Attuning Adaptation Rules via a Rule-Specific Neural Network

Tomáš Bureš[1], Petr Hnětynka[1], Martin Kruliš[1], František Plášil[1], Danylo Khalyeyev[1], Sebastian Hahner[2], Stephan Seifermann[2], Maximilian Walter[2], Robert Heinrich[2],

[1] Charles University, Czech Republic
{bures,hnetynka,krulis,plasil,khalyeyev}@d3s.mff.cuni.cz
[2] Karlsruhe Institute of Technology (KIT), Germany
{sebastian.hahner,stephan.seifermann,
maximilian.walter,robert.heinrich}@kit.edu

Abstract. There have been a number of approaches to employing neural networks (NNs) in self-adaptive systems; in many cases, generic NNs/deep learning are utilized for this purpose. When this approach is to be applied to improve an adaptation process initially driven by logical adaptation rules, the problem is that (1) these rules represent a significant and tested body of domain knowledge, which may be lost if they are replaced by an NN, and (2) the learning process is inherently demanding given the black-box nature and the number of weights in generic NNs to be trained. In this paper, we introduce the rule-specific Neural Network (rsNN) method that makes it possible to transform the guard of an adaptation rule into an rsNN, the composition of which is driven by the structure of the logical predicates in the guard. Our experiments confirmed that the black box effect is eliminated, the number of weights is significantly reduced, and much faster learning is achieved while the accuracy is preserved.

Keywords: Self-adaptive systems · adaptation rules · machine learning · neural networks

1 Introduction

The recent advances in neural networks and machine learning [8] led to their proliferation in various disciplines, and the field of self-adaptive systems is no exception [13]. In particular, they have found usage in approaches to control how systems of cooperating agents are formed and reconfigured at runtime [12,4].

These approaches employ neural networks to implement the self-adaptation loop, also known as the MAPE-K loop, which controls the runtime decisions in the system (e.g., to which service to route a particular request) and the runtime architectural changes (e.g., which services to deploy/un-deploy or reconfigure).

In typical cases, a neural network is used for the analysis and planning stages of the MAPE-K loop, replacing the traditional means of analyzing the

system state and deciding on adaptation actions. These traditional adaptation mechanisms are often specified in some form of logical rules (e.g., if-then rules or a state machine with guards and actions) [12,6,16].

Using a neural network for making decisions on adaptation actions naturally means training the network for the situations the self-adaptive system is supposed to handle. Such training typically requires a large number of system behavior examples—training data in the form of observed inputs and expected adaptation actions. This approach is significantly different from the logical rules that have been traditionally used to describe adaptation actions. Due to this substantial conceptual gap between the two approaches, it is difficult to evolve an existing self-adaptive system based on some form of logical rules into a new system that uses a neural network to make adaptation decisions. Seemingly, the typical design choice is to recreate the analysis and planning stages of the MAPE-K loop from scratch.

The existing logical rules represent a significant body of domain knowledge, especially if the system has been well-functioning and tuned to its task. Thus, when replacing the logical rules with a neural network, this body of domain knowledge is often lost, which leads to severe regress. On the other hand, applying neural networks may be advantageous as they can dynamically learn completely unanticipated relationships of stochastic character. Thus, it makes the self-adaption refined to take advantage of the specific features otherwise hidden in the system and not captured in the inherently static logical rules.

Nevertheless, if logical rules are used for determining the expected actions in training data, it is not easy to train the neural network to reliably yield actions corresponding to the existing rule-based self-adaptive system in question. The main culprit is that the neural network is often built as a black box composed of generic layers (such as a combination of recurrent and dense layers). Thus, the structure of such a generic neural network does not reflect the relationships characteristic of the domain in which the self-adaptive system resides. In other words, the neural network is built as a generic one, not exploiting the existing domain knowledge about the self-adaptive system whose adaptation actions it controls.

While this genericity is inherently advantageous in empowering the neural network to "discover" ultimately unanticipated relationships, it may also hinder the ability to adequately learn because it makes the neural network relatively complex, thus potentially increasing adaptation uncertainty.

Therefore, replacing a rule-based adaptation entirely with a generic neural network-based one might be an overly drastic change that may potentially degrade the reliability of the system (at least in the short-term perspective). Moreover, it may raise legitimate concerns since generic neural networks are much less comprehensible and predictable given their black-box nature and the typically large number of weights to be trained—there is always a danger of overfitting.

In this paper, we aim to answer the following research questions: (1) how to endow an existing rule-based self-adaptation system with the ability to learn via neural networks while still benefiting from the domain knowledge encoded in the

logical rules; and (2) how to scale the learning ability in a way that would allow the transition from logical rules to a neural network to be done on a step-by-step basis.

We address these research questions by introducing a rule-specific Neural Network (rsNN) method, which allows the transformation of an adaption rule to the corresponding rsNN to be done systematically. The key feature is that an rsNN is composable - its architecture is driven by the structure of the logical predicates in the adaption rule in question. Moreover, prior to the composition process, the predicates can be refined by predefined atomic "attunable" predicates, each having a direct equivalent in a primitive element of rsNN ("seed" of rsNN).

