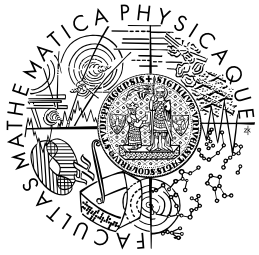


FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

Michal Töpfer

Machine-learning-based self-adaptation of component ensembles

Department of Distributed and Dependable Systems

Supervisor of the master thesis: prof. RNDr. Tomáš Bureš, Ph.D.

Study programme: Computer Science – Artificial Intelligence

Prague 2022

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources. It has not been used to obtain another or the same degree.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ....

Author's signature

I would like to thank Milad Ashqi Abdullah, MSc for all the effort he put into the development of the simulation and the framework. I would also like to thank the supervisor of this thesis – prof. RNDr. Tomáš Bureš, Ph.D. – and all of the members of the SmartArch research group, namely doc. RNDr. Petr Hnětynka, Ph.D., doc. RNDr. Martin Kruliš, Ph.D., and Mgr. Danylo Khalyeyev, for consultations and advice during the development of the project. Finally, I would like to express gratitude to my family and friends who supported me during my studies.

Title: Machine-learning-based self-adaptation of component ensembles

Author: Michal Töpfer

Department: Department of Distributed and Dependable Systems

Supervisor: prof. RNDr. Tomáš Bureš, Ph.D., Department of Distributed and Dependable Systems

Abstract: In the area of distributed self-adaptive smart systems (such as applications of Internet of Things and Cyber-Physical Systems), machine learning has been successfully used in several applications including the prediction of metrics regarding the components in the system (e.g., battery consumption), and pruning of the space of possible adaptations. It is clear that machine learning can be a useful tool in self-adaptive systems. Most of the research works focus on using the machine learning algorithms for a specific task, yet they are (at least partially) lacking in providing a systematic approach to the introduction of machine learning into the architecture of the system.

In this thesis, we propose ML-DEECo – a machine-learning-enabled component model for adaptive component architectures. It is based on the concepts of autonomous components and their ensembles (coalitions) from the DEECo component model. We enrich DEECo with abstractions for specifying machine-learning-based estimates directly in the architecture of the system. The architect can thus focus on the business logic of the application while all the tasks necessary to provide the estimates (such as collecting the data and training the model) are provided by our runtime framework. We provide an implementation of the ML-DEECo runtime in Python and evaluate it by constructing simulations of self-adaptive systems from the areas of smart farming and Industry 4.0.

Keywords: self-adaptive, machine learning, components, ensembles

Contents

1. Introduction

Smart systems such as the Internet of Things (IoT) and Cyber-Physical Systems (CPS) are becoming more and more popular in recent years. The self-adaptive smart systems are able to dynamically adapt and reconfigure based on the situation in the environment to optimize their behavior and performance to fulfill their goals. With the use of machine learning and the ability to construct predictions of the future state of the system or its components, these systems are able to make otherwise complicated decisions, and to deal with uncertainty.

The smart systems are often distributed, which has implications on the software engineering of their design and architecture. To manage the complexity and describe the dynamically evolving architecture of such systems, architectural models featuring cooperating components have been introduced. One of the approaches to engineering such systems is the concept of autonomic component ensembles, which proved to be useful for modeling dynamic architectures of cooperating autonomous components as demonstrated by several papers [1, 2, 3, 4, 5, 6]. We base our work in this thesis on the DEECo ensemble-based component model [1], which is one of the approaches for describing and modelling the architecture of a distributed smart system.

The DEECo component model features abstractions for defining components with autonomous behavior. To allow communication among the individual components, the concept of ensembles – implicit and dynamic groups of components mutually cooperating to achieve a particular goal – is introduced. The ensembles are formed and dismantled dynamically based on the changes in the attributes of the components.

In this thesis, we focus on enriching the ensemble resolution process with machine learning algorithms. In particular, we want to introduce machine-learning-based estimates into the components and ensembles. The estimates can be used to predict values (such as the future state of a component) that the self-adaptive systems can base their adaptation decisions on.

1.1 Problem statement

Although many applications of machine learning in adaptive systems have been shown (and examined in several literature reviews [7, 8]), most of the works focus on applying machine learning to a specific task. They are (at least partially) lacking in providing a systematic approach to the introduction of machine learning into the architecture of the system. It is clear that machine learning can provide useful insights for the adaptation of the system, but it is not clear how to systematically introduce machine learning into the component models and thus to the architecture of the system.

Our aim is to design an ensemble-based component model with machine learning and adaptation as first-class concepts. We want the system architect to be able to focus on the business logic of the application rather than the intricacies of machine learning (such as data collection and training). We want to provide the architect with abstractions for obtaining machine-learning-based estimates of the values in the system (e.g., attributes of the components) simply

by declaratively marking those values in the architecture. Our implementation does all the necessary work to obtain such estimates (data collection, model training, inference, etc.) so that the architect can use them without having a deep knowledge of machine learning.

1.2 Goals

In this thesis, we enrich the DEECo component model [1] with machine-learning-based estimates which can be specified directly in the architecture of the system. We call the resulting component model ML-DEECo.

The ML-DEECo component model should provide the necessary abstractions for describing the architecture of a distributed smart system. That includes specification of components of the system and their behaviors, as well as their communication (via ensembles). It should also provide abstractions for an easy inclusion of machine-learning-based estimates into the architecture of the system, which can benefit the self-adaptability of the system.

Towards the design and implementation of ML-DEECo, we address several goals:

Goal 1 Identify and analyze possible usages of machine learning in a component-based architecture of an ensemble-based adaptive system.

Goal 2 Design the component model semantics such that the machine learning can be realized without having to write data collection or machine learning procedures.

Goal 3 Map these concepts to a widely-used programming language (Python in our case).

1.3 Structure of the text

In Chapter 2, we first introduce the running example from the smart farming domain (Section 2.1), then we provide an overview of the concepts of Ensemble-Based Component Systems (EBCS) and specifically DEECo (Section 2.2) and lastly we give an overview of the machine learning concepts used in this thesis (Section 2.3).

Chapter 3 focuses on modeling machine-learning-enabled ensembles. We first analyze the possible usages of machine learning in Ensemble-Based Component Systems by providing examples of such tasks in Section 3.1 and classifying them into a taxonomy in Section 3.2, thus addressing Goal 1. Section 3.3 addresses Goal 2 and describes the design and semantics of the machine-learning-based estimators used in ML-DEECo.

In Chapter 4, the documentation of the ML-DEECo framework implementation in Python is provided and examples of its usage are given (Section 4.3). This chapter addresses Goal 3.

We evaluate our work in Chapter 5 by providing an implementation of the running example using the ML-DEECo framework to show that ML-DEECo can be used to easily develop a simulation of a self-adaptive machine-learning-enabled

ensemble-based component system. An overview of the implementation is given in Section 5.1 and the results of using machine-learning-based estimates in the simulation to aid the adaptation are given in Section 5.2. Additionally, we show a second example of usage of the ML-DEECo framework in Section 5.3.

We provide a survey of the related work in Chapter 6. Approaches to ensemble-based component modeling are outlined in Section 6.1. Other works focused on the usage of machine learning in adaptive systems are reviewed in Section 6.2.

2. Background and running example

In this chapter, we first introduce the running example which will be used throughout the thesis to illustrate the concepts. The running example comes from from the smart farming domain and it is outlined in Section 2.1. Then, in Section 2.2, we provide an overview of the concepts of Ensemble-Based Component Systems (EBCS) which are an approach to architecting distributed software systems. We focus specifically on the DEECo component model [1] as our work in this thesis builds on it. Lastly, we give an overview of the main machine learning concepts used in this thesis in Section 2.3.

2.1 Running example

The running example comes from the recently finished ECSEL JU project AFarCloud[1] that focused on smart farming. The scenario focuses on the protection of fields of crops against flocks of birds using autonomic flying drones.

A visualization of the scenario developed within the project can be seen in Figure 2.1. The yellow fields represent the crops that need protection. The flocks of birds are represented by the bird icon in the figure. The circular arrows in the middle of the figure are charger stations for charging the drones.

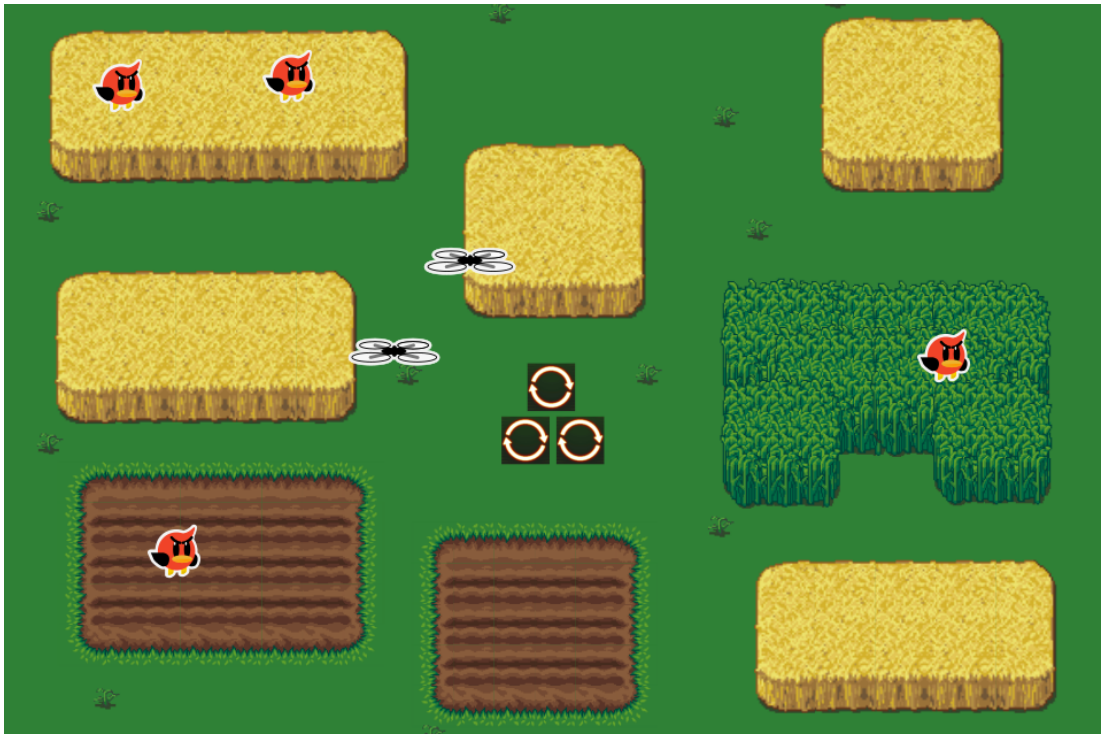


Figure 2.1: Running example visualization.

The drones are autonomous and work together on field protection. When a

[1] https://www.ecsel.eu/projects/afarcloud

flock of birds is detected in a field, the drones fly there to scare them away (using the noise from their propellers) to areas that do not need protection. Depending on the size of the field, several drones might be required for its protection as otherwise the birds will just fly to a different part of the field and eat or damage the crop there.

The drones are battery-powered so they need to be recharged in order to keep protecting. There can be several chargers in the scenario.

The autonomous drones need to cooperate in two key areas of the scenario. The first is field protection. It does not make sense for all the drones to fly to a field when it needs protection as only one or several drones are needed to scare away the birds. Furthermore, as the drones can be scattered across the whole farm, the decision of which drones will fly to the field can be also influenced by the time needed for the drone to fly there.

The second area of drone cooperation is charging. The charger stations have limited capacity, so only a subset of the drones can be charged simultaneously. The battery of the drones is limited, so we want them to be able to plan the charging and not run out of battery in the middle of a field. The drones should also work together to optimize the utilization of the charger, otherwise, too many drones might need charging at the same time resulting in some of them running out of battery.

The scenario can be considered an adaptive system with the positions of flocks of birds and the attributes of drones (i.e., position, battery level, state) constituting the inputs to the adaptation and the assignment of drones to fields and chargers constituting the adaptation actions.

2.2 Ensemble-Based Component Systems

The Ensemble-Based Component Systems (EBCS) are an approach to architecting resilient distributed systems. In EBCS, the component composition is not explicit and static. The components are autonomous and they are dynamically grouped into *ensembles* – implicit and dynamic groups of components mutually cooperating to achieve a particular goal.

The EBCS can be seen as "Distributed systems composed of components that feature autonomic and (self-)adaptive behaviors and are organized into emergent ensembles to achieve cooperation" [1].

In this thesis, we will use the concepts from the DEECo component model [1], which are summarized in the following sections. We outline the related EBCS in Section 6.1.

Distributed Emergent Ensembles of Components (DEECo), presented by Bureš et al. in 2013 [1], is a component model based on the EBCS approach. There are two first-class concepts in DEECo: *component* and *ensemble*.

2.2.1 Component

Components in DEECo are autonomous agents in the environment. They can sense the environment and also obtain information about other components via the ensembles. Based on the knowledge they have, the components can independently operate in the environment.

There are often multiple instances of a component type in the system. When we talk about components later in the text, we mean the component instances unless "type" is explicitly mentioned.

An example of a component type in our running example is a `Drone`. All the individual drones in the system, which are instances of the `Drone` component type, use the same set of rules (program) to autonomously decide what to do. They can sense the environment and use that information to alter their behavior.

2.2.2 Ensemble

An ensemble represents a group of components and their interaction. In DEECo, ensembles are the only way for components to bind and communicate with one another. The ensembles are used for knowledge exchange among a group of components – the members of the ensemble. This way, the members can obtain information about the other components in the ensemble.

The ensembles are formed and re-formed dynamically at runtime. One component plays a role of a *coordinator* of the ensemble and initiates the formation. Other member components are determined dynamically based on a membership condition.

Similar to the components, there can be multiple instances of the same ensemble type. Again, we mean the ensemble instances unless "type" is explicitly mentioned later in the text.

The ensembles can be used for example to coordinate the protection of a field. When the sensors report birds on a certain field, an ensemble of drones can be assembled to coordinate the protection of the field. The number of members of the ensemble can depend on the size of the field to fully cover the field to protect it.

Ensembles example

An example of field protection ensemble instances can be seen in Figure 2.2. We can see that there are birds in three of the five fields, so we instantiate the ensembles for the three fields which need protection. Each of the ensemble instances selects the closest drones and assigns them the task to protect the field (the drone then starts flying towards the field to scare away the birds). As we can see, the number of necessary drones can depend on the size of the field – for the red and blue ensembles, we need two drones for protection, while for the cyan ensemble, only one drone is necessary. The remaining two fields are not under attack by birds, so we do not need to protect them. We can also see that one of the drones is currently being charged (the greyed-out one in the middle), so we cannot use the drone for protection. Lastly, we have one idle drone which is not a member of any ensemble, as we already have enough drones to protect all the fields.

