2009 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2009)

Proceedings of the

2nd International Workshop on Evolutionary and Reinforcement Learning for Autonomous Robot Systems (ERLARS 2009)

Thursday, October 15 2009
St. Louis, Missouri, U.S.A.

Nils T Siebel and Josef Pauli

http://www.erlars.org/

Table of Contents

A Message from the Chairs

Welcome to the 2nd International Workshop on Evolutionary and Reinforcement Learning for Autonomous Robot Systems, ERLARS 2009, held in conjunction with the IROS 2009 conference in St. Louis, Missouri, USA on October 15 2009.

The ERLARS workshop is concerned with research on efficient algorithms for evolutionary and reinforcement learning methods to make them more suitable for autonomous robot systems. The long term goal is to develop methods that enable robot systems to learn completely, directly and continuously through interaction with the environment. In order to achieve this, methods are examined that can make the search for suitable robot control strategies more feasible for situations in which only few measurements about the environment can be obtained.

The articles that you will find in these proceedings are steps along this way. We hope that they can serve as a useful set of ideas and methods to achieve the long term research goal.

We would like to thank the program committee members who provided very good reviews in a short period of time. We are also especially indebted to the authors of the articles sent to this workshop for providing the material to make us think and discuss.

It has been a great pleasure organising this event and we are happy to be supported by such a strong team of researchers. We sincerely hope that you enjoy the workshop and we look forward, with your help, to continue building a strong community around this event in the future.

Nils T Siebel and Josef Pauli, Chairs, ERLARS 2009 Workshop.

Organisation of the ERLARS 2009 Workshop

Workshop Chairs

Nils T Siebel
Cognitive Systems Group
Institute of Computer Science
Christian-Albrechts-University of Kiel
Kiel, Germany

Josef Pauli
Intelligent Systems Group
Department of Computer Science
University of Duisburg-Essen
Duisburg, Germany

Programme Committee

Andrew Barto, Autonomous Learning Laboratory, University of Massachusetts Amherst, USA.
Peter Dürr, Laboratory of Intelligent Systems, EPFL Lausanne, Switzerland.
Christian Igel, Institut für Neuroinformatik, Ruhr-Universität Bochum, Germany.
Yohannes Kassahun, DFKI Lab Bremen, University of Bremen, Germany.
Takanori Koga, Computational Brain Science Laboratory, Yamaguchi University, Japan.
Tim Kovacs, Department of Computer Science, University of Bristol, UK.
Jun Ota, Graduate School of Engineering, University of Tokyo, Japan.
Josef Pauli, Intelligent Systems Group, University of Duisburg-Essen, Germany.
Jan Peters, Max Planck Institute for Biological Cybernetics, Tübingen, Germany.
Daniel Polani, Department of Computer Science, University of Hertfordshire, Hatfield, UK.
Marcello Restelli, Artificial Intelligence and Robotics Laboratory, Politecnico di Milano, Italy.
Stefan Schiffer, Department of Computer Science, RWTH Aachen University, Germany.
Juergen Schmidhuber, Swiss AI Lab IDSIA, Lugano, Switzerland.
Nils T Siebel, Institute of Computer Science, Christian-Albrechts-University of Kiel, Germany.
Marc Toussaint, Berlin Machine Learning and Robotics Group, TU Berlin, Germany.
Jeremy Wyatt, School of Computer Science, University of Birmingham.

2nd International Workshop on Evolutionary and Reinforcement Learning for Autonomous Robot Systems (ERLARS 2009)
St. Louis, Missouri, USA, October 15 2009

Combining Central Pattern Generators with the Electromagnetism-like Algorithm for Head Motion Stabilization during Quadruped Robot Locomotion

Cristina P. Santos, Miguel Oliveira, Vitor Matos, Ana Maria A.C. Rocha and Lino Costa

***Abstract*— Visually-guided locomotion is important for autonomous robotics. However, there are several difficulties, for instance, the head shaking that results from the robot locomotion itself that constraints stable image acquisition and the possibility to rely on that information to act accordingly.**

In this article, we propose a controller architecture that is able to generate locomotion for a quadruped robot and to generate head motion able to minimize the head motion induced by locomotion itself. The movement controllers are biologically inspired in the concept of Central Pattern Generators (CPGs). CPGs are modelled based on nonlinear dynamical systems, coupled Hopf oscillators. This approach allows to explicitly specify parameters such as amplitude, offset and frequency of movement and to smoothly modulate the generated oscillations according to changes in these parameters. We take advantage of this particularity and propose a combined approach to generate head movement stabilization on a quadruped robot, using CPGs and a global optimization algorithm. The best set of parameters that generates the head movement are computed by the electromagnetism-like algorithm in order to reduce the head shaking caused by locomotion.

Experimental results on a simulated AIBO robot demonstrate that the proposed approach generates head movement that does not eliminate but reduces the one induced by locomotion.

I. INTRODUCTION

Robot locomotion is a challenging task that involves several relevant subtasks, not yet completely solved. The motion of quadruped, biped and snake-like robots, for instance, with cameras mounted in their heads, causes head shaking.This kind of disturbances, generated by locomotion itself, makes it difficult to keep the visual frame stable and, therefore, to act according to the visual information. Head stabilization is very important for achieving a visually-guided locomotion, a concept which has been suggested from a considerable number of neuroscientific findings in humans and animals [18].

As a basic research to realize visually-guided quadruped locomotion, we aim in this article at head stabilization of a quadruped robot that walks with a walking gait. In our research, we propose a motion stabilization system of an ers-7 AIBO quadruped robot, which performs its own head motion according to a feedforward controller. Several similar works have been proposed in literature [4], [7], [6], [5]. But these methods consider that the robot moves according to a scheduled robot motion plan, which imply that space and time constraints on robot motion must be known before hand as well as robot and environment models. As such, control is based on this scheduled plan. Other works have successfully achieved gaze stabilization [5], that consists on image stabilization during head movements in space. The overall of the gaze stabilization approaches can be divided into two types of techniques. One uses specific hardware, like accelerometers and gyroscope to estimate the 3D posture of the head, and complex control algorithms to compensate the oscillations. The use of inertial information was already proposed by several authors [5], [16], [17]. Typically this kind of techniques is used in binocular robot heads, where gaze is implemented through the coordination of the two eye movements. Most of the approaches are inspired in biological systems, specifically in the human Vestibular-Ocular Reflex (VOR). In robots with fixed eyes, the fixation point procedure is achieved by compensatory head or body movements, based on multisensory information of the head.

In this work, a combined approach to generate head movement stabilization on a quadruped robot, using Central Pattern Generators (CPGs) and the electromagnetism-like algorithm is proposed. We intend to use a head controller, based on Central Pattern Generators (CPGs), that generates trajectories for tilt, pan and nod head joints. CPGs are neural networks located in the spine of vertebrates, able to generate coordinated rhythmic movements, namely locomotion [11]. These CPGs are modelled as coupled oscillators and solved using numeric integration. These CPGs have been applied in drumming [1] and postural control [3]. This dynamical systems approach model for CPGs presents multiple interesting properties, including: low computation cost which is well-suited for real time; robustness against small perturbations; the smooth online modulation of trajectories through changes in the dynamical systems parameters and phase-locking between the different oscillators for different DOFs.

In order to achieve the desired head movement, opposed to the one induced by locomotion, it is necessary to appropriately tune the CPG parameters. This can be achieved by optimizing the CPG parameters using an optimization method. The optimization process is done offline according to the head movement induced by the locomotion when no stabilization procedure was performed.

Some algorithms for solving this type of problem require substantial gradient information and aim to improve the

Cristina Santos, Miguel Oliveira and Vitor Matos are with Industrial Electronics Department, School of Engineering, University of Minho, 4800-058 Guimaraes, Portugal cristina@dei.uminho.pt, mcampos@dei.uminho.pt, vmatos@dei.uminho.pt

Ana Rocha and Lino Costa are with Production Systems Department, School of Engineering, University of Minho, 4710-057 Braga, Portugal arocha@dps.uminho.pt, lac@dps.uminho.pt

solution in a neighborhood of a given initial approximation. When the problem has more than one local solution, the convergence to the global solution may depend on the provided initial approximation. Thus, searching for a global optimum is a difficult task that could be done by using stochastic-type algorithms. The stochastic methods can be classified in two main categories, namely, the point-to-point search strategies and the population-based search techniques. From the population-based techniques, we would like to emphasize three particular algorithms, the electromagnetism-like algorithm (EM) [12], the particle swarm optimization [13] and genetic algorithms (GA) [2] that despite employing different strategies, they are easy to implement and computationally inexpensive in terms of memory requirement. The GA is well suited and has already been applied to solve this optimization problem because it can handle both discrete and continuous variables, nonlinear objective and constrain functions without requiring gradient information [14]. Recently, EM algorithm appeared as a promising algorithm for handling optimization problems with simple bounds. This technique is finding popularity within research community as design tools and problem solvers because of their versatility and ability to optimize in complex multimodal search spaces applied to nondifferentiable objective functions [15]. In this paper, we are interested in the application of the EM algorithm, proposed in [12], to optimize the CPG parameters of amplitude, offset and frequency of each head oscillator to head motion stabilization during quadruped robot locomotion.

The remainder of this paper is organized as follows. In Section II, the system architecture and how to generate locomotion and head movement is described. The main ideas concerning the optimization system, namely the problem statement that evaluates the head movement, the EM mechanism to optimize the CPG parameters and some experimental results, are described in Section III. Simulated results are described in Section IV. Conclusions are made in Section V.

II. SYSTEM ARCHITECTURE

Our aim is to propose a control architecture that is able to generate locomotion for a quadruped robot and to generate head motion such as to minimize the head movement induced by the the locomotion itself.

The overall system architecture is depicted in Figure 1.

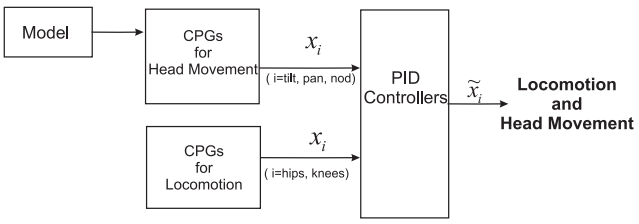


Fig. 1. Overall system architecture

The proposed movement controllers are biologically inspired in the concept of CPGs. A locomotion controller generates hip and knee trajectories. A head controller specifies the planned neck tilt, pan and nod joint values. These trajectories are used as input for the PID controllers of these joints.

The head controller parameters have to be tuned such that the resultant movement is as desired. Using our CPG approach allows us to assign explicit parameters for each of the nonlinear oscillators, independently controlling the amplitude, offset and frequency of the movement. We apply a stochastic optimization method, the EM algorithm, in order to determine the best set of CPG control parameters that results in, or close to the desired movement. This set of parameters constitute the Model module in Fig. 1.

A. Locomotion Generation

In this section we present the network of CPGs used to generate locomotion. A CPG for a given degree-of-freedom (DOF) is modelled as coupled Hopf oscillators, that generate a rhythmic movement.

1) Rhythmic Movement Generation: Rhythmic movements are generated by the following Hopf oscillator

$$\dot{x}_\mathrm{i} = \beta\left(\mu_\mathrm{i} - r_\mathrm{i}^2\right)(x_\mathrm{i} - O_\mathrm{i}) - \omega z_\mathrm{i}, \quad (1)$$

$$\dot{z}_\mathrm{i} = \beta\left(\mu_\mathrm{i} - r_\mathrm{i}^2\right) z_\mathrm{i} + \omega\,(x_\mathrm{i} - O_\mathrm{i}), \quad (2)$$

where $r_\mathrm{i} = \sqrt{(x_\mathrm{i} - O_\mathrm{i})^2 + z_\mathrm{i}^2}$, ω specifies the oscillations frequency (in rad s^{-1}), peak-to-peak amplitude of the oscillations are given by $A_\mathrm{i} = 2\sqrt{\mu_\mathrm{i}}$ and relaxation to the limit cycle is given by $\frac{1}{2\beta\mu_\mathrm{i}}$.

This Hopf oscillator contains a bifurcation from a stable fixed point at $x_\mathrm{i} = O_\mathrm{i}$ (when $\mu_\mathrm{i} < 0$) to a structurally stable, harmonic limit cycle, for $\mu_\mathrm{i} > 0$. The fixed point x_i has an offset given by O_i.

Thus, this Hopf oscillator exhibits limit cycle behaviour and describes a stable rhythmic motion where parameters A_i, ω and O_i control the desired amplitude, frequency and offset of the resultant oscillations.

2) Locomotion Controller Architecture: We have to couple the oscillators in order to ensure phase-locked synchronization between the hip and knee DOFs of the robot, and generate locomotion with a desired gait.

Fig. 2 depicts the network structure used to generate locomotion for a quadruped robot. Hopf oscillators of the hips are bilaterally coupled, these couplings being illustrated by right-left arrows, and hip Hopf oscillators are unilaterally

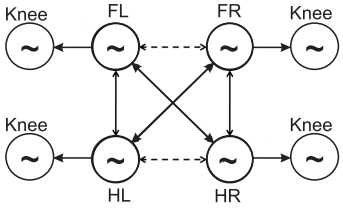


Fig. 2. Locomotion controller architecture depicting coupling structure among the CPGs for a walking gait. The footfall sequence is: HL-FL-HR-FR, with each foot lagging a quarter of a cycle from the previous.

coupled to the corresponding knee Hopf oscillators. For the hip joints, this is achieved by modifying (1) and (2) as follows:

$$\begin{bmatrix}\dot{x}_{i[1]}\\ \dot{z}_{i[1]}\end{bmatrix} = \begin{bmatrix}\beta\mu_i & \omega\\ -\omega & \beta\mu i\end{bmatrix}\begin{bmatrix}x_{i[1]}-O_{i[1]}\\ z_{i[1]}\end{bmatrix} - \beta r_{i[1]}^2\begin{bmatrix}x_{i[1]}-O_{i[1]}\\ z_{i[1]}\end{bmatrix} + \sum_{j\neq i}\mathbf{R}(\theta_{i[1]}^{j[1]})\begin{bmatrix}x_{j[1]}-O_{j[1]}\\ z_{j[1]}\end{bmatrix}$$

For the knee joints, we modify (1) and (2) as follows:

$$\begin{bmatrix}\dot{x}_{i[3]}\\ \dot{z}_{i[3]}\end{bmatrix} = \begin{bmatrix}\beta\mu i & \omega\\ -\omega & \beta\mu i\end{bmatrix}\begin{bmatrix}x_{i[3]}-O_{i[3]}\\ z_{i[3]}\end{bmatrix} - \beta r_{i[3]}^2\begin{bmatrix}x_{i[3]}-O_{i[3]}\\ z_{i[3]}\end{bmatrix} + \frac{1}{2}\mathbf{R}(\psi_{i[3]}^{j[1]})\begin{bmatrix}x_{j[1]}-O_{j[1]}\\ z_{j[1]}\end{bmatrix}$$

where $r_i[k]$ is the norm of vector $(x_i[k]-O_{i[k]}, z_i[k])^T$ (k = 1,3, that is hip and knee joints) and i, j = Fore Left (FL), Fore Right (FR), Hind Left (HL) and Hind Right (HR) limbs. The linear terms are rotated onto each other by the rotation matrices $\mathbf{R}(\theta_{i[1]}^{j[1]})$ and $\mathbf{R}(\psi_{i[3]}^{j[1]})$, where $\theta_{i[1]}^{j[1]}$ is the relative phase among the i[1]'s and j[1]'s hip oscillators and represents bidirectional couplings between these oscillators such that $\theta_{i[1]}^{j[1]} = -\theta_{j[1]}^{i[1]}$ and $\psi_{i[3]}^{j[1]}$ is the required relative phase among the i[3]'s and j[1]'s oscillators (see Fig. 2). We assure that closed-loop interoscillator couplings have phase biases that sum to a multiple of 2 π.

Each hip oscillator lags a quarter of a cycle from the previous. The relative phases between hips and knees, $\psi_{i[3]}^{j[1]}$, were all set to 180.

Due to the properties of these coupled Hopf oscillators, the generated trajectories are always smooth and thus potentially useful for trajectory generation in a robot.

This network structure constitutes the locomotion controller that generates desired trajectories, x_i, obtained by integrating the CPGs dynamical systems. These are sent online for the PID controllers of each hip and knee joints and result in the actual trajectories $\tilde{x}_i$.

3) Generating a walking gait: A gait event sequence is specified using the duty factors and the relative phases, where the first event, and the start of the stride, is chosen as the event when the fore left leg (reference leg) is set down. We have set a non-singular, regular and symmetric gait with a FL-HR-FR-HL gait even sequence $\{\varphi_{FL}, \varphi_{HR}, \psi_{FR}, \psi_{HL}, \varphi_{FR}, \varphi_{HL}, \psi_{FL}, \psi_{HR}\}$), a duty factor of 0.73 and a velocity of 19mms^{-1} (measured in the Z direction, see Fig. 3).

We have implemented in webots [8] this locomotion controller (simulation results and the experiment description is detailed explained in section).

B. Head Movement Generation

Head movement is generated similarly to locomotion, but a CPG for a given DOF is modelled as an Hopf oscillator, not coupled to any other oscillator. Each CPG, therefore, generates a rhythmic movement according to

$$\begin{bmatrix}\dot{x}_i\\ \dot{z}_i\end{bmatrix} = \begin{bmatrix}\beta\mu_i & \omega\\ -\omega & \beta\mu i\end{bmatrix}\begin{bmatrix}x_i-O_i\\ z_i\end{bmatrix} - \beta r_i^2\begin{bmatrix}x_i-O_i\\ z_i\end{bmatrix}, \quad (3)$$

where i =tilt,pan,nod.

The control policy is the x_i variable, obtained by integrating the CPGs dynamical systems, and represents tilt, pan and nod joint angles in our experiments. These are sent online for the corresponding PID controllers.

Note that the final movement for each of these joints is a rhythmic motion which amplitude of movement is specified by μ_i, offset by O_i and its frequency by ω.

The differential equations for locomotion and head movement are solved using Euler integration with a fixed time step of 1ms. The x_i trajectories represent angular positions and are directly sent to the PID controllers of the joint servomotors.

III. OPTIMIZATION SYSTEM

In this section, we explain how the head CPGs are optimized in order to reduce the camera (head) movement induced by locomotion itself. We will optimize the distance between the generated head movement for a set of head CPG control parameters and the one induced by locomotion.

In order to implement the head motion it is necessary one or several optimal combinations of amplitude, offset and frequency of each head oscillator. This is possible because we can easily modulate amplitude, offset and frequency of the generated trajectories according to changes in the A_i, O_i and ω CPG parameters and these are represented in an explicit way by our CPG. Therefore, we have to tune the head CPG parameters: amplitude A_i, offset O_i and common frequency ω. In order to optimize the combinations of the different head CPG control parameters the EM algorithm is used.

The multitude of parameter combinations is large, and it is difficult to derive an accurate model for the tested quadruped robot and for the environment. Besides, such a model based approach would also require some post-adaptation of results (because of backlash, friction, etc).

In this study, the search of parameters suitable for the implementation of the required head motion was carried out based on the data from a simulated quadruped robot. The (X,Y,Z) head coordinates, in a world coordinate system (Fig. 3), are recorded when a simulated robot walks during 30s and no head stabilization is performed. We are interested in the opposite of this movement around the (X,Y,Z) coordinates. This data was mathematically treated such as to keep only the oscillations in the movement and remove the drift that the robot has in the X coordinate and also the forward movement in the Z coordinate. From now on, this data is referred to as $(X,Y,Z)_{\text{observed}}$.

In the simulation, we have set a cycle time of 30ms, that is, the time needed to perform sensory acquisitions, calculate the planned trajectories (integrating the differential equations) and send this data to the servomotors. The $(X,Y,Z)_{\text{observed}}$ data is sampled with a sample time of 30ms, meaning we have a total of 1000 samples. A simulated time of 30s corresponds to 10 strides of locomotion. This time is arbitrary and could have been chosen differently but seems well suited to find a model representative of the head movement induced by the locomotion controller.

