Auton Robot (2008) 24: 49–67
DOI 10.1007/s10514-007-9060-9

Fault detection in autonomous robots based on fault injection and learning

Anders Lyhne Christensen · Rehan O'Grady · Mauro Birattari · Marco Dorigo

Received: 23 January 2007 / Accepted: 27 September 2007 / Published online: 17 November 2007

Abstract In this paper, we study a new approach to fault detection for autonomous robots. Our hypothesis is that hardware faults change the flow of sensory data and the actions performed by the control program. By detecting these changes, the presence of faults can be inferred. In order to test our hypothesis, we collect data from three different tasks performed by real robots. During a number of training runs, we record sensory data from the robots while they are operating normally and after a fault has been injected. We use back-propagation neural networks to synthesize fault detection components based on the data collected in the training runs. We evaluate the performance of the trained fault detectors in terms of number of false positives and time it takes to detect a fault. The results show that good fault detectors can be obtained. We extend the set of possible faults and go on to show that a single fault detector can be trained to detect several faults in both a robot's sensors and actuators. We show that fault detectors can be synthesized that are robust to variations in the task, and we show how a fault detector can be trained to allow one robot to detect faults that occur in another robot.

Keywords Fault detection · Fault injection · Learning · Model-free · Mobile robots

A.L. Christensen (✉) · R. O'Grady · M. Birattari · M. Dorigo
IRIDIA, CoDE, Université Libre de Bruxelles,
50, Av. Franklin Roosevelt, CP 194/6, 1050 Brussels, Belgium
e-mail: alyhne@iridia.ulb.ac.be

R. O'Grady
e-mail: rogrady@ulb.ac.be

M. Birattari
e-mail: mbiro@ulb.ac.be

M. Dorigo
e-mail: mdorigo@ulb.ac.be

1 Introduction

As more and more robots are introduced in space, industry, and private homes, fault detection is becoming an increasingly important issue to address. When a robot stops exhibiting its intended behavior, either due to an internal fault or to external factors, it can become a costly and/or a dangerous affair. The problem is often exacerbated if the fault is not detected in a timely manner. In a recent paper (Carlson and Murphy 2003), the reliability of seven mobile robots from three different manufacturers was tracked over a period of two years and the average mean time between failures was found to be 8 hours. The result suggests that faults in mobile robots are quite frequent.

Technically, a fault is an unexpected change in system function which hampers or disturbs normal operation, causing unacceptable deterioration in performance (Isermann and Ballé 1997). A *fault tolerant* system is capable of continued operation, possibly at a degraded performance, in the event of faults in some of its parts. Fault tolerance is a sought-after property for critical systems due to economic and/or safety concerns. What we study in this paper is the activity known as *fault detection* in autonomous robots. Fault detection is a binary decision process confirming whether or not a fault has occurred in a system. Other aspects of fault tolerance include *fault diagnosis*, namely determining the type and location of faults, and *protection* which comprises any steps necessary to ensure continued safe operation of the system (Isermann and Ballé 1997).

Fault detection can be achieved by adding special-purpose hardware such as torque and position sensors (Terra and Tinos 2001). Adding additional hardware increases cost and complexity, and it is, therefore, something that we would like to avoid in many cases. Reducing cost and complexity would, for example, be crucial in projects such as the

National Aeronautics and Space Administration's (NASA) swarm missions, in which cooperating swarms of hundred to thousands of small-scale autonomous robots explore the solar system (Hinchey et al. 2004). Given the high number of robots, simplicity and small size are high priorities. Similarly, for domestic adoption of service and leisure robots, the number and complexity of components have to be kept low in order to reach a price point that allows for high market penetration (Kochan 2005).

In this study, we propose a method for performing fault detection for autonomous robots. Our method requires no special fault detection hardware and relatively little computational resource to run the fault detection software. It relies on recording sensory data, firstly over a period of time when a robot is operating as intended, and secondly over a period of time when various types of hardware faults are present. Using knowledge of how the flow of information changes after a fault has occurred, we are able to detect faults.

2 Related work

Fault detection is based on observations of a system's behavior (for an introduction see Isermann 1997). Deviations from normal behavior can be interpreted as symptoms of a fault in the system. A specific fault detection approach is a concrete method for *observation processing*. A large body of research in *model-based fault detection* approaches exists (Gertler 1988; Isermann and Ballé 1997). In model-based fault detection some model of the system or of how it is supposed to behave is constructed. The actual behavior is then compared to the predicted behavior and deviations can be interpreted as symptoms of faults. A deviation is called a *residual*, that is, the difference between the predicted and the observed value. In the domain of mobile autonomous robots, accurate analytical models are often not feasible due to uncertainties in the environments, noisy sensors, and imperfect actuators. A number of methods have been studied to deal with these uncertainties. Artificial neural networks and radial basis function networks have been used for fault detection and diagnosis based on residuals (Vemuri and Polycarpou 1997; Terra and Tinos 2001; Patton et al. 2000). In Skoundrianos and Tzafestas (2004), for instance, the authors train multiple local model neural networks to capture the input-output relationship for the components in a robot for which fault should be detected. The authors focus on detecting faults in the wheels of a robot, and the input and the output are the voltage to the motor driving a wheel and the speed of the wheel, respectively. Supervised learning is used to train the local model neural networks. During operation, the speeds predicted by the local models are compared to the actual speed and the residuals are computed.

Another popular approach to fault detection is to operate with multiple global models of the system concurrently. Each model corresponds to the behavior of the system in a given fault state, for example a broken motor, a flat tire, and so on. The fault corresponding to a particular model is considered to be detected when that model's predictions are a sufficiently close match to the currently observed behavior. Banks of Kalman filters have been used for such state estimation (Roumeliotis et al. 1998; Goel et al. 2000). In their basic form, Kalman filters are based on the assumption that the modeled system can be approximated as a Markov chain built on linear operators perturbed by Gaussian noise (Kalman 1960). Robotics systems are, like many other real-world systems, inherently nonlinear. Furthermore, discrete fault state changes can result in discontinuities in behavior. Extensions, such as the *extended Kalman filter* (EKF) and the *unscented Kalman filter* (UKF), overcome some of these limitations. In EKFs the state transitions and the models can be non-linear functions, but they must be differentiable so that the Jacobian matrix can be computed. In UKFs, a few sample points are picked and propagated through the model allowing the mean and covariance to be estimated even for models comprised of highly non-linear functions (Julier and Uhlmann 1997). EKFs and UKFs have been extensively used for localization for mobile robots (Smith and Cheeseman 1986; Leonard and Durrant-Whyte 1991; Ashokaraj et al. 2004), but in the domain of fault detection and fault diagnosis for autonomous robots these techniques are often used in combination with other methods.

Dynamic Bayesian networks represent another technique that does not require that the underlying phenomenon can be reasonably modeled as a linear system (Lerner et al. 2000). Recently, computationally efficient approaches for approximating Bayesian belief using *particle filters* have been studied as a means for fault detection and identification (Dearden et al. 2004; Verma et al. 2004; Li and Kadirkamanathan 2001). Particle filters are Monte Carlo methods capable of tracking hybrid state spaces of continuous noisy sensor data and discrete operation states. The key idea is to approximate the probability density function over fault states given the observed data by a swarm of points or particles. One of the main issues related to particle filters is tracking multiple low-probability events (faults) simultaneously. A scalable solution to this issue has recently been proposed (Verma and Simmons 2006).

Artificial immune-systems (AIS) are a biologically inspired approach to fault detection. An AIS is a classifier that distinguishes between *self* and *non-self* (Forrest et al. 1994). In fault detection, "self" corresponds to fault-free operation while "non-self" refers to observations resulting from a faulty behavior. AIS have been applied to robotics, see for

example (Canham et al. 2003) in which fault detectors are obtained for a Khepera robot and for a control module of a BAE Systems Rascal robot. The two systems are first trained during fault-free operation and their capacity to detect faults is then tested.

Marsland et al. have suggested using a *novelty filter* based on a clustering neural network and *habituation*, for inspection and fault detection (Marsland et al. 2005). Through unsupervised learning a novelty filter learns to ignore sensory data similar to data previously perceived. The authors evaluated the approach in various configurations using a Nomad 200 robot placed in various environments and the robot correctly detected environmental differences.

The approach that we propose in this study relies on artificial neural networks that are trained to discriminate between behaviors when a robot is operating normally and behaviors when the presence of a fault is affecting the robot's performance. We rely on a single neural network, as opposed to the multiple local model neural networks and the multi-model approaches mentioned above. We do not use explicit or analytical modeling of the system (which can be complicated for all but the simplest systems). In contrast with the studies on AIS and novelty detection, we use supervised learning. When an artificial neural network is trained, we include training data with both positive and negative examples, which potentially allows the proposed method to be extended to fault diagnosis.

3 Methodology

Some methods for fault detection base classification on the most recent observations only. The approach presented in this study allows classification based on both current and past observations, since many faults can only be detected if a system is observed over time. This is especially true for mechanical faults in mobile robots; a fault causing a wheel to block, for instance, can only be detected once the robot has tried to move the wheel for a period of time long enough for the absence of movement to be detectable. This period of time could be anywhere from a few milliseconds if, for example, dedicated torque sensors in the wheels are used, to several seconds if the presence of a fault has to be inferred based on information from non-dedicated and imprecise sensors.

We assume that the correct behavior for a robot has been specified in the form of a control program that directs the robot to carry out its intended task. The fault detection problem is to determine if the robot performs this task correctly, or if some fault in the hardware or in a software sub-system (but not in the control program itself) is degrading the robot's performance. If a fault is detected, a signal can be sent

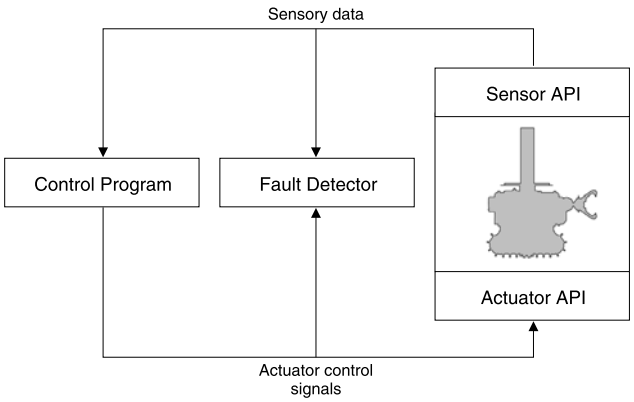


Fig. 1 The fault detection module monitors the sensory data read by the control program and the consequent control signals sent to the actuators. The fault detection module is passive and does not interact with the robot hardware or the control program

to the control program itself, another robot, or a human operator. In our design, the fault detector is an isolated software component that passively monitors the performance of the robot through the information that flows in and out of the control program. A conceptual illustration of the relationship between the control program, the robotic platform, and the fault detection module can be seen in Fig. 1. The control program is run as a succession of sense-think-act cycles. In each cycle, the control program reads data from sensors such as the on-board camera, infrared proximity sensors, light sensors, and so on, processes the data, and sends control signals to the actuators such as the motors that drive the robot. A control cycle period is typically between 0.10 s and 0.15 s, depending on the task and the amount of computation needed to extract the relevant information from the sensory data.

We take a black-box approach and consider only the inputs and the outputs of the control program, that is, the robot's flow of sensory data into the control program and the resulting flow of control signals sent from the control program to the actuators. Our hypothesis is that this information alone is sufficient to discriminate between situations in which the robot operates as intended and situations in which the presence of one or more faults hampers its performance. We record the flow of sensory data and control signals in situations where a robot is operating normally and when the robot is in a fault state.

