The 20th European Conference on Artificial Intelligence (ECAI 2012)

Proceedings of the

5th International Workshop on Evolutionary and Reinforcement Learning for Autonomous Robot Systems (ERLARS 2012)

Tuesday, August 28 2012
Montpellier, France

Nils T Siebel and Yohannes Kassahun

http://www.erlars.org/

Proceedings of the 5th International Workshop on Evolutionary and Reinforcement Learning
for Autonomous Robot Systems (ERLARS 2012)
Editors: Nils T Siebel and Yohannes Kassahun

ISSN 2190-5576 (Print)
ISSN 2190-5746 (Internet)

Published and Printed by
Nils T Siebel
Wilmsstr. 5
10961 Berlin
Germany
1st Edition, August 2012

Table of Contents

A Message from the Chairs

We would like to welcome you to the 5rd International Workshop on Evolutionary and Reinforcement Learning for Autonomous Robot Systems, ERLARS 2012, held again in conjunction with the ECAI conference, this time in Montpellier, France on August 28 20120.

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 contained 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. In order to give researchers a chance to discuss their work at an early stage this proceedings volume also includes short papers / research statements.

We would like to thank the program committee members who provided very good and helpful reviews. 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 Yohannes Kassahun, Chairs, ERLARS 2012 Workshop.

Organisation of the ERLARS 2012 Workshop

Workshop Chairs

Nils T Siebel
Building Automation Lab
Department of Engineering I
HTW University of Applied Sciences
Berlin, Germany

Yohannes Kassahun
Research Group Robotics
DFKI Lab Bremen
University of Bremen
Bremen, Germany

Programme Committee

5th International Workshop on Evolutionary and Reinforcement Learning for Autonomous Robot Systems (ERLARS 2012)
Montpellier, France, August 28 2012

Acquiring Diverse Predictive Knowledge in Real Time by Temporal-difference Learning

Joseph Modayil and **Adam White** and **Patrick M. Pilarski** and **Richard S. Sutton**[1]

Abstract. Existing robot algorithms demonstrate several capabilities that are enabled by a robot's knowledge of the temporally-extended consequences of its behaviour. This knowledge consists of real-time predictions—predictions that are conventionally computed by iterating a small one-timestep model of the robot's dynamics. Given the utility of such predictions, alternatives are desirable when this conventional approach is not applicable, for example when an adequate model of the one-timestep dynamics is either not available or not computationally tractable. We describe how a robot can both learn and make many such predictions in real-time using a standard reinforcement learning algorithm. Our experiments show that a mobile robot can learn and make thousands of accurate predictions at 10 Hz about the future of all of its sensors and many internal state variables at multiple time-scales. The method uses a single set of features and learning parameters that are shared across all the predictions. We demonstrate the generality of these predictions with an application to a different platform, a robot arm operating at 50 Hz. Here, the predictions are about which arm joint the user wants to move next, a difficult situation to model analytically, and we show how the learned predictions enable measurable improvements to the user interface. The predictions learned in real-time by this method constitute a basic form of knowledge about the robot's interaction with the environment, and extensions of this method can express more general forms of knowledge.

1 Introduction

A robot's ability to make real-time predictions about the consequences of its behaviour supports several additional capabilities. Examples of robot capabilities built on real-time predictions include collision avoidance [Fox et al., 1997], stability [Abbeel et al., 2010], and motion planning [LaValle, 2006]. The conventional approach to make these predictions is to manually construct a small one-timestep model of the system dynamics offline, and then, during real-time operation, to make temporally-extended predictions by simulating future trajectories with the model. However, this approach requires a one-timestep model of the dynamics to be available, and it requires computationally expensive simulations with the model to predict quantities of interest.

We propose an alternate approach for real-time predictions, namely to learn to directly predict the temporally-extended consequences of a behaviour in real-time. This is the technique used for the critic's value function in an actor-critic based method. We demonstrate that this direct approach scales well for learning and making many temporally extended predictions in parallel, and thus potentially opens the door to new robot capabilities.

The main contribution of this work is an empirical demonstration that thousands of temporally-extended predictions can be learned online in real-time with high accuracy. The predictions are in the form of questions about future sensor values and internal state bits. We demonstrate that a mobile robot can both learn and make thousands of predictions in real-time. In our first experimental setting, predictions are made every 100ms, and the predictions are about the robot's future sensor readings and internal state variables either at the next timestep in 100ms, or over the next short time scale of 0.5, 2, or 8 seconds. These predictions provide the robot with immediate knowledge about many distinct, temporally extended consequences of its behaviour. In a second experimental setting, we demonstrate the generality of these predictions by evaluating how they can improve the user interface for a robot arm.

The approach is novel in several respects. The predictions have the benefit of scientific empiricism—the predictions can be evaluated for their accuracy by comparison to the robot's future experience. Although directly learning the temporally extended consequences of behaviour is not a common way of representing knowledge in robotics, these predictions can also be assembled to form a conventional one-timestep model of the dynamics. The ease of acquiring this knowledge, the generality of the method, and known extensions to the prediction algorithm, suggest that this is a promising direction for further investigation.

The paper is structured as follows. First, we present the method to describe the learning setting precisely. Then, we show results from our experimental evaluation of the method on a mobile robot. We then demonstrate the generality of this method with an application to the completely different domain of predictions for a human-guided robot arm. After describing related work, we discuss how this method can be extended to more general forms of predictions.

2 Method

The method relies on learning many temporally-extended predictions, so we first review the underlying temporal-difference prediction algorithm TD(λ) [Sutton, 1988]. As input at each timestep $t \in \mathbb{N}$, the algorithm receives the feature vector $x_t \in \mathbb{R}^n$. The feature vector is the robot's description of the state of the environment s_t. Note that the description provided by x_t will be restricted to features that the robot can readily compute, and this is typically an incomplete characterization of the state of the environment. Each predictive question pertains to some signal $r_t \in \mathbb{R}$ that is observed at each timestep. The signal is called the reward in reinforcement learning, but here it is an arbitrary target signal and does not indicate

[1] Reinforcement Learning and Artificial Intelligence Laboratory, Department of Computing Science, University of Alberta, Canada. email: {jmodayil, awhite, pilarski, sutton} @cs.ualberta.ca

a quantity that the robot wishes to maximize. We assume that the robot is following a fixed behaviour and the question is to predict the *return*, G_t, which is the discounted sum of the target signal observed in the future,

$$G_t = \sum_{k=1}^{\infty} \gamma^{k-1} r_{t+k}, \tag{1}$$

where γ is a constant. A particular choice of γ will focus the question on either the next timestep (note $\gamma = 0$ implies $G_t = r_{t+1}$) or over an extended temporal horizon for $\gamma \in (0, 1)$. The linear TD(λ) algorithm learns to approximate the return by a linear function of the feature vector x_t, with

$$\hat{g}(x_t) = \theta_t^\top x_t, \tag{2}$$

where $\theta_t \in \mathbb{R}^n$. Prediction is computationally efficient in that the time and space requirements are linear in the feature vector size. The TD(λ) algorithm adjusts the weight vector θ_t at each timestep to reduce the error between predictions on adjacent timesteps with the following update rules.

$$\begin{aligned} \delta_t &= r_{t+1} + \gamma\theta_t^\top x_{t+1} - \theta_t^\top x_t & (3)\\ e_t &= \gamma\lambda e_{t-1} + x_t & (4)\\ \theta_{t+1} &= \theta_t + \alpha\delta_t e_t & (5) \end{aligned}$$

Here $e_t \in \mathbb{R}^n$ is called the trace (and is initialized to the zero vector), and $\alpha \in [0, 1)$ is a step size parameter. The value $\lambda \in [0, 1]$ is the trace decay parameter. When $\lambda = 1$ and α is slowly decreased over time to zero, this algorithm converges to a weight vector θ_* that minimizes the squared error between the predictions and the return. However, the algorithm is often used with $\lambda < 1$ for faster learning, and with α set to a constant value to enable adaptation to a dynamic environment. Note that the update at each timestep requires time that is linear in the feature vector size.

Under common assumptions [Sutton and Barto, 1998], the update rules will adapt $\hat{g}$ to approximate g, a general value function that is the expected value of the return when starting at the environmental state s.

$$\hat{g}(x(s)) \approx g(s) \equiv E[G_t | S_t = s]$$

A common oversight is to consider TD(λ) as only appropriate for learning a value function that describes the robot's behaviour. It is in fact a general algorithm for making multistep predictions, and was described as such when introduced [Sutton, 1988]. Although this algorithm is often used in reinforcement learning to pursue goal-directed behaviour, it can be used for an arbitrary function r_t of state.

We propose taking advantage of the computational efficiency of the TD(λ) algorithm to learn a set of m predictions,

$$\{(r^{(1)}, \gamma^{(1)}), \ldots, (r^{(m)}, \gamma^{(m)})\}$$

that can be learned and predicted in parallel from the single stream of robot behaviour. Each predictive question has its own target signal r and constant γ. As the learning and prediction algorithms are both linear in the n-dimensional feature vector, the computational complexity and memory requirements of this approach grow as $O(mn)$, which on modern computing systems enables the use of many features and many predictions in real-time. Moreover, this approach is intrinsically parallel and decoupled, which enables flexible deployment on parallel computing architectures.

The goal of learning to make many predictions in parallel and in real-time on a robot raises different issues than are often considered for reinforcement learning experiments. In particular, manually tuning the learning parameters for each question is impractical. Instead, the learning parameters should enable stable learning. As such, values for α and λ are shared across all the questions. Furthermore, the feature vector is shared across all the questions. This problem setting encourages the use of diverse features and a large feature vector, to enable learning better predictions for the diverse set of questions. Note that in the online setting with an abundance of data, increasing the space of features is generally not harmful.

We define this scenario of learning and predicting, in real-time on a robot as *in situ* learning. Two competing desires for *in situ* learning must be balanced. First, learning and prediction are often only one piece of the larger system, so their computational and memory footprints should be reasonable. Second, the predictions should exhibit accuracy within the robot's lifetime. Several approaches in robot learning are computationally expensive but learn from limited experience. However, modern robots have extended operational lifespans of days and years, so computationally efficient real-time algorithms can run on top of these operational systems with little overhead, and potentially enable substantial learning to occur directly from the stream of robot experience.

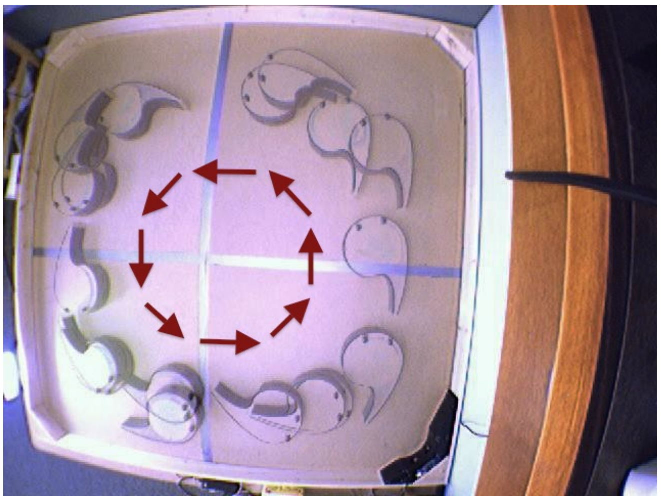


Figure 1. A robot performing a regular though non-periodic behaviour of following the walls in the pen. A lamp shines in one corner of the pen, and its light is observed by some of the robot's sensors.

3 Evaluation

To evaluate the practicality of the above method on a real system we considered *nexting* predictions, namely predictions about the future value of sensors (and many feature vector components), at a variety of time scales[2]. By predicting what will happen next, the robot gained a basic knowledge of its interaction with the environment. The robot performed an extended wall-following behaviour in a small pen (Figure 1). The observation stream contained both repeated events (such as passing a light, driving forward, and periods when the motors are cooling off), along with fine structure (such as variations in the accelerometers without readily apparent structure). The behaviour exhibited substantial variations, for example the time to complete a loop of the pen varied from 20 to 40 seconds, and there were seven-minute

[2] This experiment is described with additional details in [Modayil et al., 2012]

resting periods with no motion to allow the motors to cool off. Every 100ms, the robot generated an observation vector with 53 components. They cover 11 different sensing modalities and 4 software variables that are listed in Figure 2.

Sensor Group	Group Size	Tiling Type	(resolution, tilings)
IRDistance	10	strip	(8,8)
		strip	(2,4)
		skip(0)	(4,4)
		skip(1)	(4,4)
Light	4	strip	(4,8)
		skip(0)	(4,1)
IRLight	8	strip	(8,6)
		strip	(4,1)
		skip(0)	(8,1)
		skip(1)	(8,1)
Thermal	8	strip	(8,4)
Rotational Velocity	1	strip	(8,8)
Mag	3	strip	(8,8)
Accel	3	strip	(8,8)
MotorSpeed	3	strip	(8,4)
		skip(0)	(8,8)
MotorVoltage	3	strip	(8,2)
MotorCurrent	3	strip	(8,2)
MotorTemperature	3	strip	(4,4)
OverheatingFlag	1	strip	(2,4)
LastAction	3	strip	(6,4)

Figure 2. Summary of the tile-coding strategy for producing the feature vector from the sensory observations. Sensors values in each group were tiled either singly (strip tilings) or jointly pairwise (skip tilings). The last column indicates how many tilings of each type were made for each sensor or sensor group, and how many intervals (resolution) were involved in each dimension of each tiling. See text for explanation.

The observation vector is transformed into the agent's representation x_t by tile coding. This produced a binary vector, $x_t \in \{0,1\}^n$, with a constant number of 1 features (see [Sutton and Barto, 1998] for more details on tile coding, in short a tile coder maps data from a continuous domain into a binary representation by a set of indicator functions whose support tile the continuous domain). The features provided no history and performed no averaging of sensor values. The tile coder was comprised of many overlapping tilings of individual sensors and pairs of sensors (see Figure 2). The *resolution* of a tiling refers to the number of uniform partitions per dimension. When multiple tilings covered a space, each had a random offset. The sensory signals were partitioned based on sensor modalities into IR(InfraRed)Distance, Light, Thermal, IRLight, MotorSpeed, MotorCurrent, MotorVoltage, MotorTemperature, Acceleration, Magnetometer and LastAction. Within each sensor group, each individual sensor (e.g., Light0) was tiled independently as multiple one-dimensional overlapping grids called *strip* tilings. Additionally, pairs of sensors within a group (e.g., IRLight$i$ and IRLight$j$) were tiled together using multiple two-dimensional overlapping grids. The two-dimensional grids combined sensors in one of two ways. When they combined sensors within a group that were directly spatially adjacent on the robot, we call it a *skip(0)* tiling, whereas a *skip(1)* tiling combines sensors that are spatially adjacent with a skip of one (e.g., IRDistance1 with IRDistance3, IRDistance2 with IRDistance4, etc.). All in all, this tiling scheme produced a feature vector with $n = 6065$ components, most of which were 0s, but exactly 457 of which were 1s, including one bias feature that was always 1.

We applied TD(λ) to learn 2160 predictions in parallel. For the first 212 predictions, the target signal, $r_t^{(i)}$, was the sensor reading of one of the 53 sensors listed in Figure 2 and the discount rate was set to one of four timescales; for the remaining 1948 predictions, the target signal was set to one of 457 randomly selected bits from the feature vector and the discount rate was again set to one of four timescales. The discount rate $\gamma^{(i)}$ was one of the four values in $\{0, 0.8, 0.95, 0.9875\}$, corresponding to time scales of approximately 0.1, 0.5, 2, and 8 seconds respectively. For each question, the step-size parameter is set to $\alpha = \frac{0.1}{457}$ ($\frac{1}{10}$th of the number of active features), and the trace parameter is set to $\lambda = 0.9$. The initial weight vector was initialized to 0.

An initial performance question is scalability, in particular whether so many predictions can be made and learned in real time. We found that the total computation time for a cycle under our conditions was 55ms, well within the 100ms duty cycle of the robot. The wall-following policy, tile-coding, and the TD(λ) learning algorithm were all implemented in Java and run on a laptop connected to the robot by a dedicated wireless link. The laptop used an Intel Core 2 Duo processor with a 2.4GHz clock cycle, 3MB of shared L3 cache, and 4GB DDR3 RAM. The system garbage collector was called on every timestep to reduce variability. Four threads were used for the learning code. For offline analysis, data was also logged to disk for 120000 timesteps (3 hours and 20 minutes). The total memory consumption was 400MB. Note that with faster computers, the number of predictions or the size of the weight and feature vectors can be increased at least proportionally. This strategy for prediction should be easily scalable to millions of predictions with foreseeable increases in parallel computing power over the next decade.

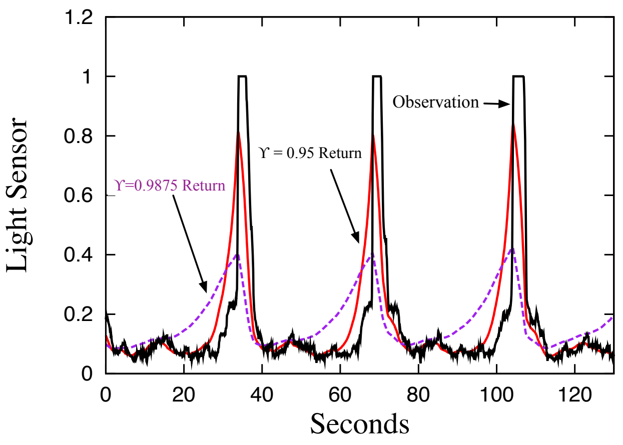


Figure 3. A plot of the returns for one light sensor. The sensor readings exhibit sharp changes when the robot passes the lamp. The returns for each question are computed at the end of the experiment.

Next, we consider one of these nexting predictions in detail. Each question asks what will happen next over a relatively short, but temporally-extended, time scale. Consider the robot's ability to anticipate when one of its light sensors will be saturated as it passes the lamp in one corner of the pen. Examples of returns for different time-scales are shown in Figure 3. The returns for each question are computed from the stored log of observations using Equation 1. For each point in time t, the value of the return constitutes the empirical ground truth answer for the question.

Once the returns are computed offline, they can be compared to

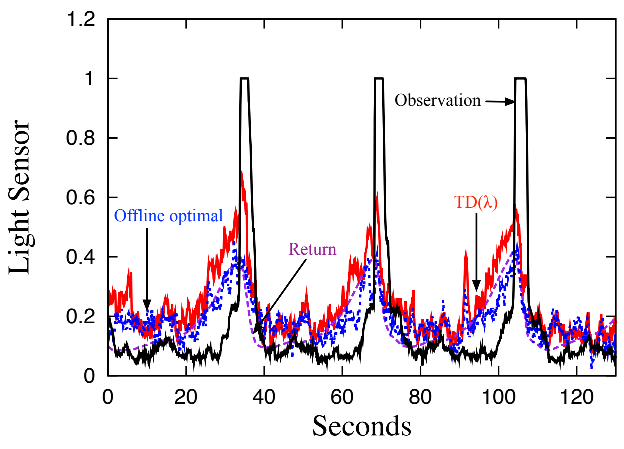


Figure 4. The learned prediction of TD(λ) closely matches the return, and the predictions are qualitatively similar to the best predictions that can be made with the given features (the best linear predictor for these features was computed offline). The prediction also has the desired qualitative structure of rising in advance of changes in the light signal (the observation).

the predictions that were made in real-time during the experiment, as seen in Figure 4. The predictions in the graph show a clear example of anticipating the increase in light. The return and the learned prediction are in close correspondence. The performance of the learned predictions is also similar to the performance of the best weights, θ_*, that minimize the mean squared error on the dataset, which were computed offline for the given set of features.