The rest of the paper is organized as follows. Section 2 presents an example that is used for motivating and illustrating rsNN. Section 3 is devoted to the key contribution of the paper—it describes the concepts and ideas of rsNN, while Section 4 discusses the methodology, results, and limitations of experimental evaluation. Section 5 discusses other approaches focused on employing neural networks in self-adaptation, and the concluding Section 6 summarizes the contribution.

2 Motivating example

As a motivating example, we utilize a realistic yet straightforward use-case from our former project focused on security in Industry 4.0 settings[3]. The example employs the MAPE-K loop principle to dynamically reconfigure the software architecture of agents—workers (represented by components) operating jointly on a common task. In the architecture, groups of workers are determined by the access policies that allow the member workers to perform their tasks. Since these tasks are subject to changes, the access control is intertwined with dynamic, runtime modification of the software architecture.

Implementation-wise, the MAPE-K controller dynamically re-establishes the groups of workers to deal with situations in the environment—e.g., when a machine breaks down, the MAPE-K controller establishes a group of workers that communicate and collaborate to fix the machine (so that the software architecture is dynamically reconfigured). It also gives these workers the necessary access rights, e.g., to access the machine's logs and physically enter the room (workplace) where the machine is located.

In the example, we pick up a particular adaptation rule from the larger use-case in the project mentioned above, and later in Section 3, we will employ it to demonstrate a step-by-step transition from this static adaptation rule to the corresponding rsNN neural network.

Let us consider a factory with several workplaces where production is organized in shifts, each determined by its start and end time, during which worker groups, each with an assigned (the only) workplace, perform their tasks. The workers are allowed to enter the factory only at a time close to a particular shift's start

[3] `http://trust40.ipd.kit.edu/home/`

and must leave soon after the shift ends. After entering, they have to pick up headgear (protective equipment) from a dispenser as a necessary condition for being permitted to enter the assigned workplace. Similarly, they are allowed to enter only the assigned workplace and only close to the shift start time (and have to leave as soon as the shift ends).

As expected in the Industry 4.0 domain, the assignment of workers to particular shifts is not static but can frequently change, and the roles of individual workers within the shift can also alternate rapidly. This leads to changes in the runtime software architecture. Consequently, the access control system of the factory cannot assign access rights statically only, thus supporting dynamic, situation-based access control.

To perform access right adaptation, the MAPE-k controller uses adaptation rules in the form of guard-actions, where the action is adding/revoking or allowing access.

Listing 1 shows an example of an adaptation rule which dynamically determines a group of workers formed for the duration of a shift, having access rights to the assigned workplace. In particular, the adaptation rule specifies whether a specific worker belongs to the group and if so, it gives the worker access to the workplace assigned for the shift.

The structure of the adaptation rule has three parts. First, there are declared data fields (in this particular case, only a single field initialized to the shift of the given worker—line 2). Second, there is a *guard*, which defines the condition when the rule is applied. This particular guard reads: To allow a worker to enter the assigned workplace, the worker needs to be already at the appropriate workplace gate (line 5), needs to have a headgear ready (line 6), and needs to be there at the right time (i.e., during the shift or close to its start or end—line 4). Finally, there is an *action* determining what has to be executed—in this case, the assignment of the *allow* access rights to the assigned workplace to the worker (line 8).

```
1 rule AccessToWorkplace(worker) {
2     shift = shifts.filter(worker in shift.workers)
3     guard {
4       duringShift(shift) &&
5         atWorkplaceGate(worker, shift.workplace) &&
6           hasHeadgear(worker)
7     }
8     action { allow(worker, ENTER, shift.workplace) }
9 }
```

Listing 1: Access to workplace rule

The predicates atWorkplaceGate, hasHeadgear, and duringShift are declared in Listing 2.

The predicate duringShift tests whether the current time is between 20 minutes (i.e., 1200 seconds) before the start of the shift and 20 minutes after the end of the shift. The global variable NOW contains the current time.

The atWorkplaceGate predicate mandates that the position of the worker has to be close (in terms of Euclidean distance) to the gate of the workplace assigned to the worker.

The predicate hasHeadgear checks whether the worker retrieved a headgear from the dispenser. To check this, we assume that each worker is associated with a list of related events (the events data field of the worker—line 11 in Listing 2). For instance, retrieving and returning the headgear are the events registered in the list of events upon performing the respective actions. Thus, the check of whether a worker has a headgear available is performed by verifying that after filtering the two specific event types from the list (line 12), the latest event is TAKE_HGEAR (the filtered events are sorted in descending order—line 13 and line 14).

```
1  pred duringShift(shift) {
2    shift.startTime − 1200 < NOW && shift.endTime + 1200 > NOW
3  }
4
5  pred atWorkplaceGate(worker, workplace) {
6      sqrt((workplace.gate.posX − worker.posX) ^ 2 +
7        (workplace.gate.posY − worker.posY) ^ 2) < 10
8  }
9
10 pred hasHeadgear(worker) {
11     worker.events
12       .filter(event −> event.type == (TAKE_HGEAR || RET_HGER))
13       .sortDesc(event −> event.time)
14       .first().type == TAKE_HGEAR
15 }
```