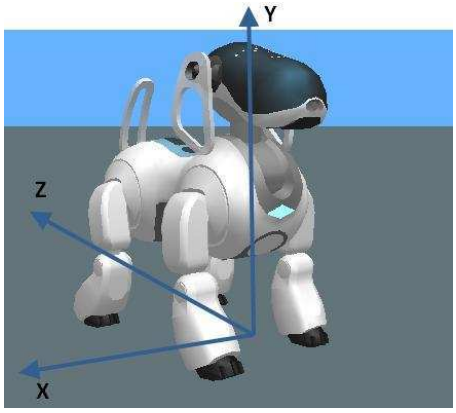


Fig. 3. World coordinate system.

The basic idea is to combine the CPG model for head movement generation with the optimization algorithm. Fig. 4 illustrates a schematics of the overall optimization system.

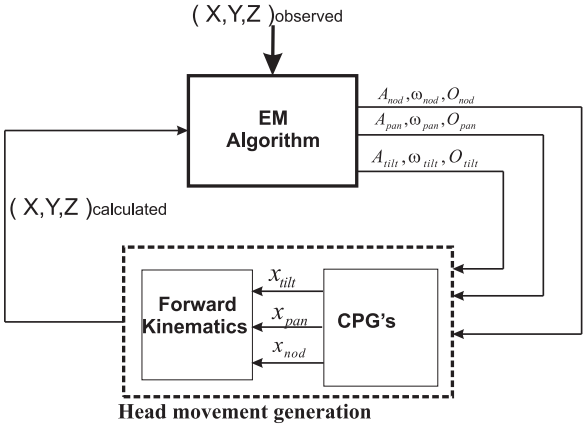


Fig. 4. Schematics of the optimization system.

Three head CPGs (3) generate during 30s rhythmic motions for the tilt, pan and nod joints. By applying forward kinematics, we calculate the resultant set of 1000 samples of $(X,Y,Z)_{\text{calculated}}$ head coordinates in the world coordinate system.

A. Problem Definition

The sum of the distances between each sample of the observed and calculated head coordinates is used as fitness function in order to evaluate the resulting head movement. Thus, the fitness of the *i*th point is given by

$$f_{\text{i}} = \sum_{j=1}^{n} \sqrt{\left(X_{\text{j}} - X'_{\text{j}}\right)^2 + \left(Y_{\text{j}} - Y'_{\text{j}}\right)^2 + \left(Z_{\text{j}} - Z'_{\text{j}}\right)^2} \quad (4)$$

where j is an head position sample (because the points are generated and acquired in a discrete manner); n is the total number of samples originated during the evaluation time; (X',Y',Z') represent the calculated head coordinates with the CPG parameters and (X,Y,Z) represent the offline observed head coordinates. Only head position errors are computed in the fitness function, because we only control three DOFs and as such cannot control head orientation.

In the optimization process each point is evaluated according to its fitness function value. Since we have a population of points the one with the smallest distance is denoted as the best point. Then, in the EM algorithm, each point is directed for a better position, inside of the allowed limits. The search ranges of the head CPG control parameters were set beforehand as shown in Table I for the purpose of efficient learning and according to the limits of the tilt, pan and nod DOFs. Search for optimal parameters is carried out by performing the overall optimization system over a preset number of iterations.

TABLE I
SEARCH RANGES OF CPG PARAMETERS

Parameter	Range	Unit
A_{tilt}	$[0,75]$	$(^o)$
ω_{tilt}	$[1,12]$	(rads^{-1})
O_{tilt}	$\left[-75+\frac{A_{\text{tilt}}}{2}, 0-\frac{A_{\text{tilt}}}{2}\right]$	$(^o)$
A_{pan}	$[0,(88+88)]$	$(^o)$
ω_{pan}	$[1,12]$	(rads^{-1})
O_{pan}	$\left[-88+\frac{A_{\text{tilt}}}{2}, 88-\frac{A_{\text{tilt}}}{2}\right]$	$(^o)$
A_{nod}	$[0,(45+15)]$	$(^o)$
ω_{nod}	$[1,12]$	(rads^{-1})
O_{nod}	$\left[-15+\frac{A_{\text{tilt}}}{2}, 45-\frac{A_{\text{tilt}}}{2}\right]$	$(^o)$

The combinations of amplitude, offset and frequency of each tilt, pan and nod oscillators, that are necessary to generate the desired head movement, form each point of the population. Each coordinate of the point consists in 9 CPG free parameters that span our vector x^i for the optimization, as follows

x^i_1	x^i_2	x^i_3	x^i_4	x^i_5	x^i_6	x^i_7	x^i_8	x^i_9
A_{tilt}	ω_{tilt}	O_{tilt}	A_{pan}	ω_{pan}	O_{pan}	A_{nod}	ω_{nod}	O_{nod}

B. Electromagnetism Algorithm

The EM algorithm starts with a population of randomly generated points from the feasible region. Analogous to electromagnetism, each point is a charged particle that is released to the space. The charge of each point is related to the fitness function value and determines the magnitude of attraction of the point over the population. The better the fitness function value, the higher the magnitude of attraction. The charges are used to find a direction for each point to move in subsequent iterations. The regions that have higher attraction will signal other points to move towards them. In addition, a repulsion mechanism is also introduced to explore new regions for even better solutions. Thus, the EM algorithm comprises 3 procedures: *Initialize* that will run only once in the start of the EM algorithm, *CalcF* and *Move*, these latter running sequentially every iteration. A more detailed explanation of the EM algorithm follows.

Initialize is a procedure that aims to randomly generate a population of points, x^i, from the feasible region, where each

coordinate of a point is assumed to be uniformly distributed between the corresponding upper and lower bounds. Note that in order to guarantee the feasibility of the initial points and all points generated during the search a repair mechanism was implemented. Thus, an infeasible solution is repaired exploring the relations among variables expressed by the box constraints.

Then to compute the fitness function value for all the points in the population, they will be the input of the head movement generation process (see Fig. 4) and by applying forward kinematics the resultant $(X,Y,Z)_{\text{calculated}}$ head coordinates are computed. With them the fitness function value for all the points is calculated and the best point, which is the point with the best fitness function value, is identified.

For the *CalcF* procedure, the Coulomb's law of the electromagnetism theory is used. Thus, the force exerted on a point via other points is inversely proportional to the square of the distance between the points and directly proportional to the product of their charges. Then, we compute the charges of the points according to their fitness function values. The charge of each point determines the power of attraction or repulsion for that point. In this way the points that have better fitness function values possess higher charges. The total force vector exerted on each point is then calculated by adding the individual component forces between any pair of points.

The *Move* procedure uses the total force vector to move the point in the direction of the force by a random step length. The best point is not moved and is carried out to the subsequent iterations. To maintain feasibility, the force exerted on each point is normalized and scaled by the allowed range of movement towards the lower or the upper bound, for each coordinate. To ensure feasibility in this movement algorithm we define the projection of each coordinate of the point to the feasible region, according to the range presented in Table I.

After the EM algorithm, each point should be evaluated in terms of fitness function value, so they should go to the head movement generation process. Then this algorithm is repeated.

C. Experimental Results

The optimization system was implemented in Matlab (Version 6.5) running in an AMD Athlom XP 2400+ 2.00Gz (512 MB of RAM) PC. The system of equations was integrated using the Euler method with 1ms fixed integration steps (similarly to the simulated robotic experiments). The evaluation time for head movement generation is 30s.

In our implementation, the optimization system ends when the number of iterations exceeds 2000 iterations. In this study the number of points in the population was set to 20. When stochastic methods are used to solve problems, the impact of the random number seeds has to be taken into consideration and each optimization process should be run a certain number of times. In this experience we set it to 10.

Table II contains the Best, Mean and standard deviation (SD) values of the solutions found (in terms of fitness function and time) over the 10 runs. We can see that the SD value, in terms of fitness function, is a large value. It denotes that fitness values obtained in each run are not similar. It can be seen by Fig. 5 that shows the evolution of the best (solid line) and mean (dashed line) fitness function value over the 2000 iterations. The best point has a fitness value of 4261 that was achieved at iteration 1150. The best run took 6h18min (CPU time) and each iteration took in average 11.16 seconds.

TABLE II
PERFORMANCE OF EM ALGORITHM IN THE OPTIMIZATION SYSTEM

Best fitness (mm)	Mean fitness (mm)	SD fitness (mm)	Best time (hours)	Mean time (hours)	SD time (hours)
4261	5325.53	870.6349	6.1047	6.5089	0.4120

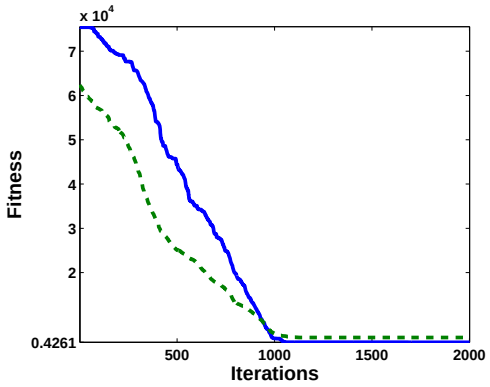


Fig. 5. Best (solid) and mean (dashed) fitness evolution.

Table III shows the tuned CPG parameters representing the best point found, over 2000 iterations, in the 10 runs.

TABLE III
BEST POINT CPG PARAMETERS

Parameter	Value	Unit
A_{tilt}	0.0001	(^{o})
y_{tilt}	-6×10^{-5}	(^{o})
w_{tilt}	6.707	(rads^{-1})
A_{pan}	7.77	(^{o})
y_{pan}	0.072	(^{o})
w_{pan}	2.12	(rads^{-1})
A_{nod}	0.0001	(^{o})
y_{nod}	-1.18	(^{o})
w_{nod}	1	(rads^{-1})

A better understanding of the evolution of the fitness function can be seen in Fig. 6 where the distance between observed and calculated values of the head movement at the beginning and at the end of the optimization system is displayed. We can observe that this distance, in each sample time for time ranging between $t = 5$ and 15s, is smaller at the end of the process. In average, we can also conclude that after 2000 iterations of the optimization system, a reduction

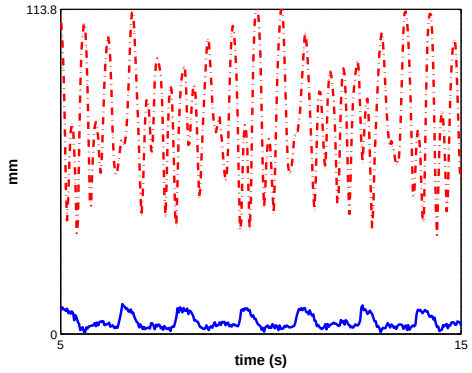


Fig. 6. Distance between observed and calculated values of the head movement at the beginning (dotted line) and at the end (solid line) of the optimization system, for time ranging from 5 to 15 seconds.

of 22,17% of the head movement is verified.

Fig. 7 depicts the time courses of the (X,Y,Z) calculated (solid line) head movement according to the head CPG control parameters of the best solution found. The observed (dotted line) head movement is also illustrated. Table IV gives the maximal movement variation in the (X,Y,Z) coordinates for the calculated and observed movements. We conclude that the generated movements are quite similar in the X coordinate. The calculated movement is quite different in the Y and Z coordinate. This results from the fact that only the pan joint controls movement in the X coordinate, while both the tilt and nod joints control the Y and Z coordinates.

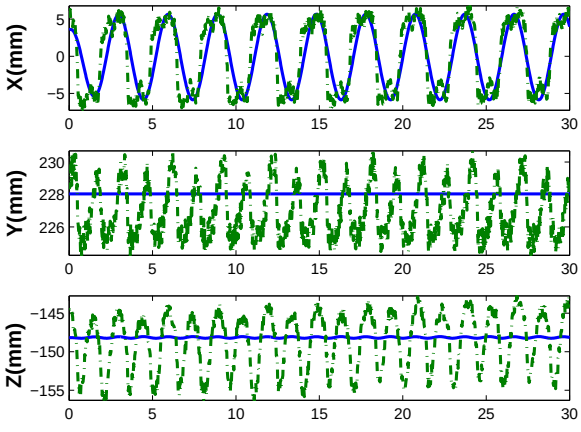


Fig. 7. (X,Y,Z) calculated (solid line) and observed (dotted line) head movement, during 30s, according to the head CPG control parameters from best point on the final of optimization system.

Fig. 8 depicts 3D calculated (solid line) and observed (dotted line) head movement for the best point.

We have also made another experiment, where we have changed the size of the population to 50 points, maintaining the number of 2000 iterations to terminate the process.

TABLE IV
MAXIMAL MOVEMENT VARIATION IN (X,Y,Z)

	Max ΔX (mm)	Max ΔY (mm)	Max ΔZ (mm)
Calculated Movement	11.47	0	0.2
Observed Movement	13.42	5.9	11.3

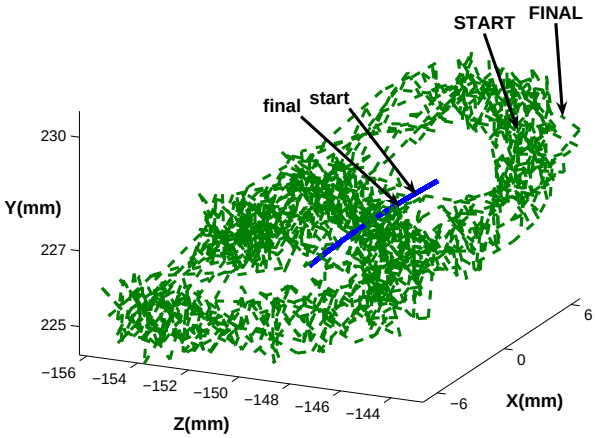


Fig. 8. 3D calculated (solid line) and observed (dotted line) head movement according to the CPG parameters of the 1150th iteration best point. START (FINAL) and start (final) indicate where the observed and calculated movement started (ended), respectively.

Running the optimization system we obtained a best fitness function value of 3991 at iteration 1760.

IV. SIMULATION RESULTS

Our aim was to build a system able to eliminate or reduce the head motion of a robot that walks in the environment. For that, we set a dynamical controller generating trajectories for the head joints such that the final head movement is opposite to the one induced by locomotion.

In this section, we describe the experiment done in a simulated ers-7 AIBO robot using Webots [8]. Webots is a software for the physic simulation of robots based on ODE, an open source physics engine for simulating 3D rigid body dynamics. The model of the AIBO is as close to the real robot as the simulation enable us to be. Thus, we simulate the exact number of DOFs, mass distributions and the visual system.

The ers-7 AIBO dog robot is a 18 DOFs quadruped robot made by Sony. The locomotion controller generates the joint angles of the hip and knee joints in the sagittal plane, that is 8 DOFs of the robot, 2 DOFs in each leg. Only walk gait is generated and tested.

The head controller generates the joint angles of the 3 DOFs: tilt, pan and nod. The other DOFs are not used for the moment, and remain fixed to an appropriately chosen value during the experiments.

The AIBO has a camera built into its head.

At each sensorial cycle (30ms), sensory information is acquired. The dynamics of the CPGs are numerically integrated using the Euler method with a fixed time step of 1ms thus specifying servo positions. Parameters were chosen in order to respect feasibility of the experiment and are given in Table V and VI.

TABLE V

PARAMETER VALUES FOR GENERATING LOCOMOTION

	β	ω (rad s^{-1})	μ_i	$\frac{1}{2\beta\mu_i}$ (s)
Front Limbs	0.1	2.044	6.25	0.8
Hind Limbs	0.025	2.044	25	0.8

TABLE VI

PARAMETER VALUES FOR GENERATING HEAD MOTION

	β	ω (rad s^{-1})	μ_i	$\frac{1}{2\beta\mu_i}$ (s)
tilt	1.25×10^{9}	4.19	2.5×10^{-9}	0.8
pan	0.041	2.09	15.13	0.8
nod	1.25×10^{9}	4.19	2.5×10^{-9}	0.8

Because we are working in a simulated environment, we are able to build a GPS into the AIBO camera, that enable us to verify how the head effectively moves in an external coordinate system. Two simulations are performed: the robot walks during 30s with and without the feedforward solution and its GPS coordinates are recorded. Results are compared for these two simulations. Fig. 9 shows the GPS coordinates for the experiments with (solid line) and without the feedforward solution (dotted line). The overall experiment can be seen in the attached video.

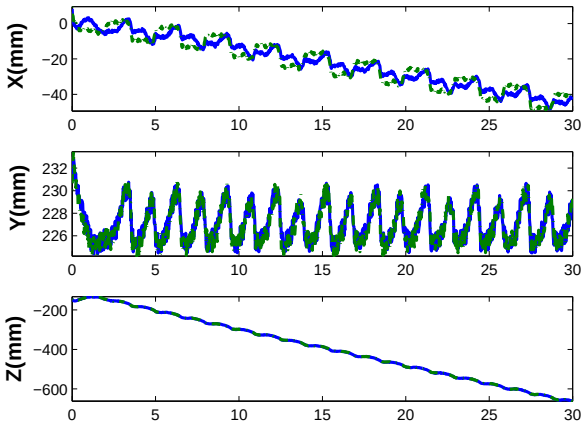


Fig. 9. (X,Y,Z) coordinates of the GPS positioned in the AIBO head when the robot walks during 30s. Solid and dotted lines indicate the experiment in which the feedforward solution is and is not implemented, respectively.

We expect that the proposed feedforward solution minimizes the variation of the GPS coordinates, meaning that the head remains near the same position during the experiment. We observe that the X coordinates of the marker position oscillate less. Note that there is some drift in the X coordinates, meaning the robot slightly deviates towards its side while walking. The observed peaks in the Y coordinate reflect the final stage of the swing phase and the begin of the stance phases of the fore legs, corresponding to an accentuated movement of the robot center of mass. This problem will be addressed in current work, by improving the locomotion controller and take into account balance control [3].

V. CONCLUSIONS AND FUTURE WORKS

In this article, we have addressed head stabilization of a quadruped robot that walks with a walking gait. A locomotion controller based on dynamical systems, CPGs, generates quadruped locomotion. The required head motion needed to eliminate or reduce the head shaking induced by locomotion, is generated by CPGs built-in in the tilt, pan and nod joints. These CPG parameters are tuned by an optimization system. This optimization system combines CPGs and the EM algorithm. As a result, set of parameters obtained by the EM allows to reduce the head movement induced by the locomotion.

Currently, we are using other optimization methods, like the particle swarm optimization, and testing other fitness functions. We will extend this optimization work to address other locomotion related problems, such as: the generation and switch among different gaits according to the sensorial information and the control of locomotion direction. We further plan to extend our current work to online learning of the head movement similarly to [9].

REFERENCES

[1] S Degallier, C P. Santos, L Righetti and A Ijspeert,*Movement Generation using Dynamical Systems: a Drumming Humanoid Robot.* In Humanoids06 IEEE-RAS International Conference on Humanoids Robots, Genova, Italy, December 4-6 2006

[2] Goldberg, D.,*Genetic Algorithms in Search*, Optimization,and Machine Learning, Addison-Wesley, (1989).

[3] Castro, Luiz, Santos, C P; Oliveira, M and Ijspeert, A, *Postural Control on a Quadruped Robot Using Lateral Titl: a Dynamical System Approach*, European Robotics Symposium EUROS 2008, Prague, 2008.