To put this formally in the DEECo concepts, we can have component types `Drone` and `Field`, where `Field` represents sensors for detecting birds present in a given field. The `FieldProtection` ensemble type can be used to coordinate the protection of the fields. When a `Field` detects birds, it will initiate the formation of an `FieldProtection` ensemble instance (the `Field` component becomes the coordinator of the ensemble). The members are selected from all the `Drone`

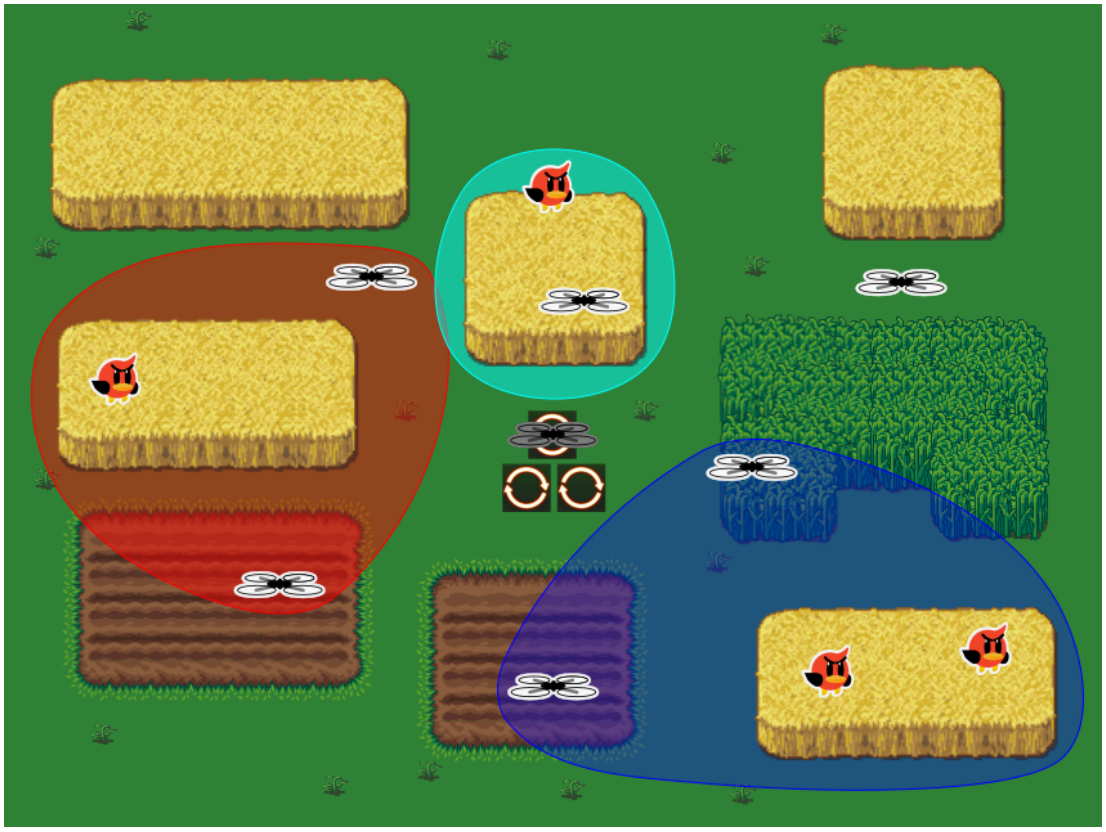


Figure 2.2: Example of three ensembles for field protection.

component instances which are available (not being charged). Based on the size of the field, a certain number of drones are selected while selecting the closest drones first. The ensemble formation can be executed for all the fields at the same time – we can thus divide the drones among the fields in an optimal way (to prevent as much damage as possible).

2.2.3 Ensemble formation

There are several approaches to the formation of ensembles. In the original DEECo [1], the ensemble formation is initiated by the coordinator and only the membership condition is used to determine the members – all the components which satisfy it become members.

There are several other approaches to the ensemble formation in DEECo, some of them described in more detail in Section 6.1.3. Most notable of the other approaches are the COP-based [9] and the ML-based [10]. In the COP-based approach, the ensemble formation is formulated as a constraint optimization problem, and the ensembles are created to maximize a given utility function. The ML-based approach represents ensemble formation as a classification problem and uses machine learning to determine which ensembles should be formed.

In this work, we use a greedy approach to the ensemble formation described in more detail in the following section.

Greedy ensemble formation

To allow for greedy ensemble formation, the ensemble definition must be expanded. In the system, we will have a list of *potential* ensembles – all the ensembles which can potentially be formed. In each time step of the system runtime, the ensemble formation process is executed and some of the potential ensembles are selected to become active in this time step – we call these the *materialized* ensembles.

Each potential ensemble will have a *priority* which determines the order in which the ensembles are considered during the ensemble formation process.

To specify the members of the ensemble, we use a concept of a *role*. There can be *static* roles, which are specified when the potential ensemble is instantiated and cannot change, and *dynamic* roles, for which the member components are dynamically selected by the ensemble formation process. The roles are resolved separately in the order of their definition.

Each role has a *select* predicate, *utility* function and *cardinality* (both minimum and maximum). During the ensemble formation process, all the components in the system are considered possible members for a role. The components are first filtered using the select predicate. Then, they are ordered by the utility function. The maximum cardinality is used to pick the correct number of components with the highest utility as the members of the ensemble in a given role. If there are not enough components to satisfy the minimum cardinality requirement, the ensemble cannot be materialized in this time step.

In the select predicate and the utility function, the information about already materialized ensembles can be used. The member components are selected one at a time and the select predicate is evaluated repeatedly, so the information about already selected members is available when deciding whether the current component should be selected to be a member of the ensemble.

To continue with the example of an ensemble for field protection, we can define its select, utility, and cardinality functions. The select predicate can pick only those drones which are idle, meaning that they do not currently work on the protection of another field. The utility function can be the inverse of the distance to the field to order the drones from the closest to the furthest. As already suggested earlier, the cardinality can be set based on the size of the field, because several drones are enough to fully cover the field and we do not need more drones than that.

2.3 Overview of machine learning concepts used in the thesis

Machine learning is a popular area of artificial intelligence. It focuses on constructing prediction models based on data. We usually use a dataset of training examples to create the model, which can be then used to generate predictions for new, previously unseen, data. The model is usually represented as a function that takes an example (one item from the dataset) as an input and produces the output.

A commonly accepted definition of machine learning was provided by Mitchell in 1997 [11]:

> A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E.

Based on the structure of the experience E we have, the area is often divided to *supervised*, *unsupervised*, and *reinforcement* learning.

In supervised learning, the examples in the dataset consist of features, which can be used as inputs to the model, and also the desired outcomes (called *targets*). The two most common tasks in supervised learning are *regression* and *classification*. In regression, the targets are real numbers, the model thus predicts a numeric quantity. For instance, we might want to construct a model to predict the price of a house based on several attributes such as the number of bedrooms, size of the house and the garden, etc. The targets for classification come from a fixed finite set of categories (classes), the model is thus constructed to predict to which category the example belongs, which can be represented either as only one class or as a probability distribution over all the classes. An example of a common classification task is image recognition. The model gets an image as its input and the objective is to identify which objects (from a predefined set of options) are on the image.

In unsupervised learning, the data come without annotations. Common tasks include clustering and embeddings into a latent space. In clustering, we group the data into several clusters of similar examples. Embeddings are a representation of examples (e.g., words in a sentence) as fixed-length vectors of numbers.

Reinforcement learning is a special category of learning, which develops a goal-seeking agent trained using rewards from the environment. The agent is placed in an environment (simulation), which he can (partially) observe. Based on the observations, the agent can make decisions and perform actions in the environment. The agent can then learn from observing the results of the action and also from an additional reward signal, which tells him whether he accomplished the goal.

In this work, we will focus only on supervised learning.

2.3.1 Input data and features

The training data for supervised learning form a set of N examples. Each example is a tuple $(\vec{x_i}, y_i)$, where $\vec{x_i} \in R^D$ is a vector of D input features and y_t is the target[2]. The examples usually correspond to the instances in the dataset, for example, in the real estate price prediction, each house is one example and the attributes of the house (such as the number of bedrooms) are used as input features.

In practice, the data are usually considered a matrix $X \in \mathbb{R}^{N \times D}$ on N examples, each with D dimensions. The targets are $Y \in \mathbb{R}^N$ for regression and $Y \in \{0, 1, \ldots, K-1\}^N$ for classification into K classes.

To obtain the real-valued features from the attributes of the instances in the dataset, some transformation is often performed. The simplest transformation is normalization, which scales the numeric attributes to a certain range in order to prevent attributes with large values from having a too big influence on the result.

[2]We can also have multiple targets; then, the y_i will also be a vector.

If all the attributes are scaled to the same range (e.g., to $[0, 1]$), the model can learn which features are important for the prediction. For categorical attributes, one-hot encoding can be performed to convert the K categories (initially represented as a single scalar with K possible values) into K binary features, each corresponding to one of the categories.

2.3.2 Neural networks

Artificial neural networks (NNs) are a popular class of models used in all areas of machine learning. They were first introduced by McCulloch and Pitts in 1943 [12]. Since then, numerous architectures of NNs were created for various tasks.

The neural networks are usually a layered architecture with several neurons in each layer. Each neuron performs a simple operation: a scalar product of the inputs with a trainable vector of weights followed by a non-linear function. The outputs of the neurons in one layer are considered inputs for the next layer. We can also think about the scalar product as connections of the inputs to the neuron. Each such connection has an associated weight. The training algorithm updates the weights of the connections in order to find a mapping from the inputs to the outputs.

In this thesis, we work with the simplest neural network architecture called *multilayer perceptron* (MLP, also known as *fully-connected* or *dense*). The MPL consists of an input layer, several hidden layers of neurons, and the output layer. The neurons in successive layers are connected in a fully-connected manner, meaning that each neuron in a layer is connected to all the outputs of neurons in the previous layer. There are no connections between neurons in the same layer, and there are also no connections between layers that are not immediately successive. There can be one or multiple hidden layers, which determine the depth of the network. Deeper networks have higher capacity and are able to learn more complex tasks, but are also more prone to overfitting.

For the training of neural networks, the stochastic gradient descend algorithm is often used. First, a loss function must be defined to assess the quality of the predictions. The loss function indicates how far the predictions are from the targets. Smaller loss means a better prediction. The gradient descend algorithm computes the partial derivatives of the loss with respect to the parameters of the model (weights in the neural network). The weights are then updated slightly against the gradient to make the loss function smaller (to improve the prediction). This process of computing the gradient of the loss and updating the weights is repeated several times for all training data.

3. Modeling machine-learning-enabled ensembles

In this chapter, we aim to address Goals 1 and 2 of this thesis.

To address Goal 1 (*Identify and analyze possible usages of machine learning in a component-based architecture of an ensemble-based adaptive system.*), we start by providing several examples of possible usages of machine learning in a self-adaptive system in Section 3.1. Based on the examples, we provide a taxonomy of the prediction tasks in a self-adaptive system in Section 3.2.

We focus on Goal 2 (*Design the component model semantics such that the machine learning can be realized without having to write data collection or machine learning procedures.*) in Section 3.3. We design the estimators needed to solve the tasks defined in Section 3.2 and deal with the problem of collecting data for training of the estimators. The designed estimators provide the abstractions needed in ML-DEECo to define the machine-learning-based estimates directly in the architecture of the system.

3.1 Examples of prediction tasks

In our approach, we focus on the application of supervised machine learning in the architecture of ensemble-based component systems. We want to be able to describe the estimated quantities in the architecture of the system and use them for decisions regarding the run and self-adaptation of the system.

In this section, we provide several examples of questions regarding the system, which can be relevant to the self-adaptation of the system and can be answered using supervised machine learning.

We can be interested in some pretty straightforward estimates:

- What will the battery level of the drone be after a given amount of time?
- How much energy (from the battery) does the drone need to fly to the nearest charger?
- Will a particular field be attacked by birds in the near future?

But we can also reason about some quite complex questions regarding the system:

- How many drones will need charging in the near future, e.g., one minute from now?
- How many drones will be waiting for a certain charger to become available at some point in the future?
- How long will a drone have to wait before it is accepted to a charger (because the charger is occupied by other drones)?

- At what time should the drone start flying towards the charger to get there before its battery runs out while trying to keep protecting a field for as long as possible?
- How long will a field stay safe (without birds)?

3.2 Taxonomy of prediction tasks

Based on the examples of prediction tasks shown in the previous section, we have identified two important dimensions for classifying the prediction tasks related to ensemble-based component systems – *where* and *what*. The taxonomy is summarized in Figure 3.1 and the dimensions are described in the following sections.

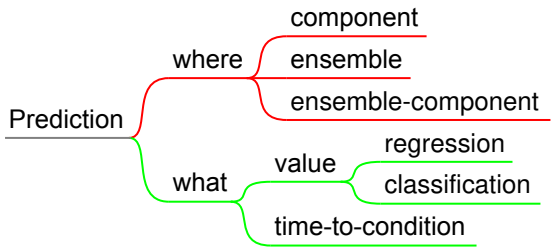


Figure 3.1: Taxonomy of prediction tasks.

3.2.1 Where

The *where* dimension represents where the estimate is needed and what features can be used as inputs. As we have only two main concepts in the ensemble-based component systems – components and ensembles, we have identified three spots where to assign the machine-learning-based estimates:

1. to a component type;
2. to an ensemble type;
3. to an ensemble–component pair (i.e. to a dynamic role in our DEECo-based approach).

For a component-assigned estimate, the attributes of the component can be the input features. That includes both the internal state of the component as well as its believed knowledge of the surrounding environment and other components. For instance, the attribute `position` of a `Drone` component can be used as the input to an estimator that predicts the energy needed to get to the closest charger.

For an ensemble-assigned estimate, the input features can be the attributes of the members of the ensemble. In our work, the ensembles are stateless, so the only attributes available are those of the members in static roles as these do not change during the runtime of the system (only the members in dynamic roles are assigned and re-assigned dynamically). In other approaches, the ensembles can have a state which can also be transformed into input features. For example, the ensemble-assigned estimate can be used to predict how many components will become members of the ensemble in the next time step.

For estimates assigned to an ensemble-component pair, both the attributes of the component and the attributes of the members of the ensemble can be used. If we want to use the estimate during the member selection process, we can use the attributes of the potential member and also the knowledge of the already selected members for the role as well as members of the ensemble in other roles. For instance, if we have an ensemble for management of the charging of the drones, we can have an estimate for the waiting time before a drone is accepted for charging – this estimate can be used by the ensembles to assess whether a potential member will have to wait for a long time and thus should ask for charging soon to survive the waiting without running out of battery.

3.2.2 What

The *what* dimension relates to the predicted quantity. We have identified two important tasks relevant to self-adaptive systems – *future-value* estimate and *time-to-condition* estimate.

Future-value estimate

The future-value estimate is a well-known task in supervised machine learning applied to the self-adaptive systems area. We use the current available observations to predict an unknown value, which will become known at some time in the future. Once the value is known, we can observe it and use that to train the machine learning model. An example of this would be a prediction of the battery level of a drone after a set interval of time. That is exactly a value that is not known now (as the battery drain can depend on the actions performed by the drone) but will become known in the future.

There are two commonly known tasks for the value estimate – *regression* and *classification*. In regression, the objective is to predict a numeric quantity, usually a real number or a vector of real numbers. The predicted quantity in classification comes from a fixed set of classes, the model thus often predicts a probability distribution over the classes, which can be interpreted as the probability of an input object (represented by the input features) belonging to each of the classes.

Time-to-condition estimate

In a time-to-condition estimate, the objective is to predict how long it will take until a defined condition becomes true. This is especially handy for proactive control in the self-adaptive system. The task is defined by specifying a condition over future values of attributes of the components or ensembles. The predictor can again use the current values of the attributes to compute the estimated time.

3.3 Estimators

In this section, we want to focus on describing the estimators that can be used for the tasks defined in the *what* dimension in the previous section.

Obviously, as we want to use machine learning to solve the tasks, we need to somehow define the inputs and outputs of the machine learning model. The inputs

are attributes of the components and ensembles, so we need to preprocess them into features that can be processed by the model. We have already suggested the most common methods of preprocessing, such as one-hot encoding for categorical features, in Section 2.3.1.

The outputs in future-value estimate are also attributes of components and ensembles, so they can be treated in the same way. For time-to-condition estimates, the only output is the predicted time, so there is no need to define it again in the architecture of the system.

The declarations of the inputs and outputs by the architect of the system can be realized as getter functions for the attributes of the components and ensembles. We use this approach in our implementation as we later show in Section 4.2.4.

