Modeling Deep Reinforcement Learning based Architectures for Cyber-Physical Systems

Nicola Gatto, Evgeny Kusmenko, Bernhard Rumpe
Chair of Software Engineering, RWTH Aachen University, Aachen, Germany, kusmenko@se-rwth.de

***Abstract*—Reinforcement learning is a sub-field of machine learning where an agent aims to learn a behavior or a policy maximizing a reward function by trial and error. The approach is particularly interesting for the design of autonomous cyber-physical systems such as self-driving cars.**

In this work we present a generative, domain-specific modeling framework for the design, training and integration of reinforcement learning systems. It consists of a neural network modeling language which is used to design the models to be trained, e.g. actor and critic networks, and a training language used to describe the training procedure and set the corresponding hyperparameters. The underlying component model allows the modeler to embed the trained networks in larger component & connector architectures. We illustrate our framework by the example of a self-driving racing car.

***Index Terms*—cyber-physical systems, machine learning, reinforcement learning, domain-specific languages**

I. Introduction

The design of intelligent cyber-physical system (CPS) is a complex process which requires appropriate tools, languages, and frameworks. The component & connector (C&C) paradigm, well known from tools such as Simulink [1] and LabView [2], is a model-driven development approach widely used in engineering research and practice; it enables the developers to decompose a system in tractable functional blocks, the so called components. Components can talk to each other over directed connectors connecting their typed output and input ports. There are two main ways to describe the behavior of a component: first, a component can be modeled as an interconnection of multiple subcomponents. Second, it can be implemented using a behavior description or a general purpose language. For instance, Simulink components can be implemented as MATLAB code.

With the advance of machine learning technology, we have been seeing more and more problems being tackled using decision trees, support vector machines, deep neural networks, and other data-driven learning techniques. In CPSs, perception tasks are often implemented as convolutional neural networks (CNNs), since this kind of networks is known to perform well in image processing applications. On the other hand, reinforcement learning (RL) is an approach which has been researched for training decision making and control networks. By trial and error, an agent learns actions maximizing a numerical reward in a given environment. Implementing and training reinforcement learning systems is therefore inherently different from standard supervised learning tasks where a fixed dataset with labeled examples is used for training. In this paper we present a model-driven and component-oriented approach for the design and generation of reinforcement learning models and their integration into larger systems. Therefore, we extend the deep learning domain-specific language (DSL) family MontiAnna [3], originally developed for supervised learning tasks, by concepts specific to the RL domain. We employ the C&C paradigm to encapsulate and couple the different parts of an RL-based system and provide a simple, yet fully functional application example modeling an agent controlling a car in the racing simulator TORCS.

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

```
1 import torcs.agent.network.TorcsActor;
2 component Master {
3   ports in Q^{29} state,
4     out Q(-1:1)^{3} action;
5
6   instance TorcsActor actor;
7
8   connect state -> actor.state;
9   connect actor.action -> action;}
```

Listing 1. The main EMADL component of a racing car control system.

II. Background

A. EmbeddedMontiArc for Deep Learning

In this section, we give a brief overview of EmbeddedMontiArcDeepLearning (EMADL), a textual modeling language family based on the C&C paradigm and designed for a type-safe and verifiable development of embedded and CPSs. The core language of EMADL is the architecture description language (ADL) EmbeddedMontiArc (EMA) [4], [5] enabling the developer to model an architecture as a system of hierarchically organized interacting components.

In EMA, a component is defined using the component keyword followed by the component's name, e.g. in line 2 of listing 1; a component interface is defined as a set of strictly typed input and output ports (lines 3-4 in listing 1). The type system supports the mathematical primitive types `B`, `Z`, `Q`, and `C` denoting Booleans, integers, rational, and complex numbers, respectively. A type can be extended to a vector or a matrix by defining its dimensionality. For instance, the input port `state` in line 3 of listing 1 is a 29-dimensional vector of rational numbers. Furthermore, a type can be refined with a range as

[GKR19] N. Gatto, E. Kusmenko, B. Rumpe:
Modeling Deep Reinforcement Learning Based Architectures for Cyber-Physical Systems.
In: Proceedings of MODELS 2019. Workshop MDE Intelligence, pp. 196--202, IEEE, Munich, Sep. 2019.
www.se-rwth.de/publications/

is done in line 4, restricting the port to output values between -1 and 1 in this case. Optionally, we can specify the physical unit the port is representing.