Listing 2: Predicates from Listing 1

3 Refining adaptation rules

The problem with the adaptation rules we presented in Section 1 is that their guards are too static, and thus they do not capture the domain-specific stochastic character of the data they act upon. As already mentioned in Section 1, we aim to employ a dedicated *rule-specific neural network* (*rsNN*) to benefit from its ability to learn from the domain characteristic data being handled. To this end, in this section, we outline the method that allows us to refine an original adaption rule to make its guard predicates "attunable" and convert the guard into an *rsNN*. In a sense, our method of employing a dedicated *rsNN* for this purpose can be viewed as paving a middle ground between the adaptation rules with static guards and the adaptation rules driven by (typically complex) generic neural networks such as in [12,17].

The main idea of our method unfolds in three stages:

1. An adaptation rule is *refined* by manually rewriting (transforming) its selected guard predicates into their attunable form—they become *attunable predicates*. This is done by applying predefined atomic attunable predicates (*aa-predicates*) listed in Section 3.1. These aa-predicates serve as *rsNN* seeds in the second stage. Nevertheless, not all the guard predicates have to

be transformed this way—those remain *static predicates* (their selection is application-specific).
2. We apply an automated step that generates an *rsNN* that reflects the guard of the refined adaptation rule, containing, in particular, the trainable parameters of aa-predicates as trainable weights.
3. We employ traditional neural network training using stochastic gradient descent to pre-train the trainable weights.

The result is an *rsNN* being a custom neural network, the composition of which is driven by the structure of the guard formula with aa-predicates. This neural network is pre-trained to match outputs of the original guard formula of the adaptation rule. Nevertheless, being a neural network, it can be further trained by running additional examples.

As to pre-training data, we assume there are sample traces of input data to the system, obtained either from historical data, simulation, or random sampling. We use the logical formulas of the original guard predicates over the input data to provide the ground truth (i.e., expected inputs) employed in the supervised learning of the *rsNN*.

Further, the developer has the ability to specify the learning capacity in many aa-predicates, which in turn determines how many neurons are used for its implementation in the *rsNN*.

3.1 Atomic attunable predicates as *rsNN* seeds

This section provides an overview of the aa-predicates defined in the *rsNN* methods. The key idea is that these predicates serve as elementary building blocks for attunable predicates forming an adaptation rule, and at the same time, each of them is easily transformable into a building block of *rsNN*—it serves as an *rsNN* seed as defined in Section 3.3.

Each aa-predicate operates on a single n-dimensional input value (i.e., a fixed-sized vector). Since each aa-predicate yields a true/false value, its corresponding *rsNN* seed solves a classification task, yielding likewise true/false.

Following the type of input value domain, we distinguish between aa-predicates that operate on domains with a metric (i.e., with the ability to measure the distance between quantities) and categorical quantities where no such metric exists:

1. **Metric Quantity:** There are two types of aa-predicates defined over a metric:

   (a) Quantity lies in a one-sided interval

   $$isAboveThreshold\_nD(x, min, max)$$
   $$isBelowThreshold\_nD(x, min, max)$$

   Here x is a value in an n-dimensional space that is compared to a learned threshold (above or below) by the corresponding *rsNN* seed. In order to

control the uncertainty that is potentially induced by learning, the *min* and *max* parameters impose the limits for the learned threshold.

(b) Quantity lies in a two-sided interval

$$hasRightValue\_nD(x, min, max, c)$$

Here it is verified whether the parameter x lies inside the learned interval of an n-dimensional space. The parameters *min* and *max* have the same meaning as in the case of the aa-predicates for a one-sided interval, while the parameter c states the learning capacity of the corresponding *rsNN* seed; technically, this is, e.g., the highest number of the neurons in a hidden layer of the *rsNN* seed.

2. **Categorical quantity:** For this type of input domain, we define an aa-predicate that decides whether a categorical quantity has the right value:

$$hasRightCategories\_nD(x, m, c)$$

Here x is an n-dimensional vector of categorical values from the same domain of the size m (the number of categories). The corresponding *rsNN* learns which combinations of categorical values in the input vector satisfy this aa-predicate. The learning capacity is determined by c.

3.2 Making guard predicates attunable

In this section, we demonstrate the first stage of the *rsNN* method (i.e., the manual rewriting of guard predicates) on the example presented in Section 2. We show two alternatives to such rewriting to demonstrate that a designer may choose several ways to make a predicate attunable depending on what quantities are to be the subject of future learning.

We start with the guard predicates shown in Listing 2. At first, we assume that the designer would like to rewrite duringShift to make it attunable, with the goal to learn the permitted time interval in which the access is allowed. For example, security reasons may require learning the typical behavior patterns of workers induced by the public transportation schedule. (On the contrary, in Listing 2, the interval is firmly set from 20 minutes before the shift starts to 20 minutes after the shift is over.)