3.3.1 Training data collection

The other key problem which needs to be solved is how to collect the data for training the machine learning model. We assume that the system is run in discrete time steps – in each time step, the ensembles are materialized and the components perform their actions.

One thing to consider is that the inputs and outputs might not be valid at all times of the simulation. For instance, the drone components might run out of battery and become inactive. In such a situation, we do not want to collect the training data as they might be misleading for the model and damage our predictions. We solve this using guard predicates which determine the validity of the inputs and outputs. These guards can use the attributes of the components and ensembles the estimate is attached to.

Future-value estimate with fixed time difference

In a future-value estimate, we want to predict an unknown value, which will become known at some time in the future. The specification of the target value can be done similarly to the specification of the inputs. The only thing which needs to be done is the matching of the inputs and the targets (true outputs).

In the simplest case, we know how long it will take before the value is known and the time difference between the inputs and the targets is fixed. This is for example the prediction of the battery level 50 time steps into the future – we predict a value that will become known after 50 time steps.

We perform the following steps in every time step:

1. If the inputs guard predicate is true, we collect the input features and tag them with the current time.
2. If the outputs guard predicate is true, we collect the target features. We then associate the target features with the input features based on time – as the difference between the inputs and the targets is fixed, we can easily link the targets with the appropriate inputs. If the output guard predicate is false, we discard the inputs which would correspond to the targets at the current time.

Future-value estimate with a range of time differences

The situation is more complicated if the time difference between the observation of the inputs and the outputs is not fixed. If we can at least define a range of possible time differences, we can generalize the data collection from the previous section. Note that the allowed range of differences only influences the training data, the model can still be able to generalize and predict values for time differences outside the range.

The steps are as follows:

1. If the inputs guard predicate is true, we collect the input features and tag them with the current time.
2. If the outputs guard predicate is true, we collect the target features. We then associate the target features with all the inputs from the allowed time range. This can thus create as many training examples as there are allowed time differences in the range (i.e. $\mathit{maximum\_allowed\_time\_difference} - \mathit{minimum\_allowed\_time\_difference} + 1$). If the output guard predicate is false, we discard the inputs which would correspond to the oldest inputs still in the allowed range (these will not be in the allowed range of time differences in the next time step).

We can illustrate this on an example. If we set the allowed range of differences to the minimum of 1 and the maximum of 10 time steps, we will create 10 training records in each time step. For example, at time 15, we will collect the outputs and pair them with the inputs collected at times $15 - 1 = 14$, $15 - 2 = 13$, ..., $15 - 10 = 5$, assuming all the guard predicates are true.

Future-value estimate with custom inputs and targets matching

If the time interval between the prediction and the observation of the true value is not known in advance, one has to link the inputs and the targets manually using a unique identifier. Otherwise, the data collection process can be very similar (we link the inputs and targets by the identifier).

Time-to-condition estimate

For a time-to-condition estimate, the objective is to predict how long it will take before a given condition over the attributes of a component or an ensemble becomes true. The predicted quantity is thus the time difference between making the prediction and the condition becoming true. We propose using a buffer to collect the data.

We perform the following steps in every time step:

1. If the inputs guard predicate is true, we add the input features with the current time as a new record into the buffer.
2. If the condition is true, we associate all the records in the buffer with the time difference between the current time and the time they were added to the buffer. We clear the buffer.

This way, we use all the time differences between making a prediction and the condition becoming true for the training. This ensures that we collect varying

time differences and we are thus able to predict reasonable values for both the near and the far future.

3.3.2 Neural networks architecture

In this work, we use neural networks, namely the multilayer perceptron, for the predictions. Different tasks can be realized by using a different output layer of the network.

Future-value estimate

For a future-value regression task with one target – prediction of one real number, we use a neural network with one output neuron. Depending on whether the target values come from a bounded range of values, one can optionally limit the values produced by the network. Our particular implementation is detailed in Section 4.4.2. If we had more than one target, each of them would have its own output neuron.

For a future-value classification task with one target – prediction of one value from a fixed set of K classes, we create a neural network with K output neurons and the softmax activation function. This is a common way of predicting a probability distribution over the classes. We then predict the class with the highest probability.

Time-to-condition estimate

The time-to-condition estimate task is a special case of regression. We again construct a model with one output neuron predicting the time difference. This time, we use the exponential function as activation in order to predict only positive values (as we know that the time difference cannot be negative).

4. ML-DEECo implementation

In this chapter, we describe our implementation of the machine-learning-enabled component model for adaptive component architectures called ML-DEECo. The implementation addresses Goal 3 of the thesis and shows a mapping of the design concepts and semantics defined in Chapter 3 to Python programming language[1].

ML-DEECo is implemented as a Python package and provides DSL[2] to allow specifying the components and ensembles, and assigning machine-learning-based estimates to them. The source code of the framework is available at GitHub [13].

We start with a description of our implementation of DEECo concepts in Python in Section 4.1. Then, we extend it with machine learning in Section 4.2. Section 4.3 shows two complete examples of ML-DEECo usage – a component with a value estimate, and an ensemble with a time-to-condition estimate. In Section 4.4, we provide details on the implementation of the neural networks used in our framework.

4.1 DEECo in Python

The ML-DEECo framework is built on top of DEECo concepts (see Section 2.2 and [1]). In this section, we provide an overview of the implementation of components and ensembles. Our Python implementation uses the greedy method of ensemble formation described in Section 2.2.3.

4.1.1 Specifying components

The autonomous components are represented using classes derived from a base class `Component`, which we provide in the `ml_deeco.simulation` module. Each component has an `actuate` method, which is periodically called by our simulation runtime (once per time step of the simulation).

Furthermore, we provide abstractions for simulations on a 2D map, namely `Point2D`, `StationaryComponent2D` and `MovingComponent2D`. The `Point2D` class represents a point on the map. The `StationaryComponent2D` is a base class for components which do not move during the simulation and have a fixed location (specified as `Point2D`). For components which do move during the simulation, the base class is `MovingComponent2D`. It adds a `move` method which can be used to move the component towards a target (a `Point2D`).

Two examples of components can be found in Listings 4.1 and 4.2. The former shows a definition of a stationary charging station and the latter show a definition of a moving drone component.

```
1 from ml_deeco.simulation import StationaryComponent2D
2
3 class Charger(StationaryComponent2D):
4
5     def __init__(self, location):
6         super().__init__(location)
7         self.charging_drones = []
```

[1] `https://www.python.org/`.
[2] Domain specific language.

```
8
9       # a drone at the location of the charger can start charging
10      def startCharging(self, drone):
11          if drone.location == self.location:
12          self.charging_drones.append(drone)
13
14      def actuate(self):
15          # we charge the drones
16          for drone in self.charging_drones:
17          drone.battery += 0.01
18              if drone.battery == 1:
19                  # fully charged
20                  drone.station = None
```

Listing 4.1: Example of definition of a stationary component.

```
1  from ml_deeco.simulation import MovingComponent2D
2
3  class Drone(MovingComponent2D):
4
5      def __init__(self, location, speed):
6          super().__init__(location, speed)
7          self.battery = 1
8          self.station = None # charging station
9
10     def actuate(self):
11         # if the drone has an assigned charger
12         if self.station:
13             # fly towards it
14             if self.move(self.station.location):
15                 # drone arrived at the location of the charger
16                 self.station.startCharging(self)
```

Listing 4.2: Example of definition of a moving component.

4.1.2 Specifying ensembles

Ensembles represent groups of components in the system. They are implemented as classes derived from the `Ensemble` class defined in `ml_deeco.simulation` module.

As stated in the description of greedy ensemble formation in DEECo (Section 2.2.3), we create all the instances for potential ensembles in the system at the beginning of the simulation. In each time step, the framework will decide which ensembles are formed in that time step and with which member components – we call these the *materialized* ensembles.

Each potential ensemble has a `priority` method, which is used to order the ensembles for materialization – the framework takes the ensembles in descending priority and decides which of them are materialized. When the ensemble is materialized, the `actuate` method is called.

The members of the ensembles are specified using static and dynamic roles. The static roles are defined when the potential ensemble is initialized and are represented as variables of the ensemble type class. The dynamic roles are assigned by the framework in every time step. The dynamic roles are represented using `someOf` (a list of components) and `oneOf` (single component) constructs assigned as properties of the ensemble type class in our DSL. For each role, the component type (class) must be set.

To select the members for a dynamic role, several conditions can be specified using decorators:

- `select` annotates a predicate, which the components must pass to be picked for the role;
- `utility` annotates a function, which is used to order the components (only those which passed the `select`) – components with the highest utility come first;
- `cardinality` annotates a function, which sets how many components can be picked for the role. The cardinality can either be a single integer (for `oneOf`, the cardinality is always `1`) or a tuple of the minimum and the maximum allowed number of components.

The member selection for a dynamic role works by first finding all components of the correct type that pass the select predicate, then ordering them by the utility (higher utility is better) and using the cardinality to limit the number of selected members. If there are not enough components passing the selection, i.e. the number of components is smaller than the minimum cardinality, the ensemble cannot be materialized in this time step.

Note that the members are picked one at a time, which means that the select predicate can depend on the previously picked members for the role.

The parameters of the `select` predicate are

- the ensemble instance (`self`),
- the component instance considered for membership (only components of correct type are considered),
- list of already materialized ensembles.

It should return a Boolean.

The parameters of the `utiliy` function are

- the ensemble instance (`self`),
- the component instance considered for membership (only components passing the select predicate are considered).

It should return a float.

The parameter of the `cardinality` function is

- the ensemble instance (`self`).

It should return an integer (maximum allowed number of members), or a tuple of two integers (minimum and maximum, both inclusive).

The members of a dynamic role can be accessed by getting the property – it will be either a list of components (for `someOf`) or a single component (for `oneOf`).

The ensemble can also define the `situation` method, which defines in what situations the ensemble can be materialized. The method should return a boolean indicating whether the ensemble instance can be materialized in this time step. This method is called before the dynamic member selection process begins, so it can only access the attributes of the static members.

Example of an ensemble definition

Listing 4.3 shows an example of an ensemble for drone charging. It groups drones which are assigned to a particular charging station and need charging. The priority

of the ensemble (line 29) is set to the number of free charging slots so that the emptier chargers will be filled sooner.

It has one static role for assigning a charging station (line 6), which is set in the `__init__` method (line 26).

It has one dynamic role for finding the drones (line 9). This role has a select predicate for selecting drones in need of charging (line 12), a utility function for ordering them by the missing battery (line 17), and a cardinality function for limiting the number of drones in the ensemble to the number of free charging slots (line 22).

When the ensemble is materialized (the `actuate` method on line 32 is called), the `station` property of all the members is assigned to indicate to them that they should fly towards the charging station and start charging. Notice that `self.drones` is used as a list here.

```
1  from ml_deeco.simulation import Ensemble, someOf
2
3  class ChargingAssignment(Ensemble):
4
5      # static role
6      charger: Charger
7
8      # dynamic role
9      drones: List[Drone] = someOf(Drone)
10
11     # we select those drones which need charging
12     @drones.select
13     def need_charging(self, drone, otherEnsembles):
14         return drone.needs_charging
15
16     # order them by the missing battery (so the drones with less
           battery are selected first)
17     @drones.utility
18     def missing_battery(self, drone):
19         return 1 - drone.battery
20
21     # and limit the cardinality to the number of free slots on the
           charger
22     @drones.cardinality
23     def free_slots(self):
24         return 0, self.charger.free_slots
25
26     def __init__(self, charger):
27         self.charger = charger
28
29     def priority(self):
30         return self.charger.free_slots
31
32     def actuate(self):
33         # assign the charger to each drone -- it will start flying
               towards it to charge
34         for drone in self.drones:
35             drone.station = self.charger
```

Listing 4.3: Example of a definition of an ensemble for drone charging.

4.1.3 Running the simulation

The `ml_deeco.simulation` module offers a function for running the simulation called `run_simulation`. It has three mandatory parameters and one optional

parameter:

- `components` – a list of all components in the system;
- `ensembles` – a list of all potential ensembles in the system;
- `steps` – the number of steps to be simulated;
- `stepCallback` – a callback function which is called after each simulation step. It can be used for example to log data from the simulation. The parameters are:
  - list of all components in the system;
  - list of the materialized ensembles (in this time step);
  - the current time step (integer).

For better control over the simulation, one can also run the simulation loop manually. The functions `materialize_ensembles` and `actuate_components` from `ml_deeco.simulation` module can be useful for that (they are used inside our `run_simulation`).

4.2 ML-DEECo

In this section, we focus on the parts of the implementation of our DSL which allow us to specify the machine learning models and connect them to the components, ensembles, and ensemble roles.

There are two types of tasks our framework focuses on – value estimate and time-to-condition estimate, both described in more detail in Section 3.2.2. The value estimate uses the currently available observations (inputs) to predict some value that can be observed only at some future point (after a fixed amount of time steps). When we observe the true value (target), we can use it to train the model. The time-to-condition estimate focuses on predicting how long it will take until some condition will become true. This is done by specifying a condition over some future values of component fields.

Obviously, we also need to specify the inputs of the machine learning model. The current state of the component or the fields of the members of the ensemble can be used as inputs. The inputs are specified using getter functions with decorators to link them to the estimate.

Last but not least, the framework automatically collects data for the training of the machine learning model. As we work with a dynamic system, some of the components might not be active at all times. We use a concept of *guards* to indicate whether the training data are valid in a particular time step.

4.2.1 Adding machine-learning-based estimates

The definition of each estimate is split into three parts:

1. The definition of the `Estimator` – a machine learning model and storage for the collected data.

2. The declaration of the `Estimate` field in the component or ensemble or its association with a role.

3. The definition of inputs, target, and guards. These are realized as decorators with getter functions on the component or ensemble.

All of these steps are realized using the `ml_deeco.estimators` module.

4.2.2 Estimator

Estimator represents the underlying machine learning model used for computing the estimates and also a storage for the collected data. The framework currently features `ConstantEstimator` and `NeuralNetworkEstimator`.

All of the implemented estimators are derived from the `Estimator` class. They all have the following common parameters for the constructor:

- `outputFolder` – folder for exporting the collected training data and results and charts from the evaluation of the training. Set to `None` to disable export.
- `name` – String to identify the `Estimator` in the printed output of the framework.
- `accumulateData` – If set to `True`, data from all previous iterations are used for training. If set to `False` (default), only the data from the last iteration are used for training. This is useful when we run the simulation and the training several times.

The `ConstantEstimator` serves as a baseline in the experiments. It always predicts the same constant value (set through the constructor) regardless of the inputs.

The `NeuralNetworkEstimator` uses TensorFlow framework [14] to implement a feedforward neural network. The number of neurons in hidden layers is specified using the `hidden_layers` parameter of its constructor. The input and output layers are constructed automatically based on the inputs and targets specified by the `Estimate`. The loss function for training can also be automatically inferred, more details are written later in Section 4.4.3.

An example of a neural-network-based estimator with two hidden layers, each with 256 neurons, can be seen in Listing 4.4

```
from ml_deeco.estimators import NeuralNetworkEstimator

futureBatteryEstimator = NeuralNetworkEstimator(
    hidden_layers=[256, 256], # two hidden layers
    outputFolder="results/drone_battery", name="Drone Battery"
)
```

Listing 4.4: Definition of a neural-network-based estimator.

4.2.3 Estimate assignment to components and ensembles

To a component

The estimates are declared as properties of the component class. For the estimate of future value – both regression and classification – the `ValueEstimate` can be instantiated. For the time-to-condition estimate, the `TimeEstimate` is used.

In the case of a value estimate, we use the `inTimeSteps` method to set a fixed time difference between the inputs and targets, or the `inTimeStepsRange` method to set a range of allowed time differences.

For both `ValueEstimate` and `TimeEstimate`, the `Estimator` (described in the previous section) must be assigned. That is done by the `using` method.