[4] Takizawa, S, Ushida, S, Okatani, T, Deguchi, K.,*2DOF Motion Stabilization of Biped Robot by Gaze Control Strategy.* 2005 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2005). Aug. 2005:3809-3814

[5] Ravi Kaushik, Marek Marcinkiewicz, Jizhong Xiao, Simon Parsons, and Theodore Raphan*Implementation of Bio-Inspired Vestibulo-Ocular Reflex in a Quadrupedal Robot* 2007 IEEE International Conference on Robotics and Automation (ICRA 2007), Roma, Italy, 10-14 April 2007: 4861-4866

[6] Yamada, H.; Mori, M.; Hirose, S.,*Stabilization of the head of an undulating snake-like robot*, it 2007 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2007), San Diego, CA, USA, Oct 29 - Nov 2, 2007 : 3566-3571

[7] Yurak Son, Takuya Kamano, Takashi Yasuno, Takayuki Suzuki,and Hironobu Harada,*Generation of Adaptive Gait Patterns for Quadruped Robot with CPG Network Including Motor Dynamic Model*, Electrical Engineering in Japan, Vol. 155, No. 1, 2006

[8] O. Michel, "Webots: Professional mobile robot simulation," *International Journal of Advanced Robotic Systems*, vol. 1, no. 1, pp. 39–42, 2004.

[9] A. Sproewitz, R. Moeckel, J. Maye, M. Asadpour, and A.J. Ijspeert. *Adaptive locomotion control in modular robotics.* In Workshop on Self-Reconfigurable Robots/Systems and Applications IROS07, pages 81-84, November 2007.

[10] Ludovic Righetti and Auke Jan Ijspeert.*Pattern generators with sensory feedback for the control of quadruped locomotion.* IEEE International Conference on Robotics and Automation ICRA 2008, California, 2008.

[11] S. Grillner. *Neurobiological bases of rhythmic motor acts in vertebrates.* Science, Vol. 228, No. 4696, pp. 143-149, 1985

[12] S.I. Birbil and S. Fang, *An electromagnetism-like mechanism for global optimization*, Journal of Global Optimization 25 (2003), pp. 263–282.

[13] J. Kennedy and R.C. Eberhart, *Particle swarm optimization*, in *IEEE International Conference on Neural Network*, 1995, pp. 1942–1948.

[14] Cristina Santos, Miguel Oliveira, Ana Maria A.C. Rocha and Lino Costa, *Head Motion Stabilization During Quadruped Robot Locomotion: Combining Dynamical Systems and a Genetic Algorithm*, IEEE International Conference on Robotics and Automation (ICRA 2009), May 12-17, Kobe, Japan 2009.

[15] Rocha, Ana Maria A.C. and Fernandes, Edite M.G.P., *Performance profile assessment of electromagnetism-like algorithms for global optimization*, AIP Conference Proceedings, Vol. 1060 (1), Springer-Verlag (2008), (15-18).

[16] F. Patane, C. Laschi, H. Miwa, E. Guglielmelli, P. Dario, and A. Takanishi, *Design and development of a biologically-inspired artificial vestibular system for robot heads*, in IEEE Conference on Intelligent Robots and Systems, IROS'04, Sendai, Japan, September 28 - Ocotber 2, 1317- 1322, 2004.

[17] G. Asuni, G. Teti, C. Laschi, E. Guglielmelli, and P. Dario, *A robotic head neuro-controller based on biologically-inspired neural methods*, in IEEE Conference on Robotics and Automation, ICRA'05, Barcelona, Spain, April 18-22, 2005.

[18] R. Cromwell, J. Schurter, Scott Shelton, S. Vora, *Head stabilization strategies in the sagittal plane during locomotor tasks*, Physiotherapy Research International, Whurr Publishers Ltd, 9(1), pp.3342, 2004

2nd International Workshop on Evolutionary and Reinforcement Learning for Autonomous Robot Systems (ERLARS 2009)
St. Louis, Missouri, USA, October 15 2009

Combination of Reinforcement Learning and Neural Networks

Anastasia Noglik and Josef Pauli

***Abstract*— Reinforcement Learning has been already successfully applied to the problem of a target oriented robot navigation. However one of its drawbacks is the usually observed low convergence rate of the learning progress. To weaken this disadvantage the present work suggests to use a control function which is represented by a neural network to prevent the agent of collisions with objects if a dangerous situation has been recognized. This leads to a better performance of the learning process and to a reduced number of collisions. Since an evolutionary algorithm is used to develop the neural network, little effort is necessary to train the network.**

I. Introduction

Reinforcement Learning (RL) is a sub-area of machine learning and describes methods for solving a class of problems. Inspired by psychological theory, RL is concerned with how an agent ought to take actions in an environment so as to maximize some notion of long-term reward. Reinforcement Learning algorithms attempt to find a policy that maps states of the world to the actions the agent ought to take in those states.

In the last few decades a lot of interesting work have been done in this area. Several algorithms have been developed to solve various problems including robot control, elevator scheduling or telecommunications. RL was also successfully applied in the development of game strategies of backgammon and chess.

However, similar to most other algorithms there are also some main drawbacks, which have to be taken into account. One common issue is the poor convergence of these approaches [9]. The reason of it usually lies in the high dimensionality of the corresponding policy/value function. Also a large number of episodes is needed to find a suitable strategy. In case of planning and learning Sutton et. al. addressed this problem by developing the Dyna-Q algorithm [11]. Here the policy/value function is influenced based not only on the real experience of an agent, but also on its simulated experience, which is produced through the model-based processes including in this algorithm.

In the work presented here, this problem is addressed by incorporating additional context knowledge about the application domain. This additional knowledge can highly improve the convergence of the reinforcement method and usually can be easily extracted from the corresponding domain. However a lot of the existing algorithms discard this kind of knowledge or do not provide any way to incorporate it in the learning process.

There are different forms of context knowledge. One example is knowledge which is extracted from the information which is already used in the learning process of the agent. This context knowledge can be provided to the learning process in form of e.g. a heuristic function, see [2], [5]. An other form of context knowledge is extracted from information about the environment which is not used in the learning process. This context knowledge is used in a control function for the agent.

The functionality of the proposed method is illustrated in case of a goal oriented navigation problem. Here the aim of the agent is to learn a short and collision free path from a given starting state to a target point through a complex environment. The agent has two sources of information available. The idiothetic source is used to specify the agent's position in a state space. The allothetic source, in this work the infrared sensors, is used to extract the required additional knowledge. This knowledge serves as a control function to save the agent from dangerous situations, like unintended bumping into the wall. In this method the control function is approximated using an evolutionary gained artificial neural network (ANN) (NEAT Method [10]).

An example for the combination of both methods is the NEAT+Q algorithm, [13]. This algorithm combines the power of RL methods with the ability of NEAT to learn effective representations. The NEAT+Q algorithm is an extension of the NEAT method using RL. The algorithm which is proposed in the present work is an extension of RL using ANN which is gained using NEAT.

The incorporation of the control function into the learning process involves several interesting aspects. First of all the two information sources have to be fused together. Secondly the question of an appropriate discretization of the action space is raised. The way the control function influences the learning process is described in more detail in the remainder of this paper.

The present work investigates the performance of the proposed method by varying the influence strength of the control function on the learning process. The investigations are performed with a Scorpion robot from Evolution Robotics with a certain mechanical design and infrared sensors. Therefore the results are not valid for all models of the robot. But the used methods for investigation and determination of optimized parameters are applicable to all types of agents.

II. Background

In the present work the linear gradient Sarsa(λ)-Algorithm is used as a basic method [7]. Tile Coding is selected as discretization method for the state space. The action-state function is approximated by a linear function. The proposed extension uses local information as context knowledge which is different from the global information in the state of the agent. Hence the basic method can be replaced by

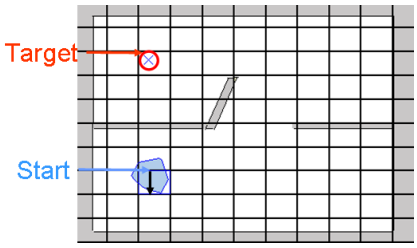


Fig. 1. Simple environment 1 for the determination of optimal parameter sets

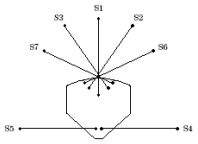


Fig. 2. Model of the ERSP Scorpion robot of the company Evolution Robotics

other Reinforcement Learning methods. The state space can be discretized or represented by different approaches.

A. Sarsa(λ) Algorithm

Reinforcement Learning is a synonym of learning by interaction. Fully adaptive control algorithms which learn both by observation and *trial-and-error* are a promising approach in machine learning. RL is defined as the learning of a mapping from situations S (S is the set of possible states) to actions A (A is the set of actions) so as to maximize an accumulated scalar reward or reinforcement signal r. Rewards r are gratifications or punishments and are a means of informing the agent about the target.
An action-state value function $Q^\pi(s,a)$ is defined as the value of taking action $a \in A(s)$ in the state $s \in S$ under a policy π and provides a measure for the quality of the (s,a)-pair. The function is defined as an expected return of future rewards:

$$Q^\pi(s,a) = E_\pi \left\{ \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s, a_t = a \right\}$$

where γ is a discount factor [12].
Sarsa(λ) is an on-policy learning method, which continually estimates Q^π for the behavior policy π [12]. Learning is an iterative process. In the beginning the agent owns a random suboptimal value function and strategy. The basic learning step updates a single Q-value. a_{t+1} is obtained from the ϵ-greedy policy that uses values from the estimated Q-function. Subsequently the Q-value for (s,a) is updated as follows:

$$Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha[r_{t+1} + \gamma Q(s_{t+1},a_{t+1}) - Q(s_t,a_t)] \quad (1)$$

where α is the learning rate. The Q-function is approximated by a linear function.

B. Tile Coding, Function Approximation, State Space, Action Space

1) Tile Coding: The success of RL with large continuous state spaces depends on an effective Q-function approximation. Of the many function approximation schemes proposed, tile coding offers an empirically successful balance among representational power, computational cost, ease of use and it has been widely adopted in recent RL work[8].
To generalize the state representation the table of Q-values can be approximated using a representation of the state space with tile coding [1]. Tile coding is a form of coarse coding. In tile coding the receptive fields of features are grouped into exhaustive partitions of the input space. Each such partition is called tiling, and each element of the tiling is called tile. Each tile is the receptive field for one binary feature. According to the chosen discretization of the state space, one state is represented by one feature only.

2) Function Approximation: The Sarsa(λ) algorithm with function approximation was first explored in [7]. The Q-function is approximated in the present work by a linear function. The representation of the Q-function by a parameter vector is defined as:

$$Q(s,a) := \sum_{i \in \mathcal{F}_s} \theta_a(i) \cdot \phi_s(i), \; \mathcal{F}_s \leftarrow \text{ set of features present in } s$$

The respective Q-value corresponds then to the value of the currently involved binary feature.

3) State Space: The three axes of the state space in the defined navigation problem correspond to the horizontal x and vertical position y of the agent and its orientation o. The discretization of the state space significantly influences the performance of the learning process. The discretization $N_x \times N_y \times N_o$ specifies the number of feature centers along the positions and orientation axes respectively. The x,y-plane was discretized according to the robot size. Each rectangle of this plane was as big as it does not exceed the size of the robot. An example of such a discretization can be seen in figure 1.

4) Action Space: An action is a vector which consists of two scalar values forward velocity and rotation velocity (v,β), where the unit of v is centimeters per minute and the unit of β is degrees per minute. The action space of the to be learned Q-Function consists of three discrete actions: forward movement $(v := 10, \beta := 0)$, turn left $(v := 0, \beta := 15°)$, turn right $(v := 0, \beta := -15°)$. The control function which is represented by an ANN gives only real values, $(v,\beta) \in [0,10] \times [-90°, 90°]$.

C. Robot and Intelligent Agent

In the present work the terms mobile robot and intelligent agent are used synonymously. The experiments with the agent are performed as a numerical simulation. But as input to the agent true sensor values are used which have been measured previously with the real robot. This results in a true-to-reality simulation [4]. That is the reason why the intelligent agent in the simulation is called robot. The model of the mobile Scorpion robot (agent) used in the simulation is

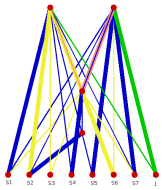


Fig. 3. Example of the automatically developed ANN

equipped with seven infrared sensors $(s_1, \ldots s_7)$, see figure 2.

D. NEAT Method

Neuro Evolution of Augmenting Topologies (NEAT) applies the evolutionary approach for the development of the topology as well as for the determination of the weights of an Artificial Neural Network (ANN). A NEAT is used in the simulation for the training of an ANN, which is able to take over the control of the robot. Each ANN is an individuum with the same input (7 sensors and 1 bias node) and output (2 control commands for rotation and forward velocity) layers. NEAT has been selected due to a high adaptation ability and relatively simple application. An extensive description of the system provides the work [10].
The counter of the covered angle from the given center in the pre-defined direction has been selected as fitness function. Examples for more fitness functions are described in [6] and [3]. Different scenarios have been used in the simulation during the training phase to train the required skills. For example the agent learns a fast reaction to suddenly occurring objects when its starting point is very close to a wall.
The ANN which represents the control function is a kind of reactive behavior pattern for robots. The ANN encodes a direct mapping of the sensor values on the control commands. The resulting ANN intervenes during dangerous situations to prevent collisions with objects.

E. Control Function appriximated by ANN

An example of an ANN which has been trained with the above described method, fitness function and scenarios is depicted in figure 3. This ANN has been also tested successfully in reality. Yellow marked edges have a positive weighting, blue marked edges have a negative weighting. The thicker the line is, the higher is the absolute value of the weight for the shown connection. The green lines are the bias connections. The red connections mean a feedback.

III. Concept of extended method

To weaken Reinforcement Learning disadvantages the present work suggests to use a control function which is represented by a neural network to prevent the agent of collisions with objects if a dangerous situation has been recognized. This leads to a better performance of the learning process and to a reduced number of collisions.
To extend the basic method by control function, the method has to be modified in such way, that the learning process is affected as little as necessary and the benefit is as high as possible. On the one hand RL should not notice the extension, and on the other hand the control function shall protect the robot from collisions as good as possible.
Such a control function can not only be represented with an ANN which is achieved with an evolutionary approach. But the control function can can also be gained e.g. by using control theory or fuzzy logic. Such a control function has to be generated only once and can be used as extension of the basic method for different problems. Therefore it is not necessary to consider the time which is needed for generation of the control function in the overall time which is needed for the solution of the problem.

A. Problems and Solutions

The robot receives sensor values which are in the range of $[0; 80]$. The directions of the robot's sensors are depicted in figure 2. A collision with the environment is signaled, if a sensor measures the minimum value below a predefined threshold, so $s_{min} = min_{i=1,\ldots,7} s_i < s_{limit}$, whereas here $s_{limit} := 15$.
The basic and extended methods of the learning process have the following sequence: After the lowest sensor value s_{min} falls below the given limit of s_{limit} the collision is signaled and the agent is displaced to the starting point. The eligibility traces are set to zero and the episode is continued.
The two important questions are: 1. When is the situation as dangerous, so that a neural network has to take over the control of the robot? 2. When does this intervention disturb dramatically? These questions have strongly influenced the development.
The balance between the intervention in the learning process and the benefit of the control function has to be right so that the method can be used. To resolve the previously mentioned dilemma, several experiments have been accomplished to find an appropriate pair of the following parameters: $s_{Critical}$ and $\beta_{Discretisation}$, see figure 4 rows 10 and 14. Thus the control function is only used, if $s_{min} \in [s_{limit}; s_{Critical}]$.
The learning process with RL has global information as basis. Each state will be extracted from odometry data and represented in the coordinate system of the starting point. The answer of the neural network is determined based on local information. The artificial neural network provides the answer to the sensor values. The result is a kind of data fusion. The control function is approximated by the neural network with sensor values (local information) as input. The local information can be different although the state from the agent's perspective is the same. The reasons are the use of different kinds of information, the discretization of the state space and the inaccuracy of the sensor values, for which the variable $\beta_{Discretisation}$ is responsible.

B. ANN+Sarsa(λ)-Algorithm

The extension of the basic Sarsa(λ) algorithm is depicted in figure 4. The difference is the addition of rows 09 to 18 and 32 to 34. This block is used, if the minimum sensor value $s_{min} = min_i s_1, \ldots, s_7$ goes below the specified $s_{Critical}$. In row 11 the answer of the ANN on the sensor values

$(s_1, \ldots, s_7)$ is taken. The rows $14-18$ are a kind of projection of the continuous action (v, β) in the discrete action space $\{forwardmovement, turnright, turnleft\}$. In this case it is a continuous action space, but it is projected into an ordinary discrete action space by means of the parameter $|\beta_{Discretisation}$.
It has been decided to use "limited learning" instead of an ANN intervention, see row 13 in figure 4. The Temporal Difference (TD)-error value δ_t is set to a constant value r_{Reflex}. That means, if the ANN takes over the control, the Q-function for the current state and the projected action is determined explicitly independent on the global target and not learned, compare Eq. (1). Hence the agent has no possibility to learn in the critical points.

C. Parameterization of the Extended Method

Two variables are responsible for the right balance between the intervention into the learning process and the improvement by using an ANN. The parameter $s_{Critical}$ determines when the ANN is taking over the control. The parameter $\beta_{Discretisation}$ is responsible for the projection of the action provided by the ANN in the discrete action space. r_{Reflex} outputs always the same value, so that the agent gets a certain value at the critical points. $r_{Reflex} = -5$ is more painful for the agent than each step ($r_{step} = -1$), but it is not worse than a reward of $r_{collision} = -20$. The gratification reward is the value $r_{target} = +1000$.
The tests have been performed with the following parameters: $\alpha = 0.2$, $\epsilon = 0.1$, $\gamma = 0.9$, $\lambda = 0.8$. This is a reasonable parametrization. No procedure for the selection of these learning parameters has been used.

IV. Evaluation Strategy

The performance of the basic method will be compared to that of the proposed method by applying both methods to the navigation problem in the three previously mentioned environments. The discretization of all environments was chosen to be standard Tile Coding (TC). Also the used reward model was equal in all conducted experiments.
The balance between the intervention in the learning process and the benefit of the control function has to be right so that the method can be used. To resolve the previously mentioned problem, several experiments have been accomplished to find an appropriate pair of the following parameters: $s_{Critical}$ and $\beta_{Discretisation}$. Thereby two criteria have been used to compare the performance of the two methods.

The first criterion is the learning progress defined as:

$$LearningProgress_{BE} := \frac{1}{N} \sum_{i=1}^{N} \bar{step}_i$$

where $\bar{step}_i$ is the average number of steps in the i-th episode. According to the definition smaller values of this criterion indicate a better learning process. In each episode in total 30 tests have been conducted. In case of the first two environments N was chosen to be 250. In the third environment it is set to $N = 500$. The computed values of this criterion can be found in the next section in tables I and IV.
The second criterion is the average number of wall contacts:

$$\bar{Wallcontacts} = \sum_{j=1}^{N} \bar{Wallcontacts}_j$$

where $\bar{Wallcontacts}_j$ is the average number of wall contacts during one episode, in which in total 30 tests have been conducted. This ensures a certain statistical significance. Again in case of the first two environments N was set to 250, and in case of the third environment to 500. This criterion was chosen, because a small number of wall contacts is important in case of a navigation problem. The computed values of this criterion can be found in tables II and V.
To evaluate the influence of the ANN on the learning progress of the extended method the following third criterion has been developed. *The average number of network applications* depends on the configuration of the parameters and is listed in table III.
Relative deviations emphasize the positive or negative influence of the extension with an ANN in comparison to the standard algorithm. The deviation value is calculated with $Deviation := (1 - \frac{(number_{parameters})}{number_{standard}}) * 100\%$. $LearningProgress$ as well as $WallContacts$ are inserted instead of $number$ for each parameter combination.