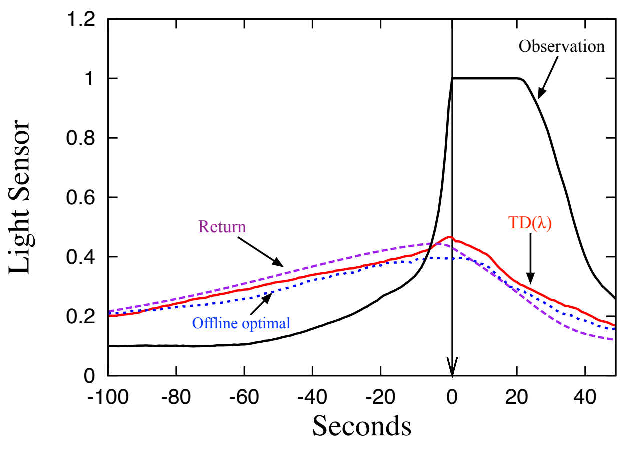


Figure 5. The observations, returns and predictions from a temporal window aligned at the event of the onset of light saturation. An average over 100 events shows the performance in the mean. The learned predictions are closely aligned with the returns and the offline optimal answers in the mean.

The solution quality for this question around a significant event is examined by aligning fixed length windows at the onset of light saturation. Within each window, the values for the observations, returns and predictions were averaged over the 100 events. Figure 5 shows that the predictions and the returns are well-matched in the mean.

Having demonstrated that accurate prediction is possible, we now consider the rate of learning in Figure 6 by comparing the root mean square errors of the prediction algorithms as compared to the true returns computed from future experience. We can see that the error comes down quickly for all the online algorithms. The comparison to the predictive performance of the offline-optimal solution shows a vanishing accuracy gap with $TD(\lambda)$ by the end of the experiment. For contrast, we also show the learning curve for a trivial representation consisting only of a bias unit (the single feature that is always 1).

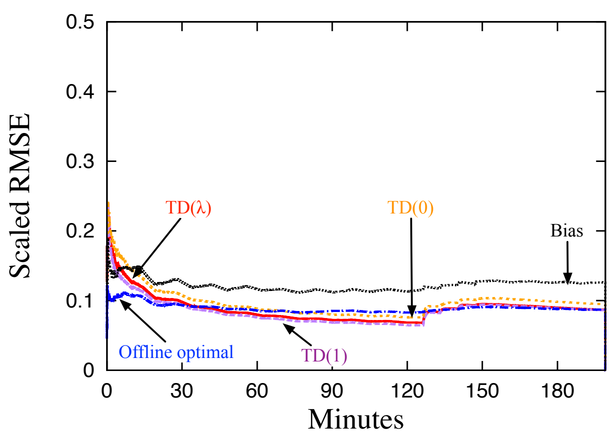


Figure 6. Learning curves for the 8-second light sensor nexting predictions. The predictions have had their root mean squared error (RMSE) scaled by $\frac{1}{1-\gamma}$. The graph compares the errors of different learning algorithms. The jog in the middle of the graph occurs when the robot stops by the light to cool off its motors, causing the online learners to start making poor predictions. In spite of the unusual event, the TD(λ) solution still approaches the offline-optimal solution. TD(λ) performs similarly to a supervised learner TD(1), and they both slightly outperform TD(0). The curve for the bias unit shows the poor performance of a learner with a trivial representation.

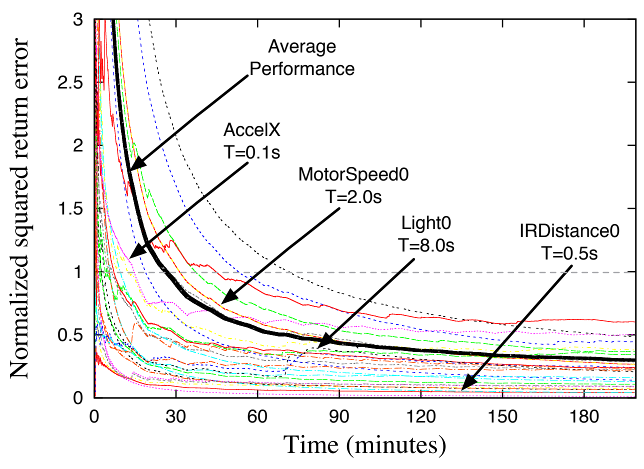


Figure 7. The presented method learned to answer 2160 questions in real-time about future sensor observations generated by the mobile robot's behaviour. The questions pertain to the expected values over the near future at timescales of 0.1, 0.5, 2, and 8 seconds. Only a handful of the learned predictions about sensors and features are shown above; sensors are diverse and include motor temperatures, currents, voltages, light sensors, infrared light sensors, ambient temperature, magnetometers, accelerometers, and others. The predictions are made in real-time at a rate of 10Hz. The answers have substantial accuracy; the error shown for each prediction is normalized by the observed variance (Equation 6). Moreover, substantial learning is achieved within the first 30 minutes of experience, as is seen in the error curve for the average performance.

Figure 7 demonstrates a key result, namely that many accurate answers to predictive questions can be learned in parallel from standard robot behaviour. To compare the accuracy of the different questions, the prediction errors are normalized by the sample variance of the returns for each question over the entire dataset. This yields a normalized squared return error (NSRE),

$$\mathrm{NSRE}(\hat{g}^i, t) = \frac{1}{t} \frac{\sum_{k=0}^{t} (\hat{g}^i_k - g^i_k)^2}{\mathrm{Var}(g^i)}. \tag{6}$$

The NSRE value represents the percentage of the variance in the return that remains unexplained by the predictor.

For every question, we can observe in Figure 7 that the error decreases along an exponential curve. Substantial learning occurs in the first 30 minutes, but errors continue to decrease with additional experience. Note that the error expresses the percentage of the sample variance unexplained, and that for every question this falls below 1. Thus, the answers are non-trivial even for a noisy sensor such as an accelerometer at long time scales—the answers learned by the system with the given experience and choice of representation outperform the best constant prediction for every single question. This highlights the potential benefit from large informative feature sets.

These results demonstrate our novel and somewhat surprising claim that it is practical to acquire a broad range of knowledge in real time directly from experience. We have shown a method with a sound theoretical foundation that learns answers to thousands of different empirical questions in practice, from regular robot experience. The method is scalable in the number of questions and features, as the amount of computation is linear in each. It is robust in practice, as no individual tuning is required for the different questions. It supports parallel implementation, which was used in dividing the computation across multiple threads. This method provides access to knowledge about multiple timescales while operating at a single fast timescale. This is an impressive set of properties.

Viewed from another perspective, this approach is one way to enable robots to tap into the phenomenon of big-data in machine learning, namely that large, simple, discriminative methods will outperform small, complex, generative models when given sufficient data. Substantially more structure often exists in the robot's experience than is predicted by small generative models. This has been seen in several domains, including games, search engines, recommender systems, and even in Jeopardy. The method described here is one simple way to provide a robot with immediate access to knowledge about many facets of its observable existence. This is both different from the way knowledge is typically considered on a robot, and opens the door to methods that can leverage a diverse body of knowledge.

4 An Illustrative Application

The method generalizes to any robot, and now we describe an application to a robot arm.[3] This application also shows how predictive empirical knowledge can be interesting, potentially useful, and difficult to acquire by other means. This demonstration takes as its setting the problem of human-machine interaction. Other researchers have presented a robotic platform for familiarizing new patients with the process of controlling a powered prosthetic arm [Dawson et al., 2012]. This *myoelectric training tool* (MTT) includes a table-top robotic arm (Figure 8), which a patient must learn to control using signals from their remaining muscles.

[3] This experiment is described with additional details in [Pilarski et al., 2012]

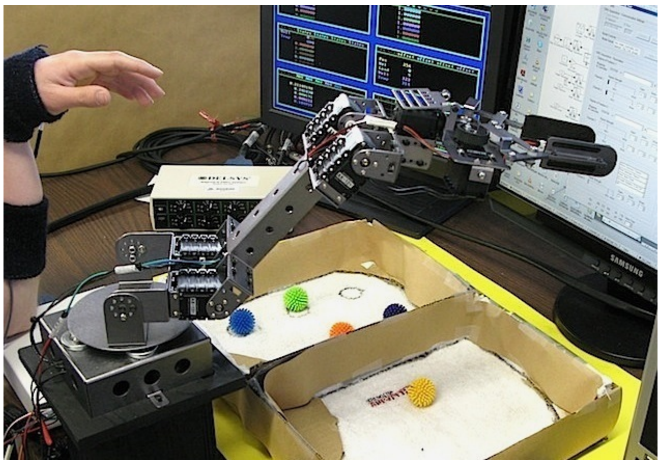


Figure 8. The myoelectric training tool (MTT), a multi-joint robotic prosthesis used to train new amputees.

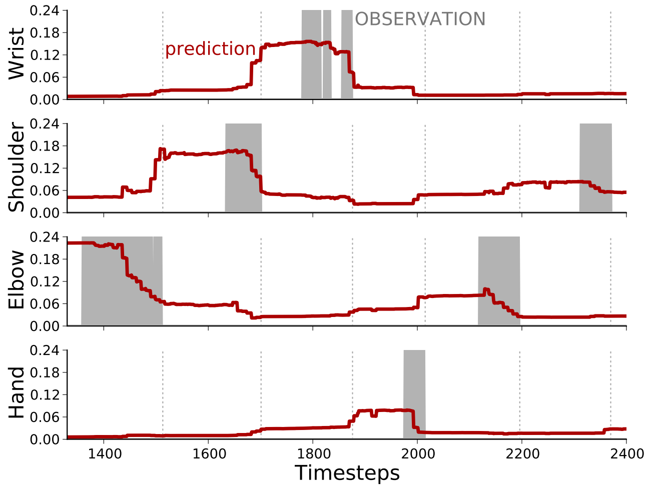


Figure 9. The user must perform a task, manually switching between four degrees of freedom. The shaded blocks indicate when a joint is active (the observed signal), and solid red lines indicate the system's predictions after less than 15min of online learning; intervals with no activity on any joint are switching times. The grey vertical bars indicate the end of a joint activity and thus the start of a switching period. The system learns to anticipate which degree of freedom the user will move next. These choices will vary across users and even within a task for a single user.

The MTT enables four degrees of freedom, but the typical lack of control sites on an amputee patient restricts the number of available control channels. As per standard commercial prostheses, control is therefore multiplexed, with one channel being used to switch between joints in a cyclic order, and a second control channel selecting the currently active joint. However, a user can spend an unacceptably large portion of their time selecting which joint they wish to move next when using switching-based approaches of this nature, as shown by the time periods with no joint activity in Figure 9.

We applied nexting predictions to examine if they can support user switching in this setting. In these experiments, four predictions were defined, one for the user-driven motion of each joint—the learning agent's goal was to predict which joint the human user would use next. These predictive questions represent temporally extended expectations for motor activity on each of the MTT's four joints. For this task, the robot was operating with a duty cycle of 20ms, and nexting questions were set to have a timescale of 2.5 seconds ($\gamma = 0.992$). The feature vector used by the learning system was generated by tiling together all four joint angles with each of the other 28 sensors provided by the MTT system. The resulting feature vector was sparse, consisting of 1,306,369 features, of which 169 were active at any given time.

Figure 9 shows the results of the nexting predictions after 15 minutes of online learning. The predictions often anticipate which joint the user will move next, as shown by the fact that the joint that is selected next often has the greatest magnitude of the four predictions. This information could be used to change the joint selection ordering, so that instead of cycling through a fixed order, the system would cycle through the joints in an order given by the nexting predictions. Nexting-based joint selection of this kind was found to decrease the number of switching commands that a user would have to provide, and thus the total projected time used for transitions, as calculated using the mean times observed for transitions involving one, two, or three user switching actions (Figure 10). The projected transition cost for the adaptive order was then compared to the cost for the best possible fixed switching order, as computed post-hoc from the recorded data. Based on this comparison, it was found that nexting predictions could facilitate a switching time decrease of more than 14% on this task (Figure 10). Moreover, the time taken for switching was found to rise monotonically with the number of switch commands. This means that with adequate feedback to the user, this predictive approach could reduce the amount of real time a patient spends on tasks.

This approach to adapting the switching order demonstrates one direct benefit of learning *in situ*. This is a scenario where, even though a person is always in control of the actions being performed, the robot can make the user's life better by learning to anticipate what the user will want next. As shown in these results, the best fixed ordering for the task is outperformed by an adaptive ordering. Given the fact that a user will switch between several tasks and can solve the same task in different ways, it is difficult to see how a non-adaptive approach could achieve the same benefits. Moreover, in spite of the relatively large number of variables and the large feature space, learning is still computationally and data efficient, as this level of performance is reached in 15 minutes. All learning related computations were completed within 5ms per iteration on the same laptop used in the earlier experiment.

Transition with 1 switching actions, mean time:	1.09 sec
Transition with 2 switching actions, mean time:	1.75 sec
Transition with 3 switching actions, mean time:	2.21 sec
Net experiment time:	20.66 min
Net observed transition time:	10.40 min
Net transition time(projected for best fixed order):	9.98 min
Net transition time(projected for adaptive order):	8.49 min

Potential time savings with adaptive control:	1.49 min
Potential time savings on transitions:	14.3%
Potential time savings on full experiment:	7.2%

Figure 10. Additional performance numbers for the switching task, including projected time savings from nexting predictions.

5 Related Work

Much previous work on reinforcement learning for real-time robotics, has focused on its role in control. For example, the Natural Actor-Critic algorithm [Peters and Schaal, 2008] has a critic that is making a single prediction. Other reinforcement learning approaches focus on policy evaluation without a predictive component, as used for example in improving a quadruped walk [Kohl and Stone, 2004]. Previous work with reinforcement learning on robots has not demonstrated learning of so many temporally-extended predictions in real-time.

Related to the idea of learning many predictions in parallel, is the idea of constructing optimal predictions for a set of tests [Talvitie and Singh, 2011]. The domains differ greatly however, as in that work the emphasis is on constructing the most accurate predictive answers for partially observable systems that have a small discrete set of observations, whereas this paper is concerned with satisfying the constraints of learning in real-time with continuous data on a robot.

The most similar work to the current system is an online variant of an offline spectral method for making many temporally-extended predictions [Boots et al., 2011]. Their work differs from the work presented here in several important ways. Although their algorithm is incremental and online, they do not demonstrate that it operates well in real-time. Moreover, is not clear how readily their approach can satisfy real-time constraints on an embedded system because their algorithm uses computationally expensive matrix operations and sophisticated data transformations. Their algorithm requires a window of past and future observations, which presents an additional memory requirement. Finally, their algorithm strongly couples the various prediction questions to discover joint structure, and coupling the problems together in this way prevents direct parallel implementations. Despite these important distinctions in implementation between the two methods, it is possible that ideas from both methods can be fruitfully combined because of their similar learning objectives.

Despite the lack of directly comparable work, considerable previous work also examines the task of predicting the temporally extended consequences of behaviour. However, the task has often been addressed with model-based roll-outs of a one timestep dynamical model. This requires the acquisition of a one timestep model of robot dynamics, either analytically or offline from logs [Thrun and Mitchell, 1995], and then using this model for adapting control. Another line of work uses online, real-time learning with a large memory of past experiences that are used to dynamically construct local models [Atkeson et al., 1997]. However, that work does not develop the use of local models for temporally extended predictions about many different targets. Moreover, their approach is computationally intractable for systems that lack an underlying low dimensional description, such as a mobile robot with a diverse set of sensing modalities.

Recent work on autonomous helicopter control [Abbeel et al., 2010] involves real-time control that balances computational complexity with a cost-to-go function (a variant of a value function) and simulations of the system dynamics at 20Hz for a two-second horizon. Similarly, another approach used roll-outs of a one timestep model to ensure that a robot with substantial inertia can both move quickly and stop safely [Fox et al., 1997]. These approaches illustrate the computational expense of generating temporally-extended predictions with one timestep models.

An important difference for robotics between reinforcement learning algorithms and the more common choice of Bayesian prediction algorithms, lies in the use of a feature vector instead of generative models of observation and state transition probabilities. In practice, it is easier to provide features of the state than to construct a full generative model for the dynamics of a sensor's interaction with the environment (such as accelerometers, or the behaviour of a person guiding a robot arm). It is not uncommon for several aspects of a robot's eventual interaction with its environment to be poorly understood by a designer. Robots are deliberately sent into novel domains with complex dynamics including underwater exploration, space, disasters, and human bodies. The control of these robots can vary between full-autonomy to full-teleoperation, with more human guidance when the situations are difficult to model a priori. System designers often have substantial knowledge gaps about the environments into which robots are deployed. If a robot had the ability to safely acquire relevant knowledge from experience, this ability would enable substantial flexibility for designers and end-users of robots.

6 Broadening the Space of Questions (Ongoing and Future Work)

We have demonstrated a method that learns to make thousands of temporally extended predictions directly from robot experience in real-time. The predictions are answering questions about the future that are empirical and multi-scale, but are of a more constrained form than can be answered by simulation rollouts with a one-timestep model. In this section, we outline generalizations of our approach that substantially broaden the space of predictions that can be expressed. The generalizations are based on the theory of options [Sutton, Precup & Singh, 1999] and on allowing general value functions to be option conditional [Sutton et al., 2011]. These extensions are not meant to be novel—our contribution is scaling—but indicate how our approach could be further generalized in future work.

The first generalization is to permit γ to vary with the state, $\gamma = \gamma_t$, which enables questions with state based (pseudo)-termination to be posed. For example the question "How many timesteps will elapse until all the wheels stop turning?" can be posed by setting $r_t = 1$ and $\gamma_t = 0$ if the observed velocity of every wheel is zero and $\gamma_t = 1$ otherwise. As another example, Figure 11 shows an example of the amount of power consumed until a light sensor is saturated or approximately two seconds have elapsed, where $\gamma_t = .95 \times \mathbb{I}_{\text{Light3}<\text{Saturation}}$, and

$$r_t = \Sigma_{i=1}^{3} \text{MotorCurrent}i_t \times \text{MotorVoltage}i_t.$$

This prediction was learned with the same parameters and feature representations used in our main result in Section III.