There are several methods for obtaining flows of sensory data and control signals for robots with faults: A broken robot can be used, readings can be obtained using a detailed software simulator, or faults can be purposefully provoked by the experimenter. In this study, we apply the latter technique: in a modified version of the low-level on-board software, we provoke (simulated) hardware faults on real robots. We apply a well-establish technique known as *software implemented fault injection* (SWIFI) used in depend-

able systems research. The technique is usually applied to measure the robustness and fault tolerance of software systems (Hsueh et al. 1997; Arlat et al. 1990). In our case, we inject faults to discover how sensory data and the control signals change when faults are present. The idea is that by actively controlling the state of the robot (for instance by injecting faults or by using a broken robot) and recording the flow of sensory data and control signals, we can use supervised learning techniques and obtain a classifier that, based on that flow, can determine the state of the system.

We use time-delay neural networks (TDNNs) as classifiers (Waibel et al. 1989; Clouse et al. 1997). TDNNs are feed-forward networks that allow reasoning based on time-varying inputs without the use of recurrent connections. In a TDNN, the values for a group of neurons are assigned based on a set of observations from a fixed distance into the past. The TDNNs used in this study are normal multilayer perceptrons for which the inputs are taken from multiple, equally spaced points in a delay-line of past observations. TDNNs have been extensively used for time-series prediction due to their ability to make predictions based on data distributed in time. Unlike more elaborate, recurrent network architectures, the properties of multilayer TDNNs are well-understood and supervised learning through back-propagation can be applied.

3.1 Formal definitions

Our aim is to obtain a function that maps from a set of current and past sensory data, S, and control signals, A, to either 0 or 1 corresponding to *no-fault* and *fault*, respectively:

$$\chi : S, A \rightarrow \{0, 1\}. \tag{1}$$

We assume that such a function exists and we approximate it with a feed-forward neural network. We let $I \subseteq (S \cup A)$ be the inputs to the network. We choose a network that has a single output neuron whose output value is in the interval $[0, 1]$. The output value is interpreted in a task-dependent way. For instance, a threshold-based classification scheme can be applied where an output value above a given threshold is interpreted as 1 (*fault*), whereas an output value below the threshold is interpreted as 0 (*no-fault*).

Sensory data, control signals and fault state We perform a number of runs each consisting of a number of control cycles (sense-compute-act loops). For each control cycle, c, we record the sensory data and control signals to and from the control program. We let i_c^r denote a single set of control program inputs and outputs (CPIO), that is, the CPIO for control cycle c in run r. We let s denote the number of values in a single CPIO set, that is $s = |i_c^r|$. We let I^r be the ordered set of all CPIO sets for r. Similarly, for each control cycle we let f_c^r denote the fault state for control cycle c in run r, where $f_c^r = 1$ if a fault has been injected and 0 otherwise. Hence, $f_c^r = 0$ when the robot is operating normally and $f_c^r = 1$ otherwise.

Tapped delay-line and input group distance The CPIO sets are stored in a tapped delay-line, where each tap has size s. The input layer of a TDNN is logically organized in a number of *input groups* $g_0, g_1, \ldots, g_{n-1}$ and each group consists of precisely s neurons, that is, one neuron for each value in a CPIO set. The activation of the input neurons in group g_t are then set according to $g_t = i_{c-t \cdot d}^r$, where c is the current control cycle and d is the *input group distance*. See Fig. 2 for an example. If we choose an input group distance $d = 1$, for example, the TDNN has access to the current and the $n - 1$ most recent CPIOs, whereas if $d = 2$, the TDNN has access to the current and every other CPIO set up to $2(n - 1)$ control cycles into the past, and so on. In this way, the input group distance specifies the relative distance in time between the individual groups and (along with the number of groups) how far into the past a TDNN "sees".

TDNN structure and activation function The input layer of the TDNN is fully connected to a hidden layer, which is again fully connected to the output layer. The output layer consists of a single neuron whose value reflects the network's classification of the current inputs. The activations of the neurons are computed layer-by-layer in a feed-forward manner and the following sigmoid activation function is used to compute the neurons' outputs in the hidden and the output layers:

$$f(a) = \frac{1}{1 + e^{-a}}, \tag{2}$$

where a is the activation of the neuron.

Classification and learning The output of the TDNN has a value between 0 and 1. The error factor used in the back-propagation algorithm is computed as the difference between the fault state f_c^r and the output o_c:

$$E_c = f_c^r - o_c. \tag{3}$$

The neural networks are all trained by a standard batch learning back-propagation algorithm to minimize the absolute value of the error factor E_c in (3) (Rumelhart et al. 1986).

In summary, sensor and actuator data is collected from a number of runs on real robots and different types of faults are injected. A TDNN is trained to discriminate between normal and faulty operation. By storing past observations in a tapped delay-line, classification based on how the flow of information changes over time is performed.

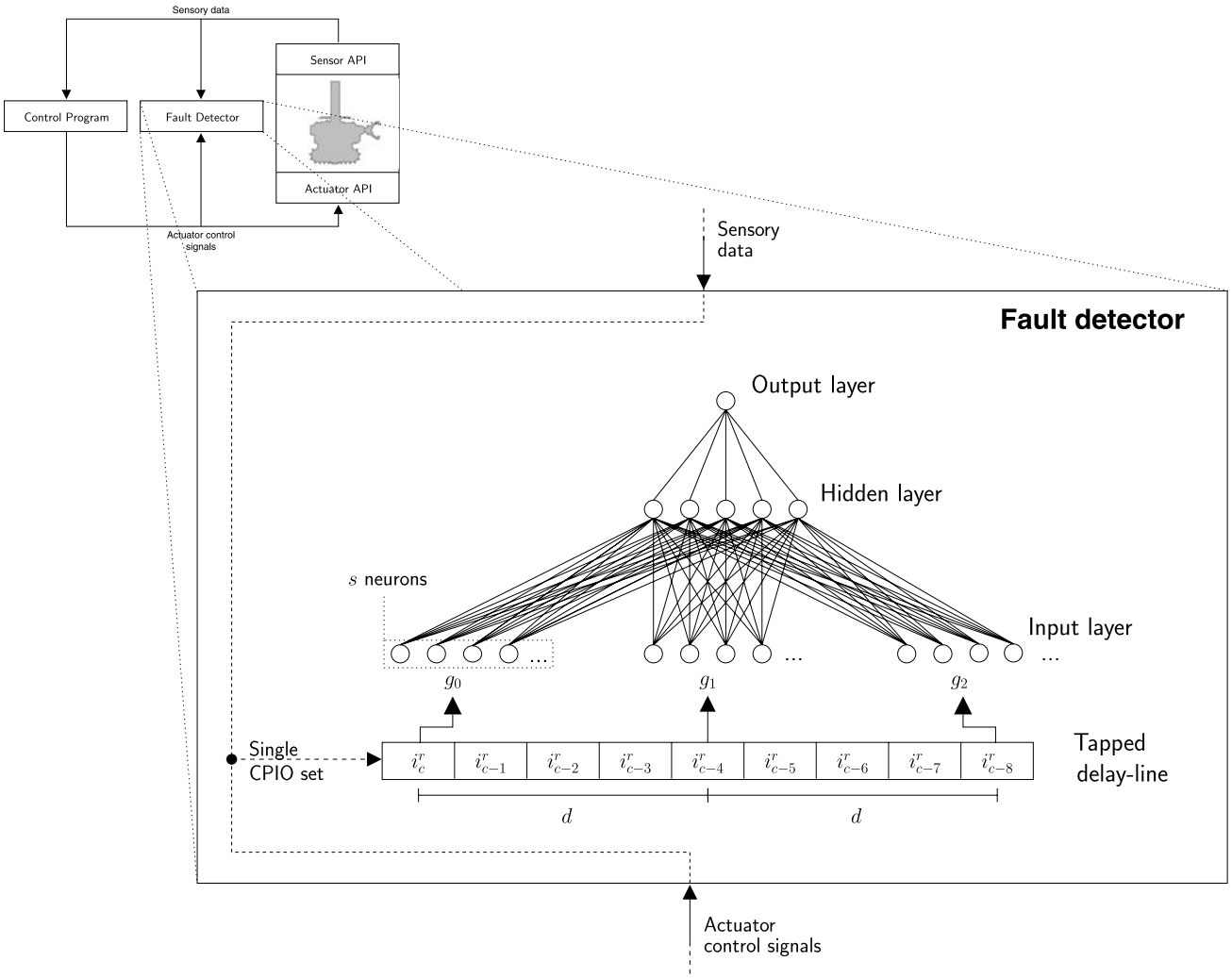


Fig. 2 An illustration of a fault detection module based on a TDNN. The current control program input and output (CPIO) is stored in the tapped delay-line and the activations of the neurons in the logical input groups are set according to the current and past CPIOs. In the example illustrated, there are 3 input groups and the input group distance d is 4

3.2 Robot hardware

We use a number of real robots known as *s-bots* (Mondada et al. 2005). The *s-bot* platform has been used for several studies, mainly in swarm intelligence and collective robotics (Dorigo et al. 2004; Trianni and Dorigo 2006; Nouyan et al. 2008). Overcoming steep hills and transport of heavy objects are notable examples of tasks which a single robot could not solve individually, but which have been solved successfully by teams of collaborating robots (Groß et al. 2006; O'Grady et al. 2005; Nouyan et al. 2006).

Each *s-bot* is equipped with an Xscale CPU running at 400 MHz, a number of sensors including an omni-directional camera, light and proximity sensors, as well as a number of actuators including a ring of 8×3 (RGB) colored leds, and a gripper that allows robots to attach to each other. The sensors and actuators are indicated in Fig. 3.

4 Faults

Faults in the mechanical system that propels the robot can be hard to detect when no special hardware to facilitate fault detection is used. Unlike faults in sensors, which are usually immediately observable due to inconsistencies or abrupt changes in the sensory values, faults in the mechanical system have to be inferred from the unexpected consequences (due to the presence of faults) of the actions performed by the robot. There is often a latency associated with the detection of faults in the mechanical systems of a robot because the consequences of the robot's actions need to become apparent before the presence of a fault can be inferred.