V. Results

In the following the results for two simple environments, see figure 1 and 5, and one more complex environment are presented. A part of the complex environment is shown in figure 6. In all experiments one tiling has been used. In environment 1 and 2 the tiling has $10 \times 10 \times 24$ tiles, in environment 3 the tiling has $49 \times 39 \times 24$ tiles.
In table I the learning progress is listed for different parameter sets, $s_{Critical} \in \{15, 20, 25, 30, 35\}$ and $\beta_{Discretisation} \in \{1, 2, \ldots 9\}$. In the lower part of table I relative deviations are listed in relation to the basic method. The tests have been performed in environment 1.
A value of 25 of the parameter $s_{Critical}$ results in a positive impact for all values of the parameter $\beta_{Discretisation}$. The best combination for the tested parameters $(s_{Critical}, \beta_{Discretisation})$ is $(25, 6)$. This combination results in an improvement of more than 20% for the selected criterion, the learning progress, in comparison to the basic method.
The results for the second criterion, the average number of wall contacts, are listed together with the relative deviations in table II. The parameter value $s_{Critical} = 30$ shows the best results with improvements between 69% and 76%. But the best parameter combination for the learning progress $(25, 6)$ provides also a relative high reduction of wall contacts per episode of 59 %. The improvement of the learning progress is much higher for the parameter combination $(25, 6)$ than for any parameter combination with a $s_{Critical}$ of 30 with

TABLE I

LEARNING PROGRESS (AVERAGE NUMBER OF STEPS) FOR ENVIRONMENT 1 FOR DIFFERENT PARAMETER SETS. COLUMNS: VARYING PARAMETER $\beta_{Discretisation}$ BETWEEN 1 AND 9. ROWS: VARYING PARAMETER $s_{Critical}$ BETWEEN 15 AND 35. LOWER PART: RELATIVE DEVIATION TO THE BASIC METHOD

	1	2	3	4	5	6	7	8	9
standard	386.31	386.31	386.31	386.31	386.31	386.31	386.31	386.31	386.31
15	404.57	423.48	391.76	377.32	376.20	374.00	356.54	405.55	385.69
20	458.71	406.55	417.48	451.70	418.66	426.97	446.28	422.27	430.91
25	362.54	344.69	325.13	333.65	314.66	301.91	336.54	334.89	313.72
30	415.17	385.20	417.87	411.52	369.50	377.83	386.47	380.90	365.78
35	1190.59	690.74	794.55	783.61	1045.50	1046.23	1795.96	696.66	790.50
standard	0%	0%	0%	0%	0%	0%	0%	0%	0%
15	-4.72%	-9.62%	-1.41%	2.32%	2.61%	3.18%	7.7%	-4.97%	0.15%
20	-18.74%	-5.23%	-8.06%	-16.92%	-8.37%	-10.52%	-15.52%	-9.3%	-11.54%
25	6.15%	10.77%	15.83%	13.63%	18.54%	21.84%	12.88%	13.31%	18.78%
30	-7.47%	0.28%	-8.16%	-6.52%	4.35%	2.19%	-0.04%	1.4%	5.31%
35	-208.19%	-78.8%	-105.67%	-102.84%	-170.63%	-170.82%	-364.89%	-80.33%	-104.62%

TABLE II

NUMBER OF COLLISIONS WITH OBJECTS (WALL CONTACTS) FOR ENVIRONMENT 1 FOR DIFFERENT PARAMETER SETS. COLUMNS: VARYING PARAMETER $\beta_{Discretisation}$ BETWEEN 1 AND 9. ROWS: VARYING PARAMETER $s_{Critical}$ BETWEEN 15 AND 35. LOWER PART: RELATIVE DEVIATION TO THE BASIC METHOD

	1	2	3	4	5	6	7	8	9
standard	3.91	3.91	3.91	3.91	3.91	3.91	3.91	3.91	3.91
15	4.7	4.85	4.33	4.21	4.18	4.17	4.04	4.33	4.18
20	4.94	4.43	4.6	5.02	4.69	4.65	4.94	4.73	4.63
25	1.91	1.8	1.65	1.75	1.56	1.6	1.72	1.69	1.58
30	1.19	1.03	1.14	1.07	0.95	0.97	1.01	1.01	0.93
35	1.39	1.16	1.21	1.3	1.38	1.28	1.73	1.16	1.32
standard	0%	0%	0%	0%	0%	0%	0%	0%	0%
15	-20.02%	-23.83%	-10.7%	-7.65%	-6.85%	-6.43%	-3.16%	-10.67%	-6.89%
20	-26.31%	-13.19%	-17.61%	-28.13%	-19.87%	-18.78%	-26.13%	-20.77%	-18.31%
25	51.19%	53.98%	57.73%	55.24%	60.08%	59.02%	55.91%	56.73%	59.55%
30	69.57%	73.7%	70.65%	72.65%	75.51%	75.08%	74.13%	73.99%	76.02%
35	64.31%	70.23%	68.88%	66.59%	64.66%	67.11%	55.79%	70.37%	66.23%

TABLE III

NUMBER OF ANN APPLICATIONS FOR ENVIRONMENT 1 FOR DIFFERENT PARAMETER SETS. COLUMNS: VARYING PARAMETER $\beta_{Discretisation}$ BETWEEN 1 AND 9. ROWS: VARYING PARAMETER $s_{Critical}$ BETWEEN 15 AND 35.

	1	2	3	4	5	6	7	8	9
standard	0	0	0	0	0	0	0	0	0
15	2.33	2.37	2.18	2.1	2.2	2.15	2	2.33	2.2
20	16.78	15.14	15.48	16.61	15.38	15.87	16.16	15.42	15.88
25	33.3	31.49	30.75	30.33	29.12	29.38	31.25	31.14	28.92
30	53.99	50.75	54.39	52.81	48.46	49.3	50.04	49.42	48.14
35	103.6	94.19	94.63	94.7	100	100.88	115.84	91.95	97.19

TABLE IV

LEARNING PROGRESS (AVERAGE NUMBER OF STEPS) FOR ENVIRONMENT 2 FOR DIFFERENT PARAMETER SETS. COLUMNS: VARYING PARAMETER $\beta_{Discretisation}$ BETWEEN 1 AND 9. ROWS: VARYING PARAMETER $s_{Critical}$ BETWEEN 20 AND 30. LOWER PART: RELATIVE DEVIATION TO THE BASIC METHOD

	1	2	3	4	5	6	7	8	9
standard	444.94	444.94	444.94	444.94	444.94	444.94	444.94	444.94	444.94
20	473.83	482.11	456.29	535.06	495.75	477.24	523.53	484.17	487.27
25	439.58	431.54	417.76	418.8	431.89	470.48	474.77	470.02	446.54
30	574.74	560.71	461.58	650.46	897	872.91	555.76	601.73	863.05
standard	0%	0%	0%	0%	0%	0%	0%	0%	0%
20	-6.49%	-8.35%	-2.55%	-20.25%	-11.42%	-7.25%	-17.66%	-8.81%	-9.51%
25	1.2%	3.01%	6.1%	5.87%	2.93%	-5.73%	-6.7%	-5.63%	-0.36%
30	-29.17%	-26.01%	-3.74%	-46.19%	-101.6%	-96.18%	-24.9%	-35.23%	-93.97%

TABLE V

NUMBER OF COLLISIONS WITH OBJECTS (WALL CONTACTS) FOR ENVIRONMENT 2 FOR DIFFERENT PARAMETER SETS. COLUMNS: VARYING PARAMETER $\beta_{Discretisation}$ BETWEEN 1 AND 9. ROWS: VARYING PARAMETER $s_{Critical}$ BETWEEN 20 AND 30. LOWER PART: RELATIVE DEVIATION TO THE BASIC METHOD

	1	2	3	4	5	6	7	8	9
standard	4.35	4.35	4.35	4.35	4.35	4.35	4.35	4.35	4.35
20	6.02	5.85	5.76	6.59	6.12	5.41	6.25	5.95	5.86
25	3.81	3.6	3.25	3.33	3.11	3.92	3.72	3.66	3.2
30	1.97	1.99	1.83	2.05	2.02	2.07	1.82	2.17	2.07
standard	0%	0%	0%	0%	0%	0%	0%	0%	0%
20	-38.41%	-34.54%	-32.34%	-51.43%	-40.63%	-24.27%	-43.64%	-36.68%	-34.69%
25	12.4%	17.15%	25.25%	23.49%	28.56%	9.77%	14.46%	15.89%	26.47%
30	54.65%	54.25%	57.83%	52.7%	53.46%	52.37%	58.19%	50%	52.35%

at the same time a similar reduction of wall contacts.

Another influence criterion is shown in table III. The number of ANN applications for different parameter sets. The correlation is not surprising. The higher the parameter $s_{Critical}$ is, the more often the ANN is used. The parameter $\beta_{Discretisation}$ has no significant impact on the number of ANN applications.

To verify the general correlations the tests are performed for different parameter sets in another environment, see figure 5. The results for the learning progress are listed in table IV and the results for the number of wall contacts are listed in table V. A smaller parameter range can be chosen, since for the parameter $s_{Critical}$ the values 20 and 35 showed negative results in previous experiments. A parameter of $s_{Critical} = 25$ showed also in this environment 2 the best results for the learning progress. But the best value for the parameter $\beta_{Discretisation}$ has shifted. The reason could be the high number of angles in this environment. A finer projection is needed in the discrete action space. For the parameter set $(25, 5)$ the learning progress has been improved by 3%. But for the parameter set $(25, 6)$ the agent showed a learning progress which is worse by 6%. The number of wall contacts is also higher in comparison to environment 1. But there is still an improvement of almost 30% using the parameter set $(25, 5)$, see table V. To understand the influence of the different parameter values on the learning progress, several corresponding curves have been plotted in figure 7. The red curve shows the learning progress using the basic method in the environment shown in figure 1. Starting with the initial value of 10000, maximal possible steps in one episode, it decreases steeply down to a value of 63 after the 20-th episode. Between the 20-th and the 250-th episode there is almost no improvement. The green, blue and violet curves show the learning progress of the proposed method for different parameter values. The green curve with the parameters $s_{Critical} = 20$ and $\beta_{Discretisation} = 5$ shows a similar development to that of the basic method, however in contrast to the basic method the number of steps starts to decrease earlier. The similarity evolves from the fact that because of the chosen $s_{Critical}$ the impact of the ANN on the learning process is too small. After its activation it does not have much elbowroom to move the robot away from the wall. As mentioned above, if the distance to the wall is smaller than s_{limit}, so $s_{min} < s_{limit} := 15$ a collision is signaled and the agent is placed to the starting point again. However, other parameter sets of the proposed method lead to different developments of the learning curves. With the greater values of the parameter $s_{Critical}$, the activated ANN if $s_{min} < s_{Critical}$ has much more impact on the learning process. As shown by the blue and violet curve the robot already reaches its target during the first episodes. Moreover it needs from the beginning less than 4000 steps to reach the target.

Considering this figure a general trend can be identified. The higher the value of the parameter $s_{Critical}$ is, the less steps are needed for the agent to reach its target during the first episodes. The overall shape of the corresponding curves also becomes more complanate. Thus an appropriate impact of the ANN on the learning process leads to a 20% better learning progress. But the disadvantage of the proposed method can also be observed in figure 7. The minimum number of required steps in one episode is reached later than using the basic method. The conclusion is, that the robot learns faster in the first few episodes. But it learns slower in the subsequent episodes.

In figure 8 the number of wall contacts is plotted over the episodes. Again the different curves correspond to the results of the basic method and the proposed method with different parameter sets. After the 15th episode all curves show a similar development. However in the first episode the basic method and the proposed method parameterized with $s_{Critical} = 20$ and $\beta_{Discretisation} = 5$ produce more than 200 wall contacts. This greatly effects the agents health. Much better are the results presented by the blue and violet curves. By applying the corresponding methods to the real robot, its probability to survive is much greater, since less wall contacts have to be endured.

The correlation between the number of ANN applications and the number of wall contacts depending on the number of episode is shown in figure 9 for a parameter combination of ($s_{Critical} = 25$, $\beta_{Discretisation} = 5$). Both curves have been scaled to the same level to point out the similar progress. The reason is that the number of situations, in which the ANN does not help is fixed. The result is a collision with a wall

01 Initialize $\vec{\theta}$ arbitrarily
02 Repeat (for each episode):
03 $\vec{e} = \vec{0}$
04 $s, a \leftarrow$ initial state and action of episode
05 $\mathcal{F}_s \leftarrow$ set of features present in s
06 Repeat (for each step of episode):
07 For all $i \in \mathcal{F}_s$:
08 $\vec{e}_a(i) \leftarrow \vec{e}_a(i) + \phi_s(i)$
09 $s_{min} \leftarrow$ MIN$(s_1, s_2, ...s_7)$
10 if $s_{min} < s_{Critical}$
11 $a' := (\beta, v) \longleftarrow$ ANN$(s_1, s_2, ...s_7)$
12 Take action a', observe reward r, and next state, s'
13 $\delta \leftarrow r_{Reflex}$
14 if $|\beta| < \beta_{Discretisation}$
15 $a \leftarrow$ forward movement
16 else
17 if $\beta > 0$
$a \leftarrow$ turn right
18 else $a \leftarrow$ turn left
19 else
20 Take action a, observe reward r, and next state, s'
21 $\delta \leftarrow r - \sum_{i \in \mathcal{F}_s} \theta_a(i) \cdot \phi_s(i)$
22 With probability $1 - \epsilon$:
23 For all $a \in \mathcal{A}(s')$:
24 $Q_a \leftarrow \sum_{i \in \mathcal{F}_{s'}} \theta_a(i) \phi_{s'}(i)$
25 $a \leftarrow \arg\max_a Q_a$
26 else
27 $a \leftarrow$ a random action $\in \mathcal{A}(s)$
28 $\mathcal{F}_{s'} \leftarrow$ set of features present in s'
29 $Q_a \leftarrow \sum_{i \in \mathcal{F}_{s'}} \theta_a(i) \phi_{s'}(i)$
30 $\delta \leftarrow \delta + \gamma Q_a$
31 $\vec{\theta} \leftarrow \vec{\theta} + \alpha \delta \vec{e}$
32 if $s_{min} < s_{Critical}$
33 $\vec{e} \leftarrow \gamma \lambda \vec{e}$
34 else $\vec{e} \leftarrow \vec{0}$

33 $s \leftarrow s'$
34 until s is target

Fig. 4. Algorithm of the proposed ANN+Sarsa(λ)-Method with ϵ-greedy policy. The difference is the addition of rows 09 to 18 and 32 to 34. In row 11 the ANN gives the answer on the sensor values $(s_1, \ldots, s_7)$. The rows $14 - 18$ are a kind of projection of the continuous action (v, β) on the discrete action space $\{forwardmovement, turnright, turnleft\}$.

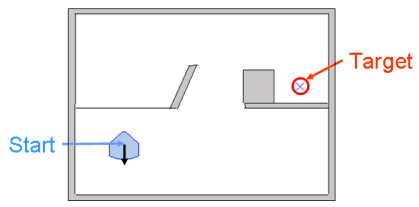


Fig. 5. Simple environment 2 for the verification of the optimal parameter sets

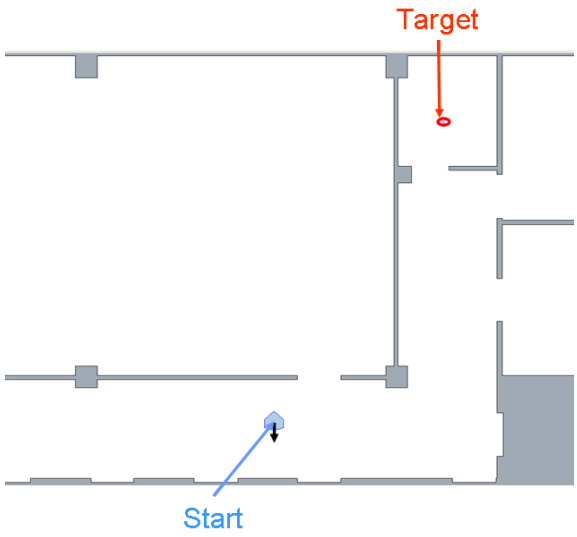


Fig. 6. Complex environment 3 for verification of the optimal parameter sets

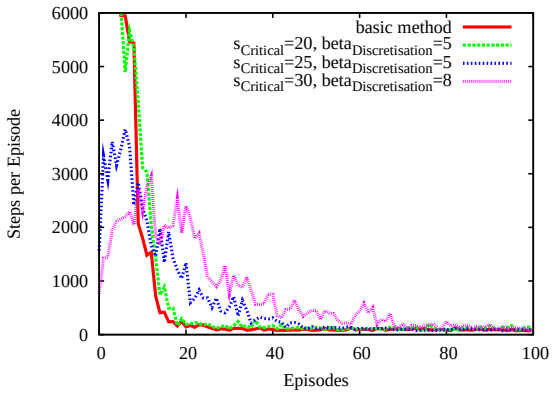


Fig. 7. Learning curves for the proposed method with 3 different parameter sets and the basic method in environment 1

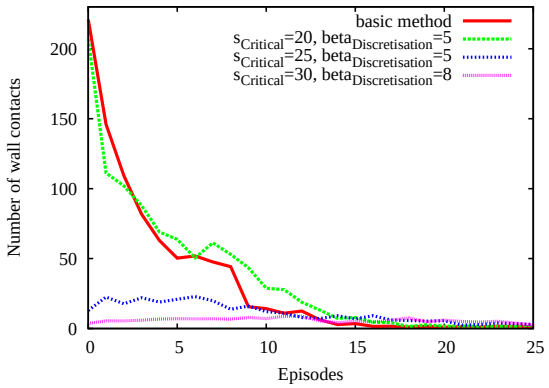


Fig. 8. Number of wall contacts resulted by the basic method and the proposed method with different parameter sets in environment 1

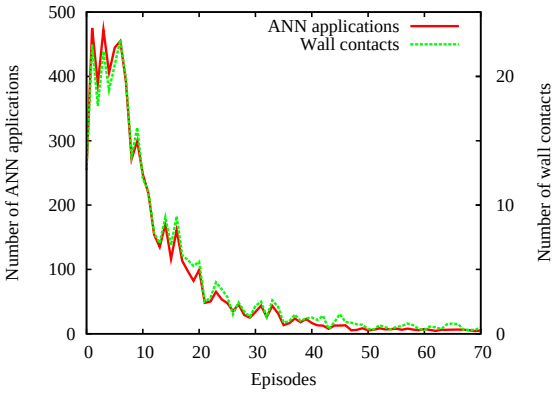


Fig. 9. Number of ANN applications and number of wall contacts depending on the episode for the parameter set $s_{Critical} = 25$, $\beta_{Discretisation} = 5$ in environment 1

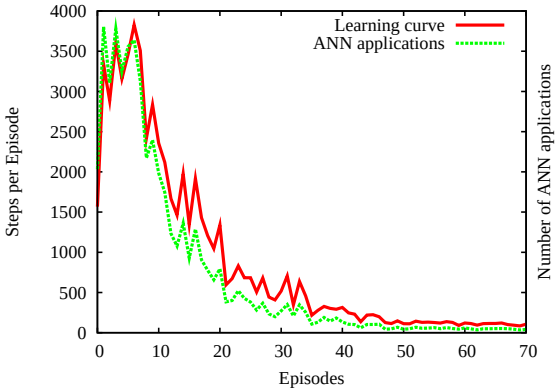


Fig. 10. Number of required steps per episode and number of ANN applications depending on the episode for the parameter set $s_{Critical} = 25$, $\beta_{Discretisation} = 5$ in environment 1

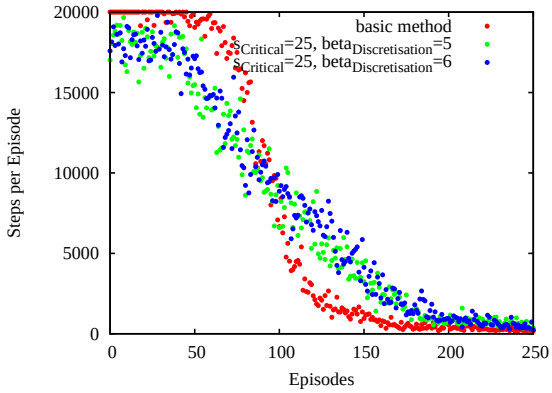


Fig. 11. Number of required steps to target depending on the episode in environment 3

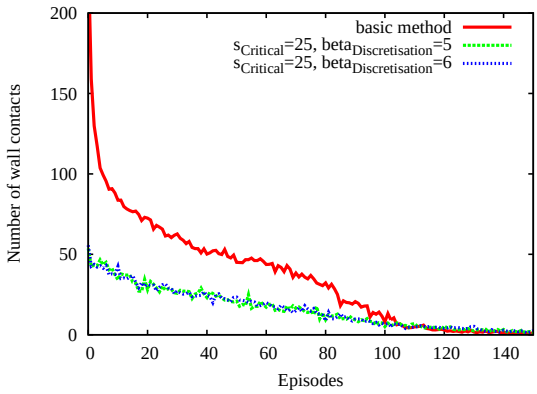


Fig. 12. Number of wall contacts depeding on the episode in environment 3

and a similar profile of the curves. The learning curve and the progress of the number of ANN applications are shown in figure 10. The profile of both curves is also similar. The more seldom dangerous situations occur during the episodes, the more seldom the ANN is used, and the faster the agent reached his target. The progress of the three values number of steps to target, number of ANN applications and number of wall contacts is very similar. The above described results are confirmed for the more complex environment 3, see figure 11. The learning progress is shown for the basic method and two sets of parameters $s_{Critical} = 25, \beta_{Discretisation} = 6$ and $s_{Critical} = 25, \beta_{Discretisation} = 5$. The optimal parameters have been determined in previous experiments.