Sub-components are instantiated using the `instance` keyword followed by the component type and the instance name, e.g. in line 6 of listing 1. Components interact only over explicit connectors connecting two compatible ports using the `connect` keyword as in lines 8-9 of listing 1.

To describe the behavior of a component, which cannot be decomposed into smaller components, EMADL offers two behavior modeling languages, MontiMath and MontiAnna. **MontiMath** is used to describe a behavior with the help of mathematical expressions [4]. The syntax is based on MATLAB, but has the same strict mathematical type system as EMADL considering ranges, units, and fixed matrix dimensions. A MontiMath implementation of a component is given in listing 5.

MontiAnna is a modeling language family for the design of deep neural networks and is hence of particular interest for this work. A MontiAnna model consists of a neural network architecture, modeled as a graph of neuron layers, and a training description capturing the hyperparameters [3]. Listing 2 shows an EMADL component whose implementation is defined using the MontiAnna network architecture language in lines 7-14. A designer can choose from a variety of neural network layer types, e.g. fully connected and convolutional layers, to assemble a network architecture.

MontiAnna provides the sequential data flow operator "->" to express that the output of the left operand is input into the right operand. Furthermore, the parallelization operator "|" can be used to split the data flow, enabling the design of complex neural network architectures. Input and output ports of an EMADL component encapsulating a MontiAnna implementation can be used as layers of the MontiAnna network. For instance, the input and output ports `state` and `action` defined in lines 3-4 of listing 2 are mapped to the input and output layers of the network used in lines 7 and 14, respectively. Further syntactic elements for the architecture definition, e.g. layer stacking using structural operators, facilitate a concise modeling of large network architectures.

While the MontiAnna architecture model describes the structure of the neural network, a separate training model is used to specify how to train it. The latter enables the developer to define a set of typed hyperparameters including the number of training episodes, batch size, loss function, etc.

B. Reinforcement Learning

RL describes a set of methods in the field of machine learning. As introduced in [6], there are two main elements in RL, the *environment* and the *agent*. The main characteristic is that the agent performs actions in order to maximize a numerical reward. In contrast to other machine learning methods, in RL the agent does not learn from an existing data set and there is no correct action for a given situation. Instead, it has to explore the action space to find action sequences maximizing the reward. We consider the interaction to be discrete. At each time step, the agent receives the state of the environment. Based on this state, it selects an action. The environment answers with a reward and samples the next state. Many RL methods try to learn the policy π, the action-value function Q, or both. A policy determines how an agent acts. It is basically a mapping of states to actions. The Q function estimates the quality of a state-action pair.

Deep RL describes the combination of deep learning techniques and RL. A lot of successful applications resulted from this combination. One current example is Alpha Zero [7] in which a trained agent was able to defeat world champions in complex board games like Go.

Two popular deep RL algorithms are the Deep Q-Network (DQN) [8] and the Deep Deterministic Policy Gradient (DDPG) [9]. DQN can be applied to RL tasks with a discrete action space. In the algorithm, a neural network is used to approximate the Q-function. The algorithm had great success in the training of an agent that learned to play Atari games. For the training, DQN utilizes experience replay [10]. After each step in the environment, the transition (state, action, reward, next-state) is stored in a buffer. Every fixed number of steps, a minibatch is drawn from the buffer to train the Q-network. The policy of the agent is derived from the trained Q-function. For each possible action, the Q-value is estimated with the help of the neural network. In every step, the agent selects the action with the highest Q-value.