Furthermore, the `withBaseline` method can be used to specify a custom function to use instead of the estimate during the first iteration (i.e. before the first training of the estimator). When no custom baseline is specified, the estimator returns values close to 0 before it is trained.

Multiple estimates can be assigned to a component, each as a separate variable.

Listing 4.5 shows an example of a drone component class with an estimate of battery level 50 time steps in the future. It uses the `futureBatteryEstimator` defined in Listing 4.4.

```
from ml_deeco.estimators import ValueEstimate

class Drone(MovingComponent2D):

    futureBatteryEstimate = ValueEstimate()\
        .inTimeSteps(50)\
        .using(futureBatteryEstimator) # defined earlier

    # more code of the component
```

Listing 4.5: Definition of a neural-network-based estimator.

To an ensemble

The estimates can be added to ensembles in a same way as to components – as properties.

To an ensemble role (ensemble-component pair)

To assign an estimate to a dynamic role of an ensemble, we work with the role definition (`someOf` or `oneOf`). It provides methods `withEstimate` (for assigning a value estimate) and `withTimeEstimate` (for assigning a time-to-condition estimate). Only one estimate can be assigned to a role.

Again, the `Estimator` must be linked by the `using` method, and the baseline can be optionally specified using the the `withBaseline` method. In the case of a value estimate, the number of time steps we want to predict into the future is again set using the `inTimeSteps` or `inTimeStepsRange` methods.

We show an example of a dynamic role with a time-to-condition estimate in Listing 4.6. It is a stub of an ensemble for grouping drones in need of charging. We want to estimate the waiting time before a drone is accepted for charging by the charger station.

```
from ml_deeco.simulation import Ensemble, someOf
from ml_deeco.estimators import NeuralNetworkEstimator

waitingTimeEstimator = NeuralNetworkEstimator(
    hidden_layers=[256, 256], # two hidden layers
    outputFolder="results/waiting_time", name="Waiting time"
)

class DroneChargingAssignment(Ensemble):

    # dynamic role with time estimate
    drones: List[Drone] = someOf(Drone)\
        .withTimeEstimate()\
```

```
14          .using(waitingTimeEstimator)
15
16      # more code of the ensemble
```

Listing 4.6: Definition of a neural-network-based estimator.

4.2.4 Configuring inputs, target and guards

The definition of inputs and targets of the machine learning model, as well as guards indicating the validity of the inputs, are realized as decorators and getter functions. For estimates assigned to components and ensembles, the decorator has a syntax `@<estimateName>.<configuration>`. For estimates assigned to roles, the syntax is `@<roleName>.estimate.<configuration>`.

The decorators are applied to methods of the component or ensemble. For estimates assigned to components and ensembles, these methods should only have the `self` parameter. For estimates assigned to roles, these methods are expected to have the `self` parameter (the ensemble instance) and a second parameter representing a component (the potential role member). For examples of decorated getter functions, see Listings 4.7 and 4.8.

Inputs

The inputs of the estimate are specified using the `input()` decorator, optionally with a feature type as a parameter. We offer a `NumericFeature(min, max)` for numeric inputs, a `CategoricalFeature(enum|list)` for categorical inputs, and a `BinaryFeature()` to represent Boolean attributes. We elaborate on the available feature types in Section 4.2.4.

Listing 4.7 shows an example of two inputs for an estimate in a component (we continue the examples from earlier Listings 4.4 and 4.5).

```
1  from ml_deeco.estimators import ValueEstimate, NumericFeature,
       CategoricalFeature
2  from ml_deeco.simulation import MovingComponent2D
3
4  class Drone(MovingComponent2D):
5
6      # create the estimate (as described earlier)
7      futureBatteryEstimate = ValueEstimate()\
8          .inTimeSteps(50)\
9          .using(futureBatteryEstimator)
10
11     def __init__(self, location):
12         self.battery = 1
13         self.state = DroneState.IDLE
14         # more code
15
16     # numeric feature
17     @futureBatteryEstimate.input(NumericFeature(0, 1))
18     def battery(self):
19         return self.battery
20
21     # categorical feature constructed from an enum
22     @futureBatteryEstimate.input(CategoricalFeature(DroneState))
23     def drone_state(self):
24         return self.state
```

Listing 4.7: Inputs of an estimate assigned to a component.

Listing 4.8 shows an example of an input for an estimate assigned to a role (we again continue the example from earlier Listing 4.6).

```
class DroneChargingAssignment(Ensemble):

    # dynamic role with time estimate (as described earlier)
    drones: List[Drone] = someOf(Drone)\
        .withTimeEstimate()\
        .using(waitingTimeEstimator)

    @drones.estimate.input(NumericFeature(0, 1))
    def battery(self, drone):
        return drone.battery
```

Listing 4.8: Inputs of an estimate assigned to a role.

Target for `ValueEstimate`

The target represents the true values which are then used for training the machine learning model. The values are collected after the defined number of time steps and associated with the corresponding inputs.

The target is specified similarly to the inputs using the `target()` decorator. A `Feature` can again be given as a parameter – this is how classification and regression tasks are distinguished. The feature is then used to set the appropriate number of neurons and the activation function of the last layer of the neural network, and the loss function used for training – see Sections 4.4.2 and 4.4.3 for more details.

An example of the target specification is given in Listing 4.9.

```
class Drone(MovingComponent2D):

    # create the estimate and inputs as described earlier
    ...

    # define the target -- regression task
    @futureBatteryEstimate.target(NumericFeature(0, 1))
    def battery(self):
        return self.battery
```

Listing 4.9: Target of an estimate assigned to a component.

Condition for `TimeEstimate`

For the time-to-condition estimate, a condition must be specified instead of the target value. The syntax is again similar – using the `condition` decorator. If multiple conditions are provided, they are considered in conjunction (all of them must be true).

An example of a condition is given in Listing 4.10.

```
class DroneChargingAssignment(Ensemble):

    # create the estimate and inputs as described earlier
    ...

    # define the condition (drone is accepted for charging)
    @drones.estimate.condition
    def is_accepted(self, drone):
```

```
9           return drone in self.charger.acceptedDrones
```

Listing 4.10: Condition for an estimate assigned to a role.

Features

When assigning the inputs and the target for the estimate, a feature type can be specified. Setting the feature type allows for preprocessing of the values before they are passed to the machine learning model (neural network in our current implementation). The specification of the feature type is technically realized by passing an instance of a `Feature` as an optional parameter of the `input()` or `target()` decorator. We offer a `NumericFeature(min, max)`, a `CategoricalFeature(enum|list)`, and a `BinaryFeature()` subclasses of `Feature`.

The `NumericFeature(min, max)` serves as a representation of a numeric value with a known range (minimum and maximum). Before passing the value to the neural network, it is normalized to the interval $[0, 1]$. This is a recommended practice as it usually improves the performance of the network. We do not offer a specific implementation of numeric features with unknown range, the default implementation (not setting the feature parameter) works fine.

For categorical inputs (inputs with a known fixed set of possible values), the `CategoricalFeature(enum|list)` should be used. The list of possible values (categories) must be specified as either a `list` of values, or an instance of `IntEnum`[3]. One-hot encoding is performed as preprocessing before passing the values to the neural network. This means that the input will be represented by as many input neurons as there are possible values. When used as a target, the last layer of the neural network has as many outputs as there are categories and uses the softmax activation function.

The binary (Boolean) inputs are represented by the `BinaryFeature()` class. It converts the Boolean values to 0 and 1 as preprocessing for the neural network. When used as a target, the neural network has one neuron with sigmoid activation in the last layer and the predicted value is the result of a comparison of the produced value with 0.5.

Internally, we also use a `TimeFeature()` to represent the time difference in the `TimeEstimate`. As we know that the time difference is always non-negative, we use the exponential function as the activation function of the last layer of the neural network.

Validity of inputs – guards

Guard functions can be specified using `inputsValid`, `targetsValid` and `conditionValid` decorators to assess the validity of inputs and targets. The data for training is collected only if the guard conditions are satisfied. This can be used for example to prevent collecting data from components that are no longer active. If multiple guards of the same type are defined, all of them must return true in order to collect the data.

An example of guards is shown in Listing 4.11. The `conditionValid` decorator is used similarly.

[3]From Python package `enum`.

```
class Drone(MovingComponent2D):

    # create the estimate, inputs and targets as described earlier
    ...

    @futureBatteryEstimate.inputsValid
    @futureBatteryEstimate.targetsValid
    def not_terminated(self):
        return self.state != DroneState.TERMINATED
```

Listing 4.11: Guards for an estimate assigned to a component.

We also provide a guard for assessing the validity of a pair of inputs and targets. It can be used to resolve more complex situations which cannot be dealt with using only the `inputsValid` and `targetsValid` decorators as we show later in Section 5.2.3. The guard is specified using the `recordValid` decorator on a function, which can have the inputs, targets and also extra information as its parameters (all in form of Python dictionaries). The extra information are collected at the same time as the inputs and can be specified using the `extra` decorator.

An example of `recordValid` guard is shown in Listing 4.12. We use it here to not collect the record if the drone was charged since collecting the inputs (more on that in Section 5.2.3).

```
    class Drone(MovingComponent2D):

        # create the estimate, inputs and targets as described
            earlier
        ...

        @futureBatteryEstimate.extra
        def current_time(self):
            """Returns the current time step of the simulation."""
            return SIMULATION_GLOBALS.currentTimeStep

        @futureBatteryEstimate.recordValid
        def not_charging(self, inputs, targets, extra):
            """Returns true if the drone was not charged in the time
                between collecting the inputs and the targets."""
            time_of_inputs = extra['current_time']
            return time_of_inputs >= self.lastChargingTime
```

Listing 4.12: Guard for a record (a pair of inputs and targets).

4.2.5 Obtaining the estimated value

The `Estimate` object is callable, so the value of the estimate based on the current inputs can be obtained by calling the estimate as a function. For estimate assigned to a role, the estimate is available in the `estimate` property of the role, and a component instance is expected as an argument of the call.

When using the `ValueEstimate` with `inTimeStepsRange` and additional parameter is required when obtaining the value to indicate how many time steps into the future the estimate should predict.

When obtaining the estimate, the input getters are called to obtain the current observations. These are then used as inputs to the machine learning model and the value produced is returned as the estimate.

The Listings 4.13 and 4.14 show how to obtain the estimated value from an estimate assigned to a component and to a role respectively. Furthermore, Listing 4.15 shows how to obtain an `ValueEstimate` with variable range of time differences.

```
class Drone(Agent):

    # create the estimate, inputs and targets as described earlier
    futureBatteryEstimate = ValueEstimate()\
        .inTimeSteps(50)\
        .using(futureBatteryEstimator)
    ...

    def actuate(self):
        estimatedFutureBattery = self.futureBatteryEstimate()
```

Listing 4.13: Obtaining the estimated value in a component.

```
class DroneChargingAssignment(Ensemble):

    # create the estimate and inputs as described earlier
    drones: List[Drone] = someOf(Drone)\
        .withTimeEstimate()\
        .using(waitingTimeEstimator)
    ...

    @drones.select
    def drones(self, drone, otherEnsembles):
        # we obtain the estimated waiting time here
        waitingTime = self.drones.estimate(drone)
        # and use it to decide whether the drone should ask for a
            charging slot
        return drone.needsCharging(waitingTime)
```

Listing 4.14: Obtaining the estimated value from an estimate assigned to a role.

```
class Drone(Agent):

    # create the estimate, inputs and targets as described earlier
    futureBatteryEstimate = ValueEstimate()\
        .inTimeStepsRange(50, 100)\
        .using(futureBatteryEstimator)
    ...

    def actuate(self):
        estimatedBatteryAfter80Steps =
            self.futureBatteryEstimate(80)
```

Listing 4.15: Obtaining the estimated value with variable estimation time in a component.

4.2.6 Running the simulation

To run the simulation in ML-DEECo, the `run_simulation` function described in Section 4.1.3 can be used. The only thing that needs to be added when running a simulation with estimates is to call the `SIMULATION_GLOBALS.initEstimators()` method (`SIMULATION_GLOBALS` is in the module `ml_deeco.simulation`) to initialize the estimators before running the simulation.

Furthermore, we provide the `run_experiment` function which is useful for running the simulation several times with training of the machine learning models in between. It is described in the following section.

Running an experiment

The `run_experiment` serves for running several runs of the simulation with the training of the machine learning models between them. It has four mandatory parameters:

- `iterations` – number of iterations to run;
- `simulations` – number of simulations in each iteration;
- `steps` – the number of steps to be simulated in each simulation;
- `prepareSimulation` – a function to prepare the components and potential ensembles for one run of the simulation, it is called before each simulation;

and four optional parameters:

- `prepareIteration` – a function to be called at the beginning of each iteration;
- `iterationCallback` – a function to be called at the end of each iteration;
- `simulationCallback` – a function to be called at the end of each simulation;
- `stepCallback` – same as in `run_simulation` (Sect. 4.1.3).

The `iterations` parameter specifies the number of iterations. In each iteration, there are several runs of the simulation (the number of runs is set by the `simulations` parameter). After that, the data from all the simulations in the current iteration are used to train the machine learning model (`Estimator`). The next iteration will use the updated model.

The `prepareSimulation` function is used to obtain the components and ensembles for the simulation. It gets the number of the current iteration and the number of the current simulation (its order within the iteration) as parameters. It is expected to return two lists: all the components to be simulated, and all the potential ensembles in the system. The simulation is then run using our `run_simulation` function for `steps` steps.

The `prepareIteration` is an optional function to be run at the beginning of each iteration. It can be used for example to initialize logs for logging data during simulations. Apart from the `stepCallback`, which is the same as in `run_simulation` (Sect. 4.1.3), we also allow specifying a `simulationCallback` (ran after each simulation) and `iterationCallback` (ran at the end of iteration after all the simulations in the iteration have been run and the training of the machine learning model has finished).

In the `run_experiment` function, the initialization of the `Estimator`s is done automatically.

4.3 Examples of ML-DEECo usage

In this section, we provide two complete examples of specification of a machine-learning-based estimate in an ensemble-based component system.

4.3.1 Drone component

The first example, shown in Listing 4.16, is a drone component with a prediction of the battery level in the future. The estimate is assigned to the component on lines 16–18, and we use neural network with two hidden layers to produce the predictions (lines 4–7). There are two inputs to the estimate – current battery

level as an example of a numeric input (line 20), and the current operational state as an example of a categorical input created from an `IntEnum` (line 24). The target, which is the battery level, is defined on line 28. The target values are collected 50 time steps later then the inputs (as specified on line 17) and together with the corresponding inputs, they are used for training of the machine learning model. We use guards (lines 32–35) to express that the inputs are only valid for training if the drone is not terminated. Lastly, the estimated value is obtained on line 39 and if the predicted future battery level is below zero, the drone decides to fly towards its nearest charger (the `find_closest_charger` method is omitted in the example).

```
1  from ml_deeco.simulation import MovingComponent2D
2  from ml_deeco.estimators import ValueEstimate, NumericFeature,
       CategoricalFeature, NeuralNetworkEstimator
3
4  futureBatteryEstimator = NeuralNetworkEstimator(
5      hidden_layers=[32, 32],
6      name="Drone battery"
7  )
8
9  class Drone(MovingComponent2D):
10
11     def __init__(self, location):
12         self.battery = 1
13         self.state = DroneState.IDLE
14         self.station = None # charging station
15
16     futureBatteryEstimate = ValueEstimate()\
17         .inTimeSteps(50)\
18         .using(futureBatteryEstimator)
19
20     @futureBatteryEstimate.input(NumericFeature(0, 1))
21     def battery(self):
22         return self.battery
23
24     @futureBatteryEstimate.input(CategoricalFeature(DroneState))
25     def drone_state(self):
26         return self.state
27
28     @futureBatteryEstimate.target(NumericFeature(0, 1))
29     def battery(self):
30         return self.battery
31
32     @futureBatteryEstimate.inputsValid
33     @futureBatteryEstimate.targetsValid
34     def not_terminated(self):
35         return self.state != DroneState.TERMINATED
36
37     def actuate(self):
38
39         estimatedFutureBattery = self.futureBatteryEstimate()
40         if estimatedFutureBattery <= 0:
41             self.station = self.find_closest_charger()
42
43         # if the drone has an assigned charger
44         if self.station:
45             # fly towards it
46             if self.move(self.station.location) and
                   self.station.has_free_slot():
47                 # drone arrived at the location of the charger
```

```
48                  self.station.start_charging(self)
```

Listing 4.16: Example of machine-learning-enabled component.