We rewrite the duringShift guard predicate as shown in Listing 3: The comparison of NOW with a particular threshold is replaced by the aa-predicates isAboveThreshold and isBelowThreshold, respectively. Each of them represents a comparison against a learned threshold.

The aa-predicates isAboveThreshold and isBelowThreshold have three parameters: (1) the value to test against the learned threshold, (2) the minimum value of the threshold, (3) the maximum value of the threshold.

Since this threshold should not depend on the actual time of the shift, the times are given relative to its start and end. By assuming a worker cannot arrive

earlier than one hour before the shift starts (+3600 seconds in line 2), the relative time 0 corresponds to that point in time (as computed by NOW + 3600 − shift.end). Similarly, by assuming a worker cannot leave later than one hour after the shift ends (−3600 seconds in line 4), the relative time 0 corresponds to that point in time (as computed by NOW − 3600 − shift.end). The minimum and maximum values of the threshold correspond to the interval of 10 hours (i.e., 36000 seconds).

```
1  pred duringShift(shift) {
2    isAboveThreshold_1D(NOW + 3600 − shift.start,min=0, max=36000)
3        &&
4    isBelowThreshold_1D(NOW − 3600− shift.end, min=−36000, max=0)
5  }
6
7  pred atWorkplaceGate(worker, workplace) {
8    sqrt((workplace.gate.posX − worker.posX) ^ 2 + (workplace.gate.posY −
         worker.posY) ^ 2) < 10
9  }
10
11 pred hasHeadgear(worker) {
12   worker.events.filter(event −> event.type == TAKE_HGEAR || RET_HGER))
13                .sortDesc(event −> event.time).first().type == TAKE_HGEAR
14 }
```

Listing 3: Guard predicates with refined duringShift by aa-predicates—one-sided intervals

The other predicates atWorkplaceGate and hasHeadgear) stay the same, as does their conjunction in the AccessToWorkplace rule.

Note that we combined static predicates with an attunable predicate. This shows that only a part of a rule can be endowed with the ability to learn while the rest can stay unchanged. At the same time, we put strict limits on how far the learning can go. In the example, these limits are expressed by the interval of 10 hours which spans from one hour before the shift to one hour after the shift (assuming the shift takes 8 hours). In other words, the value in the attunable predicate gained in the process of learning cannot exceed these bounds. This is useful if learning is to be combined with strict assurances with respect to uncertainty control.

As another alternative of the rule refinement, we assume the time of entry, place of entry, and the relation to the last event concerning the headgear is to be learned. Also, contrary to the variant of duringShift in Listing 3, we assume the time of entry is not just a single interval but can be multiple intervals (e.g., to reflect the fact that workers usually access the gate only at some time before and after the shift due to the public transportation opportunities).

To capture this, we rewrite the predicates duringShift , atWorkplaceGate, and hasHeadGear as shown in Listing 4.

The guard predicate duringShift is realized using the aa-predicate hasRightValue1D, which represents a learnable set of intervals. It has four parameters. In addition to the first three, which have the same meaning as before (i.e., value to be tested on whether it belongs to any of the learned intervals, the minimum, and the maximum value for the intervals), there is the fourth parameter capacity

which expresses learning capacity. The higher it is, the finer intervals the predicate is able to learn. Since it works relative to the min/max parameters, it is unitless. Technically, the learning capacity determines the number of neurons used for training. The exact meaning of the capacity parameter is given further in Section 3.3.

The guard predicate atWorkplaceGate is rewritten similarly. However, as the position is a two-dimensional vector, a 2D version of the hasRightValue aa-predicate is used. The meaning of its argument is the same as in the 1D version applied for the duringShift. A special feature of atWorkplaceGate is that it is specific to the workplace assigned to the worker. (There are several workplaces where the work is conducted during a shift. Each worker is assigned to a particular workplace, and their access permission is thus limited only to that workplace.) Thus, the hasRightValue2D aa-predicate has to be trained separately for each workplace. The square brackets express this after the hasRightValue2D aa-predicate, which signifies that its training is qualified by workplace ID. Since the running example assumes that there are three workplaces in a shift, there are three aa-predicates to be trained.

The hasHeadGear guard predicate is rewritten using the hasRightCategories_1D aa-predicate which assumes 1-dimensional vector of categorical values (i.e., a single value in this case) from the domain of size 2. In this simple case, the learning capacity is set to 1.

```
1  pred duringShift(shift) {
2      hasRightValue_1D(NOW − shift.start, min=0, max=36000, capacity=20)
3  }
4
5  pred atWorkplaceGate(worker) {
6      hasRightValue_2D[worker.workplace.id](worker.pos,
7                      min=(0,0), max=(316.43506,177.88289), capacity=20)
8  }
9
10 pred hasHeadGear(worker) {
11     hasRightCategories_1D(
12       worker.events.filter(event −> event.type == (TAKE_HGEAR ||
             RET_HGER))
13           .sortDesc(event −> event.time).take(1), categories=2, capacity=1
14     )
15 }
```

Listing 4: Guard predicates expressed by a two-sided interval and categorical quantity aa-predicates

3.3 Construction of *rsNN*

In this section, we formalize the second stage of the *rsNN* method, i.e., the automated construction of an *rsNN* that reflects the guard of a refined adaptation rule. First, we show how to transform a logical formula into an elementary *rsNN* (*rsNN seed*) in general, and how to combine *rsNN* seeds into larger units (and how to combine these larger units as well) via transformed logical connectives.