DDPG combines DQN, Deterministic Policy Gradient (DPG) [11], and actor-critic methods to tackle continuous action spaces. DDPG learns both a concrete policy π and an action-value function Q. The policy is called *the actor* and the Q-function is called *the critic*. For both functions, neural networks are used for the approximation. The critic is trained similar to DQN. The actor is trained with the help of the critic. It is optimized by maximizing the expected return. Hence, the policy maximizes the Q-function.

III. Requirements

MontiAnna was originally developed for the design of CNNs and recurrent neural networks (RNNs) with supervised learning in mind. In this work we investigate the suitability of the framework for the design of RL-systems given the following requirements.

(R1) There is no single RL algorithm, but rather a broad spectrum of different methods, which should be chosen carefully based on the problem to tackle. Hence, an RL framework should enable the user to specify the learning algorithm and its respective hyperparameters.

(R2) In deep RL, a neural network is used as a function approximator. The user should be able to model different architectures that fit the RL task. Thereby, the modeler should be able to assign roles to the networks, e.g. the actor and the critic role, when using algorithms like DDPG.

(R3) RL problems always involve interaction with an environment. The environment is usually not explicitly designed for the interaction with an RL agent. A modeling language

must ensure that interaction is possible for different environments. From the view of the modeler, the interface and the behavior of that environment should be time-invariant.

IV. Related Work

There are various general-purpose deep learning frameworks like TensorFlow (TF) [12] or Theano [13] that are applicable for the realization of deep RL tasks. They support the user by providing powerful utility modules from low to high-level functionality. The advantage of these frameworks is that the user gets a lot of control. From the neural network architecture to the training, each element of the algorithm can be adapted and tuned individually. TF, for instance, provides the higher-level `layers` module for modeling neural network architectures, but the user can also control operations on weights and gradients to adjust the network to his/her needs.

Every specialized neural network architecture and every special operation for the training and inference is realizable. It is possible to integrate any kind of environments, as the solution for the integration must be written and tailored to the problem domain manually. The disadvantage of this kind of frameworks is that they are rather difficult to use. First, a user requires a deep knowledge about the different library functions to be able to compose them. Second, one needs a profound understanding of the algorithms in order to be able to implement them.

More specialized frameworks tailored to the RL domain tackle the disadvantage of general-purpose frameworks. In contrast to frameworks like TF, the user is not required to implement the RL learning functionality from scratch. Such frameworks usually provide predefined agents implementing the necessary algorithms out-of-the-box. For instance, OpenAI Baselines [14] provides a number of state-of-the-art implementations of RL algorithms which a developer can utilize for his/her own RL problems. OpenAI Gym contains a series of environments helping to get up to speed with RL and well-suited for experimentation and model comparison. A disadvantage of OpenAI is that it only provides the implementation of the algorithms. The user needs to provide glue code in order to integrate these implementations in a larger system.

DeepMind TensorFlow Reinforcement Learning (TRFL) [15] builds upon TF and provides building blocks for the realization of RL algorithms. For instance, it provides loss functions and common update rules specialized to the needs of RL. In contrast to OpenAI Baselines, the library does not deliver ready-to-use implementations of RL algorithms, but can be seen as a tool that provides algorithmic components to build RL agents. Hence, it provides the same flexibility and control like the general-purpose frameworks, but is more convenient to use. Nevertheless, the user still needs knowledge about the implementation of an algorithm.

Tensorforce [16], like TRFL, builds upon TF. However, it is more of a high-level framework including agent and model libraries with pre-defined interfaces. The target of the framework is to provide modular and configurable RL components which the user can integrate directly for his/her own tasks. There are various agents for different RL algorithms like DDPG or DQN to choose from. The user needs to specify the state space, action space, the network architecture, and an optimizer. Furthermore, the user can configure the training with a lot of parameters, e.g. for the replay memory and the exploration phase. The network is specified as a Python array of layer definitions or as a JSON file. For training, the library provides a runner object, which is called with the specified agent and the environment. Tensorforce supports a lot of ready-to-use environments, for example from OpenAI Gym, OpenAI Universe, Deepmind Lab, etc. The user can also integrate proprietary environments by implementing a Python interface. This interface requires typical RL methods like reset, execution of an action, and information about the state and action space. In contrast to TRFL and the general-purpose frameworks, Tensorforce is very easy to use. The user can integrate RL components without the need for a profound knowledge about the actual implementation of the RL algorithms.