The next generalization is to add to the return an outcome z_t at termination. This allows questions to be expressed where the final state is relevant. For example, the robot's expected temperature on Motor2 when the Light3 sensor is saturated can be expressed by setting $r_t = 0$, $\gamma_t = .95 \times \mathbb{I}_{\text{Light3}<\text{Saturation}}$, $z_t = \text{MotorTemperature2}$. Incorporating z_t into the return is supported by standard TD(λ). For all the above generalizations of questions, the error in the learned predictions should decrease at rates comparable to those shown earlier.

A final generalization is to consider questions about different ways of behaving. If the robot behaves according to one policy then it is challenging to learn about the consequences of following a different policy (referred to as learning *off-policy*). The standard TD(λ) algorithm can diverge in this setting. The return can be expressed as

$$g^{r,z,\gamma,\pi}(s) = E[G_t^{r,z,\gamma,\pi} | S_t = s, S_{t+1} \sim \pi(S_t)],$$

where

$$G_t^{r,z,\gamma,\pi} = r_{t+1} + \ldots + r_T + z_T,$$

and the termination time T is distributed according to γ. This general question form is known as option-conditional prediction [Sutton et al., 1999].

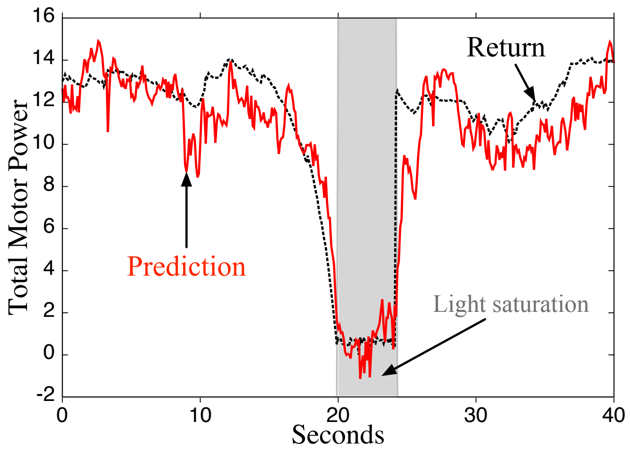


Figure 11. TD(λ) can learn to answer questions where γ varies with time for state based terminations and the addition of terminal outcomes. Here, we see predictions matching returns for the question of how much power will be used until the sensor Light3 is saturated, or spontaneous termination occurs with a 2 second horizon.

To learn to answer questions in real-time off-policy, one can use the GTD(λ) algorithm [Maei, 2011] whose update equations are shown below.

$$\begin{aligned}
\delta_t &= r_{t+1} + (1-\gamma_{t+1})z_{t+1} + \gamma_{t+1}\theta_t^\top x_{t+1} - \theta_t^\top x_t \\
\rho_t &= \frac{\pi(A_t|S_t)}{\pi_b(A_t|S_t)} \\
e_t &= \rho_t(x_t + \gamma_t \lambda_t e_{t-1}) \\
\theta_{t+1} &= \theta_t + \alpha(\delta_t e_t - \gamma_{t+1}(1-\lambda_{t+1})(e_t^\top w_t)x_{t+1}) \\
w_{t+1} &= w_t + \beta(\delta_t e_t - (x_t^\top w_t)x_t)
\end{aligned}$$

The primary computational differences between GTD(λ) and TD(λ) are an additional weight vector w, an associated step-size parameter β, and an explicit computation of the ratio ρ between π_b (the robot's behaviour policy) and π (the policy considered by the prediction).

This algorithm is a gradient-based generalization of the traditional TD(λ) algorithm. The algorithm learns an answer to the question g

specified by r,z,π, and γ as a linear function of the feature vector, with the same form of linear prediction, $\hat{g}(x_t) = \theta_t^\top x_t$, and thus the same linear complexity. The GTD(λ) algorithm maintains guarantees of stability when learning off-policy and converges to a fixed-point that minimizes the mean-squared projected Bellman error weighted by the distribution of states visited by the behaviour [Maei, 2011]. This somewhat technical objective is in some sense a natural one for online learning, as the algorithm minimizes the error that arises from the agent's limited perception (projecting environmental state onto the feature vector x), for the Bellman error (the difference between prediction and reality across adjacent timesteps), for the experience generated by the robot's behaviour that is relevant to the policy considered by the prediction.

An example of an off-policy scenario would be to learn, from the sensory experience of a car being driven by a person, to predict the time to come to a complete stop while braking. Many environmental aspects could influence stopping times while braking, including gravel roads, temperature, and rain. The ability to learn off-policy enables learning from all the snippets of experience when the driver touches the brakes, and not just the times when the car comes to a complete stop.

The space of questions that can be expressed in this final setting is quite general, and possibly covers all the interesting temporally-extended predictions that one can answer with one-step models [Sutton et al., 2011]. However, there remain numerous complications introduced by the general setting that make it unsuitable for the demonstration of scaling that is the focus of the present work. In particular, it is difficult to directly measure performance in an off-policy setting. In an off-policy setting, predictions are made about many different ways of behaving, but from each state only the predictions for one way of behaving can be tested at a time. Moreover, the tests alter the state and the state distribution from which experience is gathered. All these issues can probably be managed, but at the cost of significantly greater complexity, which would make the demonstration of scaling less compelling. Nevertheless, we plan to explore this direction in future work.

7 Conclusions

We have presented a demonstration that a robot can learn to answer temporally-extended predictive questions in real-time at scale: for thousands of questions, using thousands of features, with amounts of experience and computation that are commonly available on robots today. This approach provides a principled technique for a robot to acquire knowledge from experience in real-time about the temporally-extended consequences of its behaviour. We have described one potential use for this style of information as part of an adaptive user interface for a robot arm. The method is straightforward to deploy on different robots, and several directions are promising for further study.

ACKNOWLEDGEMENTS

The authors thank Mike Sokolsky for creating the Critterbot, and Thomas Degris for essential assistance with the experiments briefly reported in Sections 4 and 6. The authors would also like to acknowledge Michael R. Dawson and Jason P. Carey for their contributions to the myoelectric control study in Section 4. This work was supported by grants from Alberta Innovates - Technology Futures, the National Science and Engineering Research Council of Canada, the Alberta Innovates Centre for Machine Learning, and the Glenrose Rehabilitation Hospital Foundation.

REFERENCES

[Abbeel et al., 2010] Abbeel, P., Coates, A., and Ng, A. Y. (2010). Autonomous helicopter aerobatics through apprenticeship learning. *International Journal of Robotics Research (IJRR)*.

[Atkeson et al., 1997] Atkeson, C. G., Moore, A. W., and Schaal, S. (1997). Locally weighted learning. *Artificial Intelligence Review*, 11(1/5):11–73.

[Boots et al., 2011] Boots, B., Siddiqi, S., and Gordon, G. (2011). An online spectral learning algorithm for partially observable nonlinear dynamical systems. In *Proceedings of the Twenty-Fifth National Conference on Artificial Intelligence (AAAI)*.

[Dawson et al., 2012] Dawson, M. R., Fahimi, F., and Carey, J. P. (2012). The development of a myoelectric training tool for above-elbow amputees. *The Open Biomedical Engineering Journal*, in press.

[Fox et al., 1997] Fox, D., Burgard, W., and Thrun, S. (1997). The dynamic window approach to collision avoidance. *IEEE Robotics & Automation Magazine*, 4(1).

[Kohl and Stone, 2004] Kohl, N. and Stone, P. (2004). Machine learning for fast quadrupedal locomotion. In *Proceedings of the Nineteenth National Conference on Artificial Intelligence (AAAI)*, pages 611–616.

[LaValle, 2006] LaValle, S. M. (2006). *Planning Algorithms*. Cambridge University Press.

[Maei, 2011] Maei, H. R. (2011). *Gradient Temporal-Difference Learning Algorithms*. PhD thesis, University of Alberta.

[Modayil et al., 2012] Modayil, J., White, A., and Sutton, R. S. (2012). Multi-timescale nexting in a reinforcement learning robot. In *Proceedings of the International Conference on Simulation of Adaptive Behaviour (to appear)*.

[Peters and Schaal, 2008] Peters, J. and Schaal, S. (2008). Natural actor-critic. *Neurocomputing*, 71:1180–1190.

[Pilarski et al., 2012] Pilarski, P. M., Dawson, M. R., Degris, T., Carey, J. P., Sutton, R. S. (2012). Dynamic switching and real-time machine learning for improved human control of assistive biomedical robots. In *Proceedings of the 4th IEEE International Conference on Biomedical Robotics and Biomechatronics (BioRob)*, June 24–27, Roma, Italy, pages 296–302.

[Sutton et al., 1999] Sutton, R. S., Precup, D., and Singh, S. (1999). Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning. *Artificial Intelligence*, 112:181–211.

[Sutton, 1988] Sutton, R. S. (1988). Learning to predict by the methods of temporal differences. *Machine Learning*, 3:9–44.

[Sutton and Barto, 1998] Sutton, R. S. and Barto, A. G. (1998). *Reinforcement Learning: An Introduction*. MIT Press.

[Sutton et al., 2011] Sutton, R. S., Modayil, J., Delp, M., Degris, T., Pilarski, P. M., White, A., and Precup, D. (2011). Horde: A scalable real-time architecture for learning knowledge from unsupervised sensorimotor interaction. In *Proceedings of the 10th International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*.

[Talvitie and Singh, 2011] Talvitie, E. and Singh, S. (2011). Learning to make predictions in partially observable environments without a generative model. *Journal of Artificial Intelligence Research*.

[Thrun and Mitchell, 1995] Thrun, S. and Mitchell, T. (1995). Lifelong robot learning. *Robotics and Autonomous Systems*.

Continuous Adaptation of Robot Behaviour through Online Evolution and Neuromodulated Learning

Fernando Silva[1] and **Paulo Urbano**[1] and **Anders Lyhne Christensen**[2]

Abstract. We propose and evaluate a novel approach to the online synthesis of neural controllers for groups and swarms of autonomous robots. We combine online evolution of weights and network topology with neuromodulated learning in a completely decentralised manner. We demonstrate our method through a series of simulation-based experiments in which a group of e-puck-like robots must perform a dynamic concurrent foraging task. In this task, scattered food items periodically change their nutritive value or become poisonous. Our results show that when neuromodulated learning is employed, neural controllers are synthesised faster than by evolution alone. We demonstrate that the online evolutionary process is capable of generating controllers well adapted to the periodic task changes. We evaluate the performance both in a single robot setup and in a multirobot setup. An analysis of the evolved networks shows that they are characterised by specialised modulatory neurons that exclusively regulate online learning in the output neurons.

1 INTRODUCTION

Evolutionary computation techniques have been widely studied and applied in the field of robotics as a means to automate the design of robotic systems [5]. In evolutionary robotics (ER), robot controllers are typically based on artificial neural networks (ANN). The connection weights and sometimes the topology of the ANN are optimised by an evolutionary algorithm (EA), a process termed as *neuroevolution*. Evolutionary synthesis of controllers is usually performed offline in simulation, which presents a number of limitations. When a suitable neurocontroller is found, it is deployed on real robots. Since no evolution or adaptation takes place online, the controllers are fixed solutions that remain static throughout the robot's lifetime. If environmental conditions or task parameters become distinct from those encountered during offline evolution, the evolved controllers may be incapable of solving the task as they have no means to adapt.

Online evolution is a process of continuous adaptation that potentially gives robots the capacity to respond to changes in the task or in environmental conditions by modifying their behaviour. An EA is executed on the robots themselves as they perform their tasks. This way, robots are capable of long-term self-adaptation. In recent years, different approaches to online evolution have been proposed (see for instance [7, 20, 35, 36, 37, 38]). Notwithstanding, in such contributions, online neuroevolution has been limited to evolving weights in fixed-topology artificial neural networks. In a recent study [22], we proposed a novel approach called odNEAT, an online, distributed and decentralised version of NeuroEvolution of Augmenting Topologies (NEAT) [32]. NEAT is a state-of-the-art neuroevolution method that evolves the weights and the topology of an ANN. odNEAT shares some features with rtNEAT, a real-time version of NEAT designed for video games [30]. In rtNEAT, game characters are able to evolve online while they are playing against humans. Both NEAT and rtNEAT operate with access to global and centralised information. odNEAT, on the other hand, is completely distributed across multiple robots which have to solve the same task, either individually or collectively. odNEAT implements the online evolutionary process according to a physically distributed island model. Each robot acts like an island with genetic information being exchanged through intra-island variation (i.e., within a population encapsulated in one robot) and inter-island migration (between two or more robots).

By evolving neural topologies, odNEAT bypasses inherent limitations of fixed-topology online neuroevolution algorithms, in which the network topology has to be chosen *a priori*. Fixed-topology methods require a human to decide a suitable topology for a given problem, which usually involves intensive experimentation. Choosing an inappropriate topology affects the evolutionary process and, consequently, the potential for adaptation because: (i) Networks too large have extra weights, and each of these adds an extra dimension to the search space, and (ii) networks too small may be unable to represent solutions beyond a certain level of complexity, which potentially limits their performance. In odNEAT, on the other hand, a suitable network topology is the product of a continuous evolutionary process.

Online evolution is a form of online adaptation that acts at genotype level. Controllers produced are static as they do not change their parameters *while* they are controlling the robot. Whereas evolution produces phylogenetic adaptation, online learning operates on a much shorter time-scale. Online learning acts at phenotypic level and gives each individual controller the capacity to self-adjust during task-execution. Several studies indicate that learning can accelerate the evolution of good solutions, a phenomenon known as the Baldwin effect [9].

Agents controlled by ANNs can learn from experience by dynamically changing their internal synaptic strengths. This mechanism is inspired by how organisms in nature adapt to cope with dynamic and unstructured environments as a result of synaptic plasticity [18]. In this paper, we synthesise behavioural control for autonomous robots based on online evolution and online learning. In other words, we execute online both *phylogenetic adaptation*, associated with the development of the species, and *ontogenetic adaptation* which is associated with the learning processes in the individual.[3] We combine evolution of weights and network topology (odNEAT) with learning

[1] LabMAg, Faculdade de Ciências, Universidade de Lisboa (FC-UL), Lisbon, Portugal, email: {fsilva,pub}@di.fc.ul.pt
[2] Instituto de Telecomunicações, Instituto Universitário de Lisboa (ISCTE-IUL), Lisbon, Portugal, email: anders.christensen@iscte.pt

[3] These terms are in accordance with those used in [3].

through *neuromodulation* [26]. In biological organisms, neuromodulation is a form of synaptic modification involving modulatory neurons that diffuse chemicals at target synapses. Modulation has been suggested as essential for stabilising classical Hebbian plasticity and memory [2]. The combination of online evolution and neuromodulated learning allows the evolutionary process to explore two distinct kinds of plasticity[4]: *structural plasticity* is the generation of new connections and neurons, which in turn redefines the network topology; *synaptic plasticity* changes the strength of existing connections in a given topology.

We demonstrate our method in a simulated experiment where a group of e-puck-like robots [15] must perform a dynamic concurrent foraging task. Robots must locate and consume scattered food items. When a food item is consumed, a new item of the same type is randomly placed in the environment. At regular time intervals, food items change their nutritive value, or become poisonous. Besides learning to forage, robots must therefore be able to adapt and change their foraging policy in order to survive. To the best of our knowledge, the contribution presented here is novel in two aspects: (i) an online, distributed, and decentralised version of NEAT has not been studied prior to odNEAT, and (ii) this is the first demonstration of combined online evolution of *both* the weights and the ANN topology, and learning processes in multirobot systems.

2 BACKGROUND AND RELATED WORK

In this section, we first present the main features of NEAT, rtNEAT and odNEAT, and we then establish the relationship between these three neuroevolutionary methods. Finally, we discuss evolution of plastic ANNs, with a focus on the neuromodulation-based model.

2.1 NeuroEvolution of Augmenting Topologies

The NEAT method [32] is one of the most prominent neuroevolution (NE) algorithms. The method is capable of optimising both the topology of the network and its connection weights. NEAT relies on global and centralised information like canonical GAs. NEAT has been successfully applied to highly complex problems, such as double pole balancing, and has outperformed several methods that use fixed topologies [29]. The high performance of the algorithm is due to three key features: tracking genes with *historical markers* to allow meaningful crossover between topologies, *a niching scheme*, and evolving topologies incrementally from simple initial structures (*complexification*).

The network connectivity is represented through a flexible genetic encoding. Each genome is a list of connection genes, each of these referring to the two node genes connected. Furthermore, a connection gene encompasses the weight of the connection, a bit indicating if the connection gene is genetically expressed and a *global innovation number* (IN), unique for each gene in the population. INs represent a chronology of the genes introduced. With this feature, the difficulty of matching different network topologies (an NP-hard problem) is avoided and crossover can be performed without *a priori* topological analysis. During crossover, genes with the same historical markings are aligned to produce meaningful offspring. In terms of mutations, NEAT allows for common connection weights perturbations and structural changes that may lead to the insertion of: (i) a connection gene between two previously unconnected nodes, or (ii) a node gene, splitting an old connection into two new connections and disabling the former. Each new gene inserted receives an innovation number. This way, genomes representing networks of different topologies remain compatible throughout evolution because their origin is known.

The niching scheme is composed of two building blocks: speciation and fitness sharing. Speciation divides the population into non-overlapping sets of similar individuals based on a topological similarity measure. This mechanism protects new structural innovations by reducing competition between individuals representing differing structures and network complexities. In this way, newer structures have time to mature. If a species does not improve for a certain number of generations, it is removed from the population. Explicit fitness sharing dictates that individuals in the same species share the fitness of their niche. The fitness scores of existing members of a species are first *adjusted*, i.e., divided by the number of individuals in the species. Species then grow or shrink depending on whether their average adjusted fitness is above or below the population average.

The third reason why NEAT often outperforms other NE approaches is the incremental exploration of the search space. The algorithm starts with a uniform population of simple networks with no hidden nodes as in SAGA [8]. Complexity is introduced incrementally as a result of structural mutations. Since only structural mutations that have proven to be fit survive, the exploration of the search space is conducted in an incremental manner.

With the purpose of evolving increasingly complex ANNs online, rtNEAT was introduced [30]. Essentially, rtNEAT is a centralised real-time version of NEAT. rtNEAT contains some differentiating characteristics. While NEAT replaces the entire population at each generation, in rtNEAT one offspring is produced at regular intervals, every n time steps. The worst individual is removed and replaced with a child of a parent chosen among the best. Unlike NEAT, rtNEAT attempts to keep the number of species constant by adjusting a threshold C_t, which determines the topological compatibility of an individual with a species. When there are too many species, C_t is increased to make species more inclusive; when there are too few, C_t is decreased to be stricter. rtNEAT has shown to preserve the dynamics of NEAT, namely protection of innovation through speciation and complexification [29].

2.2 odNEAT: An online, distributed, and decentralised evolutionary algorithm

odNEAT runs across a distributed group of agents whose objective is to evolve and adapt while operating in the environment. Each agent is controlled by an artificial neural network that represents a candidate solution to a given task. Agents maintain a virtual energy level reflecting their individual task performance. The fitness value is defined as the average of the energy level, sampled at regular time intervals.