The agent with the proposed method reaches the target already in the first episode like in environment 1. The agent with the basic method reaches the target beginning with the 80th episode. But then it learns faster. The average number of successful target achievements is listed in table VI. The proposed method enables the agent to reach the target 430 and 439 times respectively out of 500 trials (episodes). The agent with the basic method reaches the target 417 times averaged over 30 experiments. The progress for the number of wall contacts from one episode to the next one in environment 3 is similar to the simple environment 1, see figure 12.

VI. Conclusion

A new method has been proposed which allows the integration of additional context knowledge in the learning process. The proposed method has been tested for a navigation problem. The extension with context knowledge results in a faster learning. The method uses a neural network additionally to the standard Sarsa(λ) algorithm to avoid agent collisions with the surrounding obstacles. The neural network starts working only if the actual state of an agent was recognized as dangerous. In the remaining situations the usual Sarsa(λ) algorithm is used.

Several parameters are used to control the influence of the

TABLE VI
NUMBER OF SUCCESSFUL TARGET ACHIEVEMENTS IN 500 TRIALS (EPISODES), AVERAGE OUT OF 30 EXPERIMENTS

basic method	:	417.063
Proposed method (25, 5)	:	439.476
Proposed method (25, 6)	:	430.571

ANN. Different parameter sets lead to different results. A very frequent application of an ANN yields on the one hand to a very low wall contact rate. On the other hand no improvements but disturbances of the overall learning process can be observed. Thus several experiments have been conducted to obtain optimal parameters for the three examined environments. Thereby the above described dilemma was solved: using the estimated parameters the impact of the neural network was sufficient to improve the learning progress and to reduce the number of wall contacts to an acceptable minimum.

Because of the enhanced convergence of the learning process, less computational power is required to obtain a suitable solution. Thereby the presented method can also be applied to a more complex environment.

REFERENCES

[1] J. S. Albus, *Brains, Behavior, and Robotics*, BYTE Books, Peterborough, 1981.

[2] Reinaldo A. Bianchi, Carlos H. Ribeiro, and Anna H. Costa, 'Accelerating autonomous learning by using heuristic selection of actions', *Journal of Heuristics*, **14**(2), 135–168, (2008).

[3] Y. Kassahun. Towards a unified approach to learning and adaptation, 2006.

[4] T. Köpsel, A. Noglik, and J. Pauli, 'Evolutionäre algorithmen zur topologieentwicklung von neuronalen netzen für die roboter-navigation im praktischen einsatz', in *Autonome Mobile Systeme 2007*, ed., T. Luksch K. Berns, Informatik aktuell, pp. 145–151, Berlin, (2007). Springer-Verlag.

[5] A. Noglik, M. Müller, and J. Pauli, 'Application of a Heuristic Function in Reinforcement Learning', in *Hybrid Control for Autonomous Systems Integrating Learning, Deliberation, and Reactive Control*, pp. 41–48, (2009).

[6] S. Papierok, A. Noglik, and J. Pauli, 'Application of Reinforcement Learning in a Real Environment using RBF Networks', in *Proceedings of the International Workshop on Evolutionary Learning for Autonomous Robot Systems*, pp. 17–22, (Juli 2008).

[7] G. A. Rummery and M. Niranjan, 'On-line q-learning using connectionist systems', Technical report, Cambridge University Engineering Department, (1994).

[8] A. A. Sherstov and P. Stone. Function approximation via tile coding: Automating parameter choice. Department of Computer Sciences, University of Texas at Austin, 2005.

[9] S. P. Singh and R. S. Sutton, 'Reinforcement learning with replacing eligibility traces', *Machine Learning*, **22**, 123–158, (1996).

[10] K. O. Stanley and R. Miikkulainen, 'Evolving neural networks through augmenting topologies', *Evolutionary Computation*, **10**(2), 99–127, (2002).

[11] R. S. Sutton, 'Integrated architectures for learning, planning, and reacting based on approximating dynamic programming', in *ML*, pp. 216–224, (1990).

[12] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*, The MIT Press, Cambridge, Massachusetts, 1998.

[13] S. Whiteson and P. Stone, 'Evolutionary function approximation for reinforcement learning', *Journal of Machine Learning Research*, **7**, 877–917, (May 2006).

2nd International Workshop on Evolutionary and Reinforcement Learning for Autonomous Robot Systems (ERLARS 2009)
St. Louis, Missouri, USA, October 15 2009

Using Joint Probability Densities for Simultaneous Learning of Forward and Inverse Models

Mark Edgington, Yohannes Kassahun and Frank Kirchner
Robotics Group
University of Bremen
28359 Bremen, Germany
{edgimar, kassahun, frank.kirchner}@informatik.uni-bremen.de

***Abstract*— In this position paper we propose that in many cases, instead of using standard regression methods for directly capturing relationships between variables, joint probability density estimates can and should be used for this purpose. With a good joint probability density estimate, *any* relationship which exists between variables can be extracted in the form of a regression function. Depending on the chosen density estimate representation, a regression function can be derived with relatively little computational effort. In essence, this means that by learning a joint probability density, both forward and inverse models have been captured. This method of learning the relationships between variables is demonstrated through a series of experiments.**

I. Introduction

It is common in a wide variety of scenarios to make use of regression methods, ranging from basic linear regression to more flexible variants such as Gaussian Process Regression (GPR). The common goal behind each of these methods is to learn an accurate and general mapping from some random input vector $\vec{X}$ to a random output vector $\vec{Y}$. To learn this mapping, $\vec{X} \mapsto \vec{Y}$, a regression method would operate on a set of training examples $(\vec{x}_i, \vec{y}_i)$, resulting with a regression function $\vec{y} = h(\vec{x})$. If one wants, however, to invert this function (assuming an inverse exists over some desired range of $\vec{y}$ values), and obtain the function $\vec{x} = h^{-1}(\vec{y})$, the regression method must typically be re-applied to a set of training examples in which $\vec{x}_i$ and $\vec{y}_i$ are swapped.

This illustrates one of the weaknesses of a regression-only approach to learning associations between variables: the learned relationship is unidirectional. We believe that a powerful technique for overcoming this limitation involves the combination of probability density estimation and regression.

Formally, a regression function is defined in probabilistic terms as the expected value of some random vector, given a specific value of a different random vector:

$$\vec{y} = h(\vec{x}) = E[Y \mid X = x]. \tag{1}$$

If the representation of the joint probability density by which these random vectors are distributed is appropriately chosen, the calculation of this conditional expectation is relatively inexpensive, and can be a viable alternative to standard regression methods that directly estimate a regression function from training examples.

As a simple example, if we have two random variables, U and V, which we wish to know the relationship between, we can estimate the joint probability density $f_{U,V}(u, v)$ by any number of density estimation techniques. In so doing, the relationship between these variables has been captured, and one can calculate either of the two possible regression functions

$$u = h(v) = E[U \mid V = v] = \int u\, f_{U|V}(u \mid v)\, du, \tag{2}$$

$$v = h^{-1}(u) = E[V \mid U = u] = \int v\, f_{V|U}(v \mid u)\, dv \tag{3}$$

where the relationship between a conditional density function and a joint density function can be written as

$$f_{Y|X}(y \mid x) = \frac{f_{Y,X}(y, x)}{\int f_{Y,X}(y, x)\, dy}. \tag{4}$$

Equations (2) and (3) are equally valid when U and V are random vectors.

We have successfully applied this approach to a number of problems. The probabilistic representation we use, which we have called the *Dynamic Gaussian Mixture Model*, is introduced in the next section. Following this, a method based on work done by Sung is presented which is used for deriving regression functions from a joint probability density [1]. Finally, we report on the results of three problems to which this regression technique has been applied.

II. Dynamic Gaussian Mixture Model

We have developed a set of learning algorithms for online and offline density estimation using an extended version of the standard Gaussian Mixture Model. This extended model is called a Dynamic Gaussian Mixture Model (DGMM) because of the way in which the number of Gaussian components in the model can vary dynamically to effectively capture the relevant features of an estimated joint probability density.

A DGMM represents a density function $p(\vec{x})$, as a variable-sized set of "weighted Gaussian" pairs,

$$G \equiv \{(g_1(\vec{x}), w_1), (g_2(\vec{x}), w_2), \ldots, (g_m(\vec{x}), w_m)\},$$

such that

$$p(\vec{x}) = \sum_{i=1}^{m} \hat{w}_i g_i(\vec{x}), \tag{5}$$

where $g_i(\vec{x})$ is a multivariate Gaussian distribution:

$$g_i(\vec{x}) = f^{(i)}_{\vec{X}}(\vec{x}) \sim \mathcal{N}(\mu_i, \Sigma_i), \tag{6}$$

and

$$\hat{w}_i = w_i \; / \; \sum_{k=1}^{m} w_k. \tag{7}$$

While the online estimation algorithm is discussed in detail in the context of robot motion modelling in [2], we use an offline algorithm for the experiments presented in this paper. Briefly, the offline method that was used involves generating Gaussian components for each training example, and subsequently smoothing the model through a merging process in which pairs of similar Gaussians are merged into a single representative Gaussian.

III. DGMM Based Regression

In order to calculate a regression function from a DGMM density representation, a method called Gaussian Mixture Regression (GMR) proposed by Sung was used [1]. It is a direct application of (1) to GMMs, taking advantage of the elegant mathematical properties of the Gaussian function. By using Gaussian functions as mixture components, the calculation of marginal and conditional probabilities required for the regression function becomes trivial, requiring little computational effort.

IV. Experiment and Results

A. Simultaneous Learning of the Forward and Inverse Motion Models of the SCORPION *robot*

Learning the motion model of legged robots is challenging due to the complex kinematics of the robot and the complexity of the interaction it makes with the environment during locomotion. Initial experiments were performed with the SCORPION robot [3] to test the extent to which a joint probability density of poses and commands could capture the robot's forward and inverse motion models. The pose of the robot was measured using a motion capture system installed in our laboratory. The experimental environment is a horizontal laboratory floor surface. The pose of the robot includes the Cartesian coordinates (x, y) and heading of the robot θ. The robot's command space is the Cartesian product $C_T = F \times L \times T$, where $F = \{-0.8, 0, 0.8\}$, $L = \{-0.8, 0, 0.8\}$ and $T = \{-0.8, 0, 0.8\}$. The set F stands for forward-backward movements with the maximum and minimum values of 1 and -1 respectively, and the set L stands for lateral left-right movements with the maximum and minimum values of 1 and -1 respectively. Similarly, the set T stands for left-right rotations with the maximum and minimum values of 1 and -1 respectively. We sent the SCORPION robot random commands from the command space C_T and recorded the changes in pose of the robot. In the experiment, we gave equal probabilities to all of the commands, and on average each command is repeated (non-consecutively) five times on the robot. We then built a joint probability density over (command, change in pose) tuples.

1) Extracted Forward Motion Model: In order to validate the learned joint probability density, we first extracted the expectation function of the change in pose given a command. The function was then used to estimate the pose of the robot over 50 timesteps in a separate experiment to assess its prediction quality. The expectation function represents the learned forward model. Figure 1 shows the result we obtained after we used this motion model to estimate the robot's pose. From the figure, it can be seen that the forward motion model predicts the robot's position relatively well.

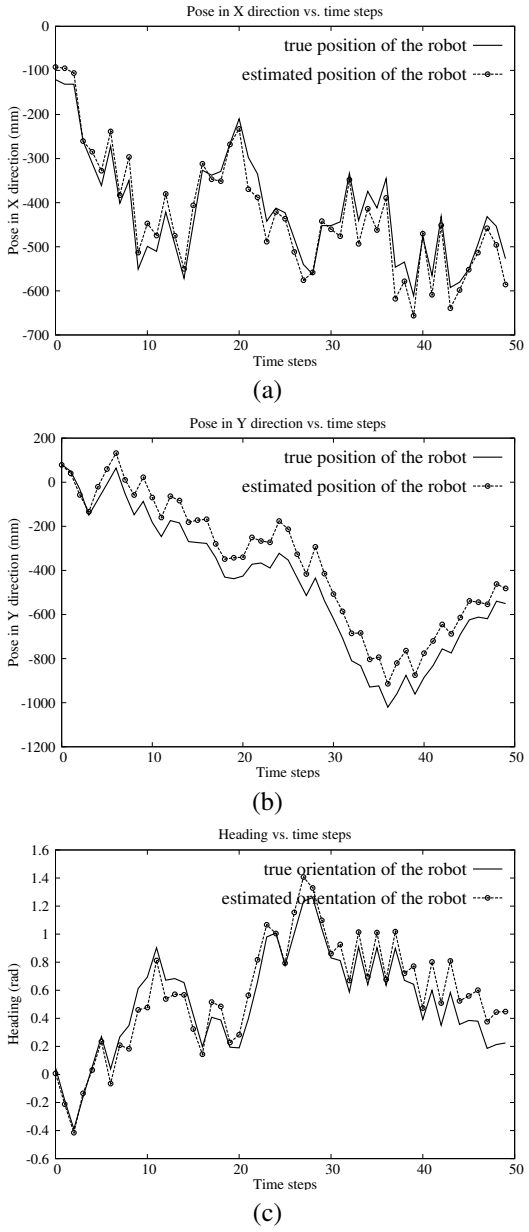


Fig. 1. Performance of the motion model in estimating the pose of the SCORPION robot on a flat surface: (a) pose estimation in the x-direction vs. timesteps. (b) pose estimation in y-direction vs. timesteps, and (c) heading estimation vs. timesteps.

2) Extracted Inverse Motion Model: The inverse motion model was also extracted from the learned joint probability density. It maps the change in robot pose to a command to be

sent to the robot. This model was used to control the robot in a closed loop manner to traverse a figure-8 shaped trajectory. Waypoints were sampled from the trajectory and the nearest waypoint to the current position of the robot was used to calculate the necessary change in pose. The inverse motion model was then used to determine which command to send to the robot, given the necessary change in pose. Figure 2 shows the results of using the inverse motion model in this way for trajectory following. As can be seen in the figure, the robot is able to follow the trajectory reasonably well. Along the target trajectory where the curvature is high, the robot tends to execute commands having larger rotational effects.

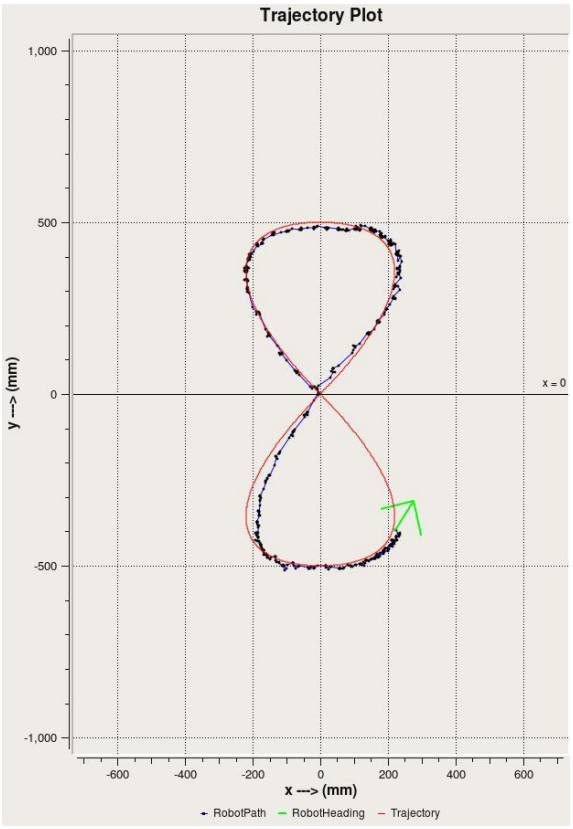


Fig. 2. The inverse motion model used in trajectory following. The figure shows the robot while following a trajectory, and the arrow in the figure shows the current orientation of robot.

B. Performance of DGMM on Classification and Prediction Problems

We have further tested the DGMM/GMR method in the areas of classification and prediction, using the Two-Spiral and Mackey-Glass time-series standard benchmark problems, respectively.

1) Two-Spiral Benchmark: The standard Two-Spiral benchmark was chosen to investigate the performance of the method on classification problems [4]. In this experiment, we learned the joint probability density function of the coordinates of a point on a spiral (x, y), and the class defined by the spiral on which the point lies. Since there are two spirals, a boolean integer in the set $\{0, 1\}$ is used to represent the class. A regression function $E[C|X = x, Y = y]$ is extracted form the joint probability density, where C is a continuous value random number. The output of the regression function is thresholded to determine the class of a point at the coordinates (x, y). Starting with Gaussians centered at each point and associated class, the merging procedure is run a number of times until the minimal number of Gaussians that resulted in a regression function with error free classification is obtained. Figure 3 shows the Gaussians of the joint probability density $f_{X,Y}(x, y)$ after the merging procedure was applied and the class variable is marginalized out. The ellipses represent equi-probability contours of the Gaussians used to form the joint probability density $f_{X,Y}(x, y)$. Note that each Gaussian has its own weight and the weighted sum of the Gaussians represents the joint probability density given by

$$f_{X,Y}(x, y) = \sum_{i=0}^{M} \hat{w}_i \phi_i, \tag{8}$$

where M is the number of Gaussians in the mixture, $\hat{w}_i$ is a mixing coefficient and ϕ_i is a Gaussian $\mathcal{N}(\mu_i, \Sigma_i)$. This initial test of the DGMM/GMR method on this classification problem suggests its suitability for classification tasks.

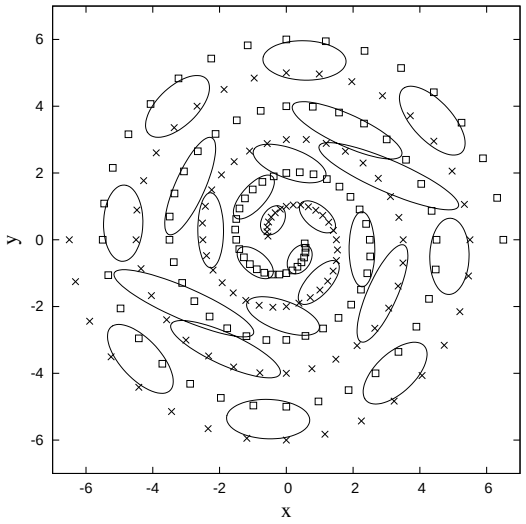


Fig. 3. Density estimation results on Two-Spiral benchmark. Each ellipse represents an equi-probability contour of a Gaussian component of the joint probability density $f_{X,Y}(x, y)$.