Google Dopamine [17], like Tensorforce, provides ready-to-use agents. The training of an agent is defined in a gin configuration file[1]. In this file, the user specifies the environment to use as well as the learning parameters. Furthermore, the user can modify an agent by subclassing one of the available agents. Dopamine provides support for the Arcade Learning Environments as well as discrete OpenAI Gym environments.

A component-based approach is given by the Reinforcement Learning Toolbox by Mathworks [18]. To model architectures with RL agents, the toolbox provides predefined agent blocks for Simulink. A block is similar to a component in EMADL. The agent block provides the input ports *observation*, *reward*, and *isdone* as well as the output ports *action* and *cumulative reward*. The block can be connected to other Simulink blocks.

The toolbox provides predefined MATLAB/Simulink environments and allows the definition of user-defined ones. A user-defined environment can be created by specifying the observation, action, and reward functionality according to a typical RL interface. Furthermore, it is possible to integrate third-party environments, e.g. by using an external language interface or a functional mockup unit (FMU). To train an agent, the toolbox provides a functional interface. The training can be configured using a configuration object.

V. Modeling Reinforcement Learning

While the discussed frameworks are powerful tools for the implementation and usage of RL solvers, they are bound to a host general purpose language (GPL), mostly Python. Furthermore, the realizations based on these frameworks often require adaptations in order to use them as components in larger architectures. In this section, we present our approach to model and train RL components using DQN and DDPG by extending the EMADL and MontiAnna DSL family. With our work, we provide a model-based and generative design approach for RL components, which is independent of the

[1] https://github.com/google/gin-config

underlying technology or programming language. Independent of a concrete implementation, the user declares how (s)he wants his/her RL component to be trained using a training DSL. The trained model can then be embedded as a standard component into any C&C architecture.

While many of the presented frameworks require a profound understanding of the used algorithms, our language extension is meant to be used for training and inference of an RL component as a black box. The hyperparameters are set in a declarative way.

In the following subsections we will present RL modeling concepts for DQN and DDPG with the help of a running example. In this example, we will model an RL agent controlling a car in the racing simulator TORCS [19]. The state space is given as $S \subseteq \mathbb{Q}^{29}$, holding information about the velocity of the car, the angle of the track axis, and data from 20 range finder sensors returning the distance to the edges of the track. An action consists of three continuous values in the range $[-1, 1]$ representing the steering, the acceleration, and the braking of the car.

A. Function Approximators

```
1  package torcs.agent.network;
2  component TorcsActor {
3    ports in Q^{29} state,
4      out Q(-1:1)^{3} action;
5
6    implementation MontiAnna {
7      state ->
8      FullyConnected(units=300) ->
9      Relu() ->
10     FullyConnected(units=600) ->
11     Relu() ->
12     FullyConnected(units=3) ->
13     Tanh() ->
14     action; } }
```

Listing 2. EMADL component of the actor. The implementation is described using the MontiAnna architecture language.

```
1  package torcs.agent.network;
2  component TorcsCritic {
3    ports in Q^{29} state,
4      in Q(-1:1)^{3} action,
5      out Q^{1} qvalue;
6    implementation MontiAnna {
7      (
8        state ->
9        FullyConnected(units=300) ->
10       Relu() ->
11       FullyConnected(units=600)
12     |
13       action ->
14       FullyConnected(units=600)
15     )->
16     Add() ->
17     FullyConnected(units=600) ->
18     Relu() ->
19     FullyConnected(units=1) ->
20     qvalue; } }
```

Listing 3. EMADL component of the critic. The implementation is described using the MontiAnna architecture language.