4.3.2 Charging ensemble

In Listing 4.17, we show an ensemble for selecting drones for charging. The ensemble has one static role (line 11) to which a charger is assigned in the `__init__` method (line 48). The priority of the ensemble (line 51) is set to the number of free charging slots of the charger.

The ensemble features one dynamic role (line 13) with an estimate of the waiting time for a free charging slot (line 14) for finding the drones, which need charging. Again, we use a neural-network-based estimator (lines 4–7) to compute the predictions. The inputs of the estimate are defined on lines 32–38, and the guards are on lines 40–42. When a drone becomes a member of the ensemble, it means that there is a free slot for it at the charger. We thus want to use that event as the end of the waiting time, which is realized in the condition on lines 44–46 – the condition returns true for members of the ensemble.

The select predicate for the role (line 17) obtains the estimated waiting time (line 21) and uses it to decide whether a drone needs charging[4]. The utility function (line 24) orders the drones by their missing battery, and the cardinality function (line 28) limits the number of ensemble members to the free charging slots.

When the ensemble is materialized (line 54), the member drones are assigned the `station` property to indicate to them that they should fly towards the charging station and start charging. Notice that `self.drones` is used as a list here.

```
1  from ml_deeco.simulation import Ensemble, someOf
2  from ml_deeco.estimators import NeuralNetworkEstimator
3
4  waitingTimeEstimator = NeuralNetworkEstimator(
5      hidden_layers=[256, 256],
6      outputFolder="results/waiting_time", name="Waiting time"
7  )
8
9  class ChargingAssignment(Ensemble):
10
11     charger: Charger
12
13     drones: List[Drone] = someOf(Drone)
14         .withTimeEstimate()\
15         .using(waitingTimeEstimator)
16
17     @drones.select
18     def need_charging(self, drone, otherEnsembles):
19         if drone.state == DroneState.TERMINATED:
20             return False
21         waitingTime = self.drones.estimate(drone)
22         return drone.needs_charging(waitingTime)
23
24     @drones.utility
25     def missing_battery(self, drone):
```

[4]The `Drone.needs_charging` method is not shown in the example, but we assume that it determines whether the drone would need charging if it had to wait for the charging slot for a given number of time steps.

```
26          return 1 - drone.battery
27
28      @drones.cardinality
29      def free_slots(self):
30          return 0, self.charger.free_slots
31
32      @drones.estimate.input(NumericFeature(0, 1))
33      def battery(self, drone):
34          return drone.battery
35
36      @drones.estimate.input(NumericFeature(0, ENVIRONMENT.size))
37      def charger_distance(self, drone):
38          return self.charger.location.distance(drone.location)
39
40      @drones.estimate.inputsValid
41      def not_terminated(self, drone)
42          return drone.state != DroneState.TERMINATED
43
44      @drones.estimate.condition
45      def is_accepted(self, drone):
46          return drone in self.drones
47
48      def __init__(self, charger):
49          self.charger = charger
50
51      def priority(self):
52          return self.charger.free_slots
53
54      def actuate(self):
55          for drone in self.drones:
56              drone.station = self.charger
```

Listing 4.17: Example of machine-learning-enabled ensemble.

4.4 Estimators

Our implementation focuses on using neural networks as the machine learning algorithms to generate predictions. We provide the `NeuralNetworkEstimator` class in `ml_deeco.estimators` module to the user. It uses the TensorFlow framework [14] inside to implement the neural networks.

4.4.1 Neural network architecture

Currently, we only work with multilayer perceptron networks (also called fully-connected or dense). The number of hidden layers and the number of neurons in each layer is done by the user as already shown in Section 4.2.2. The input layer is constructed automatically based on the number of specified inputs (for categorical inputs, more than one input neuron is used due to the one-hot encoding). The last layer is also constructed automatically based on the target. The process is described in more detail in Section 4.4.2.

To train the networks, we use the data collected during the simulation. The detail on how we perform the data collection are written in Section 3.3.1. The loss function used for training is also inferred automatically based on the specified target feature, details are in Section 4.4.3. The parameters for the training can be specified using the `fit_params` parameter of `NeuralNetworkEstimator`. It is a dictionary, which is then passed to the `fit` function of the constructed TensorFlow

model. The default training parameters are: 50 training epochs, 0.2 validation split, and early stopping[5] with patience 10. Furthermore, the `optimizer` for training can be used, the default one is Adam [15] (`tf.optimizers.Adam`).

4.4.2 Inference of the output layer of the neural network

We use the feature type specified to the `target` decorator to automatically infer the number of neurons and the activation function for the last layer of the neural network. If the user wants to use a different activation function, they can specify it using the `activation` parameter of the `NeuralNetworkEstimator` constructor.

When no feature type is specified for the target, the default `Feature` implementation is used. The output of the last layer is not processed any further and is returned as-is. Technically, this is done by using identity as the activation function.

For numeric features, the sigmoid[6] function is used as the last layer activation. This is done to ensure that the predicted value will always fall into the interval $[0, 1]$. The value is then scaled based on the provided range of the feature.

For categorical features, we use as many neurons as there are categories with the softmax[7] activation function. That produces a probability distribution over the categories and the category with highest probability is returned as the prediction.

For binary features, we use one neuron with the sigmoid activation function. The prediction is produced by comparing the output of the sigmoid with 0.5 (predicting `True` if the output of the network is bigger than 0.5).

Lastly, for `TimeFeature` (used internally by the `TimeEstimate`), we use the exponential function (e^x) as activation in order to produce non-negative prediction.

A summary of the used activations (together with losses) is provided in Table 4.1.

Target feature	Last layer activation	Loss
`Feature` (default)	identity	Mean squared error
`NumericFeature`	sigmoid (+ scaling to proper range)	Mean squared error
`CategoricalFeature`	softmax (1 neuron for each category)	Categorical cross-entropy
`BinaryFeature`	sigmoid (only 1 neuron)	Binary cross-entropy
`TimeFeature`	exponential	Poisson

Table 4.1: Summary of activation functions of the last layer of the neural network and the loss function used for training based on the target feature type.

4.4.3 Inference of the loss function for training

Similar to the last layer inference, we use the feature type specified to the `target` decorator to automatically infer the loss function for the training of the neural

[5] If the loss computed on the validation data does not improve for a given number of epochs, the training is stopped. This is realized by the `tf.keras.callbacks.EarlyStopping` callback.

[6] Sigmoid is defined as $\sigma(x) = \frac{1}{1+e^{-x}}$.

[7] The softmax of a vector $(x_1, \ldots, x_D)$ is a vector $(\frac{e^{x_1}}{z}, \ldots, \frac{e^{x_D}}{z})$, where $z = \sum_{i=1}^{D} e^{x_i}$.

network. If the user wants to use a different loss function, they can specify it using the `loss` parameter of the `NeuralNetworkEstimator` constructor.

For both the `NumericFeature` and also the `Feature` which is default when no feature type is specified, the mean squared error loss (`tf.keras.losses.MeanSquaredError`) is used.

For categorical features, we use the categorical cross-entropy loss (`tf.keras.losses.CategoricalCrossentropy`), which works well together with the softmax activation function. For binary features, the binary cross-entropy loss (`tf.keras.losses.BinaryCrossentropy`) is used.

Lastly, for `TimeFeature`, we use the Poisson loss (`tf.keras.losses.Poisson`).

A summary of the used loss functions (together with activations) is provided in Table 4.1.

4.4.4 Caching of estimates

We employ a performance optimization for the role-assigned estimates with fixed time difference. We compute the estimated values for all potential member components at the same time and cache them. It saves time as the neural network is capable of processing all the potential members in one batch. We thus only do one call to the TensorFlow backend per time step for each `Estimate` instance.

This implies that the inputs of the model cannot use the information about the already selected members for the role. To suppress this behavior, pass a keyword argument `ignoreCache=True` to the call for obtaining the estimate, which forces the prediction to be generated again from the current values of the inputs.

5. Evaluation

To prove that ML-DEECo is useful for modeling smart systems, we implemented a simulation of the running example in Python using our ML-DEECo runtime framework implementation [13]. In this chapter, we first provide a description of the simulation in Section 5.1 and then present results of using the machine-learning-based estimates for self-adaptation of the simulated system in Section 5.2. We have also included an additional example from the Industry 4.0 domain implemented using ML-DEECo in Section 5.3.

5.1 Simulation of the running example

We implemented a simulation of the running example (described in Section 2.1) using the ML-DEECo runtime framework. The source code of the simulation can be found in the replication package [16].

We use the DEECo concepts of *components* and *ensembles* (for details, see Section 2.2) to implement the simulation.

We have two types of autonomous components – drones, which protect the crops, and chargers, which are used for charging the drones. The flocks of birds, which damage the crop, are also modeled as components as it is convenient for the simulation. Nevertheless, the system cannot control the behavior of the birds, they thus belong among the "beyond control components".

The components group together into ensembles formed at runtime by our framework. We have one ensemble type for field protection (details in Section 5.1.5) and three ensemble types for management of the charging of the drones (details in Section 5.1.6). Furthermore, the charging of the drones is the place where we use a machine-learning-based estimate to predict the waiting time before a drone is accepted for charging.

5.1.1 World configuration and agricultural fields

The configuration of the simulation is loaded from a `.yaml` file[1] and it is stored in the `ENVIRONMENT` global variable. The simulation is then initialized and the dynamic information, such as the lists of drones, chargers, and birds are saved in the `WORDL` global variable.

The agricultural fields of crops which need protection are represented by the `Field` class. We model the fields as having a rectangular shape. The number of drones needed to protect the whole field from the birds depends on the size of the field. The `Field` class also records the amount of crops damaged by the birds.

5.1.2 Flocks of birds

The birds are a threat to the crops on the fields. They seek undamaged crops on the fields and eat them, which will damage that part of the field. The birds are afraid of the drones, so when a drone is nearby, the flock of birds starts to

[1]The format of the configuration file is described in the replication package [16].

flee to a random place on the map. The behavior of the birds is randomized, the randomness will thus influence the results of the simulation.

The flocks of birds are represented by the `Bird` class. Each bird is modeled as having a state (the `BirdState` enum). Based on the state, the behavior of the bird is executed in each time step. As we model the birds as components, the periodical execution of the behavior is realized in the `actuate` function.

5.1.3 Drone component

The `Drone` class represents the autonomous drones used for field protection. The drone can be assigned a position to protect and it will fly towards it to scare away the birds. As the drones run on battery, they need to be charged. The drone can detect when it needs charging and once accepted for charging by a charger, the drone will fly towards the charger.

The behavior of the drones depends on the operational state which is modeled by the `DroneState` enum.

Drone battery estimate

We employ the machine-learning-based estimate inside the `Drone` component to predict the battery level in the future. We use the `ValueEstimate` with `inTimeStepsRange` to be able to estimate the battery level at any time in the near future.

We had to be careful about the data we collect as the charging of the drone skews the estimate of the future battery. We discuss this further in Section 5.2.3.

5.1.4 Charger component

The `Charger` class represents the charger station for charging the drones. Each charger station has a limited capacity so it can charge only several drones at the same time. Furthermore, we assume that all the chargers are connected to the same power supply which creates an upper limit for the total charging rate. That means that when multiple drones are charged simultaneously, the charging rate for each drone will be lower than if there was only one drone being charged.

5.1.5 Field protection ensemble

We use ensembles of drones and fields to communicate the assignment of places that need protection. It is assumed that in a real-life scenario, the position of the birds will be detected by additional sensors and communicated through the ensembles.

We have one instance of the `FieldProtection` ensemble for each field on the map (the field is thus a static member of the ensemble instance). The ensemble also has a dynamic role for selecting a drone to protect the field. We select only one drone for each field in each time step to distribute the drones evenly among the fields. The member for the ensemble is selected from the idle drones with utility based on the distance of the drone to the field. Once a drone becomes a member of the ensemble, it is assigned the target field and its state is changed so that it is not selected again in the next time step. The priority, which defines the

order in which the potential ensembles are materialized, depends on the number of unprotected places in the field.

5.1.6 Drone charging ensembles

The most complex group behavior which we model in this example is the charging of the drones. We use three ensemble types to achieve the desired outcome:

1. `DroneChargingPreAssignment` partitions the drones among the chargers so that each drone is assigned to the closest charger;
2. `DroneChargingAssignment` selects the drones in need of charging;
3. `AcceptedDronesAssignment` groups the drones which were accepted by the charger — those start moving to the charger and start charging when they get there.

Once on the charger, the drone will charge until its battery is full, and then its state changes to "idle". We provide a more detailed description of each ensemble type in the following sections.

Furthermore, drone charging is where we apply the machine-learning-based approach to the self-adaptation of the system. We use a time-to-condition estimate in the `DroneChargingAssignment` to predict the waiting time before a drone gets accepted for charging.

`DroneChargingPreAssignment`

This ensemble has the highest priority among the drone-charging-related ensembles which ensures it is executed before the other two ensembles. It is used to divide the drones among the chargers – for each active drone, the closest charger is located in each time step. Technically, we have an instance of the `DroneChargingPreAssignment` for each charger, so its members are the drones for which this charger is the closest. The cardinality of the ensemble is unlimited (technically, it is set to the total number of drones in the simulation), so we do not need any utility function. When materialized, the ensemble will save the found drones to the `potentialDrones` list of the charger. As the ensembles are re-formed in every time step, the `potentialDrones` list always contains the drones for which the charger is the closest.

`DroneChargingAssignment`

The `DroneChargingAssignment` ensemble groups the drones which require charging. The priority of the ensemble is set so that it is run after the `DroneChargingPreAssignment`. Again, we have an instance of the ensemble for each charger and these instances are independent as each of them only works with the `potentialDrones` for the charger.

We use a time-to-condition estimate in the select predicate of the ensemble to predict the waiting time before a drone is accepted for charging. We sum the estimated waiting time and the time needed to fly to the charger to get an estimate of the time before the drone starts charging. Based on that, the drone computes

what battery level it will have at the time we estimate it to start charging. If this battery level is below a certain threshold, the drone signalizes that it needs charging.

We do not limit the cardinality of the role so all the drones in the system can become members of the ensemble. The members are then assigned to the `waitingDrones` list of the charger.

Waiting time estimate

The estimate is technically realized as a `TimeEstimate` assigned to the dynamic role `drones` in the `DroneChargingAssignment` ensemble.

As a baseline, we assume that the drone will be accepted immediately and there will be no waiting. The time before the drone starts charging is thus only the time needed to fly towards the charger.