Then, we describe how the elementary logical formulas in the guard (i.e., static predicates and aa-predicates) are transformed into *rsNN* seeds.

Transforming a logical formula and connectives.
A logical formula $L(x_1, \ldots, x_m)$ is transformed to a continuous function $N(x_1, \ldots, x_m, w_1, \ldots, w_n) \rightarrow [0, 1]$ (i.e., a neural network), where $x_1, \ldots, x_m$ are the inputs to the logical formula (e.g., the current time, position of the worker in an aa-predicate), and $w_1, \ldots, w_n$ are trainable weights. The goal is to construct the function N and to train its weights in such a way that $L(x_1, \ldots, x_m) \Leftrightarrow N(x_1, \ldots, x_m, w_1, \ldots, w_n) > 0.5$ for as many inputs $x_1, \ldots, x_m$ as possible. By convention, we interpret $N(\ldots) > 0.5$ as $true$, while if this relation does not hold it is interpreted as $false$. Also, we use the symbol $\mathcal{T}$ to denote the transformation from the logical formula L to the continuous function N—i.e., $N(\ldots) = \mathcal{T}(L(\ldots))$.

As to logical connectives, we deviate from the traditional notion in which conjunction is defined as a product and disjunction is derived using De Morgan's laws. This is because our experiments showed that the conjunctions of multiple operands are close to impossible to train (very likely due to the vanishing gradient problem [9]). Therefore we transform conjunction and disjunction as follows (similarly to in [11]):

$$\mathcal{T}(L_1 \& \ldots \& L_k) = S\left((\mathcal{T}(L_1) + \cdots + \mathcal{T}(L_k) - k + 0.5) * p\right)$$

$$\mathcal{T}(L_1 \vee \cdots \vee L_k) = S\left((\mathcal{T}(L_1) + \cdots + \mathcal{T}(L_k) - 0.5) * p\right)$$

$$\mathcal{T}(\neg L) = 1 - \mathcal{T}(L)$$

where $S(x)$ is the sigmoid activation function defined as $S(x) = \frac{1}{1+e^{-x}}$, and $p > 1$ is an adjustable strength of the conjunction/disjunction operator. The bigger it is, the stricter the results are. However, too high values have the potential to harm training due to the vanishing gradient problem.

Transformation of a static predicate. A static predicate is transformed simply into a function that returns 0 or 1 depending on the result of the static predicate. Formally, we transform a static predicate $L_S(x_1, \ldots, x_m)$ to the function $N_S(x_1, \ldots, x_m)$ as follows:

$$\mathcal{T}(L_S) = \begin{cases} 0 & \text{if not } L_S(x_1, \ldots, x_m) \\ 1 & \text{if } L_S(x_1, \ldots, x_m) \end{cases}$$

Transformation of one-sided interval aa-predicates. We transform an aa-predicate $isAboveThreshold(x, min, max)$ to the function $N_>(x, w_t)$ and an aa-predicate $isBelowThreshold(x, min, max)$ to the function $N_<(x, w_t)$ as follows.

$$\mathcal{T}(isAboveThreshold) = S\left(\left(\frac{x - min}{max - min} - w_t\right) * p\right)$$

$$\mathcal{T}(isBelowThreshold) = S\left(\left(w_t - \frac{x - min}{max - min}\right) * p\right)$$

where w_t is a trainable weight.

Transformation of two-sided interval aa-predicates. We base these aa-predicates on radial basis function (RBF) networks [14]. We apply one hidden layer of Gaussian functions and then construct a linear combination of their outputs. The weights in the linear combination are trainable. The training capacity c in the aa-predicate determines the number of neurons (i.e., points for which the Gaussian function is to be evaluated) in the hidden layer.

We set the means μ_i of the Gaussian function to a set of points over the area delimited by *min* and *max* parameters of the aa-predicate (e.g., forming a grid or being randomly sampled from a uniform distribution). We choose the σ parameter of the Gaussian function to be of the scale of the mean distance between neighbor points. The exact choice of σ seems not to be very important. Our experiments have shown that it has no significant effect and what matters is only its scale, not the exact value. The trainable linear combination after the RBF layer automatically adjusts to the chosen values of μ_i and σ.

For the sake of clarity, we show the transformation of

$$\mathit{hasRightValue\_nD}(x, \mathit{min}, \mathit{max}, c)$$

for $n = 1$ and for arbitrary n. In the 1-D case, we transform an aa-predicate $\mathit{hasRightValue\_1D}(x, \mathit{min}, \mathit{max}, c)$ to the function $N^1_{\simeq}(x, w_{a_1}, \ldots, w_{a_c}, w_b)$ as follows:

$$\mathcal{T}(\mathit{hasRightValue\_1D}) = S\left(w_b + \sum_{i=1}^{c} w_{a_i} e^{-\frac{(\mu_i - x)^2}{2\sigma^2}}\right)$$

where c is the capacity parameter of the predicate, $\mu_i \in [\mathit{min}, \mathit{max}]$ and σ are set as explained above, and $w_{a_1}, \ldots, w_{a_c}, w_b$ are trainable weights.