2) Mackey-Glass Time Series Benchmark: An experiment described in [5] using the chaotic Mackey-Glass equation was performed with the DGMM/GMR method to investigate the suitability of the method for prediction problems. The task is to predict the time series given by

$$x(t + 1) = (1 - a)x(t) + \frac{bx(t - \tau)}{1 + x^{10}(t - \tau)}, \tag{9}$$

where $a = 0.1$, $b = 0.2$, $\tau = 17$ and $x(0) = 1.2$. The function to be approximated has the form $x(t + 6) = f(x(t), x(t - 6), x(t - 12), x(t - 18))$. The DGMM/GMR

method is trained on 1000 samples of f where $124 \leq t \leq 1123$, and validated on another 1000 samples of f where $1124 \leq t \leq 2213$. Again, the joint probability density is learned first, and afterwards a regression function is extracted from the joint probability density that approximates the function f. Figure 4 shows the performance of the DGMM/GMR method in predicting the sequence for the validation set.

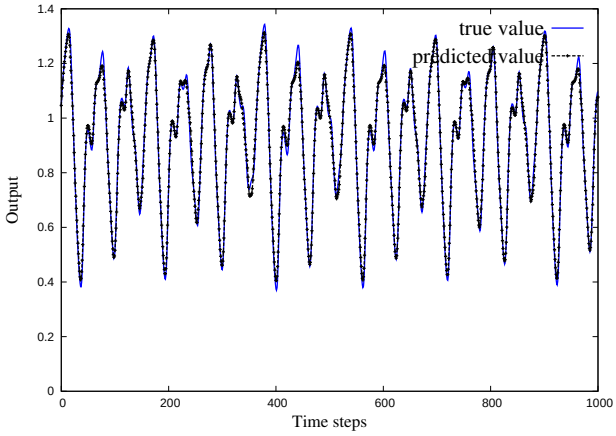


Fig. 4. Prediction performance of the DGMM/GMR method on the Mackey-Glass benchmark. The line with points represents the trajectory generated using this method.

V. Discussion

Though the computational requirements of the DGMM/GMR approach have not been addressed in this paper, the experiments presented have shown that the approach is a viable alternative to standard regression methods in typical application domains. By learning a joint probability density as an intermediate step to deriving a regression function, every observable relationship between variables is captured, regardless of its causality or lack thereof. This makes the joint probability density flexible in the ways it can be used, in contrast to a regression function, which only represents a single relationship between variables.

In the future, we will provide an in-depth analysis of the presented experiments, further validate the DGMM/GMR approach on different robot learning scenarios, and compare its computational requirements with those of standard regression methods.

Acknowledgment

This work was supported by the German Science Foundation (DFG) under contract number SFB/TR-8 (A3).

References

[1] H. G. Sung, "Gaussian mixture regression and classification," Ph.D. dissertation, Rice University, 2004.

[2] M. Edgington, Y. Kassahun, and F. Kirchner, "Dynamic motion modelling for legged robots," in *Proc. of the IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS)*, St. Louis, MO, USA, 2009, accepted.

[3] B. Klaassen, R. Linneman, D. Spenneberg, and F. Kirchner, "Biomimetic walking robot SCORPION: Control and modeling," *Robotics and Autonomous Systems*, vol. 41, no. 2, pp. 69–76, 2002.

[4] S. E. Fahlman and C. Lebiere, "The cascade-correlation learning architecture," in *Advances in Neural Information Processing Systems 2*. Morgan Kaufmann, 1990, pp. 524–532.

[5] L. C. Kiong, M. Rajeswari, and M. V. C. Raoa, "Extrapolation detection and novelty-based node insertion for sequential growing multi-experts network," *Applied Soft Computing*, vol. 3, no. 2, pp. 159–175, 2003.

2nd International Workshop on Evolutionary and Reinforcement Learning for Autonomous Robot Systems (ERLARS 2009)
St. Louis, Missouri, USA, October 15 2009

Compiling Neural Networks for Fast Neuro-Evolution

Nils T Siebel, Andreas Jordt and Gerald Sommer

***Abstract*— Any neuro-evolutionary algorithm that solves complex problems needs to deal with the issue of computational complexity. We show how a neural network (feed-forward, recurrent or RBF) can be transformed and then compiled in order to achieve fast execution speeds without requiring dedicated hardware like FPGAs. In an experimental comparison our method effects a speedup of factor 5–10 compared to the standard method of evaluation (i.e., traversing a data structure with optimised C++ code).**

I. INTRODUCTION

The use of Artificial Neural Networks (also simply "neural networks") for robotics is difficult. While neural networks have been the object of research for several decades, there is still no straightforward way to *construct* a neural network that solves a given robotics task. In many cases creating a good network requires a great deal of domain knowledge and manual intervention, e.g. to determine the network's topology ("structure"), or to adjust the parameters of one's learning algorithm ("hyperparameters") to the given problem and data. Even with manual intervention and tuning, this may still be difficult or impossible if the problem is non-trivial.

Much of the past research work has been solely on learning the *parameters* of a neural network; there are few constructive algorithms for a neural network's *topology*. Also, once a topology is found the search for optimal parameters is a difficult due to numerical ill-conditioning [1] and the so-called "curse of dimensionality" [2]. Recent neuro-evolutionary mehods aim to overcome these problems by evolving both the structure and the parameters of neural networks by evolutionary algorithms [3], [4], [5], [6]. Evolutionary algorithms are known for their good convergence even in difficult optimisation problems, and successful networks have been constructed by these methods (*idib.*). However, their main disadvantage is one inherent in all evolutionary methods: They require many trials (and thereby, evaluations of the neural network) to find a solution to any non-trivial problem.

In this article we present a method to speed up the evaluation of neural networks by first transforming them into a form that requires no branching, then compiling it into binary machine code (32- or 64-bit x86 architecture using SSE). The compiler works for feedforward, recurrent and radial basis function (RBF) networks, or any hybrid network composed of these components. The code does not require re-compilation if only network parameters are changed, which speeds up the search for optimal parameters without the need to re-compile the network.

The remainder of the article is organised as follows. Section II introduces the terminology and describes related work. Details on our neuro-evolutionary method EANT2 can be found in Section III. The neural network compilation approach is described in Section IV and validated by experiments in Section V. Section VI concludes the article.

N.T. Siebel and G. Sommer are with the Cognitive Systems Group, Institute of Computer Science, Christian-Albrechts-University of Kiel, Germany. E-Mail: {nts,gs}@ks.informatik.uni-kiel.de.

A. Jordt is with the Multimedia Information Processing Group, Institute of Computer Science, Christian-Albrechts-University of Kiel, Germany. E-Mail: jordt@mip.informatik.uni-kiel.de.

II. PRELIMINARIES AND RELATED WORK

A. Neural Network Learning Paradigms

An artificial neural network can be regarded as a function f that maps data points/vectors x from an input space $X \subseteq \mathbb{R}^n$ to vectors y in an output space $Y \subseteq \mathbb{R}^m$, i.e. $f : X \to Y,\ x \mapsto y$. For a neural network with a fixed topology this function f is parameterised by the parameters of the network, e.g. the values of synaptic weights. *Training* a neural network means to optimise these parameters such that the network is suitable for a given task. For this the optimisation process needs a measure of this suitability of given parameters, usually expressed by an *error* or *cost function* which is to be minimised during training. The main training/learning paradigms for neural networks are supervised learning, unsupervised learning and reinforcement learning.

1) Supervised Learning: Here a set of example data pairs $\{(x_i, y_i)\}_i$, $x_i \in X, y_i \in Y\ \forall i$, is given. The goal is to find a function $f : X \to Y$ (here, a neural network) that describes the mapping implied by the data points. The cost function is related to the mismatch between our mapping and the data. In classification problems, y_i is the *label* of the point x_i.

The most commonly used cost function is the mean-squared error (MSE), i.e. the mean squared difference between the network's output, $f(x_i)$, and its target value y_i, over all example pairs. A popular algorithm for the minimisation process is the *backpropagation algorithm* [7, chap. 4], which is essentially optimisation by stochastic gradient descent.

2) Unsupervised Learning: As in the supervised case, we are given data examples $\{x_i\}_i$, but not as pairs $\{(x_i, y_i)\}_i$. One form of unsupervised learning is clustering, further examples are the estimation of statistical distributions of data and blind source separation (e.g. based on Independent Component Analysis, ICA). Examples for unsupervised learning approaches by neural networks are Self-Organising Maps (SOM) and Adaptive Resonance Theory (ART) systems.

3) Reinforcement Learning: In reinforcement learning scenarios, data x is usually not given. Instead, the algorithm evaluates a candidate solution by direct interaction with the

environment. One example would be a robot controller. A given network can move the robot at each time instance t by an action y_t which is based on sensor data x_t. The environment generates an observation o_t and often also an instantaneous cost $C(o_t)$, according to the (usually unknown) dynamics of the system. The aim is to discover a policy for selecting actions that minimises a measure of a long-term cost, i.e. the expected cumulative cost.

Reinforcement learning differs from supervised learning in that correct input/output pairs are never presented, nor are sub-optimal actions explicitly corrected. While this lack of information makes reinforcement learning more flexible in its application it also means that the algorithm has no immediate hint in which direction to move in a (possibly very high-dimensional) search space. Therefore one focus is always on performance, which involves finding a balance between *exploration* (of uncharted territory) and *exploitation* (of current knowledge). The efficiency of a reinforcement learning algorithm can be measured in the number of evaluations of the cost function ("function evaluations") it needs to find a good solution. In the neuro-evolutionary methods considered in this article (but not all) an evolutionary algorithm is used for neural network training. Every function evaluation requires at least one, and sometimes very many, evaluations of the neural network. On the other hand, these methods can help to avoid local minima [8] for which backpropagation methods are well-known [9].

B. Related Work: Neuro-Evolution

Up to the late 90s only small neural networks have been evolved by evolutionary algorithms [10]. According to Yao, a main reason is the difficulty of evaluating the exact fitness (negative cost) of a newly found structure: In order to fully evaluate a *structure* one needs to find the optimal (or, some near-optimal) *parameters* for it. However, the search for good parameters for a given structure has a high computational complexity unless the problem is very simple *(ibid.)*.

In order to avoid this problem most approaches evolve the structure and parameters of the neural networks simultaneously. Examples are EPNet [11], GNARL [3] and NEAT [4]. EPNet uses a modified backpropagation algorithm for parameter optimisation (a local method). Mutation operators for searching the space of neural structures are addition and deletion of neurons and connections (no crossover is used). EPNet has a tendency to remove connections/nodes rather than to add new ones. This is done to counteract "bloat" (i.e. ever growing networks with only little fitness improvement; called "survival of the fattest" in [12]). GNARL also does not use crossover during structural mutation. However, it uses an evolutionary algorithm for parameter optimisation. Both parametrical and structural mutation use a "temperature" measure to determine whether large or small random modifications should be applied—a concept known from simulated annealing [13]. In order to calculate the current temperature, the algorithm needs some knowledge about the "ideal solution" to the problem, e.g. the best fitness expected to be reached.

NEAT, unlike EPNet and GNARL, uses a crossover operator that allows to produce valid offspring from two given neural networks. It works by first aligning similar or equal subnetworks and then exchanging differing parts. Like GNARL, NEAT uses evolutionary algorithms for both parametrical and structural mutation. However, the probabilities and standard deviations used for random mutation are constant over time. NEAT also incorporates the concept of speciation, i.e. separated sub-populations that aim at cultivating and preserving diversity in the population [12, chap. 9].

C. Related Work: Speeding up Individual Evaluation in Evolutionary Algorithms

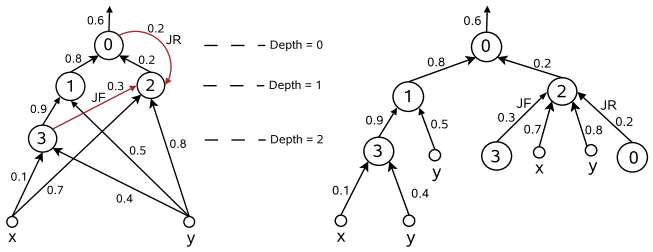
1) Neural Networks in Dedicated Hardware: Most publicised work on speeding up neural network evaluation is through the use of hardware on which neural networks are evaluated, and sometimes also trained. One main idea is to implement the parallel data processing nature of neural networks in reconfigurable circuits (such as "Field Programmable Gate Arrays", "FPGAs") that offer massive parallelism through their architecture.

Cabestany *et al.* discuss the status of some of the first approaches to implement artificial neural networks in hardware in the late 90s [14]. They conclude that at the presented stage (1996) no implementation exists that enables a feasible use of existing systems in industry. One of the reasons given is the lack of a proper software that facilitates the co-operation and data exchange between host computer and the dedicated hardware in a manner suitable for neural networks.

Moerland and Fiesler discuss technical limitations of hardware for the implementation of neural networks and suitable training algorithms [15]. These limitations are quantisation effects (stemming from limited numerical precision of network parameters in hardware implementations) and what they call "hardware non-idealities" such as non-uniformities found on analogue hardware. While the latter effects are no longer relevant on modern implementations that use digital circuitry the former still are. The authors conclude that a precision of 16 bits for network weights is sufficient. They also make a case for the integration of the training algorithm into the hardware program and suggest suitable training algorithms, since back-propagation and other standard methods cannot be easily implemented in FPGAs.

Zhu and Sutton discuss the more recent developments in an FPGA-specific survey [16]. This includes the use of reconfiguration capabilities of FPGAs during training. Even with newer hardware boards the authors still call the implementation of neural networks in FPGAs "challenging" due to the multiplication-intensive nature of network evaluations. Their conclusions are somewhat inconclusive but in line with those given by the previous authors; in particular, they see a requirement for specialised learning algorithms adapted to the nature of FPGAs since traditional algorithms cannot be implemented efficiently into FPGAs.

2) Binary Code in Genetic Programming: Nordin proposes a different approach [17]. He develops algorithms by



(a) Original neural network (b) Network in tree format

N 0 W=0.6	N 1 W=0.8	N 3 W=0.9	I x W=0.1	I y W=0.4	I y W=0.5	N 2 W=0.2	JF 3 W=0.3	I x W=0.7	I y W=0.8	JR 0 W=0.2

(c) Corresponding Linear Genome

Fig. 1. An example of encoding a neural network using a linear genome

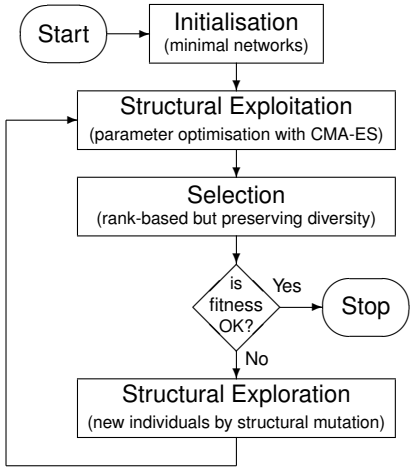


Fig. 2. The EANT2 algorithm. Please note that CMA-ES has its own optimisation loop which creates a nested loop in EANT2.

genetic programming that are directly encoded in executable machine code. The code is a fixed length program consisting of 12 CPU instructions. Naturally, the execution speed of this code is very high compared to an interpreted version of the same instruction sequence, e.g. in LISP. Nordin reports a speedup factor of approximately 1000. Even if the instruction set (+,−,⋆,/), data access (number of variables) is fairly limited, this is a very fast solution.

Harvey *et al.* evolve Java byte code with the help of genetic programming [18], [19]. Their system, which they call "bcGP" for "byte code genetic programming", stores individuals as member functions of Java class files (in memory; each represents a population), which can be run on a Java Virtual Machine (JVM). The instruction set uses basic arithmetic operations (+,−,⋆,/) for regression tasks and comparison operators for classification tasks. While the authors call the performance of their system "efficient" they also say that no speed comparison between byte code GP and a corresponding high-level has been done. Considering the code is run on a JVM, and with the overhead involved in handling the class file it is probably not quite as fast as Nordin's direct machine code solution but may enable a larger code complexity.

To our knowledge no publications exists on the subject of machine code compilation of neural networks. While the performance of genetic programming has been found to be similar to that of neural networks on a range of problems [20] the methods cannot easily be transferred.

III. OUR NEURO-EVOLUTIONARY METHOD: EANT2

A. The Algorithm

EANT2, "Evolutionary Acquisition of Neural Topologies Version 2", is an evolutionary reinforcement learning system that realises neural network learning with evolutionary algorithms both for the structural and the parametrical part. It is based on the previous method EANT [5] but uses different algorithms for structural mutation and parameter optimisation [21]. EANT2 represents neural networks and their parameters in a compact genetic encoding, the "linear genome". It encodes the topology of the network implicitly by the order of its elements (genes). The following basic gene types exist: neurons, network inputs, biases and forward connections. There are also "irregular" connections between neural genes which we call "jumper connections". Jumper genes can encode either forward or recurrent connections. Figure 1 shows an example encoding of a neural network using a linear genome. The figures show (a) the neural network to be encoded. It has one forward and one recurrent jumper connection; (b) the neural network interpreted as a tree structure; and (c) the linear genome encoding the neural network. In the linear genome, `N` stands for a neuron, `I` for an input to the neural network, `JF` for a forward jumper connection, and `JR` for a recurrent jumper connection. The numbers beside `N` represent the global identification numbers of the neurons, `x` and `y` are the inputs coded by input genes. A linear genome can be interpreted as a tree based program if one considers all the inputs to the network and all jumper connections as terminals.

Linear genomes can be evaluated, without decoding, similar to the way mathematical expressions in postfix notation are evaluated. For example, a neuron gene is followed by its input genes. In order to evaluate it, one can traverse the linear genome from back to front, pushing inputs onto a stack. When encountering a neuron gene one pops as many genes from the stack as there are inputs to the neuron, using their values as input values. The resulting evaluated neuron is again pushed onto the stack, enabling this subnetwork to be used as an input to another neuron. Connection ("jumper") genes make it possible for neuron outputs to be used as input to more than one neuron, see `JF3` in the example above. Together with bias neurons the linear genome can encode any neural network in a very compact format; its length is equal to the number of synaptic network weights.

The steps of our algorithm, shown in Figure 2, are explained in detail below.

Initialisation: EANT2 usually starts with minimal initial structures. A minimal network has no hidden layers or recurrent connections, only 1 neuron per output, connected to

some or all inputs. EANT2 gradually develops these simple initial structures further using the structural and parametrical evolutionary algorithms discussed below. On a larger scale new neural structures are added to a current generation of networks. We call this "structural exploration". On a smaller scale the current structures are optimised by changing their parameters: "structural exploitation".

Structural Exploitation: At this stage the structures in the current EANT2 population are exploited by optimising their parameters. Parametrical mutation is realised using CMA-ES ("Covariance Matrix Adaptation Evolution Strategy") [22]. CMA-ES is a variant of Evolution Strategies that avoids random adaptation of strategy parameters. Instead, the search area spanned by the mutation strategy parameters, expressed by a covariance matrix, is adapted at each step depending on the current population. CMA-ES uses sophisticated methods to avoid problems like premature convergence and is known for fast convergence to good solutions even with multi-modal and non-separable functions in high-dimensional spaces *(ibid.)*. It has been first successfully applied to reinforcement learning of neural network weights by Igel [8].

Selection: The selection operator determines which population members are carried on from one generation to the next. Our selection in the outer, structural exploration loop is rank-based and "greedy", preferring individuals that have a larger fitness. In order to maintain diversity in the population, it also compares individuals by structure, ignoring their parameters. The operator makes sure that not more than 1 copy of an individual and not more than 2 similar individuals are kept in the population. "Similar" in this case means that a structure was derived from an another one by only changing connections, not adding neurons.

Structural Exploration: In this step new structures are generated and added to the population. This is achieved by applying the following structural mutation operators to the existing structures: Adding or removing a random subnetwork, adding or removing a random connection and adding a random bias. New hidden neurons are connected to approx. 50 % of inputs; the exact percentage and selection of inputs are random.