In odNEAT, each agent maintains a local set of chromosomes in an internal repository. The repository is a genetic pool that stores a limited number of chromosomes and their respective fitnesses. The stored chromosomes are arranged into species based on the niching scheme of NEAT. The set of chromosomes include the agent's current and previous active chromosomes and those received from other agents. Each agent probabilistically broadcasts its active chromosome to agents in its immediate neighbourhood, an *inter-agent reproductive event*, with a probability computed as follows:

$$P(event) = \frac{\bar{F}_k}{\bar{F}_{total}} \quad (1)$$

[4] These terms are in accordance with those defined in [17].

where $\bar{F}_k$ is the average adjusted fitness of *local* species *k* to which the chromosome belongs and $\bar{F}_{total}$ is the sum of all *local* species' average adjusted fitnesses. Due to the broadcast of genetic information, the active chromosome of an agent may be present in another agent's repository. Such migrations approximate in a distributed manner and over time the reproduction dynamics of rtNEAT. This way, each repository is a local mirror of what happens in the population at large, but no agent has a complete global view of the system.

Besides the internal repository, each agent also maintains a local tabu list, a short-term memory which keeps track of recent *poor* solutions: chromosomes removed from the repository or that caused the robot to run out of energy. Newly received chromosomes must first be accepted by tabu list. The *acceptance condition* is only met if the received chromosomes are topologically dissimilar from all chromosomes in the tabu list. After the pre-evaluation by the tabu list and if the acceptance condition was met, a received chromosome becomes part of the repository if it has a fitness score higher than the worst local chromosome. Due to the fixed size of the repository, whenever it is full, the insertion of a new chromosome is accompanied by the pre-requisite of removing the chromosome with the *worst adjusted fitness*. When a new chromosome is removed or added, the corresponding species has one less or one more element and therefore the adjusted fitness $\bar{F}$ is recalculated. Whenever an agent receives a copy *C'* of a chromosome *C* already contained in the repository (structurally the repository does not allow copies of the same chromosome), the energy level of *C'* is used to incrementally average the fitness of the *C* and provide a more reliable estimate of the chromosome's true fitness.

A particular characteristic of NEAT is the chronology of the genes due global to innovation numbers, which are assigned sequentially. In order to allow a decentralised implementation, odNEAT uses local high-resolution timestamps instead of innovation numbers. Each agent is responsible for assigning a timestamp to each local innovation, be it a connection or a node. Using high-resolution timestamps for labels practically guarantees uniqueness and allows odNEAT to retain NEAT's concept of chronology.

When an agent's energy reaches zero (because it is incapable of accomplishing the task), a new active chromosome is created. In this process — *an intra-agent reproductive event* — a parent species is chosen with probability proportional to its average fitness, as defined in Equation 1. Then, two parents are selected from the species, each one via a tournament selection of size 2. Offspring is created based on NEAT's genetic operators: crossover of the parents' genomes and mutation of the new chromosome.

Newly created chromosomes are given a certain amount of time α during which they control the agent, a *maturation period*. The maturation period gives the new chromosomes a change to spread their genome by mating with other agents. In Algorithm 1, we summarise odNEAT as executed independently by each agent.

2.3 Artificial evolution of neuromodulated plasticity

Synaptic plasticity is considered a fundamental mechanism behind memory and learning in biological organisms [11]. In ANNs, the modification of internal synaptic connection strengths can be performed according to a generalised Hebbian plasticity rule [18]. Synaptic weights are updated based on pre- and post-synaptic neuron activities as follows:

$$\Delta w = \eta \cdot [Axy + Bx + Cy + D] \qquad (2)$$

Algorithm 1 Pseudo-code of odNEAT that runs independently on every agent (see text).

```
$initialise\_genes()$
$energy \leftarrow default\_energy$
**loop**
    **if** $broadcast?$ **then**
        $send(all\_genes, agents\_in\_range)$
    **end if**
    **if** $has\_received?$ **then**
        **for all** $element$ **in** $received$ **do**
            **if** $tabu\_and\_repository\_accept(element)$ **then**
                $add\_to\_repository(element)$
                $adjust\_repository\_size()$
                $adjust\_species\_fitness()$
            **end if**
        **end for**
    **end if**
    $act\_in\_environment()$
    $energy \leftarrow update\_energy\_level()$
    **if** $energy \leq 0$ && $not(in\_maturation\_period?)$ **then**
        $add\_to\_tabu\_list(old\_controller)$
        $generate\_offspring()$
        $assign\_as\_controller(offspring)$
    **end if**
**end loop**
```

where η is the learning rate, x and y are the activation levels of the pre-synaptic and post-synaptic neurons. w is the connection weight and $A - D$ are respectively the correlation term, pre-synaptic term, post-synaptic term, and constant weight decay or increase. By tuning these parameters, it is possible to evolve distinct forms of synaptic plasticity. ANN controllers can thus implement learning and memory by means of recurrent connections, plastic Hebbian connections, or a combination of the two.

The adaptation capabilities of fixed-topology plastic Hebbian ANNs were demonstrated in [34]. In a light-switching task, a mobile robot Khepera had to turn on a light switch and then navigate towards a gray area at the opposite end of the environment. The evolved plastic Hebbian controllers managed to solve the task much faster than fixed-weight networks. The plastic controllers also exhibited a larger variety of successful behaviours and robustness to environmental changes. With a similar setup, it was shown that dynamic environments promote the genetic expression of plastic connections over static ones [6]. Plastic ANNs have also been successfully used in a variation of the classic foraging task denominated as the *dangerous foraging domain* [31].

Although the use of plastic ANNs can increase performance, recent studies indicate that in more complex tasks, both plastic and fixed-weight ANNs have limited learning capabilities [18, 26, 31]. In this context, controlling synaptic plasticity through *neuromodulation* was presented as a more powerful and biologically plausible approach [11]. In a neuromodulated network, specialised modulatory neurons control the amount of activity-dependent plasticity between pairs of standard control neurons. Therefore, control of plasticity is separated from the signal processing. This process is illustrated in Fig. 1.

Neuromodulation has been successfully applied to various domains. In [26], the authors presented results corroborating the favourable effects of neuromodulation when evolving adaptive ANNs for navigation and reward-collecting in both single and dou-

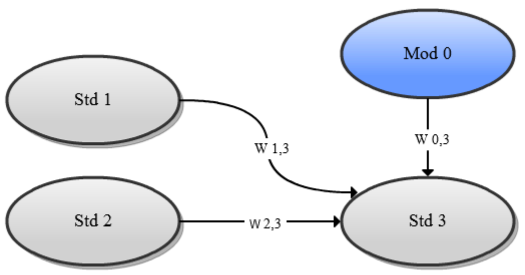


Figure 1. Neuromodulated plasticity. A modulatory neuron, Mod 0, transmits a modulatory signal to Std 3. Modulation affects the learning rate for synaptic plasticity of weights $w_{1,3}$ and $w_{2,3}$. The weights are part of the incoming connections for the standard control neuron being modulated.

ble T-maze. The experiments conducted demonstrated that, in some situations, the use of neuromodulation enables the evolution of high-performing neural controllers, whereas plain Hebbian plasticity does not. This result is similar to those presented, for instance, in [12, 23]. In [27], it was shown that neuromodulation allows the evolution of behaviours with a complex reinforcement learning dynamic. In a simulated foraging experiment, a simulated bee had to collect nectar on a field with two types of flowers. With the amount of nectar associated to each of the flowers changed stochastically, the bee evolved a value-based learning strategy which was found to perform efficiently in environments not seen during evolution. In [25], results obtained suggest that neuromodulation does not only allow for better learning, but also reduces the computation time in decision processes.

The main advantage of adding neuromodulation is that ANNs become capable of changing the degree of synaptic plasticity on specific neurons at specific times, i.e., deciding when learning should start and stop. In addition to its standard activation value a_i, each neuron i also computes its modulatory activation m_i as follows:

$$a_i = \sum_{j \in Std} w_{ji} \cdot o_j \tag{3}$$

$$m_i = \sum_{j \in Mod} w_{ji} \cdot o_j \tag{4}$$

where w_{ji} is the connection weight between pre- and post-synaptic neurons j and i. o_j is the output of a pre-synaptic neuron j. The weight between neurons j and i, with $j \in Std$, undergoes synaptic modification as follows:

$$\Delta w_{ji} = tanh(m_i/2) \cdot \eta \cdot [Ao_j o_i + Bo_j + Co_i + D] \tag{5}$$

2.4 Online evolution and learning

The current practice when combining evolution and individual learning mechanisms such as Hebbian plasticity or neuromodulation is to conduct evolution offline and learning online. Evolution is performed in a discrete and centralised manner. An external component creates an initial population and is responsible for selecting, mutating and replacing individuals. The evaluation process is based on repeated trials of experiments. For instance, in the T-maze experiments [26], an agent had to navigate through the maze, collect one reward and eventually return home. After that, a new trial started and the agent was tested for the ability to perform the same type of action. In the dangerous foraging domain [31], each trial contained only a specific type of food, either nutritious or poisonous. The ANN-controlled agent was evaluated by its ability to consume nutritious items, and stop consuming after trying a poisonous item. The robot did thus not have to explore to survive. One intuitive strategy would be to consume one poisonous item and then simply stop moving until the end of the trial. In a continuous evolutionary process, this kind of strategies is condemned to fail. In our domain, if a robot stops foraging after consuming a poisonous item, it will inevitably die. This way, online evolution presents a higher degree of difficulty as robots are evaluated continuously by their ability to perform the task.

In our proposed method, as previously mentioned, *both* evolution (of weights and topology) and learning are performed online. In order to encode neuromodulated plasticity, odNEAT's genetic encoding was augmented with a new modulatory neuron type. Each time a new neuron is added through structural mutation, it is randomly assigned either a standard or modulatory role. We augmented the genetic encoding with the learning parameters in Eq. 5. The five parameters are separately encoded and evolved in the range [-1,1] for A-D, and [-100,100] for η. It is important to note that there is no Lamarckian inheritance. Modifications in connection weights that occur as a result of neuromodulated learning are not passed on to offspring nor are part of the broadcasted chromosomes.

3 EXPERIMENTAL SETUP

This section describes the evaluation domain of our method, the robot model and the ANN's initial topology, and the common parameters across all experiments.

3.1 The concurrent foraging domain

The food foraging environment is a classical scenario to test adaptation and learning. The concurrent foraging task used in this study, a variation of the classical foraging, is performed in an environment with different types of items that can be consumed. The environment is a 3 x 3 meter square arena surrounded by blue walls. Each robot loses energy at a constant rate of 0.1 units/sec and therefore must learn to explore efficiently. The virtual energy level is limited to the range [0,100] energy units. This way, each robot is capable of surviving for approximately 17 minutes without consuming any (nutritious) food. There are two types of items, red items and pink items. Items of the same colour always have the same nutritive value, but at regular time intervals, the nutritious food items become poisonous or less nutritive and vice-versa. Robots able to sense the colour of nearby items but cannot determine the nutritive value of an item without consuming it. When an item is consumed, a new item of the same type is placed randomly in the arena. This way, the task remains dynamic while the sum of the energy value of the food items in the environment is kept constant.

In our experimental setup, the nutritive value of the different types of food changes periodically. Periods are composed of four phases of equal duration. At the beginning of each phase, the energy value of the different types of food items is set as listed in Table 1.

Table 1. The energy value of red and pink food items during the four phases. Values listed are in energy units.

	Phase 1	Phase 2	Phase 3	Phase 4
Red item	5	8	-3	3
Pink item	3	-3	8	5

To assess how robots adapt through time and what is the impact of neuromodulated learning on the task performance, we applied odNEAT with and without neuromodulation. For each configuration, we performed three sets of evolutionary experiments characterised by distinct phase durations p_d: (i) p_d = 9 min, (ii) p_d = 90 min, and (iii) p_d = 900 min.

The motivation for the concurrent foraging task is twofold: (i) since robots lose energy at a constant rate, they are required to evolve efficient exploration behaviours, (ii) when the nutritive values of the two types of food items change, the robots must be able to change their food gathering policy in order to survive.

3.2 Robot model and behavioural control

The simulated robots are modelled after the e-puck, a small (75 mm in diameter) differential drive robot capable of moving at speeds of up to 13 cm/s [15]. We have equipped each robot with an omni-directional camera similar to the one employed by the *s-bot* robots [1]. The image recorded is processed to calculate the distance, the red colour component, and the blue colour component of the closest object in each of the eight 45° sectors. The camera has a range of 50 cm and is subject to noise (simulated by adding a random Gaussian component within $\pm$ 5% of each of the three components' saturation value). Besides the camera, each robot has an internal energy level, comfort, and discomfort sensors. The energy sensor allows a robot to perceive its virtual energy level. The comfort and discomfort sensors indicate if the robot has consumed a poisonous or a nutritious food item, respectively. Note that the two sensors do not indicate *how* nutritious or poisonous a consumed food item is. That information is indirectly reflected by a new energy sensor reading, and the robot has to learn how to adapt its behaviour accordingly.

Each of the robots is controlled by an ANN synthesised by odNEAT. The ANN's connection weights $\in [-10, 10]$. The input layer consists of 27 neurons: (i) three for each 45° sector, measuring the red and blue colour components, and distance of the closest object, (ii) one neuron for each of the virtual sensors (energy, discomfort, and comfort). The output layer contains three neurons, one for each wheel of the robot, and one for the gripper. The gripper enables a robot to consume the closest food item within a range of 2 cm (if any).

3.3 Experimental parameters

When the energy level reaches zero, a new controller is generated and assigned maximum energy (100 units). In the generation of the new controller, two parents are selected from the local repository. Crossover and mutation are performed with probabilities 0.25 and 0.4, respectively. During mutation, the probability of adding a new neuron is 0.1 while a new connection is added with probability 0.05. Each connection weight is perturbed with probability 0.02 and a maximum magnitude of 2.5. Active chromosomes can be communicated up to a range of 1 meter and the local repository is capable of storing 30 chromosomes. Performance was found to be robust to moderate changes in these parameters.

4 RESULTS AND DISCUSSION

Two main experimental setups were conducted: (i) single robot setup, and (ii) multirobot setup. In the single robot setup, we evaluate the effects of neuromodulated learning when task-requirements change at different time-scales as in [21]. We also analyse the structural role of neuromodulation, i.e., how modulatory neurons are integrated in the ANN topology. In the multirobot setup, we apply with odNEAT with neuromodulation to robot groups of different sizes. We evaluate the impact of the group size on performance.

4.1 Effects of neuromodulated learning

To assess the impact of neuromodulated learning on the robots' task performance, we performed three sets of evolutionary experiments characterised by distinct phase durations p_d: (i) p_d = 9 min, (ii) p_d = 90 min, and (iii) p_d = 900 min. In order to avoid competition for food resources and interferences caused by the behaviour of other robots, only one robot was present in the environment in the first set of experiments. For each configuration, we placed five food items of each type and performed 30 independent runs. We consider those controllers stable that manage to survive at least 25 times the minimum survival time, i.e., approximately 7 hours of simulated time.

The results obtained are listed in Table 2. The average number of evaluations, i.e., the number of controllers tested by the robot to produce stable solutions, is illustrated in Fig. 2. odNEAT combined with neuromodulation required between 23.3% and 28.2% fewer evaluations than odNEAT without neuromodulation. For p_d = 9 min and p_d = 90 min, differences in the number of evaluations are not statistically significant ($\rho > 0.20$ and $\rho > 0.15$ respectively, Student's t-test). For p_d = 900 min, the differences are statistically significant ($\rho < 0.01$). These results suggest that, as the task-requirements become more stable, so does the performance of odNEAT with neuromodulation.

odNEAT alone failed to achieve stability in two evolutionary runs, one for p_d = 9 min and one for p_d = 90 min. In these runs, the longest surviving controllers were executing when the experiment as terminated after 100 hours of simulated time. At that point, the respective controllers had survived for 4.04 hours and 6.69 hours. In terms of gathered energy per period, the performance of the solutions is similar. Controllers with neuromodulation perform slightly better than solutions without neuromodulation. The most intriguing aspect is the fact *all* evolved solutions, whether modulated or not, present a similar performance. Evolution alone is thus capable of generating adequate solutions to the task.

Table 2. Summary of the results obtained for each of the three phase durations tested. The table lists the average number of evaluations required before stable solutions are evolved, and the average maximum age and gathered energy per period in each experimental setup.

Experimental setup with odNEAT			
Phase dur.	Evals.	Max Age (mins)	Gathered Energy
9 mins	39.02	3404.98 $\pm$ 1668.31	343.43 $\pm$ 35.38
90 mins	49.28	2886.88 $\pm$ 1399.20	3491.03 $\pm$ 334.49
900 mins	40.40	3041.81 $\pm$ 1446.78	42526.94 $\pm$ 6897.61
Experimental setup with neuromodulated odNEAT			
Phase dur.	Evals.	Max Age (mins)	Gathered Energy
9 min	29.52	3351.12 $\pm$ 1358.34	354.39 $\pm$ 46.19
90 min	37.79	2799.34 $\pm$ 1650.21	3530.82 $\pm$ 336.66
900 min	28.99	3074.33 $\pm$ 1283.85	45199.64 $\pm$ 6680.48

Neuromodulation allows a significant speed-up in adaptation time, which is important when adaptation is completely online. The results suggest an interplay between online evolution and learning. The idea that learning helps evolution by reducing the adaptation time is not new. There is much evidence that: (i) both processes are integral to the success of evolution in both biological and artificial systems [14, 19], and (ii) that learning can accelerate the evolution of good solutions [9], which is known as the Baldwin effect. As for

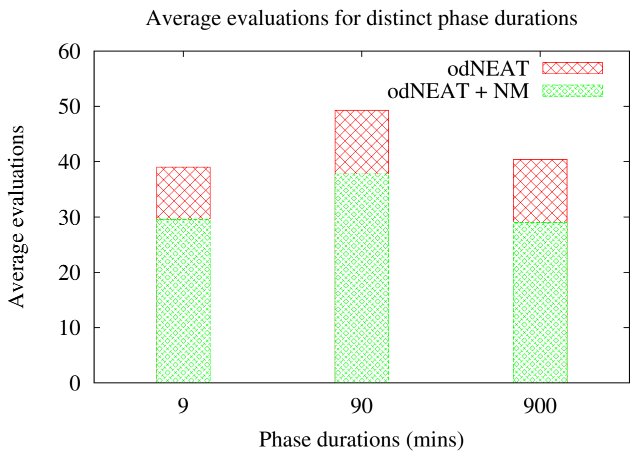


Figure 2. The average evaluations that each evolutionary method requires in order to produce stable controllers, within distinct phase durations. On average, odNEAT combined with neuromodulation required between 23.3% and 28.2% fewer evaluations than odNEAT without neuromodulation.