Both RL algorithms use neural networks as their function approximators. DQN approximates the Q-function, while DDPG approximates the Q-function (critic) and the policy (actor). Usually, the networks get the current states of the environment as inputs. The Q-network of DQN estimates how good each possible action is, while DDPG outputs the action to be performed directly. For the TORCS example we are going to apply DDPG, since the action space is continuous. Consequently, we need to model an actor and a critic network, which we can accomplish using MontiAnna's network description language in listings 2 and 3.

The actor implementation in listing 2 is a feedforward network taking the state vector sent by the TORCS environment as its input in line 7. The output in line 14 is a vector representing the three actions steering, acceleration, and braking. The network is built of three fully connected layers instantiated in lines 8, 10, and 12 and consisting of 300, 600, and 3 neurons, respectively. The activation function is set to ReLU for the first two layers in lines 9 and 11 and to TanH for the output layer in line 13.

The implementation in listing 3 models the critic network and approximates the Q-function. Hence, it takes as input the state and action vectors in lines 3 and 4. MontiAnna allows us to model parallel paths using the "|" operator as is done in line 12. We use this operator to model different paths for the state and action inputs. The path for the state input is shown in lines 8-11 while the action path can be seen in lines 13 and 14. The two paths are merged by applying the "Add" operation in line 16 which causes the vectors resulting from both paths to be summed up element-wise. As the critic outputs a single Q-value, the last layer, defined in line 19, is a linear fully connected layer with a single unit.

The critic is only used for the training of the actor component and is not needed at execution time. Therefore, the critic component is not embedded into another C&C architecture, but is rather used as a stand-alone component by the training program. The actor component, on the other hand, is run during execution time to decide which action to take next. Hence, the designer can embed the actor directly into a complex C&C architecture, e.g. a robot control system, as a standard EMADL component by instantiating and connecting it to other components as is done in listing 1. In case DQN is used as the learning algorithm, the function approximator is modeled in a similar way to the actor in DDPG.

B. Training Parameters

We extended the training language of MontiAnna to support RL-specific parameters. Listing 4 shows a training model for the TORCS actor. In lines 3-4 the designer states that the TorcsActor component is trained using RL and sets the training algorithm to DDPG. Alternatively, the user can select between DQN and Twin Delayed DDPG (TD3) [20].

For the training, we require a critic network. But how does the compiler know where to find the critic component for a given actor training? We can link a critic to an actor component

by providing the fully qualified EMADL component name of the critic network in line 5 of the training model in listing 4.

Furthermore, the designer is able to configure typical RL-hyperparameters including the number of episodes to train (line 13), the discount factor (line 14), or the target update rate (line 19). An optional target score parameter is defined in line 37 and serves as a stopping criterion: the training is stopped if the average reward over the last 100 episodes is equal or greater than the target score. The snapshot interval parameter in line 20 causes the weight parameters of the agent to be stored every 150 episodes. By default, the weights are only stored after the training of the agent.

The replay memory is defined in terms of an operation mode: `buffer` is the standard replay memory mode introduced in [8], [10], where state-action-reward pairs are stored in a buffer. For each training step, we sample a minibatch. Additionally, we provide the `online`-mode, which is effectively no replay memory, as well as the `combined` mode, representing the replay memory introduced in [21], where the last performed state-action-reward pair is added to the minibatch.

The exploration strategy is defined as an operation mode, as well, cf. lines 25-34 in our training model. We support the ε-greedy strategy for discrete action spaces and the Ornstein-Uhlenbeck (OU) process as well as Gaussian noise for continuous ones. For all operation modes, the designer can choose an epsilon ε (see line 26) to set the randomness or noise level for the action selection. The parameters following in lines 27-30 allow us to model a reduction of the epsilon parameter over the course of the training. Line 27 sets the reduction method to `linear`, i.e. ε is reduced by the value set in line 29 after each episode until the minimal value defined in line 30 is reached. In line 28 we declare that the reduction strategy has to be applied only after the 10th episode. Depending on the chosen strategy, strategy-specific parameters can be set, as well, e.g. the parameters theta, mu, and sigma for OU defined in lines 31-33. These parameters allow us to control the generated noise. The parameter mu, for example, determines the mean value to which the generated noise of the OU process will drift. We can control the noise for each action dimension individually, which is why each of the lines 31-33 holds three values. This allows us to model individual mean and standard deviation values for steering, braking and accelerating.