This is generalized to the n-D case as follows:

$$\mathcal{T}(\mathit{hasRightValue\_nD}) = S\left(w_b + \sum_{i_1=1}^{c} \cdots \sum_{i_n=1}^{c} w_{a_{i_1,\ldots,i_n}} e^{-\frac{|\mu_{i_1,\ldots,i_n} - x|^2}{2\sigma^2}}\right)$$

where $\mu_{i,j} \in [\mathit{min}_1, \mathit{max}_1] \times \cdots \times [\mathit{min}_n, \mathit{max}_n]$ and σ are set as explained above, x is an n-D vector, $|\cdot|$ stands for vector norm, and $w_{a_{1,\ldots,1}}, \ldots, w_{a_{c,\ldots,c}}, w_b$ are trainable weights.

Transformation of a categorical quantity aa-predicate. We base this aa-predicate on a multi-layer perceptron with one hidden layer, which has the number of units equal to the capacity parameter c of the aa-predicate and is activated by the ReLU activation function.

The transformation of an aa-predicate

$$\mathit{hasRightCategories\_nD}(x, m, c)$$

to the function $N_{\doteq}(x, w^h_{a_{1,1}}, \ldots, w^h_{a_{c,m}}, w^h_{b_1}, \ldots, w^h_{b_c}, w^o_{a_1}, \ldots, w^o_{a_c}, w^o_b)$ is defined as follows:

$$\mathcal{T}(\mathit{hasRightCategories\_nD}) = S\left(w^o_b + \sum_{i=1}^{c} w^o_{a_i} \mathrm{ReLU}\left(w^h_{b_i} + \sum_{j=1}^{n}\sum_{k=1}^{m} w^h_{a_{i,j,k}} \delta_{x_j,k}\right)\right)$$

where $x \in \{1, \ldots, m\}^n$ is the n-dimensional input vector of categorical values from the same domain of size m, c is the capacity, $w^h_{i,j,k}, w^h_b$ are trainable weights of the hidden layer, $w^o_{a_i}, w^o_b$ are trainable weights of the output layer, $\delta_{i,j}$ is the Kronecker delta—i.e., $\delta_{i,j} = 1$ if $i = j$ and $\delta_{i,j} = 0$ otherwise. The ReLU function is defined as $\mathrm{ReLU}(x) = \max(0, x)$. Note that the Kronecker delta in the formula stands for one-hot encoding of the categorical input values.

3.4 Training an *rsNN*

The N function we defined as the result of the transformations in Section 3.3 contains trainable weights. We train these weights using supervised learning and employing the traditional stochastic gradient descent optimization.

The samples for training are taken from existing logs obtained from the system runtime or a simulation. In the case of the motivation example, each sample contains the current time, the worker id, its position, and the history of events associated with the worker. To obtain accurate outputs for supervised learning, we exploit the fact that we have the original logical formula of the guard with static predicates available. Thus we use it as an oracle for generating the ground truth for training inputs. The exact training procedure is described in [1].

After this training step, the function N can be used as a drop-in replacement for the corresponding adaptation rule. Moreover, being a neural network, it is able to digest additional samples generated at runtime—e.g., to learn from situations when the outputs of the system have been manually corrected/overridden.

4 Evaluation

We evaluated our approach by comparing the training results of *rsNNs* created by the method proposed in Section 3 with generic NNs comprising one and two dense layers. The complete set of necessary code and data for replicating the evaluation, as well as the experiments, detailed evaluation of results, graphs, and discussion that did not fit this paper, is available in the replication package [1].

For our motivating example, we created two datasets: (a) *random* sampled dataset, which was obtained by randomly generating inputs and using the original logical formula of the guard as an oracle; (b) *combined* dataset, which combines data from a simulation and the random dataset. Both datasets have about $500,000$ data points.

The datasets were balanced in such a manner that half of the samples correspond to *true* and a half to the *false* evaluation of the guard of AccessToWorkplace. Additionally, to obtain more representative results for evaluation, the false cases were balanced so that each combination of the top-level conjunctions outcomes (i.e., duringShift & atWorkplaceGate & hasHeadGear) has the same probability.

The combined dataset combines false cases from random sampling and true cases from a simulation. The simulation was performed by a simulator we developed in the frame of an applied research project (Trust4.0[4]). The reason for combining these two sources is to get better coverage for all possible cases when the guard of the adaptation rule evaluates to *false*.