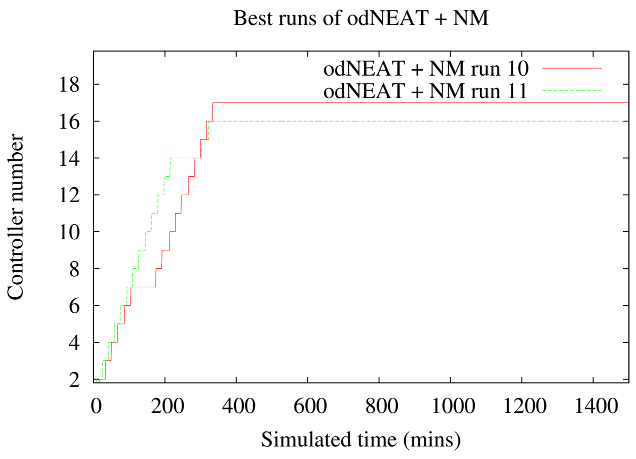


Figure 3. The best two runs for odNEAT with neuromodulation in the setup p_d = 9 mins. In both cases, stable controllers are produced in approximately 350 mins, an equivalent to 5.83 hours, and operate until the end of the respective experiment (6000 mins, 100 hours).

what neuromodulation concerns, our results further indicate that the additional complexity required to include modulatory neurons, and the corresponding increase in the search space, is compensated for by the learning ability and dynamics of modulated networks.

Depending on the experimental setup, the most stable controller of each run operated from approximately 47 hours to 57 hours of simulated time before the experiment was terminated. Figure 3 exemplifies the adaptation process that produces stable controllers for the two best runs of odNEAT with neuromodulation and p_d = 9 mins. Stable solutions are produced in less than 6 hours and operate until the end of the experiment (100 hours). These results indicate that the evolutionary process is capable of evolving controllers well adapted to the periodic changes in the nutritive value of the food items. The foraging behaviours of the two controllers are different from one another. While the controller synthesised in run 10 is greedy and therefore consumes more food items (including poisonous), the other controller exhibits the opposite strategy and restricts food consumption actions.

Table 3. Summary of the number of nodes and connections added to the initial network topology by each evolutionary method. NM stands for neuromodulation. Results for each configuration are averages over 30 independent evolutionary runs.

Evol. Method	Phase Dur.	Connections added	Neurons added
odNEAT	9 mins	26.43 ± 12.30	9.47 ± 3.95
odNEAT	90 mins	30.26 ± 17.32	10.41 ± 4.65
odNEAT	900 mins	25.17 ± 12.82	9.60 ± 4.31
odNEAT + NM	9 mins	22.89 ± 14.98	8.48 ± 4.83
odNEAT + NM	90 mins	29.32 ± 11.31	10.82 ± 3.91
odNEAT + NM	900 mins	28.91 ± 11.57	10.50 ± 3.57

ANNs evolved with and without neuromodulation have a similar topological complexity. The initial topology of stable solutions was augmented with a comparable number of connections and neurons (see Table 3). Topologies of similar complexity are synthesised faster by odNEAT with neuromodulation. This result suggests that when neuromodulation is used, odNEAT performs a more efficient exploitation of a given network topology. In fixed-weight networks, fine-grain adjustment of connection weights can only be achieved through mutation. Modulated networks allow for a different expression of a given topology's potential and are advantageous even when task requirements do not change for long periods (p_d= 900 mins). When modulatory neurons are present, solutions are synthesised after fewer controller evaluations, probably due to the modification of internal dynamics by each network.

4.2 Structural role of neuromodulation

The results presented above show that neuromodulated learning allows for faster synthesis of stable controllers. In this section, we analyse the structural role of neuromodulation on the most stable controllers of each independent run in order to determine how it affects internal neural dynamics.

Table 4. Summary of the most stable controllers in each independent run. The table lists the number of neurons and connections added to each network, and how many of these have a modulatory role.

	Phase Duration		
	9 mins	90 mins	900 mins
Neurons added	9.73 ± 4.88	11.97 ± 4.02	10.10 ± 5.07
Mod. Neurons	4.97 ± 2.92	6.07 ± 2.99	5.03 ± 3.36
Conns. added	23.93 ± 13.28	30.57 ± 10.58	25.67 ± 13.34
Mod. Conns.	6.37 ± 4.39	7.93 ± 4.34	6.97 ± 4.90

Table 4 shows the average complexity of each stable solution. Approximately half of the neurons added through structural mutation have a modulatory role. Modulatory actions are localised as each of these neurons typically connects to only one or two other neurons. A common topological characteristic between evolved solutions is that the majority of modulatory connections have output neurons as targets. Both topological aspects, the low density of connections per modulatory neuron and the fact that output neurons are the main target of modulation, have also been verified in distinct tasks and experiments [24]. This topological aspect is the main difference between solutions evolved with and without modulation, and what accounts for faster synthesis of sustainable ANNs.

Further analysis of neural topologies indicates that the evolutionary process often leads to the appearance of modulatory neurons that *exclusively* regulate output neurons. We define these units as *specialised* modulatory neurons due to the fact they only regulate the output actions. The percentage of specialised neurons from the total of modulatory neurons added is listed in Table 5. Depending on the experimental setup, 59% to 69% of the modulatory neurons inserted are specialised units. 6% to 9% of the specialised neurons modulate at least two output neurons. This way, specialised neurons are capable of simultaneous regulating, for instance, the left and right wheels and/or the gripper.

Table 5. Summary of the specialised neurons (modulatory neurons that only modulate the output neurons) for the best solutions of each evolutionary run. The table lists the percentage of modulatory neurons that are specialised in regulating the output neurons, and the percentage of specialised neurons that regulate each of the three outputs. LW and RW represent the left and right wheel, respectively. Gr represents the gripper.

Phase Dur.	Spec. Neurons (%)	LW (%)	RW (%)	Gr (%)
9 mins	69 ± 20	34	34	38
90 mins	62 ± 24	40	32	35
900 mins	57 ± 26	50	30	29

For p_d = 9 and p_d = 900 mins, differences in the number of specialised neurons are statistically significant ($\rho < 0.05$, Student's t-test). Analysis of experimental data shows that there is a higher regulatory activity of outputs for the setups of p_d = 9 mins and p_d = 90 mins, than for p_d = 900 mins. In these scenarios, controllers experience more environmental changes during task-execution. Food gathering policies must be flexible and change whenever a nutritious item becomes less nutritive or poisonous. With the increase of the duration of each phase, the task becomes less dynamic and the percentage of specialised neurons decreases. Existing specialised neurons increasingly focuses on movement (left and right wheels) and less on the gripping and food consumption actions.

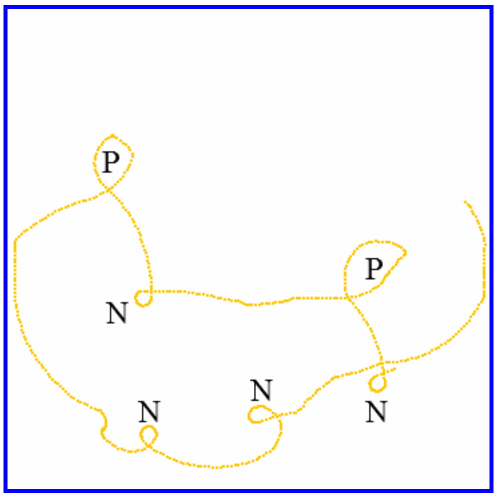


Figure 4. Evolved foraging behaviour for p_d = 9 mins. Nutritious food items, marked with 'N', are consumed by moving closely around it hence the small circular movements. Poisonous food items, marked with 'P', are avoided by performing wider circular trajectories.

In behavioural terms, foraging strategies evolved are quite distinct. Figure 4 shows one of the evolved behaviours by odNEAT with neuromodulation for p_d = 9 mins. In the presented environmental stage, there are both poisonous and nutritious food items. When a nutritious food item is detected, the robot moves closer to the item and consumes it (small circular movements). When a poisonous food item is detected, the robot moves in a wider circular trajectory and avoids the item.

4.3 Scalability experiments

odNEAT in a completely distributed evolutionary algorithm for online adaptation in groups of embodied agents such as robots. The EA is distributed across multiple robots which have to solve the same task, either individually or collectively. In odNEAT, each robot tries to evolve a solution to solve the same task, either individually or collectively. Exploration of the search space is therefore performed in parallel. Due to the physically distributed island model, each robot is able to propagate its current solution to other robots. Individual robots therefore contribute to the improvement of the entire group even in individual tasks.

In this section, we analyse the impact of group size on performance of odNEAT with neuromodulation. We performed 30 independent evolutionary runs for groups of 1, 2, 5 and 8 robots. Experiments were conducted with ten food items of each type. Phase durations were fixed at p_d = 90 mins. Active chromosomes can be communicated up to a range of 1 meter.

Table 6. Summary of the scalability experiments. Average evaluations required for producing a stable controller, average and maximum age of each stable controller. Results are averages over 30 independent runs.

Group Size	Average Evals. per Robot	Max Age (mins)
1	61.76 ± 37.43	2452.32 ± 1316.86
2	18.01 ± 9.35	5175.95 ± 1055.22
5	29.01 ± 43.73	5305.34 ± 828.30
8	59.65 ± 44.39	4814.39 ± 1162.09

Table 6 shows the experimental results obtained with each group size. For group sizes of 2 and 5 robots, the stable controllers were capable of surviving for 86.27 and 88.42 hours, respectively. Within the 8 robots group, this time decreased slightly to 80.24 hours. For groups of two robots, performance improved significantly ($\rho < 0.001$, Student's t-test) as the EA requires approximately 70.84% fewer evaluations to generate stable solutions. However, the set of 30 runs was characterised by two outlier runs in which the evolutionary required respectively 44.00 and 39.25 evaluations on average to produce sustainable solutions. For groups of five robots, performance also improved compared to the single robot setup. In this configuration, odNEAT with neuromodulation required approximately 53% fewer evaluations to evolve controllers well adapted to task changes. From the set of 30 runs, one is an outlier. In this run, odNEAT with neuromodulation required an average of 238.67 evaluations to evolve solutions well adapted to environmental changes. In fact, this outlier is the responsible for the high standard deviation in terms of average number of evaluations per robot. Excluding this run from the results, the average evaluations would be 24.81 ± 20.04. Performance levels would therefore be similar to the setup with a group of size 2.

Figure 5 exemplifies the adaptation process for a group of five robots during an experiment. In that experiment, before approximately 2000 minutes, an equivalent to 33.33 hours, *all* robots had generated stable controllers. After that, the controllers were able to survive until the end of the experiment. These results indicate that,

Table 7. Summary of the minimum and maximum evaluations required for producing stable solutions in each experimental configuration. The table also lists the number of runs considered as outliers (see text).

Group Size	Min Evals.	Max Evals.	Outlier Runs
1	8	186	0/30
2	5	44	2/30
5	7	239	1/30
8	8	164	2/30

even with task requirements changing periodically, robots are able to adapt and coexist in the environment for long periods of time.

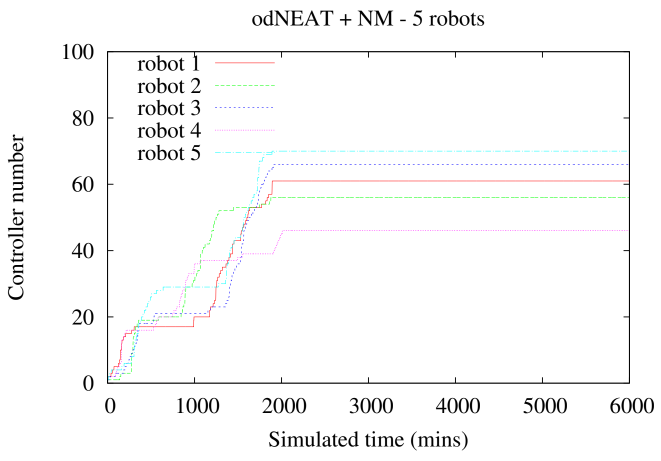


Figure 5. Adaptation process in a group of 5 robots. At the 2000th minute, all robots have a stable controller that operates until the end of the experiment (6000 mins).

For groups of 8 robots, evolution of stable controllers takes approximately the same number of evaluations as when only one robot is present. The set of runs is characterised by two outlier runs that require respectively 163.875 and 156.82 evaluations to generate sustainable solutions. Excluding these runs from the results, stable controllers would be evolved every 58.05 $\pm$ 37.03 evaluations. For groups of 8 robots, task difficulty increases significantly: for half the time, there are only ten nutritious food items to support the survival of the group. Solutions capable of coping with the setup complexity are intuitively harder to evolve, hence the number of evaluations required per robot.

One interesting aspect is that for group sizes up to 5 robots, there is a continuous decrease in the complexity added to the initial topology. These results are listed in Table 8. The reason for the reduction in the number of evaluations required, and less complex networks for groups up to 5 robots, is that each robot attempts to generate its own solution. When the setup includes only one robot, this one has to rely on its own gene pool to find a solution for the task. On the other hand, the presence of other robots in the environment makes odNEAT a parallel and distributed EA, similar to an island model [33]. In such a system, each robot acts like an island with genetic information being exchanged through inter-island migration. Good solutions are more likely broadcasted to other robots, therefore contributing to the iterative improvement of the entire group [22].

Table 8. Summary of the connections and neurons added to the initial topology of each stable controller for different group sizes. Results are averages over 30 independent evolutionary runs.

Group Size	Added Connections	Added Neurons
1	28.03 $\pm$ 22.92	9.37 $\pm$ 6.54
2	12.00 $\pm$ 9.96	4.92 $\pm$ 3.33
5	6.25 $\pm$ 2.47	3.57 $\pm$ 0.64
8	18.12 $\pm$ 12.76	7.30 $\pm$ 3.97

4.3.1 Tracking the exchange of genetic material

In order to determine to what extent is a robot affected by the gene pool of others, we analysed the origin of the information in each repository. Results are listed in Table 9. Similar solutions refer to the final, stable chromosomes that have at least 90% of their alleles in common.

Table 9. Summary of the genetic information in each robot. The table lists the percentage of chromosomes received and generated, from those stored in the repository, and the percentage of similar stable controllers for different groups sizes. Results are averages over 30 independent runs.

	Group Size		
	2	5	8
Chromosomes Received (%)	28.01	68.18	84.50
Chromosomes Generated (%)	71.99	31.82	15.50
Similar Stable Solutions (%)	53.33	54.00	20.69

With the increase of group size, the percentage of chromosomes received from other robots and stored in the repository also increases. For groups of 5 and 8 robots, the majority of the chromosomes in each repository was received from other robots (68.18% and 84.50%, respectively). Although for groups of 2 and 5 robots most of the genetic material stored is foreign to the robot, approximately 50% of the final controllers share 90% of their alleles, i.e., they are *similar*. With 8 robots, the percentage of similar solutions decreases to approximately 20.69%. For distinct groups sizes, there is a strong link between solutions exchanged among robots. Solutions propagated are frequently used by the evolutionary process embodied in the receiving robot. The results suggest that local genetic competition is an important part of the odNEAT's evolutionary dynamics.

Even stable solutions considered dissimilar, i.e., that share less than 90% of their alleles, have a relatively high percentage of genetic material in common. These values are listed in Table 10. For distinct group sizes (2, 5, and 8 robots), the average percentage of matching genes is 67.81%, 76.38% and 56.22%, respectively. The weights of matching connections are not very different between dissimilar controllers either. The average weight difference between matching connections is approximately 0.70, with each weight $w \in [-10, 10]$.

Table 10. Summary of the genetic information regarding dissimilar stable controllers, i.e., that share less than 90% of their genetic material. The table lists the percentage of matching and distinct genes, and the weight difference in matching connections. Results are averages over 30 independent runs.

Dissimilar	Group Size		
Solutions	2	5	8
Matching Genes (%)	67.81	76.38	56.22
Distinct Genes (%)	32.19	23.62	43.78
Weight Difference	0.66	0.70	0.70

5 CONCLUSIONS AND FUTURE WORK

In this paper, we introduced a novel approach to the online synthesis of behavioural control for groups and swarms of autonomous robots. We combined odNEAT and neuromodulated learning. While odNEAT evolves online both the weights and the topology of neural controllers, neuromodulation allows each individual controller to actively modify its internal dynamics. We demonstrated our method through a series of simulation-based experiments in which a group of e-puck-like robots had to perform a dynamic concurrent foraging task. When neuromodulation is present, the interplay between evolution and learning allows for a faster synthesis of solutions well adapted to task requirements. Our results further indicate that the additional complexity required to include modulatory neurons in neural topologies, and the corresponding increase in the search space, is compensated for by the learning ability and dynamics of modulated networks. We showed that neuromodulated learning accelerates evolution both when task-requirements change rapidly and when they remain stable for a long time.

Each modulatory neuron has a low density of modulatory connections as typically regulates one or two other neurons. Modulatory actions are mainly targeted at output neurons. In fact, the evolutionary process leads to the emergence of specialised modulatory neurons dedicated to *exclusively* regulating output neurons. These topological aspects are the main different between solutions evolved with and without modulation, and what accounts for the faster synthesis of sustainable controllers. The scalability experiments revealed that, for group sizes of 2 and 5 robots, odNEAT with neuromodulation scales well. Performance increases significantly in respect to the number of evaluations required to evolve stable solutions. For larger groups, of 8 robots, the complexity of the experimental setup increases drastically and stable solutions are harder to evolve.

In the future, we intend to investigate the basic requirements for truly open-ended evolution [4, 10, 28], in which the evolutionary process should be capable of producing a large variety of different and novel solutions to a given task. In this domain, open-ended evolutionary techniques such as novelty search have shown promising results [13, 16] and may complement our method.

ACKNOWLEDGEMENTS

This work was supported by FCT (Foundation for Science and Technology of Portugal) project PEst-OE/EEI/LA0008/2011.

REFERENCES

[1] C. Ampatzis, E. Tuci, V. Trianni, A.L. Christensen, and M. Dorigo, 'Evolution of autonomous self-assembly in homogeneous robots', *Artificial Life*, **4**(15), 465–484, (2009).

[2] C.H. Bailey, M. Giustetto, Y.-Y. Huang, R.D. Hawkins, and E.R. Kandel, 'Is heterosynaptic modulation essential for stabilizing hebbian plasticity and memory?', *Nature Reviews Neuroscience*, **1**(1), 11–20, (2000).

[3] *Adaptive individuals in evolving populations: models and algorithms*, eds., R.K. Belew and M. Mitchell, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, 1996.

[4] A.D. Channon and R.I. Damper, 'Towards the evolutionary emergence of increasingly complex advantageous behaviours', *International Journal of Systems Science*, **31**(7), 843–860, (2000).

[5] D. Floreano and L. Keller, 'Evolution of adaptive behaviour by means of darwinian selection', *PLoS Biology*, **8**(1), 1–8, (2010).