C. Reward Function

The reward function is an important element in the design of RL tasks. If it is provided by the environment, we obtain it from the environment interface and, thus, do not have to model it explicitly. If, however, the reward function is not provided by the environment, the designer needs to model one. As the reward is usually a mathematical function, the designer can provide its definition as an EMADL component with a MontiMath implementation.

Listing 5 shows the reward component for our TORCS RL example. In our framework, the reward component has two inputs. The first input receives the current state of the environment with its type depending on the application, while the second one is a Boolean flag indicating whether the last received state is a terminal state. Furthermore, we assume that the output, i.e. the actual reward, is always a rational number.

In lines 7-13, the actual MontiMath implementation of the reward function is shown. In lines 7-9, we extract the velocity,

```
1  training TorcsActor {
2    context : gpu
3    learning_method : reinforcement
4    rl_algorithm: ddpg-algorithm
5    critic: torcs.agent.network.torcsCritic
6    environment : ros_interface {
7     state_topic : "/torcs/state"
8     terminal_state_topic : "/torcs/terminal"
9     action_topic : "/torcs/step"
10    reset_topic : "/torcs/reset"
11   }
12   reward_function: torcs.agent.network.
         reward
13   num_episodes : 3000
14   discount_factor : 0.99
15   num_max_steps : 900000
16   training_interval : 1
17   start_training_at: 0
18   evaluation_samples: 50
19   soft_target_update_rate: 0.001
20   snapshot_interval : 150
21   replay_memory : buffer{
22    memory_size : 120000
23    sample_size : 32
24   }
25   strategy : ornstein_uhlenbeck{
26    epsilon : 1.0
27    epsilon_decay_method: linear
28    epsilon_decay_start: 10
29    epsilon_decay : 0.0001
30    min_epsilon : 0.0001
31    theta: (0.6, 1.0, 1.0)
32    mu: (0.0, 0.0, -1.2)
33    sigma: (0.3, 0.2, 0.05)
34   }
35   actor_optimizer: adam { learning_rate:
         0.0001 }
36   critic_optimizer: adam { learning_rate:
         0.001 }
37   target_score: 100000}
```

Listing 4. Configuration of the actor training modeled using the training language of MontiAnna.

```
1  package torcs.agent.network;
2  component Reward {
3    ports in Q^{29} state,
4      in B isTerminal,
5      out Q reward;
6    implementation Math {
7      Q speedX = state(22);
8      Q angle = state(1);
9      Q trackPos = state(21);
10     reward = speedX * cos(angle);
11     if abs(trackPos) > 1.0
12       reward = -20;
13     end } }
```

Listing 5. Component defining the reward of the TORCS environment with the help of MontiMath.

the angle between the car and the track axis, as well as the normalized distance to the track edges from the state vector we received from the environment. The actual reward calculation takes place in line 10. The formula consists of the two components speed and alignment of the vehicle with the street: higher speeds are rewarded, but the agent is penalized if the angle between car and street increases. Furthermore, we penalize the agent if it drives outside of the track with an absolute negative reward of -20 in lines 11-13.

Like the critic network, the reward component needs to be linked to training in the training model. This is done in line 12 of listing 4 by specifying the fully qualified name of the reward component to be used. In each step of the training, the state vector is passed to this reward component in order to retrieve a numerical reward. Of course, we can also integrate the reward component into our architecture model by connecting the corresponding ports.