As the baseline generic NNs, we selected dense neural networks. Given our experiments and consultation with an expert outside our team (a researcher from another department who specializes in practical applications of neural networks), this architecture suits the problem at hand the best. Our setup comprises networks with one and two dense layers of 128 to 1024 nodes (in the case of two layers, both of them have the same amount of nodes). The dense layers use ReLU activation, and the final layer uses sigmoid. The greatest accuracy was observed when two 256-node dense layers were used; thus, this configuration was selected as the *baseline*.

Three versions of rsNNs representing our approach were built corresponding to different levels of refinement. The first two models refined only the time condition: one used the isAboveThreshold and isBelowThreshold variant (as in Listing 3) — denoted as "*time (A&B)*", the other used hasRightValue aa-predicate (similar to Listing 3 but with hasRightValue instead of the combination of isAboveThreshold and isBelowThreshold)—denoted as "*time (right)*". The last model refined all involved inputs (time, place, and headgear events) as outlined in Listing 4 — denoted as "*all*". To verify the properness of logical connectives redefinition (Section 3.3), we built a TensorFlow[5] model with no trainable weights (i.e., just rewriting the static predicates using their transformation described in Section 3.3). By setting $p = 10$, we achieved 100% accuracy (this value of p was then used in all other experiments).

	baseline	time (A&B)	time (right)	all
Accuracy (random)	99.159%	98.878%	99.999%	99.978%
Accuracy (combined)	99.393%	92.867%	99.993%	99.575%
number of weights	68,353	2	21	1,227

Table 1: Comparison of accuracies of individual methods

[4] `https://github.com/smartarch/trust4.0-demo`

[5] `https://www.tensorflow.org/` (version 2.4)

Table 1 presents the measured accuracies on the testing set[6] of both datasets (random and combined) after 100 training epochs, comparing *rsNNs* resulting from different refinements with the baseline. The last two models outperform the baseline in terms of accuracy. The *number of Weights* line refers to the number of trainable weights in each model. While the baseline has multiple weights (as it features two dense layers), our *rsNNs* have significantly fewer weights since their composition benefits from the domain knowledge ingrained in the adaptation rules.

The lower number of trainable parameters positively impacts the performance as it makes the models train and evaluates significantly faster whilst achieving comparable accuracy levels. We did not perform a thorough performance analysis since it heavily depends on many configuration parameters (e.g., batch size) and the actual hardware (especially whether CPU or GPU is used for the training). However, in our configurations, the proposed model was trained roughly several times (up to an order of magnitude) faster than the baseline.

5 Related Work

In the domain of adaptive systems, NNs and machine learning are used in several areas. Closely related approaches use NNs in the adaptation cycle analysis phase. Namely, in [16], neural networks are applied during the analysis and planning phase to reduce a large adaptation space. We apply *rsNN* during the same phases to refine adaptation rules, thus allowing for more flexible adaptation. Similarly, in [6], NNs are applied during the restriction of the adaptation space to achieve a meaningful system after adaptation.

In [12], NNs are used to forecast values of QoS parameters, thus allowing for the progressive selection of adaptation. A similar approach is used in [2] to predict values in sensor networks and proactively perform adaptation. Multiple machine learning algorithms, including NNs, are employed in [5] to predict QoS values again.

The approaches above target either reducing the adaptation space or adapting a system proactively. They differ from our approach as we use neural networks to relax strict conditions in an adaptive system and thus to learn new unforeseen conditions. A conceptually similar approach is [7], where machine learning approaches are utilized for training a model for rule-based adaptation. Instead of NNs, approaches like the random forest, gradient boosting regression models, and extreme boosting trees are used. Similarly, paper [3] proposes a proactive learner; however, the infrastructure is mainly discussed, and details about the used machine learning techniques are omitted. In [15], the authors propose an approach to dynamic learning of knowledge in self-adaptive and self-improving systems using supervised and reinforcement learning techniques. In [10], machine learning is used to deal with uncertainty in an adaptive system (namely in a

[6] We divide the data only to the training and testing set (testing set holds 10% of data). We do not need a validation set since we do not perform any hyper-parameter training.

cloud controller). Here, the proposed approach allows users to specify potentially imprecise control rules expressed with the help of fuzzy logic, and machine learning techniques are used to learn precise rules. The approach is the complete opposite of ours, where we start with precise rules and, via machine learning, we reach attunable ones. A similar approach is in [18], where reinforcement learning is also employed for generating and evolving the adaptation rules.

6 Conclusion

In this paper, we introduced the rule-specific Neural Network (*rsNN*) method that allows for transforming the guard of an adaptation rule into a custom neural network, the composition of which is driven by the structure of the logical predicates in the guard. An essential aspect of *rsNN* is that by having the ability to combine the original static predicates with attunable ones (and, in addition, to set the training capacity of the corresponding part of *rsNN* network), one can step-by-step proceed from a static non-trainable adaptation rule to fully trainable one. This aspect allows for a gradual transition from the original self-adaptive system to its trainable counterpart while still controlling the inherent uncertainty of introducing machine learning into the system.