[6] D. Floreano and J. Urzelai, 'Evolutionary robots with on-line self-organization and behavioral fitness', *Neural Networks*, **13**(4–5), 431–443, (2000).

[7] E. Haasdijk, A.E. Eiben, and G. Karafotias, 'On-line evolution of robot controllers by an encapsulated evolution strategy', in *2010 IEEE Congress on Evolutionary Computation*, pp. 1–7. IEEE Press, Piscataway, NJ, (2010).

[8] I. Harvey, 'Evolutionary robotics and SAGA: The case for hill crawling and tournament selection', in *Workshop on Artificial Life (ALIFE '92)*, ed., C.G. Langton, volume 17 of *Sante Fe Institute Studies in the Sciences of Complexity*, pp. 299–326. Addison-Wesley, Reading, MA, (1993).

[9] G.E. Hinton and S.J. Nowlan, 'How learning can guide evolution', *Complex Systems*, **1**(3), 495–507, (1987).

[10] G. Kampis and L. Gulyás, 'Full body: The importance of the phenotype in evolution', *International Journal of Computational Intelligence and Applications*, **3**(167), 375–386, (2003).

[11] P. Katz, *Beyond Neurotransmission: Neuromodulation and its importance for information processing*, Oxford University Press, Oxford, 1st edn., 1999.

[12] T. Kondo, 'Evolutionary design and behavior analysis of neuromodulatory neural networks for mobile robots control', *Applied Soft Computing*, **7**(1), 189–202, (2007).

[13] J. Lehman and K.O. Stanley, 'Abandoning objectives: Evolution through the search for novelty alone', *Evolutionary Computation*, **19**(2), 189–223, (2011).

[14] G. Mayley, 'Guiding or hiding: Explorations into the effects of learning on the rate of evolution', in *4th European Conference on Artificial Life (ECAL'97)*, pp. 135–144. MIT Press, Cambridge, MA, (1997).

[15] F. Mondada, M. Bonani, X. Raemi, J. Pugh, C. Cianci, A. Klaptocz, S. Magnenat, J. Zufferey, D. Floreano, and A. Martinoli, 'The e-puck, a robot designed for education in engineering', in *9th Conference on Autonomous Robot Systems and Competitions*, pp. 59–65. IPCB, Castelo Branco, Portugal, (2009).

[16] J.-B. Mouret, 'Novelty-based multiobjectivization', in *New Horizons in Evolutionary Robotics: Extended Contributions from the 2009 EvoDeRob Workshop*, pp. 139–154. Springer-Verlag, Berlin, Germany, (2009).

[17] J.-B. Mouret and P. Tonelli, 'Artificial evolution of plastic neural networks: a few key concepts', in *DevLeaNN 2011: A Workshop on Development and Learning in Artificial Neural Networks*, eds., T. Kowaliw, N. Bredeche, and R. Doursat, pp. 32–36, (2011).

[18] Y. Niv, D. Joel, I. Meilijson, and E. Ruppin, 'Evolution of reinforcement learning in uncertain environments: A simple explanation for complex behaviors', *Adaptive Behavior*, **10**(1), 5–24, (2002).

[19] S. Nolfi and D. Floreano, 'Learning and evolution', *Autonomous Robots*, **7**(1), 89–113, (1999).

[20] A. Prieto, J.A. Becerra, F. Bellas, and R.J. Duro, 'Open-ended evolution as a means to self-organize heterogeneous multi-robot systems in real time', *Robotics and Autonomous Systems*, **58**(12), 1282–1291, (2010).

[21] F. Silva, P. Urbano, and A.L. Christensen, 'Adaptation of robot behaviour through online evolution and neuromodulated learning', in *13th Ibero-American Conference on Artificial Intelligence (IBERAMIA 2012)*. Springer-Verlag, Berlin, Germany, in press, (2012).

[22] F. Silva, P. Urbano, S. Oliveira, and A.L. Christensen, 'odNEAT: An algorithm for distributed online, onboard evolution of robot behaviours', in *13th International Conference on the Simulation & Synthesis of Living Systems (ALIFE13)*, pp. 251–258. MIT Press, Cambridge, MA, (2012).

[23] T. Smith, P. Husbands, A. Philippides, and M. O'Shea, 'Neuronal plasticity and temporal adaptivity: Gasnet robot control networks', *Adaptive Behavior*, **10**(3–4), 161–183, (2002).

[24] A. Soltoggio, *Evolutionary and Computational Advantages of Neuromodulated Plasticity*, Ph.D. dissertation, University of Birmingham, UK, 2008.

[25] A. Soltoggio, 'Neuromodulation increases decision speed in dynamic environments', in *8th International Conference on Epigenetic Robotics:Modeling Cognitive Development in Robotic Systems*, volume 139, pp. 119–126. Lund University Cognitive Studies, (2008).

[26] A. Soltoggio, J.A. Bullinaria, C. Mattiussi, P. Dürr, and D. Floreano, 'Evolutionary advantages of neuromodulation plasticity in dynamic, reward-based scenarios', in *ALIFE11*, pp. 569–579. MIT Press, Cambridge, MA, (2008).

[27] A. Soltoggio, P. Dürr, C. Mattiussi, and D. Floreano, 'Evolving neuromodulatory topologies for reinforcement learning-like problems', in *2007 IEEE Congress on Evolutionary Computation*, pp. 2471–2478. IEEE Press, Piscataway, NJ, (2007).

[28] R. Standish, 'Open-ended artificial evolution', *Artificial Life*, **14**(3), (2008).

[29] K.O. Stanley, *Efficient Evolution of Neural Networks through Complexification*, Ph.D. dissertation, The University of Texas at Austin, Austin, TX, 2004.

[30] K.O. Stanley, 'Evolving neural network agents in the NERO video game', in *In Proceedings of the IEEE 2005 Symposium on Computational Intelligence and Games*, pp. 182–189, (2005).

[31] K.O. Stanley, B.D. Bryant, and R. Miikkulainen, 'Evolving adaptive neural networks with and without adaptive synapses', in *2003 IEEE Congress on Evolutionary Computation*, volume 4, pp. 2557–2564. IEEE Press, Piscataway, NJ, (2003).

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

[33] R. Tanese, *Distributed Genetic Algorithms for Function Optimization*, Ph.D. dissertation, University of Michigan, Ann Arbor, MI, 1989.

[34] J. Urzelai and D. Floreano, 'Evolution of adaptive synapses: Robots with fast adaptive behavior in new environments', *Evolutionary Computation*, **9**(4), 495–524, (2001).

[35] Y. Usui and T. Arita, 'Situated and embodied evolution in collective evolutionary robotics', in *8th International Symposium on Artificial Life and Robotics*, eds., Masanori Sugisaka and Hiroshi Tanaka, pp. 212–215. Dept. of Electrical and Electronic Engineering, Oita University, Japan, (2003).

[36] R.A. Watson, S.G. Ficici, and J.B. Pollack, 'Embodied evolution: Embodying an evolutionary algorithm in a population of robots', in *1999 Congress on Evolutionary Computation*, pp. 335–342. IEEE Press, Piscataway, NJ, (1999).

[37] B. Weel, E. Haasdijk, and A.E. Eiben, 'The emergence of multicellular robot organisms through on-line on-board evolution', in *Applications of Evolutionary Computation*, eds., C. Di Chio, A. Agapitos, S. Cagnoni, C. Cotta, F. de Vega, G. Di Caro, R. Drechsler, A. Ekrt, A. Esparcia-Alczar, M. Farooq, W. Langdon, J. Merelo-Guervs, M. Preuss, H. Richter, S. Silva, A. Simes, G. Squillero, E. Tarantino, A. Tettamanzi, J. Togelius, N. Urquhart, A. Uyar, and G. Yannakakis, volume 7248 of *Lecture Notes in Computer Science*, 124–134, Springer-Verlag, Berlin, Germany, (2012).

[38] S. Wischmann, K. Stamm, and F. Wörgötter, 'Embodied evolution and learning: The neglected timing of maturation', in *European Conference on Artificial Life*, eds., Fernando Almeida e Costa, Luis Mateus Rocha, Ernesto Costa, Inman Harvey, and António Coutinho, volume 4648 of *Lecture Notes in Computer Science*, pp. 284–293. Springer-Verlag, Berlin, Germany, (2007).

5th International Workshop on Evolutionary and Reinforcement Learning for Autonomous Robot Systems (ERLARS 2012)
Montpellier, France, August 28 2012

Structured Composition of Evolved Robotic Controllers

Miguel Duarte[1] and **Sancho Oliveira**[1] and **Anders Lyhne Christensen**[1]

Abstract. In this paper, we demonstrate how an artificial neural network (ANN) based controller can be evolved for a complex task through hierarchical evolution and composition of behaviors. We demonstrate the approach in a rescue task in which an e-puck robot has to find and rescue a teammate. The robot starts in a room with obstacles and the teammate is located in a double T-maze connected to the room. We divide the rescue task into different sub-tasks: (i) exit the room and enter the double T-maze, (ii) solve the maze to find the teammate, and (iii) guide the teammate safely to the initial room. We evolve controllers for each sub-task, and we combine the resulting controllers in a bottom-up fashion through additional evolutionary runs. We conduct evolution offline, in simulation, and we evaluate the performance on real robotic hardware. The controller achieved a task completion rate of more than 90% both in simulation and on real robotic hardware.

1 Introduction

We study an approach to the hierarchical evolution of behavioral control for robots. Evolutionary robotics (ER) is a field in which evolutionary computation is used to synthesize controllers and sometimes the morphology of autonomous robots. ER techniques have the potential to automate the design of behavioral control without the need for manual and detailed specification of the desired behavior [6]. Artificial neural networks are often used as controllers in ER because of their capacity to tolerate noise [12] such as that introduced by imperfections in sensors and actuators. Numerous studies have demonstrated that it is possible to evolve robotic control systems capable of solving tasks in surprisingly simple and elegant ways [20]. To date relatively simple tasks have been solved using ER techniques, such as obstacle avoidance, gait learning, phototaxis, and foraging [18]; but as Mouret and Doncieux write: "*. . . this huge amount of work hides many unsuccessful attempts to evolve complex behaviors by only rewarding the performance of the global behavior. The bootstrap problem is often viewed as the main cause of this difficulty, and consequently as one of the main challenges of evolutionary robotics: if the objective is so hard that all the individuals in the first generation perform equally poorly, evolution cannot start and no functioning controllers will be found.*". The bootstrapping problem is thus one of the main reasons that there have been no reports of successful evolution of control systems for complex tasks.

Several different incremental approaches have been studied as a means to overcome the bootstrapping problem and to enable the evolution of behaviors for complex tasks. In incremental evolution, the initial random population starts in a simple version of the environment to avoid bootstrapping issues. The complexity of the environment is then progressively increased as the population improves (see for instance [8, 3]). Alternatively, the goal task can be decomposed into a number of sub-tasks that are then learned in an incremental manner (see for instance [10, 4, 3]). While a single ANN controller is sometimes trained in each sub-task sequentially (such as in [3, 10]), different modules can also be trained to solve different sub-tasks (see [4] for an example). The approach presented in this paper falls in the latter category: we recursively decompose the goal task into sub-tasks and train different ANN-based controllers to solve the sub-tasks. The controllers for the sub-tasks are then combined though an additional evolutionary step into a single controller for the goal task.

We use a task in which a robot must rescue a teammate. Our rescue task requires several behaviors typically associated with ER [18] such as exploration, obstacle avoidance, memory, delayed response, and the capacity to navigate safely through corridors: (i) an e-puck robot must first find its way out of a room with obstacles, (ii) the robot must then solve a double T-maze [2, 5, 22] in which two light flashes in the beginning of the maze instruct the robot on the location of the teammate, and finally (iii) the robot must guide its teammate safely to the room. We evolve behaviors in simulation and evaluate their performance on a real robot. While there are several studies on incremental evolution of behavioral control for autonomous robots, the study presented in this paper is novel in three respects: (i) sub-tasks are solved by one or more continuous time recurrent neural networks that are evolved independently, (ii) we introduce the concept of derived fitness functions during composition for sequential tasks, and (iii) we demonstrate a fully evolved behavioral controller solving a complex task on real robotic hardware.

The paper is organized as follows: in Section 2, we discuss related work; in Section 3, we detail our proposed methodology; in Section 4, we introduce the e-puck robot and our simulator; in Section 5, we describe our experimental setup and analyze the results; and finally, in Section 6, we discuss the applicability and the limitations of our approach.

2 Background and Related Work

Several approaches to incremental evolution of robotic controllers have been proposed. The approaches fall into three different categories: (i) incremental evolution where controllers are evolved with a fitness function that is gradually increased in complexity; (ii) goal task decomposition in which a single ANN is trained sequentially on different sub-tasks; (iii) goal task decomposition in which hierarchical controllers are composed of different sub-controllers evolved for different sub-tasks along with one or more arbitrators that delegate control.

[1] Instituto de Telecomunicações & Instituto Universitário de Lisboa (ISCTE-IUL), e-mail: {miguel_duarte,sancho.oliveira,anders.christensen}@iscte.pt

A methodology belonging to the first category, namely in which controllers are evolved with a fitness function that is gradually increased in complexity, was proposed by Gomez and Miikkulainen [8]. They used a prey-capture task for their study. First, a simple behavior was evolved to solve a simplified version of the global task, in the prey doesn't move. Gradually, by repeatedly increasing the prey's speed, they evolved a more general and complex behavior that was able to solve the prey-capture task. The controllers that they obtained through the incremental approach were more efficient and displayed a more general behavior than controllers evolved non-incrementally. They also found that the incremental approach helped to bootstrap evolution.

Harvey et al. [10] proposed an approach that falls in the second category, namely where a single ANN is trained sequentially on different sub-tasks. The authors describe how they evolved a controller to robustly perform simple visually guided tasks. They incrementally evolved the controller starting with a "Big Target", then a "Small Target", and finally to a "Moving Target". The controller was evolved in few generations and it performed well on real robotic hardware.

Christensen and Dorigo [3] compared two different incremental evolutionary approaches, to evolve a controller for a swarm of connected robots that had to perform phototaxis while avoiding holes. They found no benefits in using neither an incremental approach where the controllers were trained on different sub-tasks sequentially nor an incremental increase in environmental complexity over a non-incremental approach for their highly integrated task.

There are several examples of studies on incremental evolution that fall in the third category, namely in which the global controller is composed of different sub-controllers that have been trained on different sub-tasks. Moioli et al. used a homeostatic-inspired GasNet to control a robot [16]. They used two different sub-controllers, one for obstacle avoidance and one for phototaxis, that were inhibited or activated by the production and secretion of virtual hormones. The authors evolved a controller that was able to select the appropriate sub-controller depending on internal stimulus and external stimulus.

Nolfi introduced the *emergent modular architecture* in the early 1990s [19]. In his approach, the designer of the experiment has minimal impact on the architecture of the network. Evolution is allowed to explore the modular properties of the selected ANN: each actuator corresponds to multiple output neurons that compete for activation. Controllers were evolved for a garbage collection task and were tested successfully on a real Khepera robot. Soltoggio et al. used plastic networks with neuromodulation in a double T-maze task [22]. Although their controllers were able to solve the task correctly, the robot and environmental model used in their study was very simplified: the inputs of the network consisted of high-level information of the environment ("at turn", "at starting position", "at destination") and the movement of the robot consisted of discrete steps through the maze. Furthermore, their controllers were not transferred to real robotic hardware.

Lee [14] proposed an approach in which different sub-behaviors were evolved for different sub-tasks and then combined hierarchically through genetic programming. The approach was studied in a task where a robot had to search for a box in an arena and then push it towards a light source. By evolving different reactive sub-behaviors such as "circle box", "push box" and "explore", the authors managed to synthesize a robotic controller that solved the task. The author claims that his controllers were transferable to a real robot, but only some of the sub-controllers were tested on real hardware. Larsen et al. [13] extended Lee's work by using reactive neural networks for the sub-controllers and the arbitrators instead of evolved programs. However, the chosen goal task used by both Lee and Larsen is relative simple and the scalability of their respective approaches to more complex tasks was never tested.

Our approach shares many similarities with Lee's [14] and Larsen et al.'s [13] approaches in that controllers are evolved and composed hierarchically based on task decomposition. However, as we demonstrate in this study, our approach scales to complex tasks because (i) we use non-reactive controllers, and (ii) during the composition of sub-controllers into larger and more complex controllers, the fitness function for the composed task can be derived directly from the decomposition. We also demonstrate transfer of behavioral control from simulation to real robotic hardware without a significant loss of performance (we cross the reality gap [11]), and we discuss the benefits of transferring controllers incrementally.

3 Methodology

The main purpose of the proposed methodology is to allow for the synthesis of behavioral control for complex tasks using an evolutionary approach. The controller has a hierarchical architecture and it is composed of several ANNs (see Figure 1). Each network is either a *behavior arbitrator* or a *behavior primitive*. These terms were used in [14] to denote similar controller components. A behavior primitive network is usually at the bottom of the controller hierarchy and directly controls the actuators of the robot, such as the wheels. If it is relatively easy to find an appropriate fitness function for a given task, a behavior primitive (a single ANN) is evolved to solve the task. An appropriate fitness function is one that (i) allows evolution to bootstrap and to achieve good performance, (ii) evolves a controller that is able to solve the task consistently and efficiently, and (iii) evolves a controller that transfers well to real robotic hardware. In case an appropriate fitness function cannot be found for a task, the task is recursively divided into sub-tasks until appropriate fitness functions have been found for each sub-task.

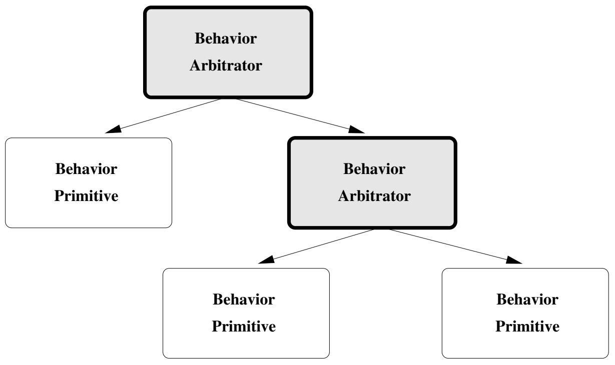


Figure 1. A representation of the hierarchical controller. A behavior arbitrator network delegates the control of the robot to one or more of its sub-controllers. A behavior primitive network can control the actuators of the robots directly.

Controllers evolved for sub-tasks are combined through the evolution of a behavior arbitrator. A behavior arbitrator receives either all or a subset of the robot's sensory inputs, and it is responsible for delegating control to one or more of its sub-controllers. Each behavior arbitrator can have a different *sub-controller activator*. The sub-controller activator activates one or more sub-controllers based on the outputs of the ANN in the behavior arbitrator. The behavior arbitrators used in this study have one output neuron for each of its immediate sub-controllers. The sub-controller activator we use activates the sub-controller for which the corresponding output neuron of the arbitrator has the highest activation. The state of a sub-controller is reset when it stops being activated. Alternative sub-controller activators could be used, such as activators that allow for multiple sub-controllers to be activated in parallel. Parallel activation of different sub-controllers could, for instance, allow a robot to communicate at the same time as it executes motor behaviors.