B. Comparison with Other Methods

EANT2 is closely related to the methods described in the related work section above. One main difference is the clear separation of structural exploration and structural exploitation. By this we try to make sure a new structural element is tested ("exploited") as much as possible before a decision is made to discard it or keep it, or before other structural modifications are applied. Another main difference is the use of CMA-ES in the parameter optimisation. Further differences of EANT2 to other recent methods, e.g. NEAT, are the absence of algorithm parameters that need to be tuned to the problem (the method should be as universal as possible) and the explicit way of preserving diversity in the population (unlike speciation). More details on the algorithm and an experimental comparison to NEAT on a robot learning task can be found in [6].

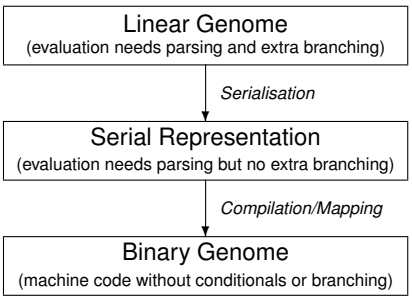


Fig. 3. The compilation process where a Linear Genome is transformed into the Binary Genome (machine code).

One main feature of EANT2 is that the structure remains fixed during structural exploitation. During this time the network is evaluated thousands of times (depending on the given task and fitness function, sometimes even millions of times) before it is changed again during structural exploration. This motivated us to examine how these many recurring sequences of operations (same sequence of additions, multiplications and activation function evaluations) on differing data could be sped up.

IV. COMPILING NEURAL NETWORKS

The evaluation of a neural network, whether it be stack based, tree based or based on any other complex structure, is equivalent to the execution of a fixed set of operations. They usually include recursive (or other) evaluation of neuron (output) values used as input to other neurons as well as other non-sequential operations. When these steps are executed this results in several `if` statements and branches in the machine code, e.g. in the compiled C++ code that implements network evaluation by traversing the relevant structures in a NeuralNetwork class. From a computational point of view branches usually imply a speed penalty at the CPU level, where any branch that was not predicted by CPU internal mechanisms usually means that the instruction pipeline needs to be flushed and rebuilt. Further slowdowns occur when the machine code in the branch is not contained in the instruction (memory) cache. As a result of these issues is a considerate overhead in the number of CPU instructions and wait times is introduced by the non-serial nature of the traversal through the neural network representation in memory. This is all in addition to the actual parsing, which usually involves switch/case statements e.g. when determining the type of a neuron input (network input, bias, output of other neuron, simple connection, recurrent connection etc.).

The goal in our network compilation is to analyse the operations needed for the evaluation of a neural network, log them and discard those operations to maintain the structural overhead, i.e. object management, conditionals, casts, function calls, jumps etc. The logged set of operations is then coded in binary (machine) code. To achieve this, the mathematical operations that are necessary for the calculation of the output values are first extracted from the network in a step we call *Serialisation*. Afterwards these operations

are translated into processor opcodes (*Compilation*). For technical reasons explained below, these opcodes need to be mapped into a data area that allows their execution (*Memory Mapping*). The result of this process is then called the "binary genome". Figure 3 illustrates this process.

A. Serialisation

Serialisation is basically done by recursively evaluating the network and at the same time protocolling the neuron dependencies. Assume for a moment that no recurrent connection is present. Then an ordered list $(L, <)$ of neurons can be defined such that

$$\forall n, m \in L, n \neq m : n < m \vee m < n, \tag{1}$$

where a neuron n can be directly calculated from the elements of $\{m \in L | m < n\}$. The existence of such a list is given by the existence of an evaluation of the neural network.

Obviously, all input neurons are on the bottom of the ordered list. If a recurrent connection is present, it will be handled like an input neuron with value 0 in the first evaluation and the preceding neuron value in the following evaluations.

In the given pseudo-code, `In` denotes the list of input neurons and `Out` the list of output neurons. For every neuron `n`, `n->incident[]` is the list of incident neurons, i.e. the list of neurons its value is calculated from. For every list let `->size` denote the number of elements in the list. The generation of the complete list L is accomplished by the following algorithm:

```
function evaluate(n, list)
{
  for (i := 1 to n->incident->size)
    if (n->incident[i] is not in L)
      evaluate (n->incident[i],L);

  L->push(n);
  return L;
}

function generate_list()
{
  L := empty;

  for (i := 1 to In->size)
    L->push(In[i]);

  for (i := 1 to Out->size)
    L := evaluate(Out[i], L);
  return L;
}
```

It can easily be seen that after execution L contains every neuron that is needed to calculate the output neuron values in the defined order. This order now allows to generate an ordered set of instructions needed to evaluate the network.

Let a be the activation function. Each neuron value $v(n), n \in L$, is calculated by the weighted sum of its incident neurons and the activation function:

$$\forall n \in L : v(n) = a\left(\sum_{j=0}^{|i(n)|} w(n, i(n)_j) \cdot v\left(i(n)_j\right)\right), \tag{2}$$

where $i(n)$ is the list of neurons incident to n and $w(n, m)$ denotes the corresponding weight for each incident neuron $m \in i(n)$ to n. The network evaluation is now simplified to the iterative application of the following steps (for each neuron):

1) pop the first non-input neuron item from the list of neurons
2) add up the weighted inputs of that neuron
3) apply the activation function, and
4) store the neuron value.

B. Compilation

The generation of the corresponding opcode is now straightforward except for the activation function. In most cases this function is not available as a machine instruction (e.g. hyperbolic tangent). Since implementation of such functions can be tedious, the usage of the algorithm provided by the C++ libraries is expedient. Such a function call is easy, if the calling convention as well as the position to jump at is known. This is not the case, if the function is a member function or is defined via templates. In this case a reliable jump point has to be created manually in the C++ code, forwarding the program jump to its actual destination.

In the following the 32-bit case without SSE support is described. For a better readability assembler commands are used to represent the opcode (since `FADD` is easier to read than 0xDC 0xC0).

Before a program is generated, memory is allocated to store the current neuron values and the current weights. This way, the memory addresses, the opcode has to access, are fixed and can be "hard coded" into the opcode, so a runtime calculation of memory addresses is not necessary.

To prepare the CPU for the evaluation code, three things have to be done:

1) back up the register that is used
2) pass the jump destination address of the activation function
3) back up the floating point unit and floating point stack[1].

The first two tasks are accomplished directly by the C++ assembler framework, (which is used to actually jump from C++ code into the generated opcodes), if the usage of certain registers is proposed. The activation function address can be written directly into a register by the embedded assembler.

The floating point units state can be saved by calling:

```
fldsv [backup_addr]
```

An appropriate memory space has to be allocated in advance. After evaluation, the unit state and the stack can be restored by executing:

[1]The floating point stack is part of the floating point unit of the processor. Floating point values are usually not stored in registers like integer values but on a register stack.

```
fldrst [backup_addr]
```

These commands are the first and the last commands in every generated program.

The calculation of a neuron value starts by pushing a zero onto the floating point stack as the current weighted input sum:

```
fldz
```

For every incident neuron connection first the corresponding neuron value is pushed onto the stack followed by a command that multiplies the top value from the stack with a value fetched from memory, which, in this case, is the corresponding weight. This operation is followed by an operation that pops two values from the floating point stack and pushes their sum onto the stack again. Now the new sum is on top of the stack and the next incident neuron can be evaluated:

```
fld [neuron_addr]
fmul [weight_addr]
fadd
```

After regarding every incident neuron value this way, the activation function is called and the resulting value is stored in the neuron array:

```
call eax
fst [neuron_addr]
```

After every neuron is calculated this way, the generated code jumps back to its location it was called from:

```
return
```

The neuron value array now contains all calculated values. Because of the ordered evaluation, each calculation only accessed already calculated neuron values.

C. Memory Mapping

Execution of generated data is an activity every modern operation system is intended to terminate immediately due to security reasons. The execution of data memory is a strong indication of a buffer overflow, which could be exploited by malicious software, or maliciously crafted data. Most Unix systems have been performing such checks via software for many years. The recent introduction of the NX-bit—a hardware solution to prevent the execution of program generated data—brought an even more powerful execution prevention, i.e. the execution of generated data is impeded by the system in every way possible.

The realtime compilation and execution of EANT2 networks is based on a trick applying the I/O memory mapping functionality of Unix systems. I/O memory mapping is usually used to load files or other peripheral device data directly into memory to circumvent the standard I/O data loading behaviour. Such data is loaded completely into a dedicated memory area and is then mapped onto a memory address the executed program can access. A file can either be mapped as read-only-executable or as read-write-accessible memory. What EANT2 does is to map a virtual file twice into memory, once in executable mode and once in read-write mode. These two file mappings are presented to EANT2 by the system as two different memory locations, but a memory access will be resolved to the same file buffer location in the physical memory. To execute a neural network, the generated opcode is copied to the read-write memory location and is then executed by jumping (via embedded assembler) to the execution-memory. A slow down due to hard disk access does not exist because the file is "fully buffered" in the RAM, so no byte of the generated data is actually written to disk.

Network size	40	84	130
Linear Genome	5445.7 ms	18796.7 ms	30171.3 ms
Binary Genome	1037.9 ms	1552.4 ms	2042.3 ms
Compilation time	8.8 ms	16.0 ms	24.2 ms

TABLE I
EXECUTION TIMES DEPENDING ON NEURAL NETWORK COMPILATION

V. EXPERIMENTS

To get an idea about the speed up, measurements of several evaluations using networks of different sizes were performed. All results are compared to the standard evaluation time of the linear genome. For a correct interpretation of this comparison it is important to note that the linear genome structure is already designed for efficient evaluation. The stack based, linear representation (see Section III-A) allows a straight evaluation in one pass and avoids repeated evaluations of sub-networks through a caching mechanism. The C++ code was compiled using the GNU C++ compiler with the "-O3" optimisation flag. The time measurements considered are the CPU times for 1,000,000 evaluations, which in our robotics experiments is an average number of evaluations for the parameter optimisation in EANT2. The tests were performed on an AMD Athlon 3000+ GNU/Linux system, using GCC version 4.3.3. Measurements of the binary genome's compilation time and the 1,000,000-fold evaluation of networks of size 40, 84 and 130 (number of synaptic connections) are given in Table I.

It can be seen that the execution speed of the "binary" (compiled) genome if about 5–10 times faster for typical networks, and increases with the network size. It can also be seen that the compilation time is insignificant compared to the speed gained by the binary evaluation if many evaluations are performed.

In practice, the fitness function that is used by the optimiser does not only consist of the network evaluation but also the interpretation of the network output. Depending on the task at hand and its implementation, the speed advantage may of course be more or less significant for the overall time needed to calculate a network's fitness value. In one of our practical applications of EANT2, where a robot movement and image acquisition is simulated after each network evaluation, the overall speed increased by a factor of 3–4 through the introduction of the binary genome. Over

time, as the networks increase in size, the speed advantage becomes more pronounced.

VI. CONCLUSIONS

We have presented a method to speedup the calculation of a neural network's output by transforming the internal representation into binary machine code. This helps to alleviate one problem neuro-evolutionary algorithms still have nowadays: they are slow, i.e. they need many evaluations of the fitness function/neural networks before they find a good solution to a robot learning problem.

In the evolutionary process that optimises the network's parameters the network only needs to be "compiled" once; new parameter values are taken from a given array of current values.

Our experiments have shown that the compilation speeds up the evaluation time of a neural network by a factor of approximately 5–10, depending on the network size. With larger networks, the speedup factor is also larger.

REFERENCES

[1] W. S. Sarle, "Ill-conditioning in neural networks," Website, SAS Institute Inc., Cary, USA, September 1999, ftp://ftp.sas.com/pub/neural/illcond/illcond.html.

[2] R. E. Bellman, *Adaptive Control Processes*. Princeton, USA: Princeton University Press, 1961.

[3] P. J. Angeline, G. M. Saunders, and J. B. Pollack, "An evolutionary algorithm that constructs recurrent neural networks," *IEEE Transactions on Neural Networks*, vol. 5, no. 1, pp. 54–65, 1994.

[4] K. O. Stanley and R. P. Miikkulainen, "Evolving neural networks through augmenting topologies," *Evolutionary Computation*, vol. 10, no. 2, pp. 99–127, 2002.

[5] Y. Kassahun and G. Sommer, "Efficient reinforcement learning through evolutionary acquisition of neural topologies," in *Proceedings of the 13th European Symposium on Artificial Neural Networks (ESANN 2005)*, Bruges, Belgium, April 2005, pp. 259–266.

[6] N. T. Siebel and G. Sommer, "Evolutionary reinforcement learning of artificial neural networks," *International Journal of Hybrid Intelligent Systems*, vol. 4, no. 3, pp. 171–183, October 2007.

[7] C. M. Bishop, *Neural Networks for Pattern Recognition*. Oxford, UK: Oxford University Press, 1995.

[8] C. Igel, "Neuroevolution for reinforcement learning using evolution strategies," in *Proceedings of the IEEE Congress on Evolutionary Computation (CEC 2003)*. IEEE Press, 2003, pp. 2588–2595.

[9] J. C. Spall, *Introduction to Stochastic Search and Optimization: Estimation, Simulation, and Control*. Hoboken, USA: John Wiley & Sons, 2003.

[10] X. Yao, "Evolving artificial neural networks," *Proceedings of the IEEE*, vol. 87, no. 9, pp. 1423–1447, September 1999.

[11] X. Yao and Y. Liu, "A new evolutionary system for evolving artificial neural networks," *IEEE Transactions on Neural Networks*, vol. 8, no. 3, pp. 694–713, May 1997.

[12] Á. E. Eiben and J. E. Smith, *Introduction to Evolutionary Computing*. Berlin, Germany: Springer Verlag, 2003.

[13] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by simulated annealing," *Science*, vol. 220, no. 4598, pp. 671–680, May 1983.

[14] J. Cabestany, P. Ienne, J. M. Moreno, and J. Madrenas, "Is there a future for ANN hardware?" in *Proceedings of the Workshop on Mixed Design of Integrated Circuits and Systems*, Lodz, Poland, May 1996, pp. 419–424.

[15] P. Moerland and E. Fiesler, "Neural network adaptations to hardware implementations," Dalle Molle Institute for Perceptive Artificial Intelligence, Martigny, Valais, Switzerland, Research Report 97-17, 1997.

[16] J. Zhu and P. Sutton, "FPGA implementations of neural networks – a survey of a decade of progress," in *Proceedings of the 13th International Conference on Field Programmable Logic and Applications (FPL 2003)*, Lisboa, Portugal, September 2003, pp. 1062–1065.

[17] P. Nordin, "A compiling genetic programming system that directly manipulates the machine code," in *Advances in Genetic Programming*, K. E. Kinnear, Ed. Cambridge, USA: MIT Press, 1994, vol. 2, ch. 14, pp. 311–â331.

[18] B. Harvey, J. A. Foster, and D. Frincke, "Byte code genetic programming," in *Late Breaking Papers at the Genetic Programming 1998 Conference (GP-98)*, Madison, USA, July 1998, (no page numbers).

[19] ——, "Towards byte code genetic programming," in *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 1999)*, vol. 2, Orlando, USA, 1999, p. 1234.

[20] M. Brameier and W. Banzhaf, "A comparison of linear genetic programming and neural networks in medical data mining," *IEEE Transactions on Evolutionary Computation*, vol. 5, no. 1, pp. 17–26, February 2001.

[21] N. T. Siebel and Y. Kassahun, "Learning neural networks for visual servoing using evolutionary methods," in *Proceedings of the 6th International Conference on Hybrid Intelligent Systems (HIS'06), Auckland, New Zealand*, December 2006, p. 6 (4 pages).

[22] N. Hansen and A. Ostermeier, "Completely derandomized self-adaptation in evolution strategies," *Evolutionary Computation*, vol. 9, no. 2, pp. 159–195, 2001.

2nd International Workshop on Evolutionary and Reinforcement Learning for Autonomous Robot Systems (ERLARS 2009)
St. Louis, Missouri, USA, October 15 2009

Path Planning for a Mobile Robot Using Self Tuning Fuzzy Logic Controller

Iraj Hassanzadeh, and Sevil M. Sadigh

***Abstract*—This paper presents a fuzzy logic controller self tuning using Self Organizing Map neural network (SOM) for near the optimal time path planning for a mobile robot to avoid obstacles in unknown environments. A SOM neural network is applied to modify the input and output membership functions of the fuzzy controller automatically. A Matlab application, Kiks II, is used to simulate a Khepera II robot. Also, this approach is implemented by Khepera II robot. It is also shown that the proposed method outperforms the FLC approach.**

I. INTRODUCTION

Over the last few years, a number of studies were reported concerning machine learning, and how it has been applied to help mobile robots to improve their operational capabilities. One of the most important issues in the design and development of intelligent mobile system is the navigation problem. This consists of the ability of a mobile robot to plan and execute collision-free motions within its environment. However, this environment may be imprecise, vast, dynamical and either partially or non-structured [4]. In such environment, motion planning depends on the sensory information of the environment, which might be associated with imprecision and uncertainty. Thus, to have a suitable motion planning scheme in a cluttered environment, the controller of such kind of robots must have to be adaptive in nature. Recently, [2] have made an extensive survey on the navigational schemes of mobile robots moving among dynamic obstacles.

Soft computing includes fuzzy logic, genetic algorithm, neural network and their different combinations [2, 3] and it can solve such complex real-world problems within a reasonable accuracy. The computational complexity of such methods is also expected to be low, due to their heuristic nature.

Since artificial neural networks (ANN) have the ability to learn the situations, many investigators have successfully applied the neural network [4, 5 and 6] to develop the model related to the navigation problem of a mobile robot. Janglová [4] used two NNs, one to determine the free space using ultrasound range finder data and the other to find a safe direction for the next robot section of the path in the workspace while avoiding the nearest obstacles. Kian Hsiang Low [5] used self-organizing neural network to perform fine, smooth motor control that moves the robot through the checkpoints. Botelho [6] used Boolean NNs such as RAM and GSN models for controlling of robot navigation. Since these NNs have high speed processing, the decision rate are increased.

We know that NNs have the ability to learn the situations, but with some neural networks, knowledge representation and extraction are difficult.

Fuzzy systems have the ability to make use of knowledge expressed in the form of linguistic rules, thus they offer the possibility of implementing expert human knowledge and experience. Their main drawback is the lack of a systematic methodology for their design. Usually, tuning parameters of membership functions is a time consuming task. Genetic Algorithm or Neural network learning techniques can automate this process, significantly reducing development time, and resulting in better performance. Kun Hsiang Wu [8] used a genetic-based adaptive fuzzy controller to navigate the robot.

In this paper is used fuzzy logic controller (FLC) for solving the navigation problems of a mobile robot and for tuning parameters is used SOM that the parameters are tuned automatically. The performance of this approach, to generate collision-free path of a robot, are compared with FLC.

The organization of the paper is as follows: section 2 describes the fuzzy logic controller (FLC) approach for path planning. Self tuning membership functions using SOM describes in section 3. Simulation and implementation results will be included in section 4. Section 5 will summarize our conclusions.

II. FUZZY LOGIC CONTROLLER (FLC)

In actual navigation, information of the input variables collected by using the camera or sensor might be imprecise in nature [2]. Thus, fuzzy logic controller could be a potential candidate for solving this problem. Two major approaches FLC are Mamdani Approach and Takagi and Sugeno Approach.

In Mamdani Approach, the condition and action variables of the FLC are expressed in terms of membership function distributions. Figure1 shows the membership function distributions of both the input and output variables. The input variables are distance and angles that explain condition`s obstacle to robot. The range of distance is divided into three linguistic terms, namely Near (NR), mid

Iraj Hassanzadeh is with the Faculty of electrical and computer engineering University of Tabriz, Tabriz, Iran (e-mail: Izadeh@tabrizu.ac.ir).
Sevil M. Sadigh is with the Faculty of electrical and computer engineering University of Tabriz, Tabriz, Iran (e-mail: s_msadigh@yahoo.com).