In this study, we focus principally on faults in the mechanical system that propels the *s-bots*. This system consists of a set of differential *treels*, that is, combined tracks and wheels (Mondada et al. 2004). Given that the treels contain

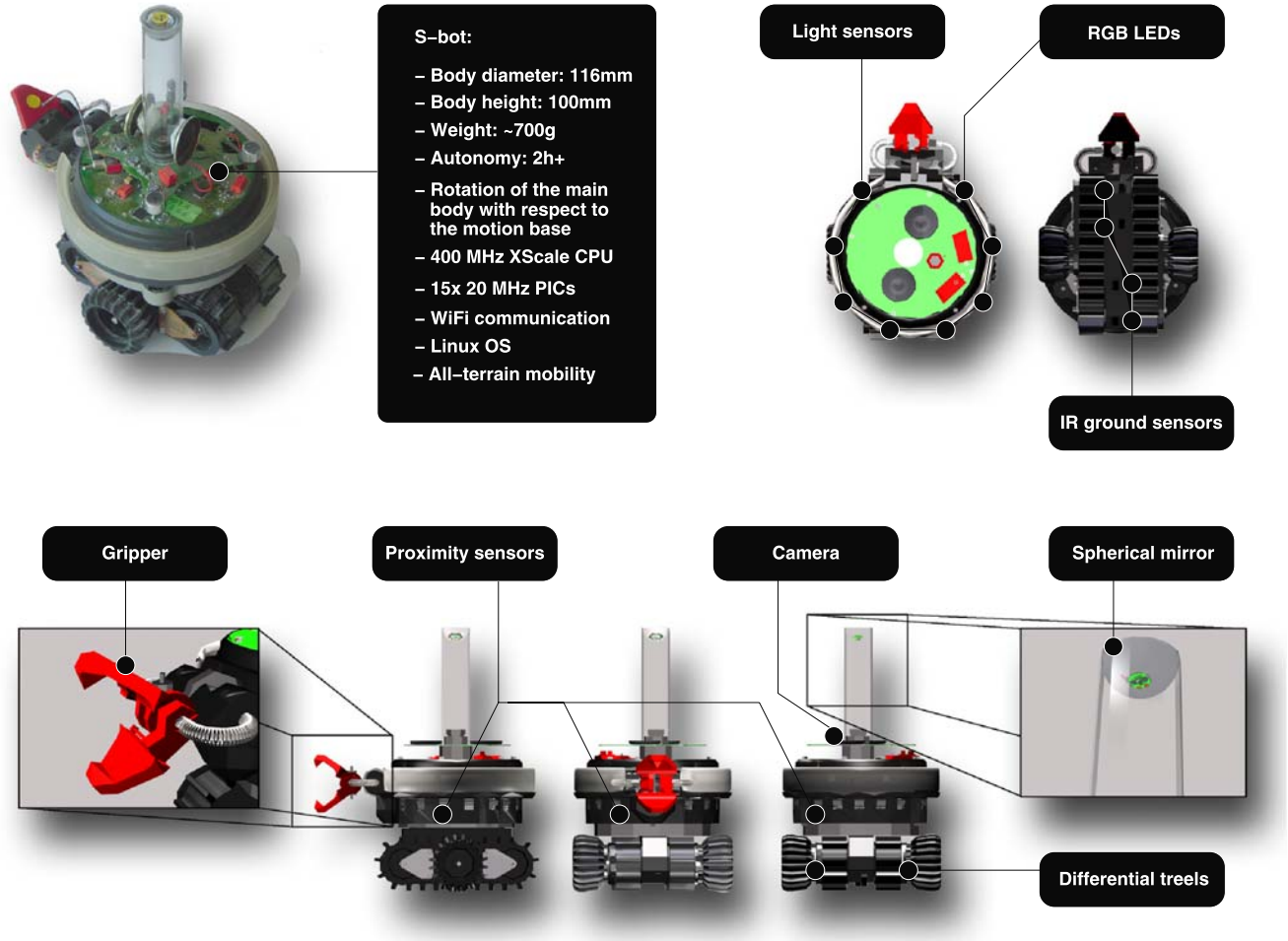


Fig. 3 The *s-bot* platform, sensors, and actuators

moving parts and that they are used continuously in most experiments, they are the components in which the majority of faults occur.

We analyze two types of faults. Both types can either be isolated to the left or the right treel or they can affect both treels simultaneously. The first type of fault causes one or both treels to stop moving. This usually happens if the strap that transfers power from the electrical motors to the treels breaks or jumps out of place. Whenever this happens, the treels stop moving. We denote this type of fault as *stuck-at-zero*. The second type of fault occurs when an *s-bot*'s software sub-system crashes leaving one (or both) motor(s) driving the treels running at some undefined speed. The result is that a treel no longer can be controlled by the on-board software. We refer to this type of fault as *stuck-at-constant*.

To collect training data, a number of runs are conducted. In each run the *s-bot* starts in a nominal state and during the run, a fault is injected. The fault is implemented in the on-board software by discarding the control program's commands to the failed part and by substituting them according to the type of fault injected. If, for instance, a *stuck-at-constant* fault is injected in the left treel, the speed of that treel is set to a random value, and all future changes in speed requested by the control program are discarded.

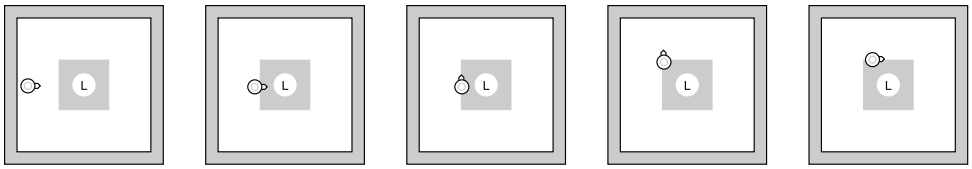
5 The three experimental setups

We have chosen three setups in which to study fault detection based on fault injection and learning. The setups are called *find perimeter*, *follow the leader*, and *connect to s-bot*, respectively, and they are described in Fig. 4. In all setups, we use a 180×180 cm^2 arena surrounded by walls.

In the *find perimeter* setup an *s-bot* follows the perimeter of a dark square drawn on the arena surface. In this setup, the four infrared ground sensors are used to discriminate between the normal light-colored arena surface and the dark square.

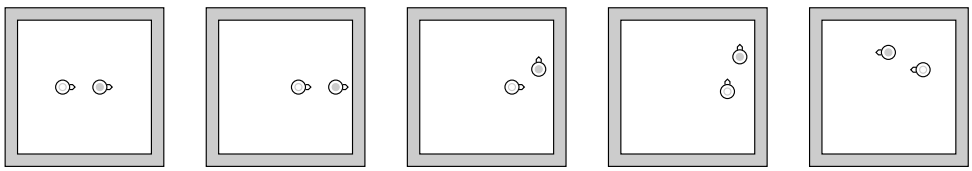
In the *follow the leader* setup, an *s-bot* (the *leader*) performs a random walk in the environment and another *s-bot* (the *follower*) follows. The two robots perceive one another

Find perimeter: An *s-bot* is situated in an arena with a dark colored square drawn on an otherwise light floor. A light source is placed in the center of the square. The task for the *s-bot* is to follow the perimeter of the square.



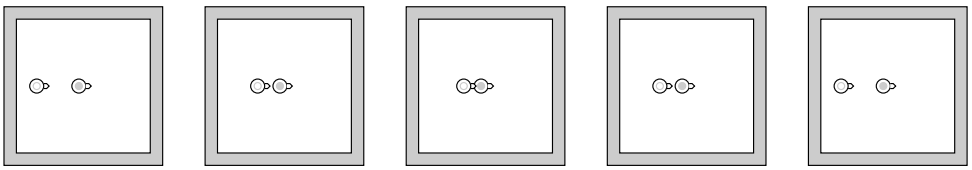
Sensors: IR ground (4 inputs), light (8 inputs)
Control period: 100 ms

Follow the leader: Two *s-bots* are placed in a square environment bounded by walls. One of the *s-bots* has been preassigned the *leader* role, while the other has been preassigned the *follower* role. The *leader* moves around in the environment. The *follower* tails the *leader* and tries to stay at a distance of 35 cm. If the *follower* falls behind, the *leader* waits. Faults are injected in the *follower* only.



Sensors: Camera (16 inputs), IR proximity (15 inputs)
Control period: 150 ms

Connect to s-bot: One *s-bot* attempts to physically connect to a stationary *s-bot* using its gripper. When a successful connection has been made, the *s-bot* waits for 10 seconds, disconnects, moves back, and tries to connect again. Faults are injected in the connecting *s-bot* only.



Sensors: Camera (16 inputs), optical sensors in gripper (4 inputs)
Control period: 150 ms

Fig. 4 Description of the three setups: *find perimeter*, *follow the leader*, and *connect to s-bot*. For each setup a list of sensors used and the control cycle period for the controllers are shown. The number in brackets after each sensor listed corresponds to the number of input values the sensor provides to the fault detector at each control cycle

using their omni-directional cameras. The infrared proximity sensors are used to detect and avoid walls. Objects up to 50 cm away can be seen reliably through the camera. Infrared proximity sensors have a range from a few centimeters up to 20 cm depending on the reflective properties of the obstacle or object. Faults are injected in the *follower* only.

In the *connect to s-bot* setup, one *s-bot* tries to connect to another, stationary *s-bot*. The connection is made using the gripper. The connecting *s-bot* uses the camera to perceive

the location of the other robot. Faults are only injected in the *s-bot* that is trying to form the connection.

Readings from sensors such as infrared ground sensors are straightforward to normalize and feed to the input neurons of a neural network. The camera sensor, in contrast, captures 640 × 480 color images. For these more complex sensor readings to serve as input to a neural network, relevant information must be extracted and processed beforehand. The *s-bots* have sufficient on-board processing power to scan the entire images and identify objects based on color information. The image processor is configured to detect the location of colored leds of the *s-bots* only, and discard any other information. The *s-bot* camera captures images of the robot's surroundings reflected in a semi-spherical mirror. Since the robots operate on flat terrain, the distance in pixels from the center of an image to a perceived object corresponds to the physical distance between the robot and the object. In order to make this information available to a neural network, we divide the image into 16 non-overlapping slices of equal size in terms of the field of view they cover. Each slice corresponds to a single input value. The value is set depending on the distance to the closest object perceived in the slice. If no object is perceived, the value for a slice is 0. Used in this way, the camera sensor effectively becomes a range sensor for colored leds.

6 Data collection, training and performance evaluation

6.1 Data collection

A total of 60 runs on real *s-bots* are performed for each of the three setups. In each run, the robot(s) start in a nominal state, and at some point during the run a fault is injected. The fault is injected at a random point in time after the first 5 seconds of the run and before the last 5 seconds of the run according to a uniform distribution. Hence, a robot spends on average 50% of the time that a run lasts in a nominal state. When a fault is injected, there is a 50% probability that a fault affects both treels instead of only one of the treels, and faults of the type *stuck-at-zero* and *stuck-at-constant* are equally likely to occur. Each run consists of 1000 control cycles and for each control cycle the sensory data, control signals, and the current fault state are recorded. In the *find perimeter* setup 1000 cycles correspond to 100 seconds, while for the *follow the leader* and in the *connect to s-bot* setups 1000 cycles correspond to 150 seconds, due to the longer control cycle period.

6.2 Training and evaluation data

The data sets recorded in each setup are partitioned into two subsets, one consisting of data from 40 runs, which is used for training; and one consisting of the data from the remaining 20 runs, which is used for a final performance evaluation. The TDNNs all have a hidden layer of 5 neurons and an input layer with 10 input groups.

6.3 Performance evaluation

The performance of the trained neural networks is computed based on the 20 runs in each setup reserved for evaluation. A network is evaluated on data from one run at a time, and the output of the network is recorded and compared to the fault state.

The two main performance criteria for a fault detection are reliability of detection and speed of detection. In our approach, the interpretation of the output of the trained neural network has an important impact on both criteria. The simplest interpretation mechanism is to define a threshold. An output value above this threshold is considered an indication that a fault is present, whereas an output value below the threshold is considered an indication that the robot is in a nominal state. In the next section, we present results for five such thresholds: 0.10, 0.25, 0.50, 0.75, and 0.90.