For our machine-learning-based estimate, we use a neural network to predict the waiting time. The input features we use are:

- `battery` – the battery level of the drone;
- `drone_state` – the operational state of the drone (categorical with categories from `DroneState` enum);
- `charger_distance` – the distance between the drone and the charger;
- `charger_capacity` – the number of total charging slots;
- `charging_drones_count` – the number of drones currently being charged by the charger;
- `charging_drones_missing_battery` – the missing battery (i.e., `1 - battery`) of the drones currently being charged by the charger;
- `accepted_drones_count` – the number of drones already accepted for charging;
- `accepted_drones_missing_battery` – the missing battery (i.e., `1 - battery`) of the drones already accepted for charging;
- `waiting_drones_count` – the number of waiting drones (from the previous time step);
- `waiting_drones_with_lower_battery` – the number of waiting drones (from the previous time step) with a battery lower than the current battery of the drone;
- `potential_drones` – the number of drones pre-assigned to the charger;
- `potential_drones_with_lower_battery` – the number of drones pre-assigned to the charger with a battery lower than the current battery of the drone;
- `neighbor_drones` – the number of drones protecting the same field;
- `neighbor_drones_average_battery` – the average battery level of the drones protecting the same field.

As we work with a time-to-condition estimate, we need a condition indicating the end of the waiting time. For that, we check whether the drone is in the `acceptedDrones` list of the charger (the list is set by the `AcceptedDronesAssignment` ensemble).

We also have a guard for the validity of the inputs checking that the drone is pre-assigned to the charger. We only want to collect the data for training from the drones belonging to this charger and not those belonging to other chargers.

`AcceptedDronesAssignment`

The last ensemble used for charging drones is the `AcceptedDronesAssignment`. This ensemble is used to select some of the `waitingDrones` and accept them for charging. When a drone is accepted, its state is changed to "moving to charger" and the drone starts flying towards the charger. The select function is constructed in such a way that when the drone reaches the charger, there is a free slot for it and it can immediately start charging. We also save the accepted drones in the `acceptedDrones` list of the charger.

The select predicate selects the drones that either were already accepted in the previous time step (and are still flying towards the charger), or are in the `waitingDrones` list and their time to reach the charger is bigger than the time needed to finish charging one of the currently charging drones and thus making the charging slot free.

The utility of function orders the drones by the time they need to finish charging. That is the sum of the time needed to fly to the charger and time to fully charge, considering the energy used for the flight. The drones which will finish charging the first become members of the ensemble first.

The cardinality is set to the number of charging slots the charger has. This effectively means that we can have an accepted drone for each charging slot. Note that the accepted drones are those which fly towards the charger and will start charging when they get there, not the drones currently being charged.

5.2 Results

In this section, we focus on the results of using the machine-learning-based estimates in the simulation. We use two estimates in our simulation:

- waiting time before a drone is accepted for charging (5.1.6),
- future battery level of a drone (5.1.3).

This section first describes the evaluation metrics in Section 5.2.1. Then, we show the baselines we compare our machine-learning-based estimates to in Section 5.2.2. Sections 5.2.3 and 5.2.4 discuss the specifics of training data collection in our use-case. Lastly, we summarize the results in Section 5.2.5.

5.2.1 Evaluation metrics

We focus on two evaluation metrics – the *survived drones* and the *damage rate*.

The *survived drones* is the number of drones that are active at the end of the simulation – these are the drones that were not terminated during the simulation due to their battery running out. It is clear that when we have an optimized schedule for charging the drones, more of the drones survive until the end of the simulation. A better estimate of the waiting time should thus result in more surviving drones.

The *damage rate* is the relative percentage of the crops eaten by the birds. If birds are on a field and they are not scared away by the drones for a certain amount of time, they damage the crop on the field. We accumulate the amount of damaged crops throughout the simulation. To get the damage rate, we normalize

the damage amount by the damage amount done by the birds if no drones are present in the simulation to scare them away (i.e., the maximum damage the birds can do).

At first, it might seem that these two metrics are highly correlated as more drones can protect a bigger area of the fields. However, this is not really true as it does not take into account the time needed to charge the drones. If drones spend most of their time at the charger (to charge their battery and therefore survive), they cannot spend the time protecting the fields.

5.2.2 Baselines

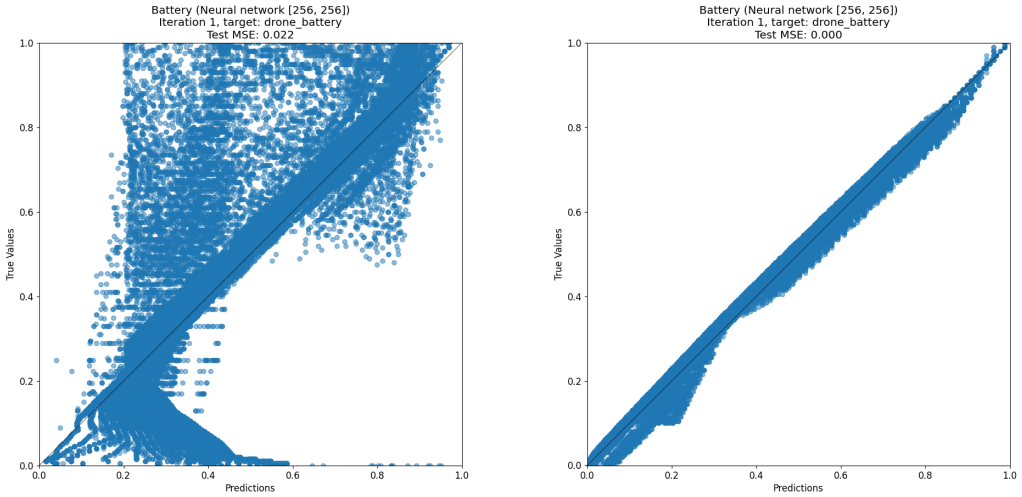
When the simulation starts, we do not have a trained machine learning model for computing the estimates. In the first several runs of the simulation, we thus use the values predicted by the untrained neural network. The weights of the neural network are initialized in a common way that results in producing 0 as the output of the network. Using the uninitialized neural network is thus almost equivalent to the baseline of predicting a constant 0 for the waiting time (we call this approach *Baseline 0*).

To show the benefits of using a machine-learning-based approach over simply increasing the threshold for indicating whether a drone needs charging (i.e., consider all or most of the drones as if they need charging), we construct another baseline (*Baseline 100*) which predicts a constant 100 as the waiting time. As the results show, using this baseline is exactly the case in which the number of survived drones increases, but the damage rate is also bigger than for the Baseline 0.

5.2.3 Using guards to collect appropriate data for the battery estimate training

As we want to use the estimate to decide whether the drone needs charging or not, we want to be able to use the estimate to predict the future battery of the drone assuming it is not charged in the meantime. However, the drone gets charged during the simulation, so the data we collect from the simulation also contain training examples with the battery level in the future higher than the battery level now. These examples skew the estimate of the battery. If we just collect all the training examples from the simulation, the trained estimate will predict the future battery including the possible charging of the drone, which is not really useful for the decision we want to make.

We use the record guards to address this issue and simply discard all training examples in which the drone was charging in the time between collecting the inputs and the targets. We show a comparison of using the guard and not using the guard in Figure 5.1. The scatter plots show the correlation of the true values and the predictions. The benefits of using the guard are clearly visible – the data are much closer to the diagonal. When the guard is not used, the predictions are very noisy and the estimator is basically useless.



(a) Using all data for training.

(b) Using only data where the drone is not charging between collecting the inputs and the targets.

Figure 5.1: Comparison of results of the estimator with regard to what training data are collected. Horizontal axis represents the predicted values, vertical axis represents the true values.

5.2.4 Strategies for collecting training data

We use the `run_experiment` function from ML-DEECo (see Section 4.2.6) to run our experiments. We set the number of iterations to 10 and the number of simulations to 5. This means that the whole experiment consists of 10 iterations. In each iteration, the simulation is run 5 times and then the training of the neural network is performed. The training data are collected during each simulation run.

In each iteration, we concatenated the data from all the simulation runs. Furthermore, we have experimented with several strategies for selecting the training data from the already finished iterations:

- only the data from the current iteration,
- data from all previous iterations,
- data from the past three iterations.

Data from current iteration

The first strategy is to use only the data collected during the current iteration. The data from an experiment with 16 drones can be seen in Figure 5.2. We can see that even one training of the neural network is enough to save most of the drones and to get similar results as with Baseline 100. However, the damage rate also increases as the drones spend a lot of time at the charging stations.

With the second training of the neural network (which uses the training data collected during the 5 runs in the second iteration), the system adapts and improves again, this time saving approximately the same number of drones, but reaching a significantly lower damage rate, even smaller than with the Baseline 0.

As the system adapts further, it seems that it is oscillating between two types

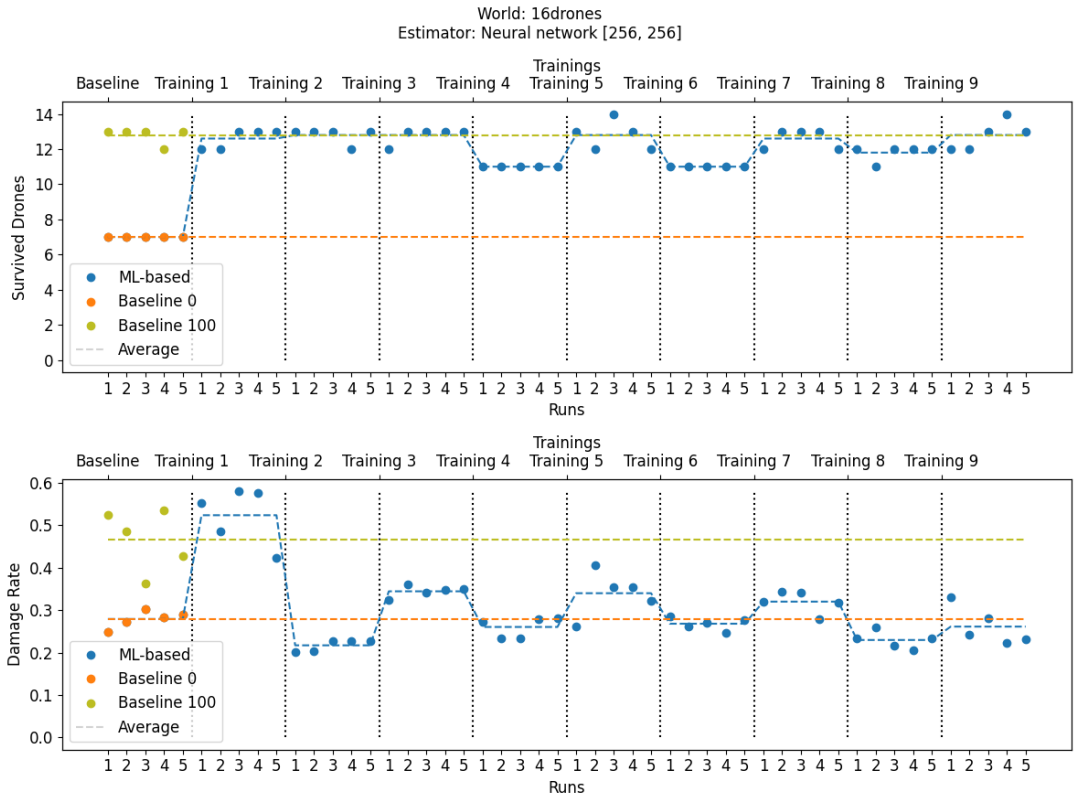


Figure 5.2: Survived drones and damage rate in a simulation with 16 drones while using data from one iteration for estimator training.

of states – in one of the types, the system saves most of the drones, but the damage rate is high because of the time spent charging; in the other type, the system saves fewer drones, but the damage rate is lower because the drones spend their time better. Clearly, the adaptation of the system changes the distribution of the data which we use for training the neural network. As we only use the data from the last iteration, the network adapts to the data from one type of the states which causes the system to change its state to the other type and vice versa, causing a feedback loop.

Data from all previous iterations

The feedback loop can be resolved by using the data from all the previous iterations for training as we can observe in Figure 5.3. The decrease of the damage rate is slower than in the previous case as the adaptation of the system is slower, because we adapt it to different training data. On the other hand, the performance of the system seems significantly more stable.

Data from previous three iterations

Using the data from all previous iterations has the disadvantage of the growing number of training data we have to keep in the memory and the training of the neural network taking longer. To remove this disadvantage while still breaking the feedback loop, we experimented with using data from a subset of the previous

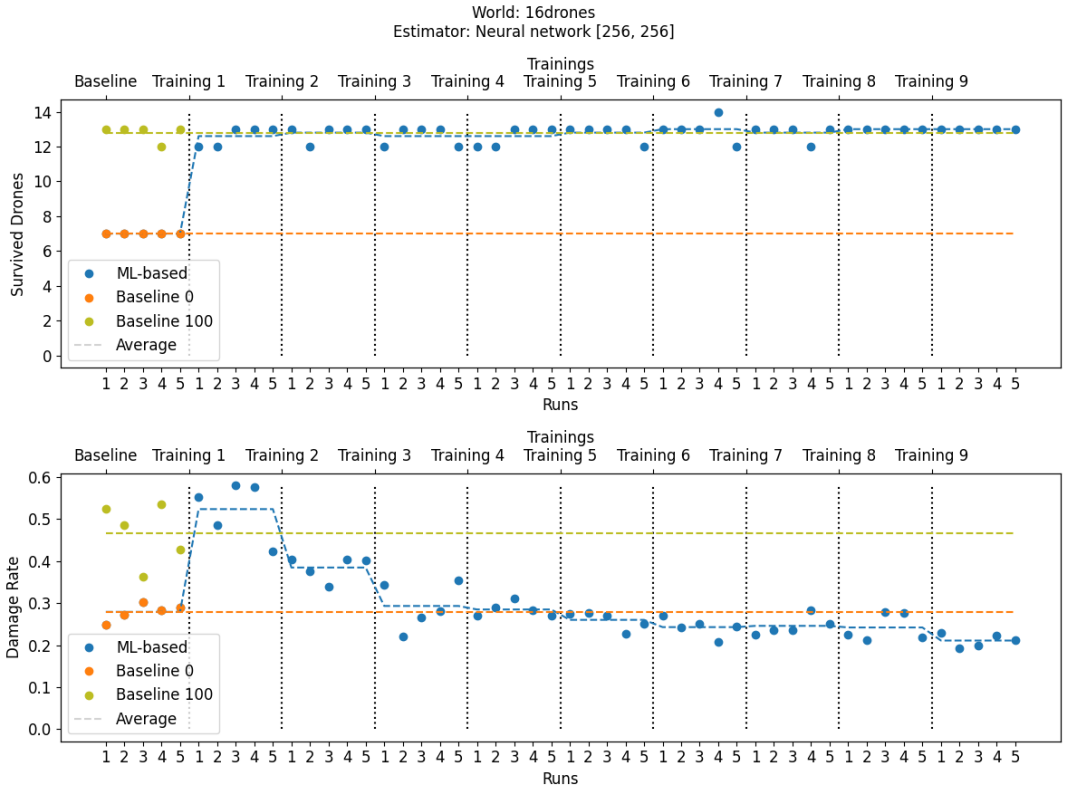


Figure 5.3: Survived drones and damage rate in a simulation with 16 drones while using data from all iterations for estimator training.