(MID) and far (FR). The range of angle is divided into five terms: left (LT), ahead left (AL), ahead (AH), ahead right (AR) and right (RT). For output variables, five linguistic terms are considered: back slow (BS), zero (Z), forward slow (FS), forward mid (FM) and forward fast (FF).

In Takagi and Sugeno Approach, the membership function distributions of the input variables have been assumed to be the same as shown in Fig. 1, whereas the outputs may be expressed like the following:

IF A_1 AND B_1, THEN $z_1=a_1 x +b_1 y +c_1$ & $z_2=a_2 x +b_2 y +c_2$

Where A_1, B_1 are labels of fuzzy sets, x and y are input variables, z_1 and z_2 are output of FLC and a_1, b_1, a_2, b_2 are the coefficients of the input variables and c_1, c_2 are the constants. With three choices for *distance* and five choices for *angle*, there could be 3 × 5 or 15 possible combinations of two different condition variables. Thus, there is a maximum of 15 rules present in the rule base that are shown in Table I.

FLC structure has 5 stage or layer. The first layer transmits input values to the next layer using linear transfer function. The next layer is the fuzzification layer, in which the membership function values of the input variables are determined corresponding to input conditions. Third layer is the layer of rule base that defines the fuzzy rules. The output of every neuron in this layer is the multiplication of their two incoming signals:

$$O_{3n} = O_{2i} \times O_{2j} \quad (1)$$

Where O_{3n} is the output of the neuron n in layer 3 and O_{2i}, O_{2j} are the outputs of the neurons i, j in layer 2, respectively.

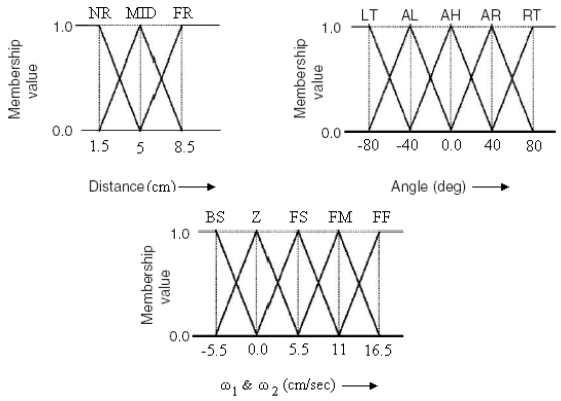


Fig. 1. *Membership function distributions for input and output variables of the FLC*

TABLE I
RULE BASE OF THE FLC FOR DITERMINING VELOCITY OF THE WHEELS.

Angle

	LT	**AL**	**AH**	**AR**	**RT**
NR	FF	FF	FF	FS	FF
MID	FF	FF	FF	FS	FF
FR	FF	FS	FM	Z	FF

ω_1

Angle

	LT	**AL**	**AH**	**AR**	**RT**
NR	FF	FS	BS	FF	FF
MID	FF	FS	BS	FF	FF
FR	FF	BS	Z	FM	FF

ω_2

ω_1= the speed of left wheel, ω_2= the speed of right wheel

4th layer is the layer of Consequence, which identifies the fired rules for a set of inputs. The connecting weights between the 4th and 5th layers indicate the membership function distributions of the output variables. Once the membership function distributions are known, this layer calculates the output of all the fired rules.

The last layer is defuzzification layer, which converts the fuzzified output to its corresponding crisp value. A center of sums method is adopted for defuzzification. The final output O_i of the i*k*th neuron lying in this layer can be expressed as follows:

$$O_i = \frac{\sum_k A_{ik} M_{ik}}{\sum_k A_{ik}} \quad (2)$$

Where A_{ik} and M_{ik} are the area and center of area for kth fired rule of ith output, respectively.

The performance of an FLC is influenced by its knowledge base (KB). Thus, it is essential to tune the KB of the fuzzy logic controller to get a better performance. Since the tuning can be viewed as an optimization process, either a neural network (NN) or a genetic algorithm (GA) offers a possibility to solve this problem. Here we use Self Organizing Map neural network (SOM) for tuning KB.

III. SELF TUNING FLC

The aim of this paper is to tune the centers of membership functions (MFs) automatically. The Self Organizing Map (SOM) is an unsupervised neural network. Thus, could be

useful for reaching this aim. Topology of SOM neural network is shown in Fig. 2.
The algorithm has two steps, each step having different method. The centers of input membership functions are tuned first. After, tuning of output membership functions is processed.

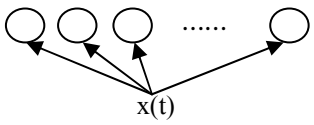


Fig. 2. Topology of SOM neural network

A. Tuning of input membership functions

Tuning of input MFs is begins by feeding the input variables of FLC to SOM networks. The neuron whose value has smallest distance with the input is selected as winning neuron. The winning vector and its neighbors are updated by

$$w(t) = w(t-1) + [\alpha\, hv(t)\, (x(t) - w(t-1))] \quad (3)$$

Where w(t) is the weight vector of neurons, x(t) denotes the input network, $\propto$ represents the learning rate. $\propto$ decrease monotonically with t and hv(t) is the neighborhood of the winning neuron and is obtained by

$$hv(t) = c + d\, e^{\left(\frac{-h(vn-i)^2}{\sigma^2}\right)} \quad (4)$$

$$\sigma_t = a + b\, e^{-ct} \quad (5)$$

Where nv is the number of the winning neuron, i denotes the number of the updating neuron, a,b,c,d and h are constant coefficients and determine the radius neighborhood. hv(t) and σ_t decrease with t. also, hv(t) decrease with keeping out of the way from the winning neuron.

Fig. 3. *Khepera II*

After updating neurons, the next input is fed to SOM networks, the winning neuron is selected and the neurons are updated again. Iterations proceed until a pre-specified number is satisfied.

B. Tuning of output membership functions

This step is different from before step at how selecting of the winning neuron and updating neurons. In the before step, the neuron whose value is smallest distance with the input is selected as winning neuron. But in this step, the neuron whose value has the most firing strength of the fuzzy rule is selected as winning neuron.
The winning vector and its neighbors are updated by

$$w(t) = w(t-1) + [\alpha\, hv(t)\, (c(t) - w(t-1))] \quad (6)$$

Where w(t) is the weight vector of neurons, c(t) denotes the value of the winning neuron, $\propto$ represents the learning rate and decrease monotonically with t. hv(t) is the neighborhood of the winning neuron and is obtained by equation (4) and (5).
Algorithm proceeds similar to previous method.

IV. SIMULATION AND IMPLEMENTATION RESULTS

In order to compare the self tuning FLC by SOM with the FLC, two approaches have been applied to control the Khepera mobile robot [1] for the obstacle avoidance task by using kiks [9] (Khepera Simulator). KiKS is an abbreviation for"Kiks is a Khepera Simulator". The program is a Matlab application that simulates a Khepera II robot connected to the computer in a very realistic way. The simulated Khepera is controlled from Matlab in the same way as real, physical Kheperas. (For more details see [9]).
Khepera is a miniature mobile robot (Fig. 3) developed in the Microcomputing Laboratory of Swiss Federal Institute of Technology [1]. It has a cylindrical shape, measuring 55 mm in diameter and 30 mm in height and its weight is only 70 g. The robot has two DC motors and eight analogue infra-red (IR) proximity sensors (Fig. 4).

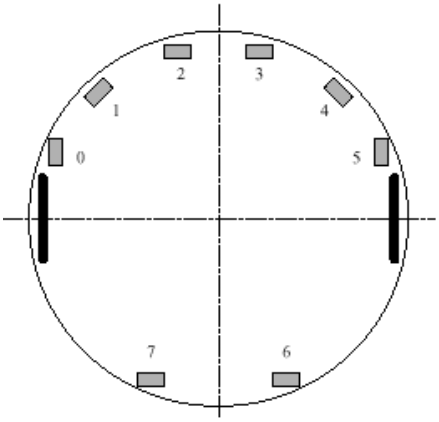


Fig. 4. Position of the sensors on the Khepera mobile robot

The task is to an autonomous mobile robot with a path planning and intelligent control should move among obstacles without collision them to reach the target. First, the robot rotates toward target and moves with a fixed speed for a time step while detecting obstacle. When robot detects any obstacle by IR sensors, FLC is to be activated. Otherwise, the robot moves with a fixed speed for a time step. The inputs of FLC are distance and angle between robot and obstacle. The value of distance and angle are calculated using data sensors of the robot and with the vectorial course. For simplicity calculations and since the robot move toward front, for determining distance and angle are used data of the front sensors (i.e. 0, …, 5). The outputs of FLC are the speed of wheels. They are applied to robot's wheels and the robot rotates until doesn't collide obstacle then it will rotate toward target. This process will continue, until the robot reaches the target.

In this work, comparison studies among the FLC and the self-tuning FLC algorithms are carried out. The rule base is set manually based on intuition and is same for the both algorithms. The knowledge base of FLC is set manually based on intuition but the knowledge base of self tuning FLC algorithm is tuned by SOM (see section3).

For the proposed self tuning FLC, the maximum learning rate is $\alpha_{max} = 1.5$; the constant coefficients of input MFs are a = 1.5, b=7.5, c = 0.5, d = 0.05 and h = 10; the constant coefficients of output MFs are a = 1.5, b=5.5, c = 1, d = 0.05 and h = 10;

The testing scenarios were doing in the environments that as not use for tuning parameters of FLC, i.e. the unknown environment for the robot. The simulation is carried out for three different scenarios (path planning with 2, 4 and 12 obstacles in the unknown environments in the first, 2nd and 3rd scenarios, respectively) and is presented in figs. 5-10. Also, the implementation results are presented in fig. 11.

Numerical calculation results for these two methods are shown in Tables II, III and IV. TOC denotes the *time of competition* (the time of robot motion from start point to target), NoH represents the number of hits of the robot with the obstacles. ASR is *average of speed robot* during path and RDP denotes error of deviation of travelling path from optimal path for each scenario.

The results show that total time traveling of robot and length travelling path of robot with self tuning FLC by SOM is lower and smoother than FLC.

V. Conclusion

In this paper, self tuning FLC method is used to plan path for a mobile robot while avoiding obstacles. In this method, MFs are tuned automatically. Also, this method is simple. Results show this method acts successfully and robot move among obstacles without collision them in unknown environment. Also, the simulation and implementation results show that total time traveling of robot and length travelling path of robot with self tuning FLC is lower and smoother than FLC.

TABLE II
FLC

	TOC (sec)	ASR (cm/sec)	No. H	RDP (mm)
Scenario 1	12.4530	11.9130	0	90
Scenario 2	17.5700	8.8814	0	134.5
Scenario 3	18.2145	8.9002	0	127.23

TOC = time of competition, ASR = average of speed robot, No.H = Number of hits, RDP = error of deviation of travelling path from optimal path for each scenario

TABLE III
SELF TUNING FLC WITH SOM.

	TOC (sec)	ASR (cm/sec)	No. H	RDP (mm)
Scenario 1	10.6480	13.301	0	31.19
Scenario 2	14.3980	10.3190	0	36.99
Scenario 3	15.2010	10.3158	0	78.39

TOC = time of competition, ASR = average of speed robot, No.H = Number of hits, RDP = error of deviation of travelling path from optimal path for each scenario

TABLE IV
IMPLEMENTATION RESULTSUSING FLC AND SELF TUNING FLC

	TOC (sec)	ASR (cm/sec)	No. H	RDP (mm)
FLC	17.594	8.0552	0	70.22
Self tuning FLC	10.969	9.9046	0	19.85

TOC = time of competition, ASR = average of speed robot, No.H = Number of hits, RDP = error of deviation of travelling path from optimal path for each scenario

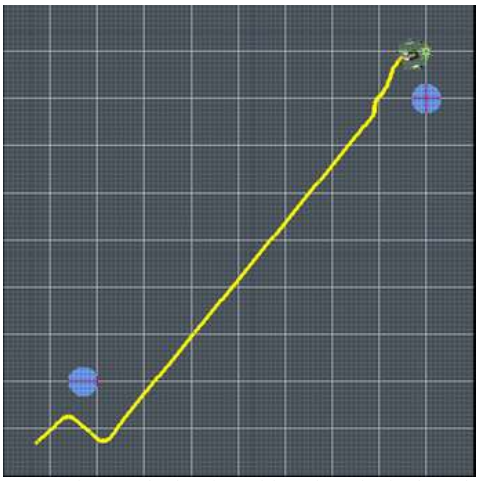


Fig. 5. Scenario 1: path planning with 2 obstacles in the unknown environment with FLC

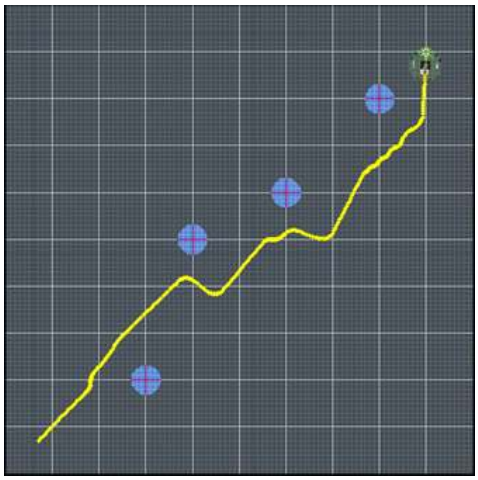


Fig. 6. Scenario 2: path planning with 4 obstacles in the unknown environment with FLC

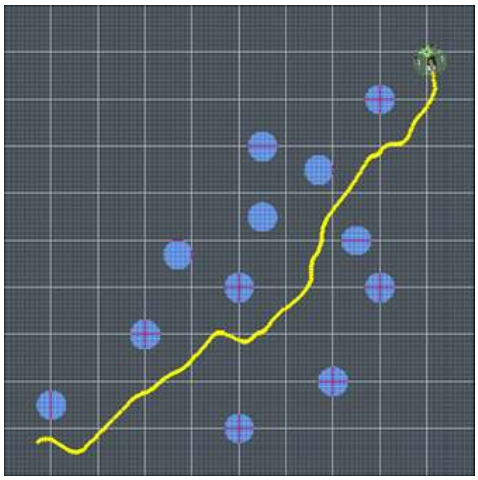


Fig. 7. Scenario 3: path planning with 12 obstacles in the unknown environment with FLC

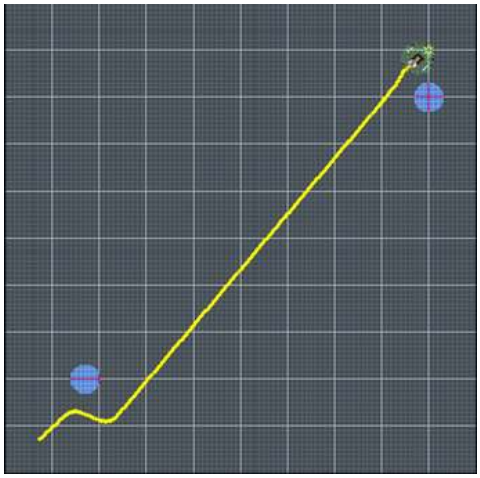


Fig. 8. Scenario 1: path planning with 2 obstacles in the unknown environment with self tuning FLC by SOM

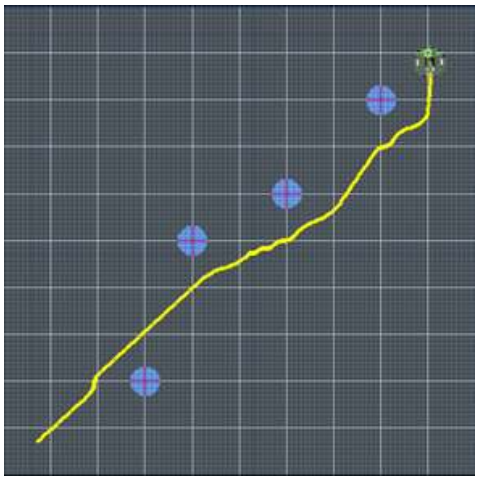


Fig. 9. Scenario 2: path planning with 4 obstacles in the unknown environment with self tuning FLC by SOM

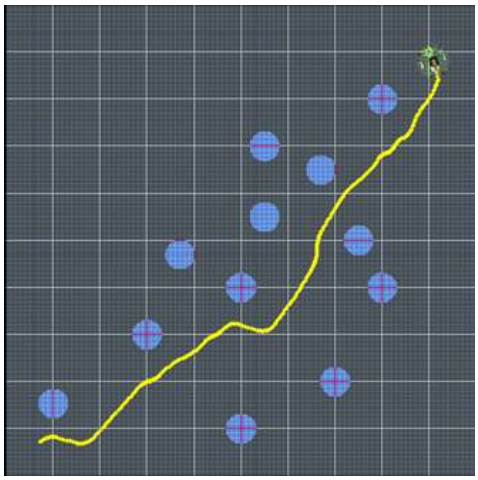


Fig. 10. Scenario 3: path planning with 12 obstacles in the unknown environment with self tuning FLC by SOM

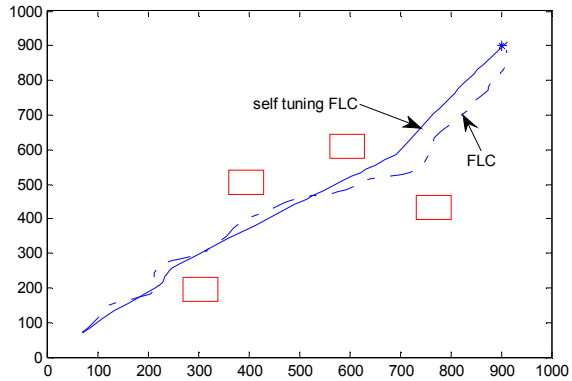


Fig. 11. Implementation of path planning in the unknown environment with FLC and self tuning FLC by SOM

REFERENCES

[1] K-Team, "Khepera user manual version 5.02," Lausanne, 12 March 1999.

[2] NirmalBaran Hui,V.Mahendar, Dilip Kumar Pratihar, "Time-optimal, collision-free navigation of a car-like mobile robot using neuro-fuzzy approaches," Fuzzy Sets and Systems 157 (2006) 2171-2204.

[3] Jelena Godjevac, Nigel Steele, "Neuro-fuzzy control of a mobile robot," Neurocomputing 28 (1999) 127-143.

[4] Danica Janglová, "Neural Networks in Mobile Robot Motion," Neural Networks in Mobile Robot Motion, pp. 15-22, Inernational Journal of Advanced Robotic Systems, Volume 1 Number 1 (2004), ISSN 1729-8806.

[5] Kian Hsiang Low, Wee Kheng Leow, Marcelo H.Ang Jr., "Integrated Planning and Control of Mobile Robot with Self-Organizing Neural Network," Proc.18th IEEE ICRA'02,vol.4,pp.3870-3875,May11-15, 2002, Washington, D.C..

[6] Sílvia S.C. Botelho, Eduardo do Valle Simões, Luís Felipe Uebel & Dante Barone "High speed Neural Control for Robot Navigation," Porto Alegre,RS 91501-970,Brazil.

[7] Jyh-Shing Roger Jang, "ANFIS: Adaptive-Network-Based Fuzzy Interence System," IEEE Transactions on systems, man, and cybernetics, vol.23, no. 3, may/june 1993.

[8] Kun Hsiang Wu, Chin Hsing Chen and Jiann Der Lee, "Genetic-Based Adaptive Fuzzy Controller for Robot Path Planning," 0-7803-3645-3/96 © 1996 IEEE.

[9] Theodor Storm, "*KiKS is a Khepera Simulator*, user guide"

[10] Sung Hoe Kim, Chongkug Park, Member, IEEE, and Fumio Harashima, Fellow, IEEE, "A Self-Organized Fuzzy Controller for Wheeled Mobile Robot Using an Evolutionary Algorithm" IEEE TRANSACTIONS ON INDUSTRIAL ELECTRONICS, VOL. 48, NO. 2, APRIL 2001.