D. Environment

During the training time of the RL system, the trainer needs to interact with the environment in order to train the agent. Likewise, during the execution time of the model, the predictor or rather the whole C&C model interacts with the environment. We employ the Robot Operating System (ROS) [22], a publish/subscribe middleware mostly used in the robotics domain for the communication of distributed modules, to realize the communication between the agent, the critic, and the environment. Therefore, we require the environment to provide the following publish/subscribe interface, based on the general interface of OpenAI Gym environments: `state` is the current state of the environment. Usually, it is represented by a feature vector or a matrix of rational numbers. The concrete shape depends on the application. `terminal` is a Boolean flag indicating whether the last received state is a terminal state. `reward` is an optional topic, which can be used by the environment to publish a reward for each action performed. If it is not present, the designer must define a reward function as a MontiMath-based EMADL component.

The agent publishes the action to be performed via the `step` topic. If the action space is discrete, the data type is an integer, otherwise it is either a scalar or vector of rational numbers. A step always causes the environment to send its next state, the terminal information and, optionally, a reward. The environment listens to a further topic called `reset`. If the message is set to "true", the environment restarts itself. A reset should always be followed by a message holding the initial state, a terminal flag that is false and, optionally, a reward.

Any application providing the interface described above can be used as training environment for our framework by providing a ROS adapter. The environment can be any software application, e.g. a simulator, or a physical machine with sensors and actuators.

For training, the designer specifies a mapping of the interface to the names of the corresponding ROS topics in the training model. In the TORCS example, the environment is an adapter that enables us to communicate with TORCS via ROS. Lines 6-11 of listing 4 show how we connect the interface of the environment with the trainer. Eventually, the designer can connect the ports of the final, trained C&C model to any environment via ROS using the middleware tagging approach as described in [23].

E. Code Generation

EMADL provides a generator mapping models to executable C++ code. Depending on the selected deep learning backend, the generation of MontiAnna models is delegated to a backend generator. Thereby, *predictor* and *trainer* artifacts are generated. Creation and training of the neural network are carried out by the trainer. It instantiates the neural network with the help of the chosen deep learning backend, e.g. MxNet or TF, and trains it according to the training model. After training, the architecture of the neural network and its weight parameters are exported to a file format, which can be loaded by the predictor. The predictor is a C++ component which is embedded into the executable EMADL code. Its task is to load the neural network together with its parameters to make predictions for incoming inputs at runtime.

We extended the EMADL code generator to be able to generate and train a fully functional RL system from an EMADL model including a MontiAnna neural network description and a corresponding RL training model. If the newly added parameter `learning_method` is set to `reinforcement` in the training model as is done in line 3 of listing 4, the generation strategy is changed. First, we check RL-specific context conditions on the composed architecture and training models in order to verify that the requirements of the chosen RL algorithm are satisfied. For instance, in the case of DDPG we ensure that the state and action dimensions of the critic correspond to those of the actor. Then, a fully functional solution including the training and final deliverable code is created.

Figure 1 shows the overall architecture of the generated artifacts. Training and execution time are depicted on the lhs and the rhs, respectively. The trainer is generated as Python code using the deep learning library MXNet Gluon. The figure shows the different components the trainer utilizes in order to train the network. The `RLTrainer` module provides an entry point for the training and creates the required neural networks based on the defined architectures. Furthermore, it possesses a Python dictionary with all the parameters and values defined in the training model.

Both the created neural networks and the training parameters are passed to the `Agent` component which implements the actual RL algorithm for the training. During the training, the agent interacts with the environment in order to exchange states and actions. In case a reward model is declared, executable C++ code for this model is generated, as well. Then, all states are passed to the executable reward model in order to obtain the reward value. Furthermore, the training system provides information such as the actor loss and the average reward. After training, weight parameters of the neural network are exported to a file such that the