A graphical representation of TDNN's output during an evaluation run is shown in Fig. 5. In the run shown, a fault was injected at control cycle 529. The number of false positives is the number of control cycles before a fault is injected for which the output of the TDNN exceeds the given threshold. Choosing a threshold of 0.50 would, for example, result in one false positive since the output of the network is higher than 0.50 for one control cycle (cycle number 304) *before* the fault was injected. If we choose a higher threshold, either 0.75 or 0.90, false positives are avoided. However, choosing a higher threshold has a negative impact on another aspect of a fault detector's performance, namely its *latency*. Latency is the number of cycles between the occurrence and detection of a fault. In the example in Fig. 5 the fault is detected at control cycle 553, 561, 570, 572, and 574 for the thresholds 0.10, 0.25, 0.50, 0.75, and 0.90, yielding latencies of 24, 32, 41, 43, 45 control cycles, respectively.[1]

For some tasks, the recovery procedure is costly, and fewer false positives might be desirable even at the cost of a higher latency. For other tasks, undetected faults can have serious consequences and a low latency is more important than reducing the number of false positives. We can gain fine control over the balance between latency and number of false positives by choosing an appropriate threshold.

[1] It is important to note that a latency of 24 control cycles may seem long, but the faults that we are trying to detect do not always have an immediate impact on the performance of a robot. If, for instance, the fault injected causes a treel to block (a fault of the type *stuck-at-zero*), the fault can only be detected if the control program tries to set the treel to a non-zero speed. In particular, if the control program is setting low speeds (values close to zero) it might take a long time before a fault can be detected.

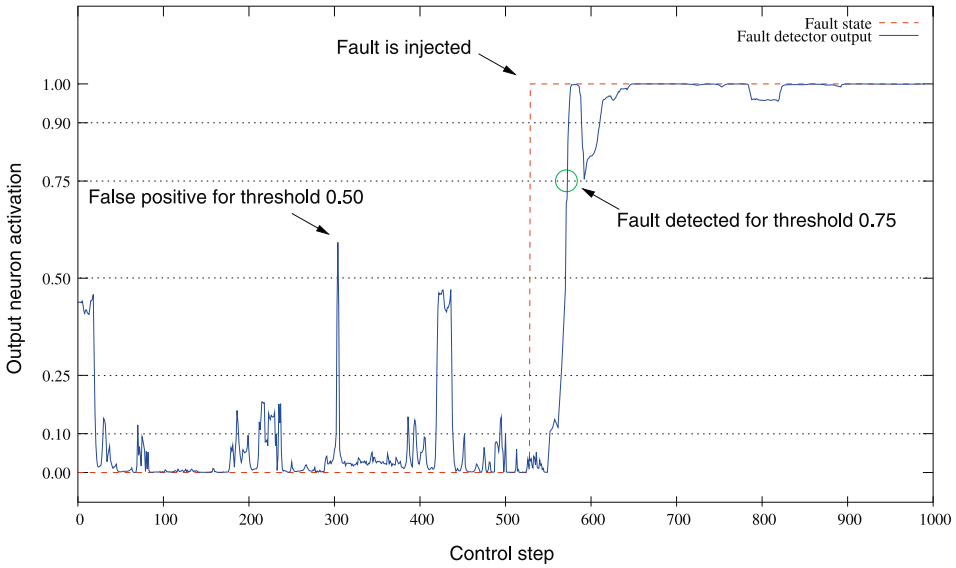


Fig. 5 An example of the output of a trained TDNN during a run. The dotted line shows the optimal output. At control cycle 529 a fault is injected. Five different thresholds are indicated, 0.10, 0.25, 0.50, 0.75, and 0.90, and a false positive for threshold 0.50 is shown at control cycle 304 (the output has a value greater than 0.50 *before* the fault was injected at control cycle 529). The latency for a threshold is the number of control cycles from the moment the fault is injected till the moment the output value of the TDNN becomes greater than the threshold. In the example above, the latency for threshold 0.75 is 43 control cycles because the output of the TDNN reaches 0.75 only at control cycle 562, that is, 43 control cycles after the fault was injected

7 Results

We first evaluate the effect of the input group distance on the performance of a fault detector with respect to its latency and number of false positives (Sect. 7.1). The input group distance determines how far into the past a fault detector "sees". We then evaluate the performance of the fault detectors in the *follow the leader* and *connect to s-bot* setups (Sect. 7.2). In some situations false positives can be costly and we show how the output of a TDNN can be reinterpreted to avoid nearly all false positives (Sect. 7.3). We test if the proposed method is applicable when faults in both sensors and actuators are considered (Sect. 7.4). We show that it is possible to obtain a fault detector that can generalize if the task varies between runs (Sect. 7.5). Finally, we demonstrate that fault detection through fault injection and learning can be extended to *exogenous fault detection*, that is, the capacity for a robot to detect faults in another, physically separate robot.

7.1 Tuning the input group distance

To find an input group distance that performs well, we trained fault detectors with input group distances ranging from 1 to 10. Fig. 6 and Fig. 7 show respectively a box-plot of the latencies[2] and a box-plot of the number of false positives observed during 20 evaluation runs in the *find perimeter* setup. Results are shown for the five thresholds: 0.10, 0.25, 0.50, 0.75, and 0.90. The median latencies and number of false positives for each configuration of input group distance and threshold are summarized in Table 1 and Table 2, respectively.

The latency results in Fig. 6 show no clear correlation between latency and input group distance. The false positive results in Fig. 7, however, show that for low input group distances, 1 and 2 in particular, the fault detector in general detects a large number of false positives. No clear trend is observed regarding the number of false positives for fault detectors with input group distances above 4.

An input group distance of 1 means that the TDNN is provided with data from the past 10 control cycles (because there are 10 input groups). 10 control cycles are equal to 1 second in the *find perimeter* setup. Similarly, an input group distance of 2 means that the TDNN is provided with data from the past 2 seconds, but only from every other control cycle. The false positive results indicate that data from a period longer than 2 seconds (i.e., an input group distance higher than 2) is needed for accurate classification.

The results in Fig. 6 and Fig. 7 show that the performance of the fault detectors, both in terms of latency and in terms

[2]For the latency results, we only include data from runs in which the fault was detected. See Table 3 for the number of undetected faults for different input group distances and thresholds.

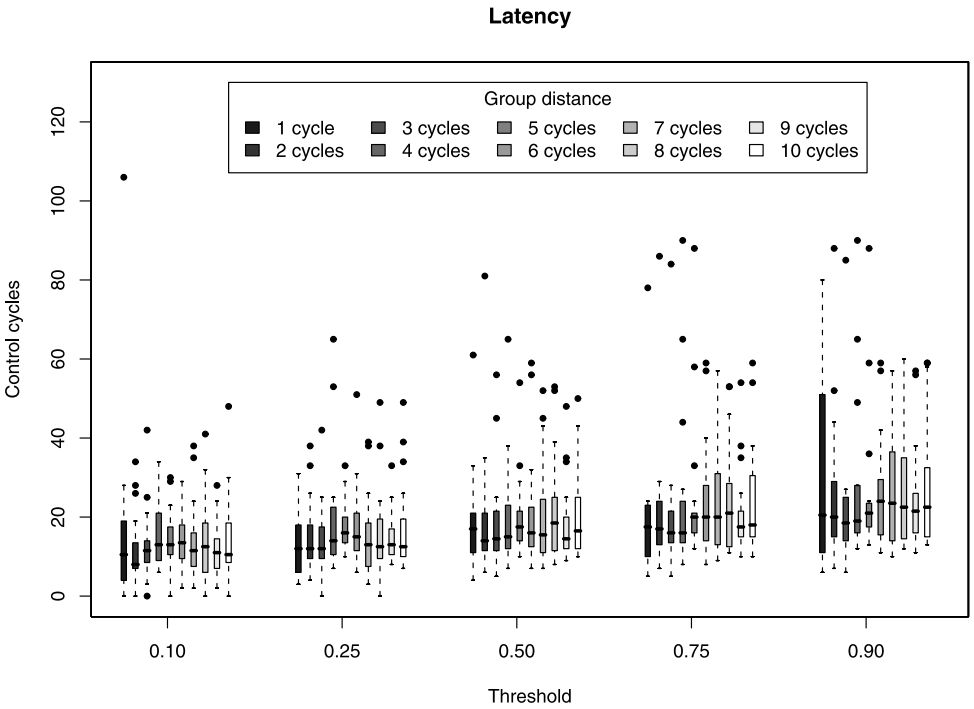


Fig. 6 Box-plot of the latencies observed in 20 evaluation runs in the *find perimeter* setup using fault detectors with input group distances from 1 to 10. Results are shown for the thresholds 0.10, 0.25, 0.50, 0.75, and 0.90. Each box comprises observations ranging from the first to the third quartile. The median is indicated by a horizontal bar, dividing the box into the upper and lower part. The whiskers extend to the farthest data points that are within 1.5 times the interquartile range. Outliers are shown as dots. The results show that the input group distance does not have a major influence on the latency of a fault detector, while larger thresholds yield longer latencies

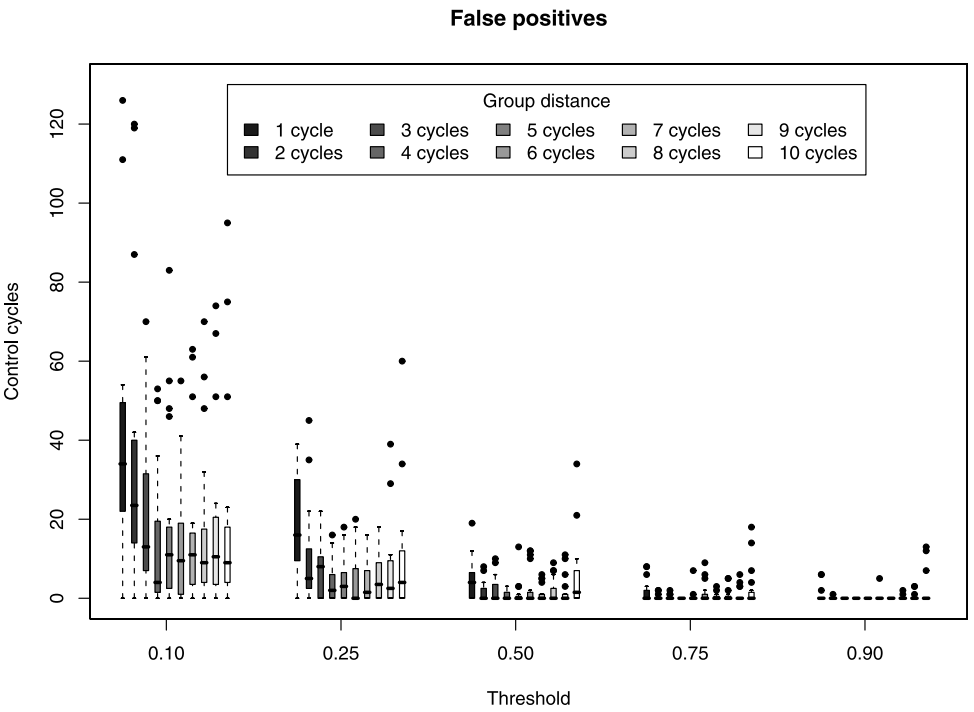


Fig. 7 Box-plot of the number of false positives observed in 20 evaluation runs in the *find perimeter* setup using fault detectors with input group distances from 1 to 10. Results are shown for the thresholds 0.10, 0.25, 0.50, 0.75, and 0.90. For low input group distances, 1 and 2 in particular, the fault detector in general detects a large number of false positives, while no clear trend is observed for fault detectors with input group distances above 4. See the caption of Fig. 6 for details on box-plots

of the number of fault positives, is clearly affected by the choice of the classification threshold: the lower the threshold, the lower the latency of the fault detector and the more false positives are observed. For the fault detector with an input group distance of 5, for instance, the median latency is 13 control cycles when a threshold of 0.10 is used, whereas the median latency is 21 when a threshold of 0.90 is used. For the same fault detector, the median number of false positives is 11 if a threshold of 0.10 is used, while no false positives are observed when a threshold of 0.90 is used.