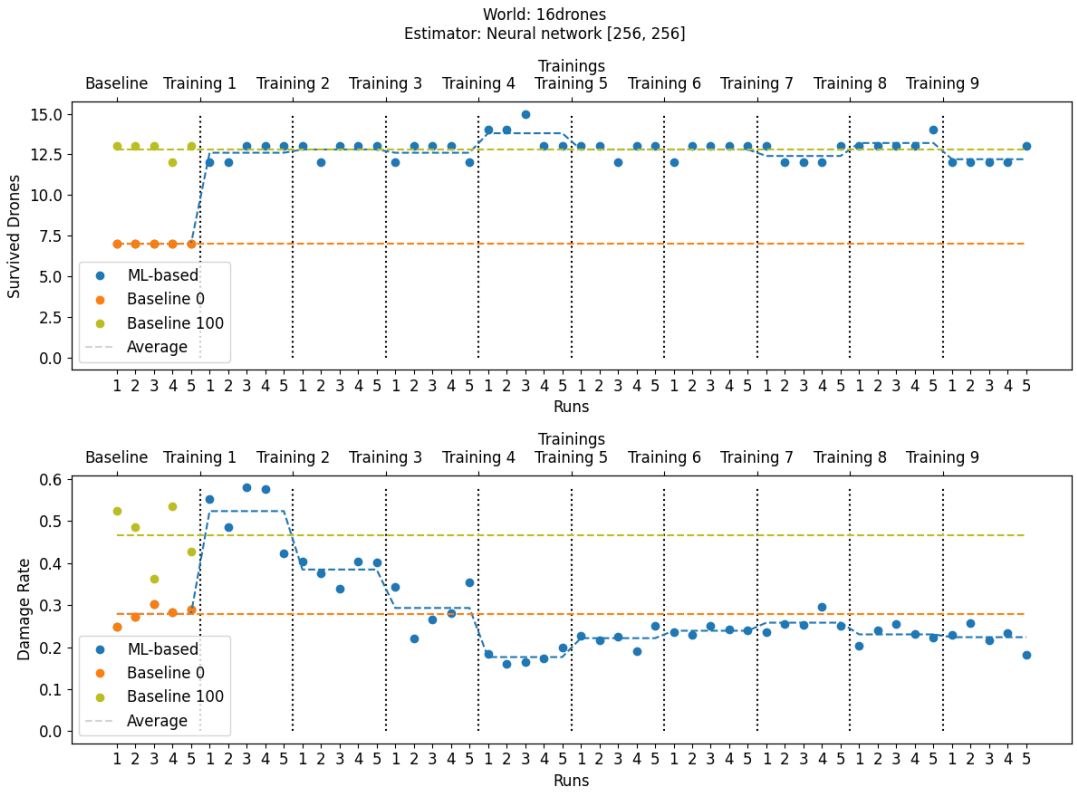


Figure 5.4: Survived drones and damage rate in a simulation with 16 drones while using data from previous three iteration for estimator training.

iterations. Specifically, we use the data from the previous three iterations to obtain the results in Figure 5.4.

The behavior is quite similar to using all the previous iterations. We can see that the adaptation is slower than when using only the last iteration, but the results are almost as stable as when using the data from all the previous iterations.

5.2.5 Summary of the results

We observed similar results for running the simulation with a different number of drones. When training only on data from one iteration at a time, the system falls into a feedback loop. Using the data from all the previous iterations resolves this issue, and using the data from the previous three iterations performs similarly.

Figure 5.5 shows the results of the simulation for a different number of initial drones. For machine-learning-based approaches, the training on the previous three iterations strategy was used and the results from the best iteration are shown (it was the fifth iteration for 12 and 16 drones, and the sixth iteration for 20 drones).

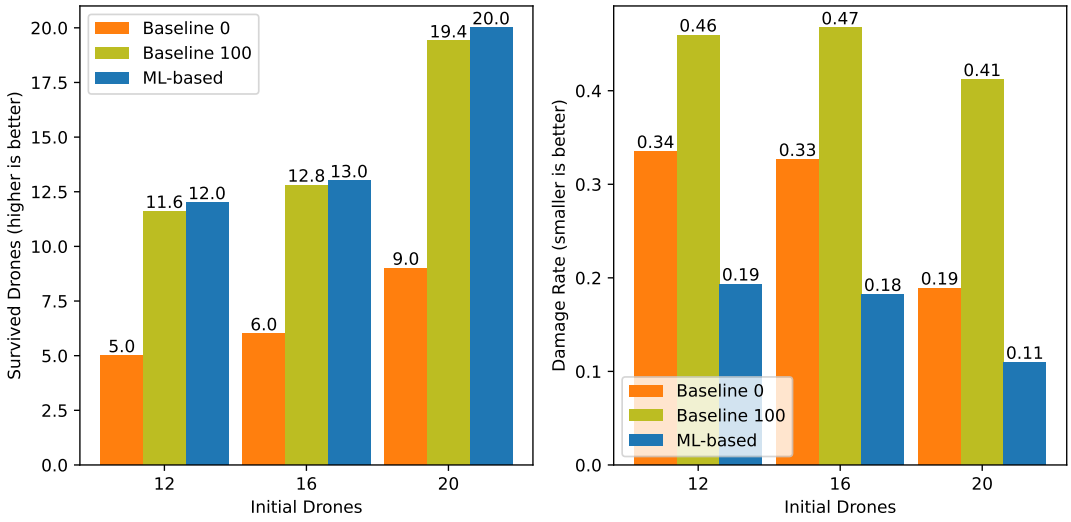


Figure 5.5: Comparison of survived drones and damage rate for baseline and ML-based approach.

Though the results are only indicative and have no generality beyond our use-case, they still show that using machine-learning-based estimates can improve the self-adaptive system. More importantly, the inclusion of machine-learning-based estimates is relatively simple with ML-DEECo, so it can be easily experimented with to see whether it is helpful in a particular use-case.

5.3 Security rules example

In this section, we provide an overview of another use case that uses the ML-DEECo framework. The scenario builds on recent work by Al-Ali et al. [17] and focuses on security rules in a smart factory. A replication package with the source code is available online [18].

5.3.1 Use case

In this example, we model a simulation of a smart factory with security rules to access doors, etc. The factory has multiple working places, each with a team of workers. The teams of workers work on projects for several different customers. The workers from one team are thus allowed only to enter their workplace and cannot enter the other workplaces (to protect the intellectual property of the customers).

In the morning, a shift of workers is assigned to each workplace. These workers are granted permission to enter the factory 30 minutes before their shift starts. Then they have to take a protective headgear from a dispenser inside the factory. Only with the headgear, they are allowed to enter their workplace.

Apart from the workers assigned to the shift, there are also several standby workers in case some of the assigned workers do not arrive in time. When a worker does not arrive at the factory 16 minutes before their shift starts, they get canceled for the day, and a standby worker is called to replace them. We assume that it will take approximately 30 minutes before the standby arrives. The cancellation of the worker includes revoking the permission to enter the factory and granting that permission to the standby.

The scenario is dynamic as replacing the workers in the shift with standbys requires changing the permissions to enter the factory and the workplaces.

5.3.2 Modelling the scenario using components and ensembles

The scenario can be easily modeled using components and ensembles. We consider the following components: `Door`, `Dispenser` (of protective headgear), `Factory`, `WorkPlace`, `Shift` and `Worker`. As we can see, the components can be used to represent all entities in the system, including those which are not physical objects, such as shifts of workers.

We then employ several ensembles to define the security rules. For each shift, we create a `ShiftTeam` ensemble instance (the shift is a static role of the ensemble). In this ensemble, we select all workers of the shift using a dynamic role – we can easily express that we want to include all workers assigned to the shift except those canceled and also include all called standbys. We then use `AccessToFactory` and `AccessToDispenser` ensembles to grant the permission to enter the factory and to obtain the headgear from the dispenser to the workers selected by the `ShiftTeam` ensemble. The `AccessToWorkPlace` is similar, but it further requires that the workers are wearing the protective headgear. The time during which the permission should be granted is easily expressed using the `situation` of the ensembles.

Figure 5.6 (from Al-Ali et al. [17]) shows an example of two ensembles in the scenario. The grey ensemble represents the access to the main gate (`AccessToFactory`) and the blue ensemble is `AccessToWorkPlace` (we can see that all the workers wear the necessary protective headgear).

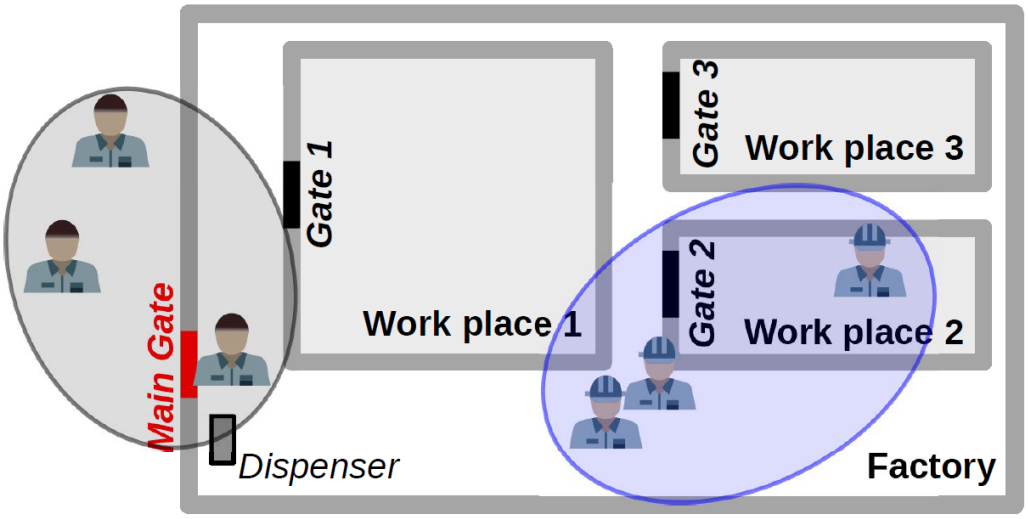


Figure 5.6: Ensembles in the Security rules example. From Al-Ali et al. [17].

5.3.3 Use of machine learning for adaptation

To show a possible usage of machine learning in this scenario, we decided to replace the static rule of canceling a worker 16 minutes before the shift starts with a dynamic threshold based on a machine-learning-based estimate.

We construct an ensemble for selecting the late workers and replacing them with standbys. We start by deciding which workers are potentially late – those who are not yet at the factory and belong to a shift that is about to start. We use the estimate to predict whether the worker will arrive at the factory before the shift start. If the estimate predicts that the worker will not arrive on time, they get canceled and they are replaced by a standby.

To further emphasize the benefits of adaptive rules, we assume that the workers behave differently on business days and during the weekend. We outline the process of generating data for the simulation in the next section.

Generating data for the simulation

For the sake of simplicity, we assume that all the workers arrive by bus at a bus stop a few minutes away from the main gate of the factory. During business days, the bus arrives 24 minutes before the shift starts, and during the weekend, the bus arrives 30 minutes before the shift. Furthermore, we assume that 10% of the workers are late and arrive by a later bus – 18 minutes before the shift starts on business days and 15 minutes before the shift starts during the weekend. To simulate uncertainty in the scenario, we add a random delay to the arrival of each worker with an exponential distribution.

5.3.4 Results

We ran the simulation with 100 workers in each shift for three iterations – in the first iteration, the rigid rule of canceling workers 16 minutes before the shift starts is used, and the two following iterations use the learned estimate. The results are summarized in Figure 5.7. The *lateness* is computed as the square of the delay of workers who arrive late at their workplace.

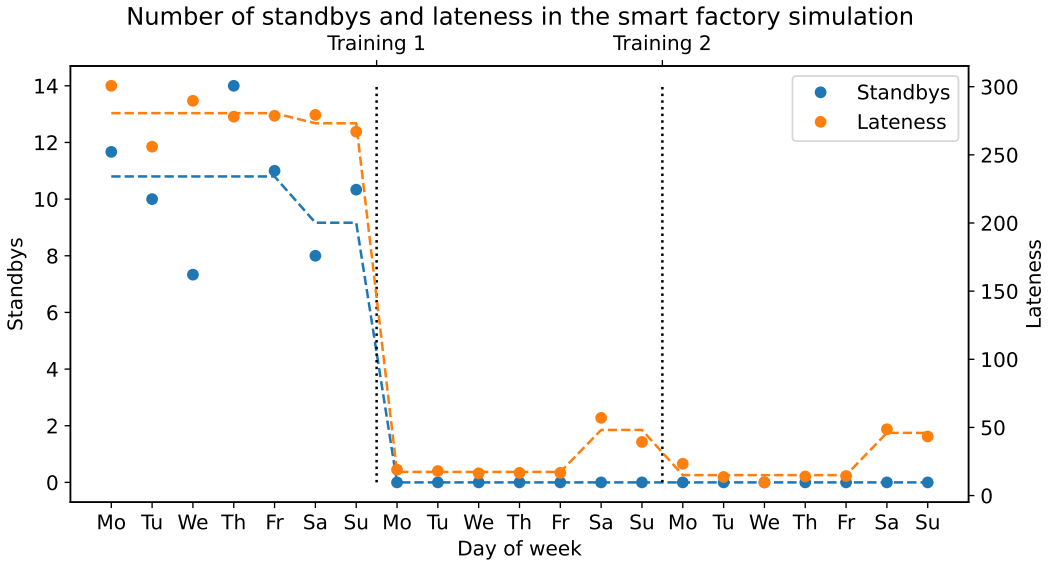


Figure 5.7: Results of the simulation with 100 workers.

We can clearly see that for this configuration, the rules based on learned estimates perform significantly better than the rigid rule. For the rigid rule, we have on average 13 standbys called, which is expected as we assume that 10% of the workers arrive by the later bus and these will not arrive at the main gate on time and thus are canceled. The learned estimate uses a later threshold (as we discuss in the following section) and thus lower number of standbys is needed and the overall lateness is smaller.

Outputs of the neural network

To inspect the decisions of the neural network, we have plotted its outputs in Figure 5.8. The plot shows the output of the network based on its input values – the day of the week and the time before the shift starts. Green values indicate that the network predicts that the worker will arrive at the factory before the shift starts while the red values indicate that the worker will not arrive on time and should be canceled and replaced by a standby.

We can see that the network was able to learn the pattern of business days and weekends and has a different threshold for them. For business days, the threshold is slightly later than the original 16 minutes – it cancels the workers which are not at the factory 12 minutes before the shift starts. This is a reasonable decision as the workers who arrive by the late bus are canceled by the original threshold but they are not canceled by the new threshold. They might come to the workplace a few minutes late, but they still arrive earlier than the standby would. For the

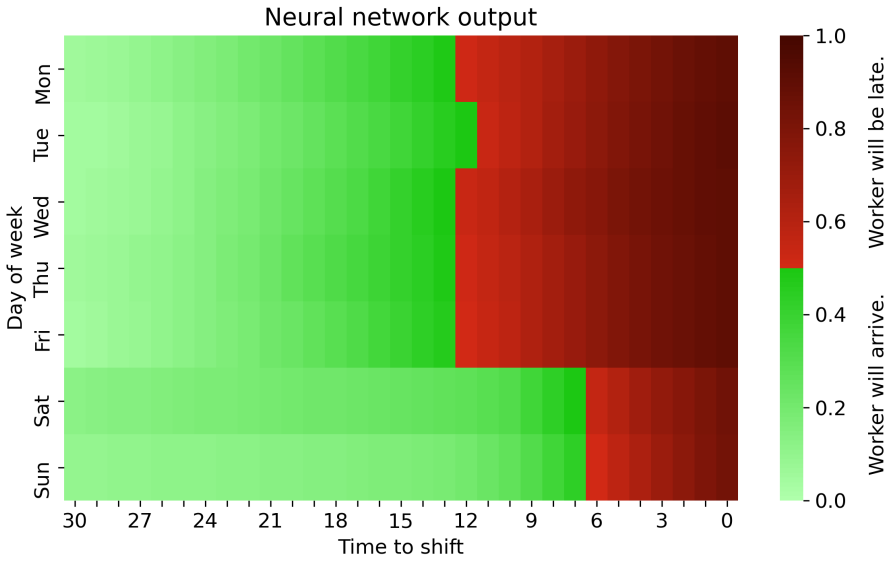


Figure 5.8: Outputs of the neural network for predicting whether a worker will arrive to the factory before their shift starts (green) or not (red).

weekends, we see similar behavior. The network still prefers the slightly late workers instead of the standbys. This time, the late bus arrives later than on business days, so the learned threshold is also later.