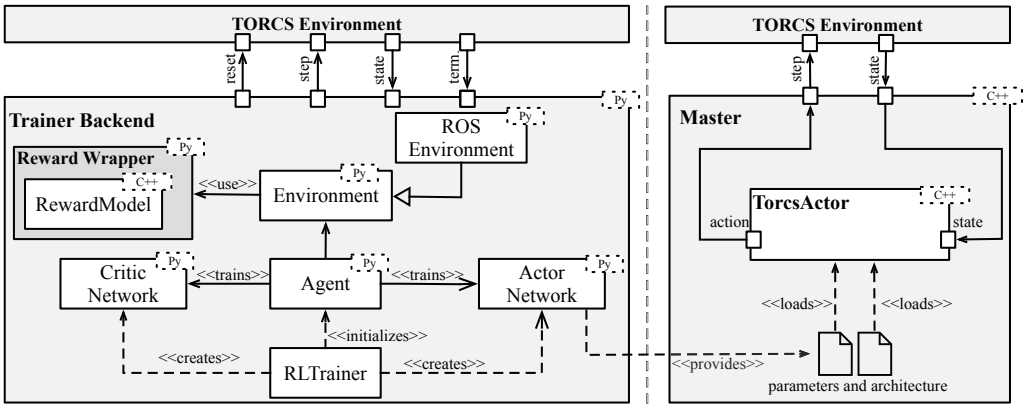

Fig. 1. Generated Architecture: The left side shows the architecture during the training. The right side is the architecture during execution time of the model.

predictor of the executable model is able to load it. A report containing statistical information about the training process is saved for analysis, as well. At this stage, the training of the RL component is finished and the user can execute the model.

VI. Conclusion

In this paper, we introduced a model-driven framework enabling us to design deep reinforcement learning systems using the MontiAnna language family and to embed the resulting networks as components in larger C&C architectures. Therefore, we extended the MontiAnna training language by reinforcement learning concepts and designed a component-based training system, where EMADL components represent different roles of the reinforcement learning process such as actor, critic, and reward. Domain-specific context conditions defined for this component model ensure the consistency of the overall system. A complete modeling example of a self-driving racing car was used to illustrate the concepts.

References

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

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

[3] Evgeny Kusmenko, Sebastian Nickels, Svetlana Pavlitskaya, Bernhard Rumpe, and Thomas Timmermanns. Modeling and Training of Neural Processing Systems. In *Conference on Model Driven Engineering Languages and Systems (MODELS'19)*. IEEE/ACM, September 2019.

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

[5] Evgeny Kusmenko, Bernhard Rumpe, Sascha Schneiders, and Michael von Wenckstern. Highly-Optimizing and Multi-Target Compiler for Embedded System Models: C++ Compiler Toolchain for the Component and Connector Language EmbeddedMontiArc. In *Conference on Model Driven Engineering Languages and Systems (MODELS'18)*, pages 447 – 457. ACM/IEEE, October 2018.

[6] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, 1998.

[7] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *arXiv preprint arXiv:1712.01815*, 2017.

[8] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing Atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

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

[10] Long-Ji Lin. Reinforcement learning for robots using neural networks. Technical report, Carnegie-Mellon Univ Pittsburgh PA School of Computer Science, 1993.

[11] David Silver, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, and Martin Riedmiller. Deterministic policy gradient algorithms. In *Proceedings of the 31st International Conference on International Conference on Machine Learning - Volume 32*, ICML'14, pages I–387–I–395. JMLR.org, 2014.

[12] Martın Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. *Software available from tensorflow.org*, 1(2), 2015.

[13] Theano Development Team. Theano: A python framework for fast computation of mathematical expressions. *CoRR*, abs/1605.02688, 2016.

[14] Prafulla Dhariwal, Christopher Hesse, Oleg Klimov, Alex Nichol, Matthias Plappert, Alec Radford, John Schulman, Szymon Sidor, Yuhuai Wu, and Peter Zhokhov. OpenAI Baselines. https://github.com/openai/baselines, 2017.

[15] DeepMind Technologies Limited. Tensorflow reinforcement learning: TRFL. https://github.com/deepmind/trfl, 2018.

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

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

[18] Inc. The MathWorks. Reinforcement learning toolbox. https://www.mathworks.com/products/reinforcement-learning.html.

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

[20] Scott Fujimoto, Herke van Hoof, and Dave Meger. Addressing function approximation error in actor-critic methods. *CoRR*, abs/1802.09477, 2018.

[21] Shangtong Zhang and Richard S. Sutton. A deeper look at experience replay. *CoRR*, abs/1712.01275, 2017.

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

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