In a few cases, a fault is never detected. Undetected fault occur when a TDNNs output never exceeds the chosen threshold after a fault has been injected. The number of undetected faults for different thresholds and input group

Table 1 Median latencies during 20 evaluation runs in the *find perimeter* setup with fault detectors using input groups distances from 1 to 10 and for the thresholds: 0.10, 0.25, 0.50, 0.75, and 0.90

Input group distance	Threshold				
	0.10	0.25	0.50	0.75	0.90
1	10.5	12.0	17.0	17.5	20.5
2	8.0	12.0	14.0	17.0	20.0
3	11.5	12.0	14.5	16.0	18.5
4	13.0	14.0	15.0	16.0	19.0
5	13.0	16.0	17.5	20.0	21.0
6	13.5	15.0	16.0	20.0	24.0
7	11.5	13.0	15.5	20.0	23.5
8	12.5	12.5	18.5	21.0	22.5
9	11.0	13.0	14.5	17.5	21.5
10	10.5	12.5	16.5	18.0	22.5

Table 2 Median number of false positives observed during 20 evaluation runs in the *find perimeter* setup with fault detectors using input groups distances from 1 to 10 and for the thresholds: 0.10, 0.25, 0.50, 0.75, and 0.90

Input group distance	Threshold				
	0.10	0.25	0.50	0.75	0.90
1	34.0	16.0	4.0	0.0	0.0
2	23.5	5.0	0.0	0.0	0.0
3	13.0	8.0	0.0	0.0	0.0
4	4.0	2.0	0.0	0.0	0.0
5	11.0	3.0	0.0	0.0	0.0
6	9.5	0.0	0.0	0.0	0.0
7	11.0	1.5	0.0	0.0	0.0
8	9.0	3.5	0.0	0.0	0.0
9	10.5	2.5	0.0	0.0	0.0
10	9.0	4.0	1.5	0.0	0.0

Table 3 Number of undetected faults observed during 20 evaluation runs in the *find perimeter* setup with fault detectors using input group distances from 1 to 10 and for the threshold: 0.10, 0.25, 0.50, 0.75, and 0.90

Input group distance	Threshold				
	0.10	0.25	0.50	0.75	0.90
1	0	1	1	2	2
2	1	1	1	3	3
3	0	0	0	1	3
4	0	0	1	1	2
5	0	0	0	1	1
6	0	0	0	0	0
7	0	0	0	0	0
8	0	0	0	0	0
9	0	0	0	0	0
10	0	0	0	0	0

distances is shown in Table 3. All undetected faults were observed when low input group distances were used.

In the other two setups, *follow the leader* and *connect to s-bot*, we did a similar study of the effect of the input group distance and the performance of the fault detectors. We found that an input group distance of 5 performed well in all setups. The experiments that we present in the following sections are all, therefore, conducted using an input group distance of 5. Since we use TDNNs with 10 input groups, an input group distance of 5 means that a TDNN can see 4.5 s into the past in the *find perimeter* setup, in which the control period is 0.10 s. TDNNs in the *follow the leader* and *connect to s-bot* setups see 6.75 s into the past since the controllers in these setups run with a control period of 0.15 s.

7.2 Fault detection performance in the *follow the leader* and *connect to s-bot* setups

We trained a fault detector to detect faults in the *follow the leader* setup and another fault detector to detect faults in the *connect to s-bot* setup. Box-plots of the false positives and the latency results observed in 20 evaluation runs in the *follow the leader* and *connect to s-bot* setups are shown in Fig. 8 and Fig. 9, respectively. In both setups, the fault detector was configured to use an input group distance of 5. The number of undetected faults observed during the evaluation runs in the two setups is shown in Table 4. Two interesting tendencies can be seen: The number of false positives observed in the *follow the leader* setup are comparatively high, while in the *connect to s-bot* the observed latencies are high when compared with the results obtained in the two other setups. In the *follow the leader* setup there are two robots moving around and the fault detector for the *follower*, in which faults were injected, has to infer the presence of faults based on its interactions with the *leader*. Misclassification of the *follower*'s state can occur in situations where, for instance, the *leader* and the *follower* are moving at constant speeds in a given direction. In these cases the *follower* receives sensory data similar to those in situations where both its treels are *stuck-at-zero*: The *leader* waits for the *follower*, but due to the fault, the *follower* does not move. The fact that the control program (and therefore the fault detector) depends on a dynamic feature of the environment (the *leader*) seems to complicate the classification of the robot's state. However, the performance of the fault detector is still quite good, especially considering that the *leader* is often the *only* object perceivable by the *follower* (the proximity sensors will only sense the presence of walls at distances lower than approximately 10 cm).

The comparatively high latencies observed in the *connect to s-bot* setup are similarly due to a task-dependent feature: After a successful connection has been made, the connecting robot waits for 10 seconds before disconnecting, moving

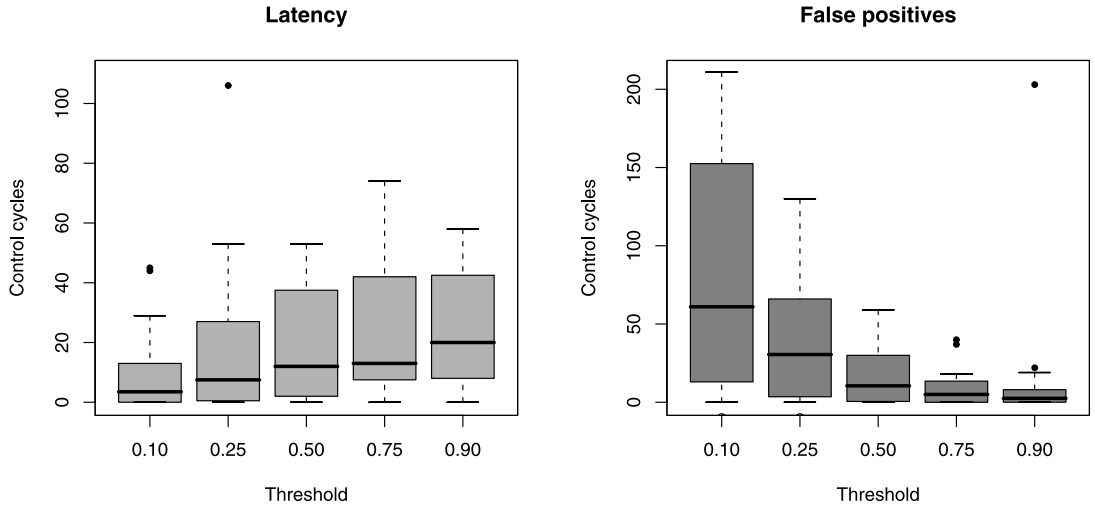


Fig. 8 Box-plots of the latencies and number of false positives observed during 20 evaluation runs in the *follow the leader* setup for different thresholds and an input group distance of 5. See the caption of Fig. 6 for details on box-plots

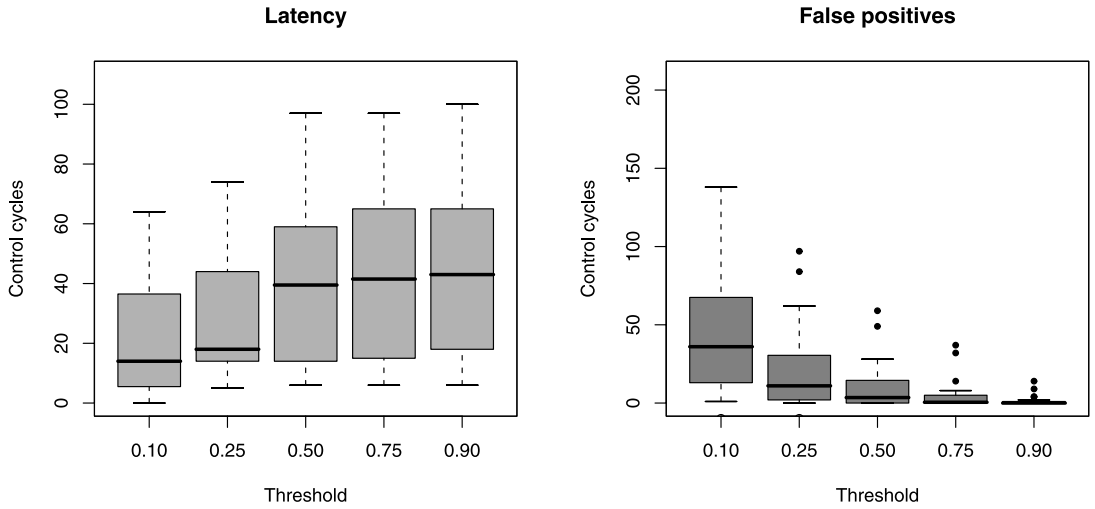


Fig. 9 Box-plots of the latencies and number of false positives observed during 20 evaluation runs in the *connect to s-bot* setup, for different thresholds and an input group distance of 5. See the caption of Fig. 6 for details on box-plots

Table 4 Number of undetected faults observed during 20 evaluation runs in the *follow the leader* and *connect to s-bot* setups, for different thresholds, using an input group distance of 5

	Threshold				
	0.10	0.25	0.50	0.75	0.90
Follow the leader	0	0	0	0	0
Connect to s-bot	1	2	2	2	3

back, and attempting to make a new connection. During the waiting period it is not possible to detect if a fault has occurred in the treels or not. Even if a *stuck-at-constant* fault is injected, causing one or both treels to be assigned a random and non-changeable speed, the outcome is the same: The robot does not move because it is physically connected to the other robot. Thus, it can take longer to detect a fault due to these particular situations in which a fault does not have an effect on the behavior of the robot.

7.3 Reducing the number of false positives

The simplest way to interpret the output of a TDNN trained to detect faults, is to compare the value of the output neuron against a threshold. Values above the threshold are interpreted as evidence of a fault whereas values below the threshold mean that no fault is detected. The fault detectors presented so far follow this simple classification scheme and results have been presented for five thresholds: 0.10, 0.25, 0.50, 0.75, and 0.90.