Threats to validity

We are aware that the data are constructed under our control. We still think that they illustrate that the machine-learning-based estimates can be useful for adaptation in specific situations as the neural network is able to learn the patterns from the data and adapt the security rules accordingly.

6. Related work

6.1 Ensemble-based component modeling

As already stated in Chapter 2, we base our work on the DEECo [1] component model. Here, we mention several other approaches to ensemble-based component modeling, namely SCEL (Section 6.1.1) and Helena (Section 6.1.2). Then, we provide more details on alternative approaches to ensemble formation in DEECo in Section 6.1.3.

6.1.1 SCEL

SCEL (Service Component Ensemble Language) [19] is a language for rigorous modeling and programming of autonomic components and their interaction. The authors focus on defining programming abstractions to model the evolutions and interactions of autonomic components and ensembles. Similar to DEECo, components can be selected for the ensemble based on a predicate over their attributes.

The authors focus on describing behaviors (modeled as processes executing actions), knowledge (information representation), aggregations (design of autonomic components and the construction of the software architecture of ensembles), and policies (control and adaptation of the actions of the components).

The abstractions have also been materialized in a Java-based runtime framework [20].

6.1.2 Helena

Another approach is to ensemble-based component modeling is Helena [21]. It also features autonomous components and their ensembles. Each ensemble has a set of roles and the components can enter the ensemble in a particular role to take responsibility for a certain part of the task realized by the ensemble. The ensembles form a structure to represent the goals of the system. Similar to DEECo, components can be members of several ensembles simultaneously.

One key difference between Helena and our work based on DEECo is that Helena models a transition system, called ensemble automata, for the state of the ensemble. The ensemble automaton describes the evolution of the ensemble over time – which components represent which role. In our work, the ensembles are state-less and formed anew in every time step of the simulation.

6.1.3 Ensemble formation in DEECo

We now return back to the DEECo [1] component model, which we base our work on, and elaborate on the approaches to forming ensembles at runtime, that we briefly mentioned in Section 2.2.3.

Bureš et al. [9] presented a language and framework for specifying dynamic component ensembles in Scala[1]. In their work, the ensembles can overlap – a

[1]High-level programming language running on JVM. See `https://www.scala-lang.org/`.

component can be a member of multiple ensembles at the same time – and they can be nested – members of a child ensemble must be members of its parent ensemble too. During the ensemble formation (or *instantiation* in the terms of the paper), all potential member components are considered for the ensemble instances. Each possible assignment of the members is evaluated using a *utility* function and the whole ensemble formation is formulated as a constraint optimization problem. The COP is solved and the ensembles are instantiated in an optimal way.

Another approach [10], presented by the same group, is based on machine learning. They argue that solving the COP takes a lot of time and propose formulating the problem as classification and using neural networks or decision trees as trained classifiers. The inputs and the outputs of the machine learning models are the same as for the CSP solver – the inputs comprise the component knowledge and the outputs are Boolean variables that represent the membership of a component instance to an ensemble instance. As the inputs and outputs are the same, the data from the CSP solver can be used as training data for the machine learning models. The authors show they were able to train the classifier with high enough accuracy. Using the trained classifiers can produce a solution that does not comply with the hard constraints, but the authors claim that if the system is well designed, the approximate solutions still work.

6.2 Machine learning in self-adaptive systems

In a systematic literature review of applications of machine learning in self-adaptive systems [8], the authors observe that there is a clear increasing trend of employing machine learning techniques in the area over the recent years. Machine learning is mostly used for directly updating the adaptation rules. Another important area is the prediction and analysis of resource usage. The authors also use the insights from the study to provide an outline of a design process for applying machine learning in self-adaptive systems. Another systematic literature review [7] confirms the same findings. Most of the surveyed works focus on the application of machine learning methods in a specific task. Our goal, on the other hand, is to design a component model with the ability to specify the machine-learning-based estimates in the architecture of the system. We still list the most relevant applications of machine learning in this section.

One of the possible applications of machine learning methods in self-adaptive systems is to reduce a large space of possible adaptations. As the analysis of all possible adaptations can be time-consuming, the authors of DLASeR (Deep Learning for Adaptation Space Reduction) [22] propose an approach for reduction of the adaptation space to consider only relevant adaptations. Their work can handle both threshold and optimization goals. Another work [23] of a similar group of authors combines the machine-learning-aided adaptation space reduction with a cost-benefit analysis to assess the costs of performing the adaptation. The machine-learning-based adaptation space reduction is further analyzed by Ghebi et al. [24] to provide a theoretical bound on the impact of the machine learning when analyzing the system by a formal verifier to provide guarantees for the decision made by the system.

Muccini and Vaidhyanathan [25] show a rather straightforward use of time-series forecasting using recurrent neural networks (namely LSTM) for the predic-

tion of quality of service (QoS) parameters such as the energy consumption of the system. They use the predictions for a proactive adaptation of the system rather than waiting for the QoS parameters to drop below a certain threshold and adapting the system afterward. They further develop the method together with Cámara [26] and employ formal quantitative verification (probabilistic model checking) to check the feasibility of the adaptation decision and provide feedback to the machine learning training for faster convergence towards optimal decisions.

The use of online reinforcement learning for self-adaptation and to address design-time uncertainties is investigated by Palm et al. [27]. They present an approach for automatic fine-tuning of the exploration rate of the reinforcement learning algorithm and quantization of the environment states.

For continuous monitoring and resource demand estimation of self-adaptive systems, Grohmann et. al present the SARDE framework [28]. SARDE dynamically executes and tunes a set of resource demand estimation approaches to select the most reliable approach for the current state of the environment and thus minimize the estimation errors. They use machine learning for model selection.

For verification and quality assurance of the self-adaptive systems with machine learning in mind, Gabor et. al. [29] present a formal framework. They use the coevolution of the adaptive system and the tests to adapt the tests to the adaptation of the system.

7. Conclusion

In this thesis, we have presented ML-DEECo, a machine-learning-enabled component model for architecting distributed smart systems based on the autonomic component ensembles from DEECo component model [1]. ML-DEECo provides primitives for easily enriching the system with machine-learning-based estimates and thus allows adaptivity of the system.

We started by identifying and analyzing possible usages of supervised machine learning in a component-based architecture of an ensemble-based adaptive system. We have identified two important dimensions in which the estimates can be categorized – where the estimate is used and what is the estimated quantity. Based on that, we have designed estimators for the two important tasks – future value prediction (both regression and classification) and time-to-condition prediction. We have described the semantics of the estimators including the process of data collection for training the machine learning models.

Based on that, we have implemented the ML-DEECo runtime framework in Python and made it available as open source [13]. The framework provides abstractions for defining autonomous components and their ensembles as well as machine-learning-based estimates of the attributes of the components and ensemble members. The framework handles all the necessary tasks for providing the estimates, such as the construction of the machine learning models (neural networks in our case), collection of the training data, and training of the model. We have accompanied the documentation of the ML-DEECo framework with examples of declaration of components and ensembles with machine-learning-based estimates.

To evaluate our approach, we have used the ML-DEECo framework to architect a simulation of a use-case from the area of smart farming. The source code of the simulation is available in the replication package [16]. The simulated system consists of autonomous drones which protect fields with crops against birds. We use a machine-learning-based estimate to optimize and adapt the charging of the drones. The results of the simulation show that the addition of machine learning was beneficial as the adapted charging allows the drones to save more crops.

In our future work, we would like to build on this approach and focus on other machine learning approaches, most importantly on reinforcement learning.

Bibliography

[1] Tomas Bures, Ilias Gerostathopoulos, Petr Hnetynka, Jaroslav Keznikl, Michal Kit, and Frantisek Plasil. DEECo: an ensemble-based component system. In *Proceedings of the 16th International ACM Sigsoft symposium on Component-based software engineering - CBSE '13*. ACM Press, 2013. doi:10.1145/2465449.2465462.

[2] Tomas Bures, Ilias Gerostathopoulos, Petr Hnetynka, Jaroslav Keznikl, Michal Kit, and Frantisek Plasil. Gossiping Components for Cyber-Physical Systems. In *Proceedings of ECSA 2014, Vienna, Austria*, volume 8627 of *LNCS*, pages 250–266. Springer, 2014. doi:10.1007/978-3-319-09970-5_23.

[3] Tomas Bures, Frantisek Plasil, Michal Kit, Petr Tuma, and Nicklas Hoch. Software Abstractions for Component Interaction in the Internet of Things. *Computer*, 49(12):50–59, 2016. ISSN 0018-9162.

[4] Tomas Bures, Petr Hnetynka, Jan Kofron, Rima Al Ali, and Dominik Skoda. Statistical Approach to Architecture Modes in Smart Cyber Physical Systems. In *Proceedings of WICSA 2016, Venice, Italy*, pages 168–177. IEEE, April 2016. doi:10.1109/WICSA.2016.33.

[5] Filip Krijt, Zbynek Jiracek, Tomas Bures, Petr Hnetynka, and Ilias Gerostathopoulos. Intelligent Ensembles - A Declarative Group Description Language and Java Framework. In *Proceedings of SEAMS 2017, Buenos Aires, Argentina*, pages 116–122. IEEE, 2017. doi:10.1109/SEAMS.2017.17.

[6] Petr Hnetynka, Tomas Bures, Ilias Gerostathopoulos, and Jan Pacovsky. Using Component Ensembles for Modeling Autonomic Component Collaboration in Smart Farming. In *Proceedings of SEAMS 2020, Seoul, Korea*, pages 156–162. ACM, 2020. doi:10.1145/3387939.3391599.

[7] Theresia Ratih Dewi Saputri and Seok-Won Lee. The Application of Machine Learning in Self-Adaptive Systems: A Systematic Literature Review. *IEEE Access*, 8:205948–205967, 2020. doi:10.1109/ACCESS.2020.3036037.

[8] Omid Gheibi, Danny Weyns, and Federico Quin. Applying Machine Learning in Self-adaptive Systems: A Systematic Literature Review. *ACM Transactions on Autonomous and Adaptive Systems*, 15(3):9:1–9:37, August 2021. doi:10.1145/3469440.

[9] Tomas Bures, Ilias Gerostathopoulos, Petr Hnetynka, Frantisek Plasil, Filip Krijt, Jiri Vinarek, and Jan Kofron. A language and framework for dynamic component ensembles in smart systems. *International Journal on Software Tools for Technology Transfer*, 22(4):497–509, feb 2020. doi:10.1007/s10009-020-00558-z.

[10] Tomáš Bureš, Ilias Gerostathopoulos, Petr Hnětynka, and Jan Pacovský. Forming ensembles at runtime: A machine learning approach. In *Leveraging Applications of Formal Methods, Verification and Validation: Engi-*

neering Principles, pages 440–456. Springer International Publishing, 2020. doi:10.1007/978-3-030-61470-6_26.

[11] Tom M. Mitchell. *Machine Learning*. McGraw-Hill, New York, 1997. ISBN 978-0-07-042807-2.

[12] Warren S. McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The Bulletin of Mathematical Biophysics*, 5 (4):115–133, dec 1943. doi:10.1007/bf02478259.

[13] ML-DEECo, 2022. URL `https://github.com/smartarch/ML-DEECo`.

[14] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL `https://www.tensorflow.org/`. Software available from tensorflow.org.

[15] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, December 2014.

[16] Replication package (smart farming), 2022. URL `https://github.com/Mnaukal/ML-DEECo-replication-package`.

[17] Rima Al-Ali, Petr Hnetynka, Jiri Havlik, Vlastimil Krivka, Robert Heinrich, Stephan Seifermann, Maximilian Walter, and Adrian Juan-Verdejo. Dynamic security rules for legacy systems. In *Proceedings of the 13th European Conference on Software Architecture - ECSA '19 - volume 2*. ACM Press, 2019. doi:10.1145/3344948.3344974.

[18] Replication package (security rules), 2022. URL `https://github.com/Mnaukal/ml-deeco-security-isola`.

[19] Rocco De Nicola, Diego Latella, Alberto Lluch Lafuente, Michele Loreti, Andrea Margheri, Mieke Massink, Andrea Morichetta, Rosario Pugliese, Francesco Tiezzi, and Andrea Vandin. The SCEL Language: Design, Implementation, Verification. In *Software Engineering for Collective Autonomic Systems*, number 8998 in LNCS, pages 3–71. Springer, 2015. doi:10.1007/978-3-319-16310-9_1.

[20] jRESP: Java Runtime Environment for SCEL Programs, 2013. URL `http://jresp.sourceforge.net/`. Accessed:2022/01/20.

[21] Rolf Hennicker and Annabelle Klarl. Foundations for Ensemble Modeling – The Helena Approach. In *Specification, Algebra, and Software*, number 8373 in LNCS, pages 359–381. Springer, 2014. doi:10.1007/978-3-642-54624-2_1.

[22] Jeroen Van Der Donckt, Danny Weyns, Federico Quin, Jonas Van Der Donckt, and Sam Michiels. Applying deep learning to reduce large adaptation spaces of self-adaptive systems with multiple types of goals. In *Proceedings of SEAMS 2020, Seoul, South Korea*, pages 20–30. ACM, 2020. doi:10.1145/3387939.3391605.

[23] Jeroen Van Der Donckt, Danny Weyns, M. Usman Iftikhar, and Sarpreet Singh Buttar. Effective Decision Making in Self-adaptive Systems Using Cost-Benefit Analysis at Runtime and Online Learning of Adaptation Spaces. In *Evaluation of Novel Approaches to Software Engineering*, volume 1023 of *LNCS*, pages 373–403. Springer, 2019. doi:10.1007/978-3-030-22559-9_17.

[24] Omid Gheibi, Danny Weyns, and Federico Quin. On the Impact of Applying Machine Learning in the Decision-Making of Self-Adaptive Systems. In *Proceedings of SEAMS 2021, Madrid, Spain*, pages 104–110. IEEE, May 2021. doi:10.1109/SEAMS51251.2021.00023.

[25] Henry Muccini and Karthik Vaidhyanathan. A machine learning-driven approach for proactive decision making in adaptive architectures. In *Companion Proceedings of ICSA 2019, Hamburg, Germany*, pages 242–245, 2019. doi:10.1109/ICSA-C.2019.00050.

[26] Javier Cámara, Henry Muccini, and Karthik Vaidhyanathan. Quantitative Verification-Aided Machine Learning: A Tandem Approach for Architecting Self-Adaptive IoT Systems. In *Proceedings of ICSA 2021, Salvador, Brazil*, pages 11–22. IEEE, March 2020. doi:10.1109/ICSA47634.2020.00010.

[27] Alexander Palm, Andreas Metzger, and Klaus Pohl. Online Reinforcement Learning for Self-adaptive Information Systems. In *Proceedings of CAiSE 2020, Grenoble, France*, volume 12127 of *LNCS*, pages 169–184. Springer, 2020. doi:10.1007/978-3-030-49435-3_11.

[28] Johannes Grohmann, Simon Eismann, André Bauer, Simon Spinner, Johannes Blum, Nikolas Herbst, and Samuel Kounev. SARDE: A Framework for Continuous and Self-Adaptive Resource Demand Estimation. *ACM Transactions on Autonomous and Adaptive Systems*, 15(2):1–31, June 2021. doi:10.1145/3463369.

[29] Thomas Gabor, Andreas Sedlmeier, Thomy Phan, Fabian Ritz, Marie Kiermeier, Lenz Belzner, Bernhard Kempter, Cornel Klein, Horst Sauer, Reiner Schmid, Jan Wieghardt, Marc Zeller, and Claudia Linnhoff-Popien. The scenario coevolution paradigm: adaptive quality assurance for adaptive systems. *International Journal on Software Tools for Technology Transfer*, 22 (4):457–476, March 2020. doi:10.1007/s10009-020-00560-5.

List of Figures

List of Listings