If the fitness function for the evolution of a behavior arbitrator is difficult to define, it can be derived based on the task decomposition. The derived fitness function is constructed to reward the arbitrator for activating a sub-controller that is suitable for the current sub-task, rather than for solving the global task. The use of derived fitness functions in the composition step circumvents the otherwise increase in fitness function complexity as the tasks considered become increasingly complex.

The topology of each network in the hierarchy (such as the number of input neurons, the number of hidden neurons, and the number output neurons) is completely independent from one another. The basic behavior primitives are evolved first. The behavior primitive are then combined though the evolution of a behavior arbitrator. The resulting controller can then be combined with other controllers through additional evolutionary steps to create a hierarchy of increasingly more complex behavioral control. Each time a new sub-controller (either a behavior primitive or a composed controller) has been evolved, its performance on real robotic hardware can be evaluated. The experimenter can thus address issues related transferability incrementally as the control system is being synthesized.

4 Robot and Simulator

We used an e-puck [17] robot for our experiments. The e-puck is a small circular (diameter of 75 mm) differential drive mobile robotic platform designed for educational use (see Figure 2). The e-puck's set of actuators is composed of two wheels, that enable the robot to move at speeds of up to 13 cm/s, a loudspeaker, and a ring of 8 LEDs which can be switched on/off individually. The e-puck is equipped with several sensors: (i) 8 infrared proximity sensors which are able to detect nearby obstacles and changes in light conditions, (ii) 3 microphones (one positioned on each side of the robot, and one towards the front), (iii) a color camera, and (iv) a 3D accelerometer. Additionally, our e-puck robots are equipped with a range & bearing board [9] which allows them to communicate with one another.

We use JBotEvolver for offline evolution of behavioral control. JBotEvolver is an open source, multirobot simulation platform, and neuroevolution framework. The simulator is written in Java and implements 2D differential drive kinematics. Evaluations of controllers can be distributed across multiple computers and different evolutionary runs can be conducted in parallel. The simulator can be downloaded from: `http://sourceforge.net/projects/jbotevolver`.

We use four of the e-puck's eight infrared proximity sensors: the two front sensors and the two lateral sensors. We collected samples (as advocated in [15]) from the sensors on a real e-puck robot in order to model them in JBotEvolver. Each sensor was sampled for 10 seconds (at a rate of 10 samples/second) at distances to the maze wall ranging from 0 cm to 12 cm. We collected samples at increments of 0.5 cm for distances between 0 cm and 2 cm, and at increments of 1 cm for distances between 2 cm and 12 cm. Distance-dependent noise was added to the sensor readings in simulation corresponding to the amount of noise measured during the sampling of the sensors. We furthermore added a 5% offset noise to the sensor's value. The e-puck's infrared proximity sensors can also measure the level of ambient light. In this study, we use ambient light readings from the two lateral proximity sensors to detect light flashes in the double T-maze sub-task. When a light flash is detected, the activation of one of the two dedicated neurons is set to 1 depending on the side from which the light flash is detected. The input neuron stays active with a value of 1 for 15 simulation cycles (equivalent to 1.5 seconds) to indicate that a flash has been detected. We also included a boolean "near robot" sensor that lets the robot know if there is any other robot within 15 cm. For this sensor, we use readings from the range & bearing board. In simulation, we added Gaussian noise (5%) to the wheel speeds in each control cycle.

If the control code does not fit within the e-puck's limited memory (8 kB), it is necessary to run the control code off-board. When the control code is executed off-board, the e-puck starts each control cycle by transmitting its sensory readings to a workstation via Bluetooth. The workstation then executes the controller, and sends back the output of the controller (wheel speeds) to the robot. We use off-board execution of control code in the real robot experiments conducted in this study.

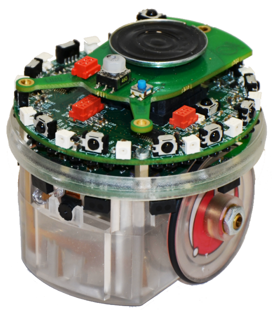


Figure 2. The e-puck is a differential drive robot with a diameter of 75 mm and is equipped with a variety of sensors and actuators, such as a color camera, infrared proximity sensors, a loudspeaker, 3 microphones, and two wheels. Our e-pucks are also equipped with a range & bearing board that allows for inter-robot communication.

5 Experiments and Results

In our experiments, a robot must rescue a teammate that is located in a particular branch of a maze. The robot must find the teammate and guide it to safety. The environment is composed of a room, in which the robot starts, and a double T-maze. A number of obstacles are located in the room. The room has a single exit that leads to the start of a double T-maze (see Figure 3). In order to find its teammate, the robot should exit the room and navigate to the correct branch of the maze. Two rows of flashing lights in the main corridor of the maze give the robot information regarding the location of the teammate. Upon navigating to the correct branch of the maze, the robot must guide its teammate back to the room.

The rescue task is relatively complex, especially given the limited amount of sensory information available to the robot, and it would be difficult to find an appropriate fitness function that allows evolution to bootstrap. We therefore divided the task into three sub-tasks: (i) exit the room, (ii) solve T-maze to find teammate, and (iii) return to the room with the teammate. Below, we detail how we evolved the controllers to solve the individual sub-tasks, and how we combined them to obtain a controller for the complete rescue task.

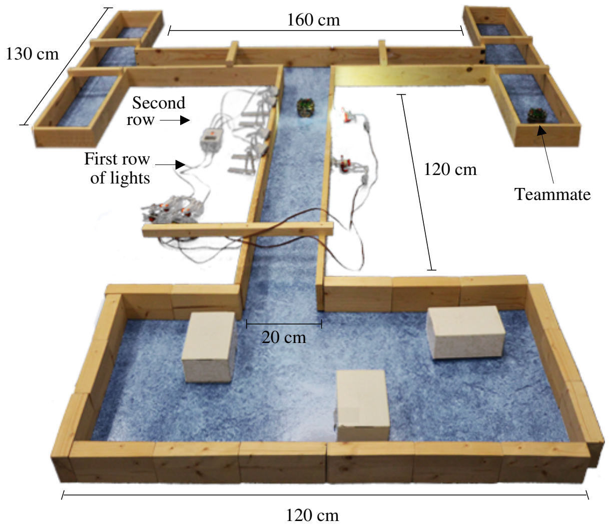


Figure 3. The environment is composed of a room with obstacles and a double T-maze. The room is rectangular and its size can vary between 1 m and 1.2 m. The double T-maze has a total size of 2 m × 2 m. The two rows with the lights are located in the central maze corridor. The activation of these two rows of lights indicate the location of a teammate.

5.1 Controller Architecture

The structure of the controller for the complete rescue task can be seen in Figure 4. We recursively divided the task into sub-tasks until an appropriate fitness function could be found, and we then evolved the sub-controllers in a bottom-up fashion, starting with the behavior primitives.

For each evolutionary run, we used a simple generational evolutionary algorithm with a population size of 100 genomes. The fitness score of each genome was averaged over samples 50 with varying initial conditions, such as the robot's starting position and orientation. After the fitness of all genomes had been sampled, the 5 highest scoring individuals were copied to the next generation. 19 copies of each genome were made and each gene was mutated with a probability of 10% by applying a Gaussian offset. All the ANNs in the behavior primitives and in the behavior arbitrators were time-continuous recurrent neural networks [1] with one hidden layer of fully-connected neurons.

5.1.1 Exit Room Sub-task

The first part of the rescue task was an exploration and obstacle avoidance task in which the robot must find a narrow exit leading to the maze. The room was rectangular with a size that varied between 1 m and 1.2 m. We placed either 2 or 3 obstacles in the room depending on its size. Each obstacle was rectangular with side lengths ranging from 5 cm to 20 cm selected at random. The location of the room exit was also randomized in each trial.

We found that an ANN with 4 input neurons, 10 hidden neurons, and 2 output neurons could solve the task. Each of the input neurons

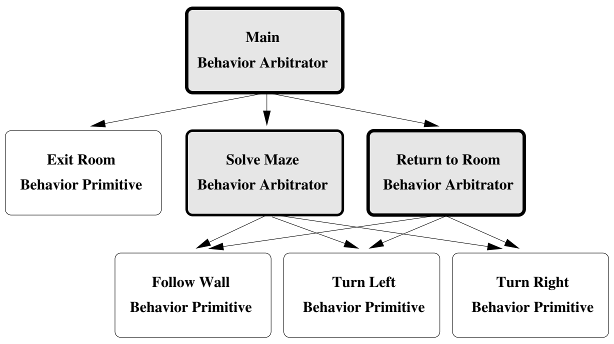

Main Behavior Arbitrator
Exit Room Behavior Primitive
Solve Maze Behavior Arbitrator
Return to Room Behavior Arbitrator
Follow Wall Behavior Primitive
Turn Left Behavior Primitive
Turn Right Behavior Primitive

Figure 4. The controller used in our experiments is composed of 3 behavior arbitrators and 4 behavior primitives.

was connected to an infrared proximity sensor, and the output neurons controlled the speed of the robot's wheels. In order to evolve the controller, the robot was randomly oriented and positioned near the center of the room at the beginning of each sample.

Each controller was evaluated according to one of two possible outcomes: (a) the robot managed to exit the room, and it was assigned a fitness according to $f_{1,a}$, or (b) the robot did not manage to find the exit to the room within the allotted time (100 seconds), and it was assigned a fitness according to $f_{1,b}$. Fitness $f_{1,a}$ and $f_{1,b}$ are defined by:

$$f_{1,a} = 5 + \frac{maxCycles - spentCycles}{maxCycles} \quad (1)$$

$$f_{1,b} = \frac{distanceToExit - closestDistanceToExit}{distanceToExit} \quad (2)$$

where $distanceToExit$ is the distance from the center of the room to its exit, and $closestDistanceToExit$ is the closest point to the exit that the robot navigated to.

The "exit room" controllers were evolved until the 500th generation and each sample was evaluated for 1000 control cycles, in a total of 10 evolutionary runs. The controllers achieved an average solve rate of 52%, with a solve rate of 96% in the best evolutionary run. The best performing controller starts by moving away from the center of the room until it senses a wall, which it then follows clockwise until the room exit is found. 3 of the 10 evolutionary runs produced controllers capable of finding the exit of the room in over 90% of the samples. The remaining runs did not produce successful behaviors: the robots would spin/circle around, sometimes finding the exit by chance and often crashing into one of the surrounding walls or into an obstacle.

5.1.2 Solve Double T-maze Sub-task

In the second sub-task, the robot had to solve a double T-maze in order to find the teammate that had to be rescued. The robot was evaluated according to one of three possible different outcomes: (i) if the robot successfully navigated to its goal, it was assigned a fitness based on $f_{2,a}$, (ii) if the robot navigated to an incorrect branch or if it collided into a wall, it was assigned a fitness based on $f_{2,b}$, and (iii) if the time expired, the robot was assigned a fitness of 0. $f_{2,a}$ and $f_{2,b}$ are defined by:

$$f_{2,a} = 1 + \frac{maxCycles - spentCycles}{maxCycles} \quad (3)$$

$$f_{2,b} = \frac{totalDistance - currentDistanceToDest}{3 \cdot totalDistance} \quad (4)$$

where $totalDistance$ is the distance from the start of the maze to the teammate, and $currentDistanceToDest$ is the final distance from the main robot to its destination.

We experimented with using a single ANN to solve this sub-task. The ANN was composed of 6 input neurons, 10 hidden neurons, and 2 output neurons. The input neurons were connected to the 4 proximity sensors and the 2 light sensors. The output neurons directly controlled the speed of the wheels.

We conducted 10 evolutionary runs, each lasting 1000 generations. The controllers were post-evaluated and the fitness of every controller was sampled 100 times for each of the 4 possible light configurations. The evolved controllers had an average solve rate of only 40%. The best controller had a solve rate of 83%, with just 3 other controllers were able to correctly solve the T-maze in more than 50% of the samples.

Since we could not obtain controllers that could solve the task consistently, we followed our methodology and further divided the solve maze sub-task into three different sub-tasks: "follow wall", "turn left" and "turn right", for which appropriate fitness functions could easily be specified. The behavior primitive network for each of these three sub-tasks had 4 input neurons, 3 hidden neurons, and 2 output neurons. The input neurons were connected to the infrared proximity sensors and the outputs controlled the speed of the wheels. The three behavior primitives were evolved in corridors of different lengths. The environment for the "turn" controllers was also composed of either left or right turns, depending on the controller.

A total of 5 evolutionary runs were simulated for each of the basic behaviors ("follow wall", "turn left" and "turn right"). The evolutionary process lasted 100 generations, and the best controller from each evolutionary run was then sampled 100 times in order to evaluate the controller's solve rate. The "turn left" controllers achieved an average solve rate of 89%, with a solve rate of 100% for the controller that obtained the highest fitness; the "turn right" controllers achieved an average solve rate of 69%, with a solve rate of 100% for the controller that obtained the highest fitness; and the "follow wall" controllers achieved an average solve rate of 99%, with a solve rate of 100% for the controller that obtained the highest fitness. The controllers for the basic behaviors achieved a good performance in relatively few generations and the majority of the evolutionary runs converged to the optimal solve rate, with an occasional run getting stuck in a local optimum.

We then evolved a behavior arbitrator with the three best behavior primitives as sub-controllers. The behavior arbitrator network had 6 inputs, 10 hidden, and 3 output neurons. The inputs were connected to the 4 infrared proximity sensors and the 2 light sensors. At the beginning of each trial, the robot was placed at the start of the double T-maze and had to navigate to the correct branch based on the activations of the lights that were placed on the first corridor (see Figure 3). For instance, if the left light of the first row and the right light of the second row were activated, the robot should turn left at the first junction and right at the second junction. The fitness awarded was either: (i) $f_{2,a}$, if the robot successfully navigated to its teammate's location, (ii) $f_{2,b}$ if the robot navigated to an incorrect branch of the maze or collided into a wall, or (iii) a fitness of 0 if the time expired before the robot managed to enter a branch of the maze. The sample was terminated if the robot collided into a wall or if it navigated to a wrong branch of the maze.

The evolution process lasted until the 1000th generation, in a total of 10 evolutionary runs. The controllers achieved an average solve rate of 93%, with a solve rate of 99.5% for the highest performing controller.

To test the controller on real robotic hardware, we built a double T-maze with a size of 2 m $\times$ 2 m (see Figure 3). In the real maze, the flashing lights were controlled by a Lego Mindstorms NXT brick. The brick was connected to four ultrasonic sensors that detected when the robot passed by. Lights were turned on by the 1st and 3rd ultrasonic sensor and turned off by the 2nd and 4th ultrasonic sensor. The brick controlled the state of the lights using two motors.

5.1.3 Return to Room Sub-task

The final sub-task consisted of the robot guiding its teammate back to the first room. For this sub-task, we reused the behavior primitives previously evolved for maze navigation ("follow wall", "turn left" and "turn right") and we evolved a new behavior arbitrator. The behavior arbitrator network was trained in the double T-maze with the robot starting in one of the four branches of the maze (chosen at random in the beginning of each trial). The behavior arbitrator had 4 input neurons, 10 hidden neurons, and 3 output neurons. The input neurons were connected to the robot's infrared proximity sensors and the output neurons selected which sub-controller should be active.

The teammate being rescued was preprogrammed to follow the main robot once it was within 15 cm. We used the e-puck range & bearing extension board to determine the distance between the two robots. Since this was a task in which the robot had to navigate correctly through the maze, we used the same fitness function as in the solve double T-maze sub-task described in the previous section. The only difference was the objective: the robot was evaluated based on its distance to the entrance of the maze, not the distance to the teammate.

We conducted a total of 10 evolutionary runs until the 500th generation for the "return to room" behavior. The controllers achieved an average solve rate of 75%, with a solve rate of 100% for the highest performing controller.

5.2 Evolving the main controller

For the composed task, we evolved a behavior arbitrator with the controllers for the exit room, the solve maze, and the return to room tasks as sub-controllers. The robot had to first find the entrance to the double T-maze, then navigate the maze in order to find its teammate, and finally guide the teammate safely back to the room. The behavior arbitrator for the complete rescue task had 5 input neurons, 10 hidden neurons, and 3 output neurons. The inputs were connected to the 4 infrared proximity sensors and to a boolean "near robot" sensor, which indicated if there was a teammate within 15 cm (based on readings from the range & bearing board).

We evolved the controller with a derived fitness function that rewards the selection of the right behaviors for the current sub-task. The controller was awarded a fitness value between 0 and 1 for each sub-task (for a maximum of 3 for all sub-tasks), depending on the amount of time that it selected the correct behavior. The fitness function is a sum of the equation f_3 for each of the 3 sub-tasks. f_3 is defined as follows:

$$f_3 = \frac{correctBehaviorCycles}{totalBehaviorCycles} \quad (5)$$

where $totalBehaviorCycles$ is the number of simulation cycles that the controller has spent in a particular sub-task, and $correctBehaviorCycles$ is the number of cycles in which the controller chose the sub-controller for that particular sub-task.

We ran 10 evolutionary runs until the 1000th generation for the composed task controller. The fitness of each genome was sampled 20 times and the average fitness was computed. Each sample lasted a maximum of 2000 control cycles (equivalent to 200 seconds). The 10 resulting controller achieved an average solve rate for the composed task of 85%, with a solve rate of 93% for the highest performing controller.

We analyzed how the main controller managed to solve each part of the composed task. On the "exit room" task, all 10 controllers averaged a solve rate of 91%. This means that all the controllers successfully learned that they should activate the exit room behavior primitive in the first part of the composed task.

After exiting the room, the controller should activate the "solve maze" behavior in order to find the robot's teammate. An important detail is that once the controller selects this behavior, it should not switch to another one until it reaches the end of the maze: switching resets the state of the selected sub-controller, meaning that the "solve maze" behavior arbitrator would forget which light flashes previously sensed. The average solve rate dropped from 91% to 88%, which means that only 3% of all the samples failed at solving the maze sub-task.

Upon finding the teammate, the robot should return to the starting point, completing the composed task. Ideally, this should be done by activating the return behavior at the end of the maze. The 10 controllers achieved an average solve rate of 85%.

5.3 Transfer to the real robot

After evaluating all the different evolutionary runs, the best performing controller from the simulation was tested on a real e-puck. The robot had to solve the composed task: find the exit of the initial room, navigate the double T-maze to the correct branch, and return to the room. We used a room with a size of 120 cm × 100 cm for our real robot experiments. Three identical obstacles with side lengths of 17.5 cm and 11 cm were placed in the room as shown in Figure 3. We sampled the controllers 6 times for each light combination, for a total of 24 samples. Since the purpose of these experiments was to test the transferability of the evolved controller, the teammate was not used and the near-robot sensor was remotely triggered if the robot reached the correct maze branch.