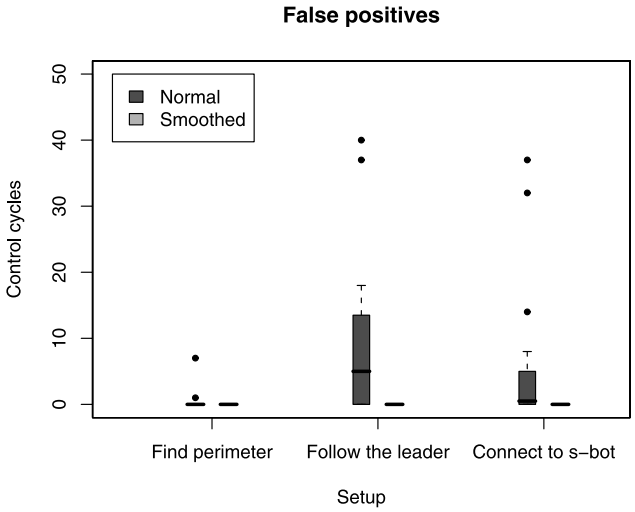


Fig. 10 Box-plots of false positives results observed in 20 runs in each of the three setups using fault detectors in which the output of the TDNN is used directly and fault detectors in which the output is smoothed by computing the moving average over 25 control cycles. A threshold of 0.75 was used for all fault detectors. False positives were only observed during one run in the *follow the leader* setup when the TDNN's output was smoothed. The run is not shown in the figure since it is out of scale (164 false positives were detected during this run). See the caption of Fig. 6 for details on box-plots

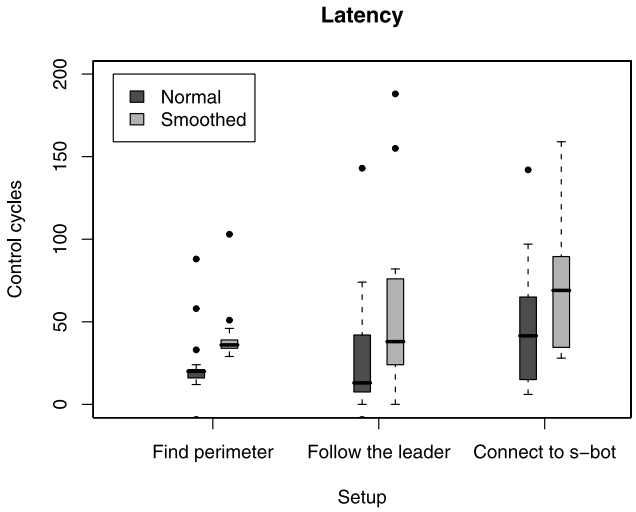


Fig. 11 Box-plots of latency results observed in 20 runs in each of the three setups using fault detectors in which the output of the TDNN is used directly and fault detectors in which the output is smoothed by computing the moving average over 25 control cycles. A threshold of 0.75 was used for all fault detectors. See the caption of Fig. 6 for details on box-plots

For many robotics tasks, a latency of a few seconds does not represent a risk: as long as a fault is eventually detected, the robot is able to communicate this to a human operator or to other robots, who can then take the necessary steps to ensure that the task is progressed. Accommodating a fault, on the other hand, is usually expensive, as other robots need to take action or a human operator needs to evaluate and solve the situation. Frequent false positives, therefore, are likely to have a negative impact on the performance.

One way of reducing the number of false positives is to choose a high threshold, e.g. 0.90, which results in fewer false positives than lower thresholds (see for instance Fig. 9). Many of the false positives observed occur for a single or few consecutive control cycles only (like in the example in Fig. 5). This suggests an alternative way of reducing the number of false positives: to smooth the output of the trained neural networks. We do this by computing the moving average of a trained TDNN's output value and basing the classification on this moving average rather than on the TDNN's output directly. We configured the fault detectors to use a moving average over 25 control cycles of the TDNN's output and a threshold of 0.75. Figures 10 and 11 show respectively the number of false positives and the latencies observed in 20 evaluation runs for each task.

By computing the moving average and thereby smoothing the output of the TDNN, we almost completely eliminate false positives. As the results in Fig. 11 show, however, this is at the cost of a higher latency. Since the moving average increases latency, it can result in more undetected faults as more runs finish before faults are detected. In the *find perimeter* setup, 2 faults were not detected when averaging the output over 25 control cycles, compared to only 1 when averaging was not used. Similarly, in the *connect to s-bot* setup, 5 faults were not detected when a moving average was used, compared to 2 when the output of the TDNN was used directly. In the *follow the leader* setup all faults were detected in both cases.

7.4 Faults in both sensors and actuators

Possible faults are not limited to the mechanical systems that propel robots; other hardware, such as sensors, can also fail. In this section, we demonstrate that our approach is equally applicable to faults in the sensors. We also show that a single appropriately trained fault detector is capable of detecting faults in both sensors and actuators. We first evaluate our approach when only faults in sensors are considered. We then go on to evaluate the approach when faults in both sensors and the treels are considered. All experiments are conducted in the *find perimeter* setup.

We conducted a set of runs in which we injected faults in the front infrared ground sensor, that is, the infrared ground sensor located closest to the gripper (see Fig. 3). We conducted another set of runs in which we injected faults in the first and second light sensor counter-clockwise from the

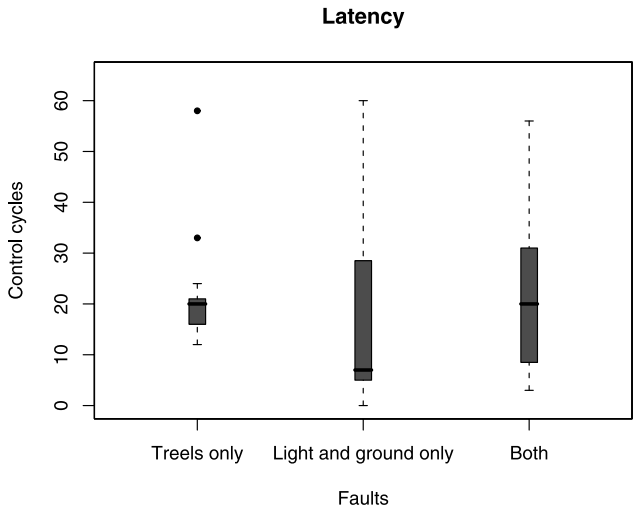


Fig. 12 Box-plots of latency results for fault detectors trained to detect faults in the treels only (from Sect. 7.1), in the ground and light sensors only, and a fault detector trained to detect faults in both the ground and light sensors and the treels. In each case, the fault detector was evaluated on 20 runs in which faults corresponding to those the fault detector was trained to detect were injected. All three fault detectors were configured to use the output of the TDNN directly and to use a threshold of 0.75. See the caption of Fig. 6 for details on box-plots

gripper when an *s-bot* is seen from above (see Fig. 3).[3] We trained a TDNN with a input group distance of 5 on 40 runs: 20 runs during which a fault was injected in the front ground sensor and another 20 runs during which a fault was injected in the light sensors. The fault detector was evaluated on 20 runs: 10 runs in which a fault was injected in the ground sensor and another 10 runs in which a fault was injected in the light sensors.

We performed a set of experiments to determine if a single fault detector can be trained to detect faults in both the sensors and actuators. We trained a fault detector on a training set consisting of 40 runs: 20 runs in which a fault was injected in either one or both treels and 20 runs in which a fault was injected in either the ground sensor or light sensors. The fault detector was evaluated on 20 runs: 10 runs in which a fault was injected in either the ground sensor or in the light sensors, and another 10 runs in which a fault was injected in either one of the treels or in both treels.

Figure 12 shows the latencies observed for the fault detector trained to detect faults in the sensors only and for the fault detector trained to detect faults in both the sensors and actuators. For each detector, we performed 20 evaluation runs. We have included the results for a fault detector trained to detect faults in the actuators (treels) only from Sect. 7.1 to allow for comparison.

The results show that a fault detector can be trained to detect faults in the ground sensor and faults in the light sensors. Furthermore, we have shown that we can train a single fault detector to detect faults in both the sensors and the treels. When a threshold of 0.75 (or higher) is used, no false positives were observed during any of the evaluation runs using the respective fault detectors. The median latency observed for the sensor-only fault detector was 7 control cycles (0.7 seconds). The median latency observed for the sensor and actuator fault detector was 20 control cycles (2.0 seconds), when a threshold of 0.75 was used, which is equivalent to the median latency observed for the treels-only fault detector (see Fig. 12).

7.5 Robustness to variations in the task

Autonomous mobile robots often navigate in environments in which the exact conditions and task parameters are unknown and sometimes even change over time. A fault detector has to be robust to such changes in order to be generally applicable. We trained a fault detector on data from three variations of the *connect to s-bot* setup. In addition to the original setup in which one *s-bot* connects to another stationary *s-bot* (see Fig. 4), we collected data from runs in two additional setups, namely *connect to moving s-bot* and *connect to swarm-bot*, illustrated in Fig. 13.

In the *connect to moving s-bot* setup, the *s-bot* to which a connection should be made (the seed) initially moves around instead of being passive. The seed only stops moving when the two robots get within a distance of 30 cm or less of one another. In the *connect to swarm-bot* setup, the connecting *s-bot* connects to a *swarm-bot*.[4] In our experiment, the *swarm-bot* consists of three *s-bots* connected in a linear formation.

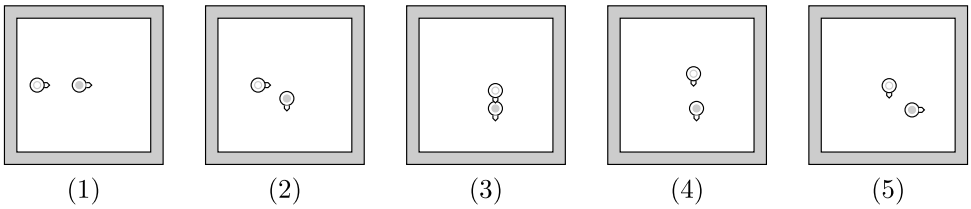
We trained a fault detector with an input group distance of 5 and 10 hidden nodes on a training set consisting of 60 runs: 30 runs in the original setup described in Fig. 4, and 15 runs in each of the two setups illustrated in Fig 13. The fault detector was evaluated on data from 20 runs: 10 runs in the original setup and 5 in each of the new setups. In order to reduce the number of false positives the moving average of the TDNN's output was computed (as explained in Sect. 7.3) and compared against a threshold of 0.90. The results observed in 20 evaluation using the output of the TDNN directly and using a moving average window length of 25 control cycles are shown in Fig. 14.

When the output of the TDNN was used directly one fault was not detected, whereas two faults were not detected when a moving average window length of 25 was used. When the

[3] We initially tried to inject faults in the first light sensor only, but a fault in a single light sensor had no effect on the performance of the robot: with seven out of eight light sensors working the robot still completed the task. We therefore injected faults in both the first and the second light sensor.

[4] A *swarm-bot* is a connected robotic entity consisting of multiple, physically connected *s-bots* (Mondada et al. 2004).

Connect to moving s-bot: One *s-bot* attempts to physically connect to another *s-bot* (the seed) using its gripper. As long as the connecting *s-bot* is farther than 30 cm from the seed, the seed moves around. When the two robots get within 30 cm of one another, the seed stops. When a successful connection has been made, the connecting *s-bot* waits for 10 seconds, disconnects, moves back, and tries to connect again. Faults are injected in the connecting *s-bot* only.



Connect to swarm-bot: One *s-bot* attempts to physically connect to a *swarm-bot* using its gripper. In our experiment, the *swarm-bot* consists of three *s-bots* physically connected in a linear formation. When a successful connection has been made, the connecting *s-bot* waits for 10 seconds, disconnects, moves back, and tries to connect again. Faults are injected in the connecting *s-bot* only.