The aspect of being able to control the uncertainty inherent to machine learning is a distinguishing factor of the *rsNN* method. This stems primarily from two facts: (1) The structure of the *rsNN* generated from an adaption rule directly relates to the composition of its predicates, and the static predicates can be combined with attunable ones. (2) An *rsNNs* is a neural network with almost two orders of magnitude fewer neurons than a generic neural network (e.g., a multi-layer perceptron network with several hidden dense layers) solving the same task. This makes the *rsNN* less prone to overfitting, which, in general, may lead to unexpected results in real environments. Moreover, given the significant difference in the number of neurons and thus trainable weights, rsNN networks train much faster, as showcased in the results of the experiments. In future work, we aim to extend the set of the predefined aa-predicates to provide a tool for applications also featuring other than metric and categorical quantities. Furthermore, we are looking into ways of supporting the process of gradual transformation of static predicates into attunable ones with the aim to make this process semi-automatic.

Acknowledgment

This work has been funded by the DFG (German Research Foundation) – project number 432576552, HE8596/1-1 (FluidTrust), supported by the Czech Science Foundation project 20-24814J, partially supported by Charles University institutional funding SVV 260588 and the KASTEL institutional funding, and partially supported by the Charles University Grant Agency project 408622.

References

1. Paper results replicaton package. https://github.com/smartarch/attuning-adaptation-rules-replication-package
2. Anaya, I.D.P., Simko, V., Bourcier, J., Plouzeau, N., Jézéquel, J.M.: A prediction-driven adaptation approach for self-adaptive sensor networks. In: Proc. of SEAMS 2014, Hyderabad, India (2014)
3. Bierzynski, K., Lutskov, P., Assmann, U.: Supporting the Self-Learning of Systems at the Network Edge with Microservices. In: Smart Systems Integration; 13th Int. Conf. and Exhibition on Integration Issues of Miniaturized Systems (2019)
4. Bureš, T., Gerostathopoulos, I., Hnětynka, P., Pacovský, J.: Forming Ensembles at Runtime: A Machine Learning Approach. In: Proc. of ISOLA 2020, Rhodes, Greece (2020)
5. Chen, T., Bahsoon, R.: Self-Adaptive and Online QoS Modeling for Cloud-Based Software Services. IEEE Trans. on Software Engineering **43**(5) (2017)
6. Gabor, T., Sedlmeier, A., Phan, T., Ritz, F., Kiermeier, M., Belzner, L., Kempter, B., Klein, C., Sauer, H., Schmid, R., Wieghardt, J., Zeller, M., Linnhoff-Popien, C.: The scenario coevolution paradigm: adaptive quality assurance for adaptive systems. Int. Journ. on Software Tools for Technology Transfer **22**(4) (2020)
7. Ghahremani, S., Adriano, C.M., Giese, H.: Training Prediction Models for Rule-Based Self-Adaptive Systems. In: Proc. of ICAC 2018, Trento, Italy (2018)
8. Goodfellow, I., Bengio, Y., Courville, A.: Deep Learning. MIT Press (2016), http://www.deeplearningbook.org
9. Hochreiter, S., Bengio, Y., Frasconi, P., Schmidhuber, J.: Gradient flow in recurrent nets: the difficulty of learning long-term dependencies. In: A field guide to dynamical recurrent neural networks. IEEE Press (2001)
10. Jamshidi, P., Pahl, C., Mendonça, N.C.: Managing Uncertainty in Autonomic Cloud Elasticity Controllers. IEEE Cloud Computing **3**(3) (2016)
11. Mańdziuk, J., Macukow, B.: A Neural Network Performing Boolean Logic Operations. Optical Memory and Neural Networks **2**(1), 17–35 (1993)
12. Muccini, H., Vaidhyanathan, K.: A Machine Learning-Driven Approach for Proactive Decision Making in Adaptive Architectures. In: Companion Proc. of ICSA 2019, Hamburg, Germany (2019)
13. Salehie, M., Tahvildari, L.: Self-Adaptive Software: Landscape and Research Challenges. ACM Trans. on Autonomous and Adaptive Systems **4**(2) (2009)
14. Schwenker, F., Kestler, H.A., Palm, G.: Three learning phases for radial-basis-function networks. Neural Networks **14**(4) (2001)
15. Stein, A., Tomforde, S., Diaconescu, A., Hähner, J., Müller-Schloer, C.: A Concept for Proactive Knowledge Construction in Self-Learning Autonomous Systems. In: Proc. of FAS*W 2018, Trento, Italy (2018)
16. Van Der Donckt, J., Weyns, D., Quin, F., Van Der Donckt, J., Michiels, S.: Applying deep learning to reduce large adaptation spaces of self-adaptive systems with multiple types of goals. In: Proc. of SEAMS 2020, Seoul, Korea. ACM (2020)
17. Weyns, D., et al.: Towards Better Adaptive Systems by Combining MAPE, Control Theory, and Machine Learning. In: Proc. of SEAMS 2021, Madrid, Spain (2021)
18. Zhao, T., Zhang, W., Zhao, H., Jin, Z.: A Reinforcement Learning-Based Framework for the Generation and Evolution of Adaptation Rules. In: Proc. of ICAC 2017, Columbus, OH, USA (2017)