.org.
[14] Nicola Gatto, Evgeny Kusmenko, and Bernhard Rumpe. 2019. Modeling Deep Reinforcement Learning Based Architectures for Cyber-Physical Systems. In *MODELS 2019. Workshop MDE Intelligence* (Munich). 196–202.
[15] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten. 2009. The WEKA Data Mining Software: An Update. *SIGKDD Explor. Newsl.* 11, 1 (Nov. 2009), 10–18.
[16] Nicolas Harrand, Franck Fleurey, Brice Morin, and Knut Eilif Husa. 2016. ThingML: A Language and Code Generation Framework for Heterogeneous Targets. In *Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems (MODELS '16).*
[17] Thomas Hartmann, Francois Fouquet, Assaad Moawad, Romain Rouvoy, and Yves Le Traon. 2018. GreyCat: Efficient What-If Analytics for Data in Motion at Scale. arXiv:1803.09627 [cs.DB]
[18] Thomas Hartmann, Assaad Moawad, Francois Fouquet, and Yves Le Traon. 2017. The Next Evolution of MDE: A Seamless Integration of Machine Learning into Domain Modeling. In *2017 MODELS'17.* 180–180. https://doi.org/10.1109/MODELS.2017.32
[19] T. Hartmann, A. Moawad, F. Fouquet, and Y. Le Traon. 2019. The next evolution of MDE: a seamless integration of machine learning into domain modeling. *Software and System Modeling (SoSyM)* 18 (May 2019), 1285–1304. https://doi.org/10.1007/s10270-017-0600-2
[20] Nils Kaminski, Evgeny Kusmenko, and Bernhard Rumpe. 2019. Modeling Dynamic Architectures of Self-Adaptive Cooperative Systems. *The Journal of Object Technology* 18, 2 (July 2019), 1–20. https://doi.org/10.5381/jot.2019.18.2.a2 The 15th European Conference on Modelling Foundations and Applications.
[21] 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), 1–21. https://doi.org/10.1016/j.jss.2021.111087
[22] Evgeny Kusmenko. 2021. *Model-Driven Development Methodology and Domain-Specific Languages for the Design of Artificial Intelligence in Cyber-Physical Systems.* Shaker Verlag. http://www.se-rwth.de/phdtheses/Diss-Kusmenko-Model-Driven-Development-Methodology-and-Domain-Specific-Languages-for-the-Design-of-Artificial-Intelligence-in-Cyber-Physical-Systems.pdf
[23] Evgeny Kusmenko, Sebastian Nickels, Svetlana Pavlitskaya, Bernhard Rumpe, and Thomas Timmermanns. 2019. Modeling and Training of Neural Processing Systems. In *MODELS'19* (Munich). IEEE, 283–293.
[24] Evgeny Kusmenko, Svetlana Pavlitskaya, Bernhard Rumpe, and Sebastian Stüber. 2019. On the Engineering of AI-Powered Systems. In *ASE19. Software Engineering Intelligence Workshop (SEI19)* (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
[25] Evgeny Kusmenko, Alexander Roth, Bernhard Rumpe, and Michael von Wenckstern. 2017. Modeling Architectures of Cyber-Physical Systems. In *ECMFA'17* (Marburg) *(LNCS 10376).* Springer, 34–50.
[26] 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 *MODELS'18* (Copenhagen). ACM, 447 – 457.
[27] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. 2015. Deep learning. *nature* 521, 7553 (2015), 436–444.
[28] Jure Leskovec, Anand Rajaraman, and Jeffrey David Ullman. 2014. *Mining of Massive Datasets* (2nd ed.). Cambridge University Press, USA. http://www.mmds.org
[29] T. Minka, J. M. Winn, J. P. Guiver, Y. Zaykov, D. Fabian, and J. Bronskill. 2018. Infer.NET 0.3. Microsoft Research Cambridge, http://dotnet.github.io/infer, accessed: 2020-09-08.
[30] ML-Quadrat 2020. ML2. https://github.com/arminmoin/ML-Quadrat. Accessed: 2020-09-12.
[31] Armin Moin, Moharram Challenger, Atta Badii, and Stephan Günnemann. 2022. A model-driven approach to machine learning and software modeling for the IoT. *Software and Systems Modeling (SoSyM)* (2022). https://doi.org/10.1007/s10270-021-00967-x
[32] Armin Moin, Moharram Challenger, Atta Badii, and Stephan Günnemann. 2022. Supporting AI Engineering on the IoT Edge through Model-Driven TinyML. https://arxiv.org/abs/2107.02690
[33] ONNX (publication date not applicable). Open Neural Network Exchange. https://github.com/onnx. Accessed: 2021-03-09.
[34] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.
[35] RapidMiner (publication date not applicable). Depth for Data Scientists, Simplified for Everyone Else. https://rapidminer.com. Accessed: 2021-09-08.
[36] TensorBoard (publication date not applicable). TensorFlow's visualization toolkit. https://www.tensorflow.org/tensorboard. Accessed: 2021-09-08.
[37] Things Modeling Language 2015. ThingML. https://github.com/TelluIoT/ThingML. Accessed: 2020-04-29.
[38] Jon Whittle, John Hutchinson, and Mark Rouncefield. 2014. The State of Practice in Model-Driven Engineering. *IEEE Software* 31, 3 (2014), 79–85. https://doi.org/10.1109/MS.2013.65
[39] Xtext (publication date not applicable). LANGUAGE ENGINEERING FOR EVERYONE! https://www.eclipse.org/Xtext/. Accessed: 2022-03-20.