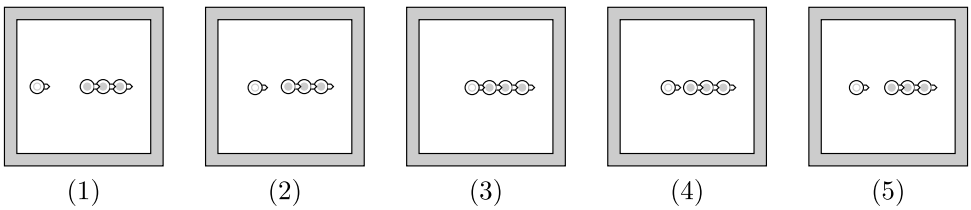


Fig. 13 Two additional setups for the *connect to s-bot* controller used to evaluate if a fault detector can generalize over variations of the task

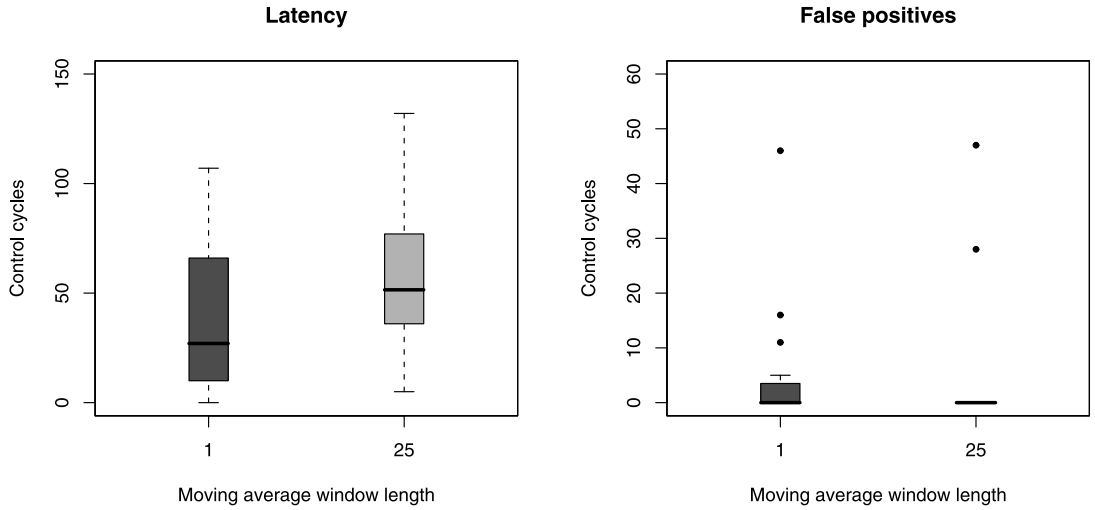


Fig. 14 Box-plots of the latencies and the number of false positives observed during 20 evaluation runs using a fault detector trained on data from a total of 60 runs in all three variations of the *connect to*... setup. Results are shown for moving average window lengths of 1 (equivalent to using the output of the TDNN directly) and 25. A threshold of 0.90 was used. See the caption of Fig. 6 for details on box-plots

output of the TDNN is smoothed over 25 control cycles, false positives only occur in two out of the 20 evaluation runs. Hence, our results indicate that it is possible to train fault detectors that are robust to variations in the task.

7.6 Exogenous fault detection

In robotics, exogenous fault detection is the activity in which one robot detects faults that occur in other, physically sep-

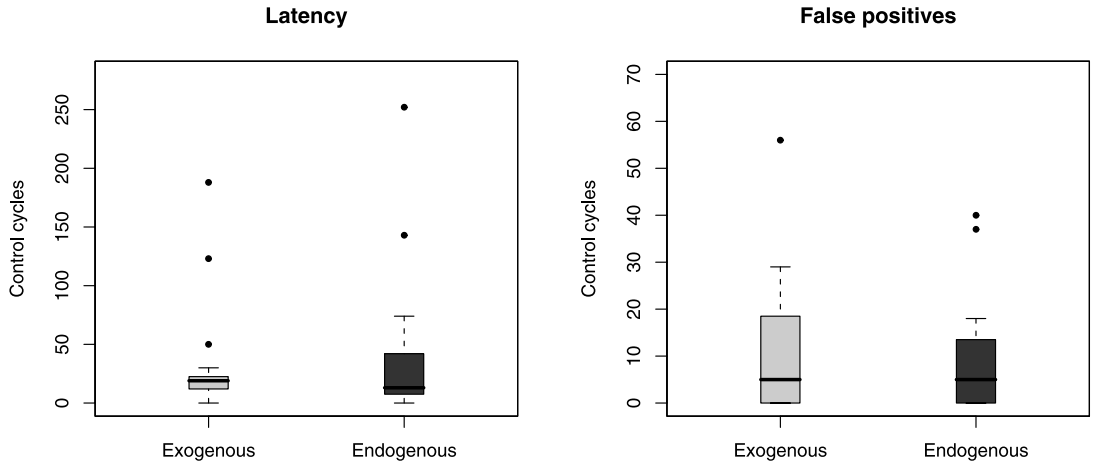


Fig. 15 Box-plots of the performance results in terms of latency and number of false positives observed in 20 evaluation runs for the *follower* performing endogenous fault detection and for the *leader* performing exogenous fault detection during the same runs. For both sets of results, the output of the TDNN is used directly and compared against a threshold of 0.75. See the caption of Fig. 6 for details on box-plots

arate robots. The *s-bot* hardware platform used for the experiments in this paper was originally designed and built in order to study multi-robot and swarm-robotics systems. Such systems have the potential to achieve a high degree of fault tolerance: if one robot fails while performing a task, another robot can take over and complete the task. Various approaches to fault detection and fault tolerance in multi-robot systems have been proposed, such as Parker's ALLIANCE (Parker 1998), Lewis and Tan's *virtual structures* (Lewis and Tan 1997), Gerkey and Matarić's MURDOCH (Gerkey and Matarić 2002b; Gerkey and Matarić 2002a), and Dias et al.'s TraderBots (Dias et al. 2004) among others.

In order to evaluate the applicability of our method to exogenous fault detection, we attempted to get the *leader* to detect faults injected in the *follower* in the *follow the leader* setup. We recorded sensory data for the *leader* robot while faults were injected in the *follower*. The sensory data (camera and infrared proximity sensors) from the *leader* were correlated with the fault state of the *follower* and a TDNN was trained on 40 runs to detect exogenous faults.

Box-plots of the latencies and number of false positives observed during 20 evaluation runs are shown in Fig. 15. In the figure we have plotted results for the *leader* performing exogenous fault detection and the results for the *follower* performing endogenous fault detection during the same runs to allow for comparison. Both fault detectors were configured to use an input group distance of 5 and a classification threshold of 0.75. The median latency for the *leader* performing exogenous fault detection is 19 control cycles, while the median latency for the *follower* performing endogenous fault detection is 14 control cycles. This difference of 5 control cycles corresponds to 750 ms. The median number of false positives is 5 control cycles for both the exogenous and endogenous detector. Visual inspection of Fig. 15 confirms that the performance of the two fault detectors is comparable. In every trial, the fault injected was detected by both the *leader* and the *follower*.

In order to reduce the number of false positives, we can compute the moving average of the trained TDNN's output as explained in Sect. 7.3. If we configure the fault detector to use the moving average of the trained TDNN's output over 25 control cycles and a threshold of 0.75, false positives are only observed in 2 out of the 20 evaluation runs. The results show that a fault detector can be trained that enables one robot to detect faults in another robot.

8 Conclusions

The detection of faults and the subsequent accommodation in autonomous robots are central issues that need to be addressed before widespread adoption for both domestic and industrial purposes can occur. Due to concerns over safety and potential costs incurred by malfunctioning robots, the scope of the tasks with which robots can be entrusted will remain fairly narrow until a high level of dependability can be attained. In this paper, we have suggested a new method for synthesizing fault detectors for autonomous mobile robots. The method is based on learning from examples in which robots operate normally and in which faults are present, respectively.

Our results, obtained with real robots, suggest that fault detection through fault injection and learning is a viable method to generate fault detectors for autonomous mobile robots. The robots need not be equipped with dedicated or

redundant sensors for the method to be applicable. For all the results presented, only the information flowing between the control program and the robots' actuators and sensors used for navigation was used. Although the performance of fault detectors could probably be improved if data from more (possibly dedicated) sensors were used, our results show that a fairly small amount of key information is sufficient to obtain good fault detectors.

We applied the proposed method in three different tasks and we explored various aspects of the method: We showed that it is possible to train fault detectors to detect faults in both actuators and sensors. We demonstrated that a single fault detector is capable of detecting faults of different types and locations. We showed that we can train a fault detector to be robust to variations in the task performed by a robot. Finally, we showed how the proposed method can be extended to exogenous fault detection: In a follow the leader task, we trained a fault detector for the leader robot to detect faults that occurred in the follower robot.

In ongoing research, we are studying extensions to our approach that will allow for fault diagnosis. Our aim is to obtain neural networks that can not only detect the presence of a fault but also the location of the fault. If the control program is made aware that a component is broken, it could direct the robot to perform tasks for which the component is not needed or only use behaviors not requiring the faulty component to be operational. For instance, in case a robot's gripper breaks while the robot is pulling an object, the control program could change the behavior from grasping and pulling to pushing the object, if made aware of the presence *and* the location of the fault. One way of extending the proposed methodology to include fault diagnosis would be to add more output neurons to the classifying neural network. Different output neurons would then correspond to different faults. Another approach could be to use multiple neural networks, one for each component in which faults should be diagnosed.

Acknowledgements This work was supported by the *SWARMANOID* project, funded by the Future and Emerging Technologies programme (IST-FET) of the European Commission, under grant IST-022888. Anders Christensen acknowledges support from COMP2SYS, a Marie Curie Early Stage Research Training Site funded by the European Community's Sixth Framework Programme (grant MEST-CT-2004-505079). The information provided is the sole responsibility of the authors and does not reflect the European Commission's opinion. The European Commission is not responsible for any use that might be made of data appearing in this publication. This work was supported by the ANTS project, an *Action de Recherche Concertée* funded by the Scientific Research Directorate of the French Community of Belgium. Mauro Birattari and Marco Dorigo acknowledge support from the Belgian F.R.S.-FNRS, of which they are a Research Associate and a Research Director, respectively.

References

Arlat, J., Aguera, M., Amat, L., Crouzet, Y., Fabre, J., Laprie, J., Martins, E., & Powell, D. (1990). Fault injection for dependability validation: a methodology and some applications. *IEEE Transactions on Software Engineering*, *16*(2), 166–182.

Ashokaraj, I., Tsourdos, A., Silson, P., & White, B. A. (2004). Sensor based robot localisation and navigation: using interval analysis and unscented Kalman filter. In *Proceedings of the 2004 IEEE/RSJ international conference on intelligent robots and systems (IROS 2004)* (Vol. 1, pp. 64–70). Las Vegas: IEEE Press.

Canham, R., Jackson, A., & Tyrrell, A. (2003). Robot error detection using an artificial immune system. In *Proceedings of NASA/DoD conference on evolvable hardware* (pp. 199–207). Washington: IEEE Computer Society.

Carlson, J., & Murphy, R. (2003). Reliability analysis of mobile robots. In *Proceedings of IEEE international conference on robotics and automation, ICRA'03* (Vol. 1, pp. 274–281). Los Alamitos: IEEE Computer Society Press.

Clouse, D., Giles, C., Horne, B., & Cottrell, G. (1997). Time-delay neural networks: representation and induction of finite-state machines. *IEEE Transactions on Neural Networks*, *8*, 1065–1070.

Dearden, R., Hutter, F., Simmons, R., Thrun, S., Verma, V., & Willeke, T. (2004). Real-time fault detection and situational awareness for rovers: report on the Mars technology program task. In *Proceedings of IEEE aerospace conference* (Vol. 2, pp. 826–840). Los Alamitos: IEEE Computer Society Press.

Dias, M. B., Zinck, M. B., Zlot, R. M., & Stentz, A. (2004). Robust multirobot coordination in dynamic environments. In *Proceedings of IEEE conference on robotics and automation, ICRA'04* (Vol. 4, pp. 3435–3442). Piscataway: IEEE Press.

Dorigo, M., Trianni, V., Şahin, E., Groß, R., Labella, T. H., Baldassarre, G., Nolfi, S., Deneubourg, J.-L., Mondada, F., Floreano, D., & Gambardella, L. M. (2004). Evolving self-organizing behaviors for a swarm-bot. *Autonomous Robots*, *17*(2–3), 223–245.

Forrest, S., Perelson, A., Allen, L., & Cherukuri, R. (1994). Self-nonself discrimination in a computer. In *Proceedings of the 1994 IEEE symposium on research in security and privacy* (Vol. 212, pp. 202–212). Los Alamitos: IEEE Computer Society.

Gerkey, B., & Matarić, M. J. (2002a). Pusher-watcher: an approach to fault-tolerant tightly-coupled robot coordination. In *Proceedings of IEEE international conference on robotics and automation, ICRA'02* (pp. 464–469). Piscataway: IEEE Press.

Gerkey, B. P., & Matarić, M. J. (2002b). Sold!: Auction methods for multirobot coordination. *IEEE Transactions on Robotics and Automation*, *18*(5), 758–768.

Gertler, J. J. (1988). Survey of model-based failure detection and isolation in complex plants. *IEEE Control Systems Magazine*, *8*, 3–11.

Goel, P., Dedeoglu, G., Roumeliotis, S., & Sukhatme, G. (2000). Fault detection and identification in a mobile robot using multiple model estimation and neural network. In *Proceedings of IEEE international conference on robotics and automation, ICRA'00* (Vol. 3, pp. 2302–2309). Los Alamitos: IEEE Computer Society Press.

Groß, R., Bonani, M., Mondada, F., & Dorigo, M. (2006). Autonomous self-assembly in swarm-bots. *IEEE Transactions on Robotics*, *22*(6), 1115–1130.

Hinchey, M., Rash, J., Rouff, C., & Truszkowski, W. (2004). NASA's swarm missions: the challenge of building autonomous software. *IT Professional*, *6*, 47–52.

Hsueh, M., Tsai, T., & Iyer, R. (1997). Fault injection techniques and tools. *Computer*, *30*(4), 75–82.

Isermann, R. (1997). Supervision, fault-detection and fault-diagnosis methods—an introduction. *Control Engineering Practice*, *5*(5), 639–652.

Isermann, R., & Ballé, P. (1997). Trends in the application of model-based fault detection and diagnosis of technical processes. *Control Engineering Practice*, *5*(5), 709–719.

Julier, S. J., & Uhlmann, J. K. (1997). A new extension of the Kalman filter to nonlinear systems. In *Proceedings of the 11th international symposium on aerospace/defense sensing, simulation and controls* (Vol. 3, pp. 182–193). Bellingham: SPIE.

Kalman, R. (1960). A new approach to linear filtering and prediction problems. *Journal of Basic Engineering*, *82*(1), 35–45.

Kochan, A. (2005). A bumper year for robots. *Industrial Robot: An International Journal*, *32*, 201–204.

Leonard, J. J., & Durrant-Whyte, H. F. (1991). Mobile robot localization by tracking geometric beacons. *IEEE Transactions on Robotics and Automation*, *7*(3), 376–382.

Lerner, U., Parr, R., Koller, D., & Biswas, G. (2000). Bayesian fault detection and diagnosis in dynamic systems. In *Proceedings of the 7th national conference on artificial intelligence* (pp. 531–537). Cambridge: AAAI Press/MIT Press.

Lewis, M. A., & Tan, K. H. (1997). High precision formation control of mobile robots using virtual structures. *Autonomous Robots*, *4*(4), 387–403.

Li, P., & Kadirkamanathan, V. (2001). Particle filtering based likelihood ratio approach to fault diagnosis in nonlinear stochastic systems. *IEEE Transactions Systems, Man Cybernetics, Part C*, *31*(3), 337–343.

Marsland, S., Nehmzow, U., & Shapiro, J. (2005). On-line novelty detection for autonomous mobile robots. *Robotics and Autonomous Systems*, *51*(2–3), 191–206.

Mondada, F., Pettinaro, G. C., Guignard, A., Kwee, I., Floreano, D., Deneubourg, J.-L., Nolfi, S., Gambardella, L., & Dorigo, M. (2004). Swarm-bot: a new distributed robotic concept. *Autonomous Robots*, *17*(2–3), 193–221.

Mondada, F., Gambardella, L. M., Floreano, D., Nolfi, S., Deneubourg, J.-L., & Dorigo, M. (2005). The cooperation of swarm-bots: physical interactions in collective robotics. *IEEE Robots and Automation Magazine*, *12*(2), 21–28.

Nouyan, S., Groß, R., Bonani, M., Mondada, F., & Dorigo, M. (2006). Group transport along a robot chain in a self-organised robot colony. In T. Arai, R. Pfeifer, T. Balch, & H. Yokoi (Eds.), *Intelligent autonomous systems* (Vol. 9, pp. 433–442). Amsterdam: IOS Press.

Nouyan, S., Campo, A., & Dorigo, M. (2008). Path formation in a robot swarm: self-orgenized strategies to find your way home. *Swarm Intelligence*, *1*(2).

O'Grady, R., Groß, R., Mondada, F., Bonani, M., & Dorigo, M. (2005). Self-assembly on demand in a group of physical autonomous mobile robots navigating rough terrain. In M. S. Capcarrere, A. A. Freitas, P. J. Bentley, C. G. Johnson, & J. Timmis (Eds.), *Lecture notes in artificial intelligence: Vol. 3630. Advances in artificial life: 8th European conference, ECAL 2005* (pp. 272–281). Berlin: Springer.

Parker, L. E. (1998). ALLIANCE: an architecture for fault tolerant multirobot cooperation. *IEEE Transactions on Robotics and Automation*, *14*(2), 220–240.

Patton, R., Uppal, F., & Lopez-Toribio, C. (2000). Soft computing approaches to fault diagnosis for dynamic systems: a survey. In A. Edelmayer, C. Banyasz (Eds.), *Proceedings of 4th IFAC symposium on fault detection supervision and safety for technical processes* (Vol. 1, pp. 298–311). Oxford: Elsevier.

Roumeliotis, S., Sukhatme, G., & Bekey, G. (1998). Sensor fault detection and identification in a mobile robot. In *Proceedings of IEEE/RSJ international conference on intelligent robots and systems* (Vol. 3, pp. 1383–1388). Los Alamitos: IEEE Computer Society Press.

Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986). Learning internal representations by back-propagating errors. *Nature*, *323*, 533–536.

Skoundrianos, E. N., & Tzafestas, S. G. (2004). Finding fault—fault diagnosis on the wheels of a mobile robot using local model neural networks. *IEEE Robotics and Automation Magazine*, *11*(3), 83–90.

Smith, R., & Cheeseman, P. (1986). On the representation and estimation of spatial uncertainty. *The International Journal of Robotics Research*, *5*(4), 56.

Terra, M., & Tinos, R. (2001). Fault detection and isolation in robotic manipulators via neural networks: a comparison among three architectures for residual analysis. *Journal of Robotic Systems*, *18*(7), 357–374.

Trianni, V., & Dorigo, M. (2006). Self-organisation and communication in groups of simulated and physical robots. *Biological Cybernetics*, *95*, 213–231.

Verma, V., & Simmons, R. (2006). Scalable robot fault detection and identification. *Robotics and Autonomous Systems*, *54*(2), 184–191.

Verma, V., Gordon, G., Simmons, R., & Thrun, S. (2004). Real-time fault diagnosis. *IEEE Robotics and Automation Magazine*, *11*(2), 56–66.

Vemuri, A., & Polycarpou, M. (1997). Neural-network-based robust fault diagnosis in robotic systems. *IEEE Transactions on Neural Networks*, *8*(6), 1410–1420.

Waibel, A., Hanazawa, T., Hinton, G., Shikano, K., & Lang, K. (1989). Phoneme recognition using time-delay neural networks. *IEEE Transactions Acoustics, Speech, and Signal Processing*, *37*, 328–339.

Anders Lyhne Christensen is currently a researcher at IRIDIA, CoDE, Université Libre de Bruxelles, Belgium. He has spent several years in the private sector and worked on software projects ranging from 3D acoustics to high performance computing. In 2002, he obtained his first master's degree in computer science and bio-informatics from Aalborg University, Denmark. Later that year, he received a Marie Curie Fellowship hosted by the Dependable Systems Group at the Universidade de Coimbra and Critical Software, Portugal. He completed his DEA at Université Libre de Bruxelles in 2005. He has published work in bio-informatics, high performance computing, and autonomous robotics. His current research interests are in dependable swarm robotics, autonomous self-assembly, and evolutionary robotics.

Rehan O'Grady is a researcher at the IRIDIA, CoDE, Université Libre de Bruxelles, Belgium. He graduated in 1999 from Edinburgh University with First Class Honors in mathematics and computer science. He received the Kevin Clark memorial prize, awarded each year to the top mathematics and computer science graduate. He subsequently worked in the software industry for several years. During his time at Micromuse PLC he developed a system for monitoring usage outages in Internet network services which was subsequently patented. He received the DEA from the Université Libre de Bruxelles in 2005. His current research interests are in swarm robotics and self-assembling robotic systems.

Mauro Birattari received a master's degree in electrical and electronic engineering from the Politecnico di Milano, Milan, Italy, in 1997, and a doctoral degree in Information Technologies from the Faculty of Engineering of the Université Libre de Bruxelles, Brussels, Belgium, in 2004. He is currently with IRIDIA, CoDE, Université Libre de Bruxelles, as a research associate of the fund for scientific research F.R.S.-FNRS of Belgium's French Community. Dr. Birattari co-authored about 50 peer-reviewed scientific publications in the field of computational intelligence. His research interests focus on swarm intelligence, ant colony optimization, machine learning, and on the application of artificial intelligence techniques to the automatic design of algorithms. Dr. Birattari is an Associate Editor for the journal *Swarm Intelligence* and has served in the organizing committee of the third, fourth, and fifth edition of the *International Workshop on Ant Colony Optimization and Swarm Intelligence*.

Marco Dorigo received the master degree in industrial technologies engineering in 1986 and the doctoral degree in information and systems electronic engineering in 1992 from Politecnico di Milano, Milan, Italy, and the title of Agrégé de l'Enseignement Supérieur, from the Université Libre de Bruxelles, Belgium, in 1995. From 1992 to 1993 he was a research fellow at the International Computer Science Institute of Berkeley, CA. In 1993 he was a NATO-CNR fellow, and from 1994 to 1996 a Marie Curie fellow. Since 1996 he has been a tenured researcher of the F.R.S.-FNRS, the fund for scientific research of Belgium's French Community, and a research director of IRIDIA, the artificial intelligence laboratory of the Université Libre de Bruxelles. He is the inventor of the ant colony optimization metaheuristic. His current research interests include swarm intelligence, swarm robotics, and metaheuristics for discrete optimization. Dr. Dorigo is the Editor-in-Chief of the Swarm Intelligence journal, and an Associate Editor or member of the editorial board for many journals in computational intelligence and adaptive systems. In 1996 he was awarded the Italian Prize for Artificial Intelligence, in 2003 the Marie Curie Excellence Award, in 2005 the Dr A. De Leeuw-Damry-Bourlart award in applied sciences and in 2007 the Cajastur International Prize for Soft Computing.