The controller solved the composed task on the real robot in 22 out of 24 samples (a solve rate of 92%). It consistently chose the correct sub-network at each point of the task, and only failed in the return to room behavior twice.

We ran additional proof-of-concept experiments in which we include a teammate that was preprogrammed to follow the main robot back to the initial room. Videos of these experiments can be found in `http://home.iscte-iul.pt/~alcen/erlars2012/`.

6 Conclusions

In this study, we demonstrated how controllers can be composed in a hierarchical fashion to allow for the evolution of behavioral control for a complex task. We started by decomposing the goal task into sub-tasks until a controller for each sub-task could easily be evolved. When we combined the sub-controllers, we used a derived fitness function that rewarded controllers for activating the sub-controller corresponding to the current sub-task rather than for solving the global task. We evaluated the evolved behavior on a real e-puck performing a rescue task. The real robot managed to solve the task in 22 out of 24 experiments (solve rate of 92%), which is similar to the robot's performance in simulation (solve rate of 93% in 400 experiments).

Our approach overcomes a number of fundamental issues in evolutionary robots. Often the experimenter has to go through a tedious trial and error process in order to design a suitable fitness function for the task at hand. In our approach, we recursively divide tasks into sub-tasks until a simple fitness function can easily be specified. We tried to evolve a single ANN-based controller for the solve maze sub-task, for instance, but since bootstrapping proved difficult, we divided the solve maze task into sub-tasks (follow wall, turn left, and turn right). For each of these simple tasks, fitness functions that allowed evolution to bootstrap were straightforward to specify.

Although more complex evolutionary algorithms, such as novelty search [21], might allow evolution to find solutions for more complex tasks, they would have their limitations. In our study, we show that, by following a divide and conquer approach, we can evolve control for a complex task using a very simple evolutionary algorithm which cannot evolve control for the complete task. For a more advanced algorithm, the divisions may be more coarse, but we could apply the same principles.

During the composition of sub-behaviors, we use a fitness function directly derived from the immediate decomposition, that is, a fitness function that rewards a controller for activating an appropriate sub-controller given the current situational context: after we had obtained controllers for each of the three sub-tasks, exit room, solve maze, and return to room, we combined them in an additional evolutionary step. During evolution, an arbitrator (an ANN) was rewarded for (i) activating the exit room sub-controller while the robot was in the room, (ii) the solve sub-controllers while the robot was in the maze, and (iii) the return to room behavior after the teammate had been located. In this way, we avoid that the complexity of the fitness function increases with the task complexity as sub-behaviors are combined.

The transfer of behavioral control from simulation to a real robot is usually a hit or miss because a controller for the goal task is completely evolved in simulation before it is tested on real hardware. In our approach, the transfer from simulation to real robotic hardware can be conducted in an incremental manner as behavior primitives and sub-controllers are evolved. This allows the designer to address issues related to transferability immediately and locally in the controller hierarchy.

The applicability of our approach depends on if the task for which a controller is sought can be broken down into reasonably independent sub-tasks. For highly integrated tasks where it is unclear if or how the goal task can be divided into sub-tasks [3], our approach may not be directly applicable. However, in cases where a controller for an indivisible sub-task cannot be evolved, either because a good fitness function cannot be found or because evolved solutions do not transfer well, the evolved control may be combined with preprogrammed behaviors [5].

The potential cost of an engineered approach, such as the approach proposed in this paper, is that evolution is constrained. Surprisingly simple and elegant solutions that the experimenter did not foresee may therefore never be discovered. Some researchers advocate the use of implicit, behavioral, and internal fitness functions [7], because fitness functions with such characteristics, in theory, allow for solutions to emerge through an autonomous self-organization process. In practice, however, such fitness functions, which are supposed to be redeemed from any constraints imposed by a priori knowledge, are often the result of a series of unsuccessful experiments. After each unsuccessful experiment, the fitness function is modified based on the results of the experiment and based on the experiment's guess concerning what may be "wrong". As a result, the fitness function used in the final successful experiment often contains factors and values, and sometime even entire terms that seem arbitrary.

We do not dismiss the potential benefits of implicit, behavioral, and internal fitness functions in our approach. Instead, we suggest to divide the task into two or more sub-tasks, when such a fitness function cannot easily be found. In this way, controllers for complex tasks can be synthesized in a hierarchical fashion, while at the same time, they can benefit from evolutionary robotics techniques, namely (i) automatically synthesis of control, and (ii) evolution's ability to exploit the way in which the world is perceived through the robot's (often limited) sensors. Our long-term goal is to combine the benefits of manual design of behavioral control with the benefits of automatic synthesis though evolutionary computation to obtain capable, efficient, and robust controllers for real robots.

Acknowledgements: This work was partly supported by Foundation of Science and Technology (FCT) under the grants PTDC/EEACRO/104658/2008, SFRH/BD/76438/2011, and PEst-OE/EEI/LA0008/2011.

REFERENCES

[1] R. D. Beer and J. C. Gallagher, 'Evolving dynamical neural networks for adaptive behavior', *Adaptive Behavior*, **1**, 91–122, (1992).

[2] J. Blynel and D. Floreano, 'Exploring the t-maze: Evolving learning-like robot behaviors using CTRNNs', in *Applications of Evolutionary Computing*, pp. 593–604. Springer, Berlin, Germany, (2003).

[3] A. L. Christensen and M. Dorigo, 'Evolving an integrated phototaxis and hole avoidance behavior for a swarm-bot', in *Proceedings of Tenth International Conference on the Simulation and Synthesis of Living Systems (ALIFEX)*, pp. 248–254. MIT Press, Cambridge, MA, (2006).

[4] R. de Nardi, J. Togelius, O. E. Holland, and S. M. Lucas, 'Evolution of neural networks for helicopter control: Why modularity matters', in *Proceedings of IEEE Congress on Evolutionary Computation (CEC'06)*, pp. 1799–1806. IEEE Press, Piscataway, NJ, (2006).

[5] M. Duarte, S. Oliveira, and A. L. Christensen, 'Automatic synthesis of controllers for real robots based on preprogrammed behaviors', in *Proceedings of the 12th International Conference on Adaptive Behaviour*. Springer, Berlin, Germany, (2012). in press.

[6] D. Floreano and L. Keller, 'Evolution of adaptive behaviour in robots by means of Darwinian selection', *PLoS Biology*, **8**, 1–8, (2010).

[7] D. Floreano and J. Urzelai, 'Evolutionary robots with on-line self-organization and behavioral fitness', *Neural Networks*, **13**(4–5), 431–443, (2000).

[8] F. Gomez and R. Miikkulainen, 'Incremental evolution of complex general behavior', *Adaptive Behavior*, (5), 317–342, (1997).

[9] A. Gutierrez, A. Campo, M. Dorigo, D. Amor, L. Magdalena, and F. Monasterio-Huelin, 'An open localization and local communication embodied sensor', *Sensors*, **8**(11), 7545–7563, (2008).

[10] I. Harvey, P. Husbands, and D. Cliff, 'Seeing the light: artificial evolution, real vision', in *Proceedings of the Third International Conference on Simulation of Adaptive Behavior: From Animals to Animats 3*, pp. 392–401. MIT Press, Cambridge, MA, (1994).

[11] N. Jakobi, 'Evolutionary robotics and the radical envelope-of-noise hypothesis', *Adaptive Behavior*, **6**, 325–368, (1997).

[12] J. Kam-Chuen, C.L. Giles, and B.G. Horne, 'An analysis of noise in recurrent neural networks: convergence and generalization', *IEEE Transactions on Neural Networks*, **7**, 1424–1438, (1996).

[13] T. Larsen and S.T. Hansen, 'Evolving composite robot behaviour - a modular architecture', in *Proceedings of the Fifth International Workshop on Robot Motion and Control, 2005. RoMoCo '05.*, pp. 271–276. IEEE Press, Piscataway, NJ, (2005).

[14] W.-P. Lee, 'Evolving complex robot behaviors', *Information Sciences*, **121**(1-2), 1–25, (1999).

[15] O. Miglino, H. H. Lund, and S. Nolfi, 'Evolving mobile robots in simulated and real environments', *Artificial Life*, **2**, 417–434, (1996).

[16] R. C. Moioli, P. A. Vargas, F. J. Von Zuben, and P. Husbands, 'Towards the evolution of an artificial homeostatic system', in *IEEE Congress on Evolutionary Computation*, pp. 4023–4030. IEEE Press, Piscataway, NJ, (2008).

[17] F. Mondada, M. Bonani, X. Raemy, J. Pugh, C. Cianci, A. Klaptocz, S. Magnenat, J.-C. Zufferey, D. Floreano, and A. Martinoli, 'The e-puck, a robot designed for education in engineering', in *In Proceedings of the 9th Conference on Autonomous Robot Systems and Competitions*, pp. 59–65. Springer, Berlin, Germany, (2009).

[18] A. L. Nelson, G. J. Barlow, and L. Doitsidis, 'Fitness functions in evolutionary robotics: A survey and analysis', *Robotics and Autonomous Systems*, **57**(4), 345–370, (2009).

[19] S. Nolfi, 'Using emergent modularity to develop control systems for mobile robots', *Adaptive Behavior*, **5**, 343–363, (1996).

[20] S. Nolfi and D. Floreano, *Evolutionary robotics: The biology, intelligence, and technology of self-organizing machines*, MIT Press, Cambridge, MA, 2000.

[21] S. Risi, S. D. Vanderbleek, C. E. Hughes, and K. O. Stanley, 'How novelty search escapes the deceptive trap of learning to learn', in *Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, GECCO '09. ACM, New York, NY, USA, (2009).

[22] A. Soltoggio, J. A. Bullinaria, C. Mattiussi, P. Dürr, and D. Floreano, 'Evolutionary Advantages of Neuromodulated Plasticity in Dynamic, Reward- based Scenarios', in *Proceedings of the 11th International Conference on Artificial Life (Alife XI)*, pp. 569–576, MIT Press, Cambridge, MA, (2008).

5th International Workshop on Evolutionary and Reinforcement Learning for Autonomous Robot Systems (ERLARS 2012)
Montpellier, France, August 28 2012

Improved State Aggregation with Growing Neural Gas in Multidimensional State Spaces

Michael Baumann[1] and **Timo Klerx**[2] and **Hans Kleine Büning**[2]

Abstract. Q-Learning is a widely used method for dealing with reinforcement learning problems. However, the conditions for its convergence include an exact representation and sufficiently (in theory even infinitely) many visits of each state-action pair—requirements that raise problems for large or continuous state spaces. To speed up learning and to exploit gained experience more efficiently it is highly beneficial to add generalization to Q-Learning and thus enabling the transfer of experience to unseen but similar states. In this paper, we report on improvements for *GNG-Q*, an algorithm that solves reinforcement learning problems with continuous state spaces and simultaneously learns a proper abstract state space. This approach combines Q-Learning and growing neural gas (an adaptive vector quantizer) to compute a state space abstraction. It starts with a coarse resolution that is gradually refined based on information achieved during learning. We improve the dealing with the non-determinism that may emerge in abstracted state spaces, suggest a new refinement strategy and propose a new criterion to decide when a refinement is necessary. Furthermore, we argue that this criterion offers an implicit local stopping condition for changes made to the approximation. Additionally, we employ eligibility traces to speed up learning. We evaluate the improved method in continuous state spaces with up to four dimensions and compare the results with several approaches from literature. Our experiments confirm that the modifications highly improve the efficiency of the abstract state space and that our approach is well competitive with existing methods.

1 Introduction

In reinforcement learning (RL), an agent has to learn a policy to solve a given problem—just from interaction with a (probably unknown) environment. It does so by trying its available actions in the different states of the environment and uses the (maybe delayed) scalar feedback—the reward—from the environment to update its estimation of the policy. The environment has to offer rewards in a way that the agent can learn a useful behavior by maximizing these rewards over time. RL problems are often modeled as Markov decision processes (MDP) [17] and can be solved with temporal difference methods that usually store the learned behavior in a tabular representation for each possible combination of states and actions. One well-known RL algorithm is Q-Learning [21] (for detailed information on other approaches we refer to [18, 17]) that is proven to converge to an optimal policy under several conditions [20]. These conditions include to visiting each state-action pair infinitely often. This requirement is fraught with problems in large or continuous state spaces that can be found in realistic settings. To use aforementioned table-based methods, continuous state spaces have to be discretized—a step that often needs domain knowledge to find a proper resolution of the approximation.

Large state spaces suffer from two severe problems: First, the *curse of dimensionality* (the search space grows exponentially in the number of states) induces high memory requirements. Second, the large amount of state-action pairs inhibits the agent to gather enough knowledge for each possible state as the probability to experience a certain state more than once decreases as the size of the state space increases. One way to cope with these problems is the use of generalization—i.e. transfer knowledge to unseen but similar states—that can e.g. be achieved by aggregating states [11]. For detailed overviews of other approaches, we refer to [4] or [19].

In this paper we analyze the *GNG-Q* approach [1] and investigate its behavior on a continuous state RL task: An agent is situated in a 2-dimensional environment and has to learn the shortest path from any position to the goal (cf. Sec. 5). Furthermore, we investigate two multi-dimensional problems: a d-dimensional continuous world and the acrobot swing up problem [16].

The idea of *GNG-Q* is to learn the behavior and its representation in parallel using a combination of Q-Learning and the unsupervised growing neural gas (GNG) vector quantizer [9]. *GNG-Q* assumes that similar states need similar behavior and computes a Voronoi tessellation of the state space and treats all states in one region equally. The core idea of *GNG-Q* is as follows: The approximation is initially very coarse and is refined in regions that contain incompatible states. In each learning step, the agent uses the current approximation to update its estimated policy. Simultaneously, changes in the learned policy point out regions that have to be refined. Thus, an *abstracted state space* is built by aggregating compatible states and this abstraction is adjusted based on the interaction during learning without knowing the environment in advance.

Our contribution is as follows: 1. We argue, that abstracted state spaces may introduce non-determinism and adapt the Q-update to better cope with this situation. 2. A new operation for refining regions of states is introduced. 3. We provide new criteria for the refinement and adaptation of the approximation and argue how these criteria lead to an implicit stopping condition for adjustments on the approximation. 4. Eligibility traces are incorporated to speed up learning. 5. We experimentally evaluate the influences of the parameters in the approach and compare its performance to other approaches. The enhancements in the updated algorithm called *GNG-Q*$^+$ lead to a significant decrease in the size of the approximation and an improved regulation of the refinement and adaptation. Furthermore, our adjusted use of the edges in the graph offers a means to model the state transition function for the abstracted states and allows to removing

[1] International Graduate School "Dynamic Intelligent Systems", University of Paderborn, 33095 Paderborn, Germany, email: mbaumann@upb.de
[2] University of Paderborn, 33095 Paderborn, Germany

dead regions. Our experiments confirm, that *GNG-Q^+* is well competitive with other approximative approaches and that *GNG-Q^+* is able to efficiently compute a useful policy in parallel with a compact state space approximation. The proposed approach operates on-line and does not need the underlying model of the considered reinforcement learning problem. Additionally, it does not need to incorporate domain knowledge and the approximation offers flexible and adaptive shapes. After learning, the policy can be stored very efficient as only the Voronoi centers of the approximation and the associated action values are needed. The mapping of one state of the original state space to its abstraction is realized by a nearest neighbor rule and is thus very fast and easy to implement.

2 Related Work

Another approach that uses vector quantization was introduced in [11]. There, an adaptive vector quantizer is used to partition the state space while the agent is learning. The partitioning is carried out based on proximity in the state space and similarity regarding the rating of actions generated by Q-Learning. The approximation is refined if the reward accumulated in one region is exceeding some threshold and a predefined minimal distance to all neighboring code words is kept. In this approach, the centers of the created regions are not able to move and domain knowledge is required to determine useful values for the thresholds used. Fernández et al. [8] present the VQQL model that consists of the generalized Lloyd algorithm for vector quantization and Q-Learning. It uses vector quantization to obtain a set of codebook vectors that represent the state space. In a subsequent step, Q-Learning is used to learn a policy based on this reduced representation. The key difference to the *GNG-Q* and *GNG-Q^+* approaches is that Fernández et al. construct the state space representation independent from learning. Ratitch and Precup [13] also follow a similar approach as they place units as centers for an approximation. Their goal is to add a new unit if for the current state the number of nearby units is below some threshold.

Konidaris et al. [10] approximate continuous environments with the Fourier basis, a method that employs Fourier series to approximate the optimal value function. They compare their approach empirically to various other basis function approaches and conclude that its performance is similarly good. Unfortunately, this approach seems to be rather run time consuming.

Adaptive Tile Coding [22] is an approach that learns a policy in parallel with an appropriate state space abstraction. Starting with a very coarse approximation consisting of just one tile, it is refined based on information (e.g. based on the policy) from learning. The refinement operation splits one tile evenly in half, which will often lead to problems as it may happen that many splits are needed until incompatible states are separated. Another Tile Coding approach was presented in [12] where a genetic algorithm was used to decide upon refinements allowing unevenly splits. A drawback of this algorithm may be that it is executed for several resolutions of the approximation.

3 State Space Abstraction in Reinforcement Learning

This section presents the theoretical model of state space abstraction and shows, how the agent uses the transition and reward functions of the original Markov decision process (MDP) to update the abstracted MDP. Additionally, we describe the *GNG-Q* approach that we adapt in Sec. 4.

3.1 Theoretical Model

Single agent reinforcement learning problems are usually modeled as Markov decision processes (MDP). In this work, we consider continuous state spaces in deterministic environments with discrete time steps and discrete actions. Thus, we define a MDP as $\mathrm{M} = (S, A, \mathrm{T}, r)$ where the transition function $\mathrm{T} : S \times A \to S$ returns the succeeding state $\mathrm{T}(s_t, a_t) = s_{t+1} \in S$ after performing action $a_t \in A$ in state $s_t \in S$. The reward function $r : S \times A \to \mathbb{R}$ reflects the immediate merit of this execution.

Q-Learning [21] is one frequently employed algorithm to learn an optimal policy $\pi^\star$. It approximates the action-value function $Q^\star(s, a)$ that expresses the expected accumulated reward for performing action a in state s and following an optimal policy afterwards. The agent incrementally updates its approximation $\widehat{Q}$ of the action-value function $Q(s_t, a_t)$ during interaction with the environment: it executes a_t in s_t and observes the succeeding state s_{t+1} and the reward $r_t = r(s_t, a_t)$. The approximation is then updated according to $\widehat{Q}_{t+1}(s_t, a_t) := (1 - \alpha_t)\widehat{Q}_t(s_t, a_t) + \alpha_t\big[r(s_t, a_t) + \gamma \max_{a' \in A} \widehat{Q}_t(s_{t+1}, a')\big]$ with the learning rate α and discount factor γ. Q-Learning is proven to converge to the true Q-function given that each state-action pair is updated infinitely often, an exact representation of the policy is used and the learning rate α_t fulfills $\sum_t \alpha_t = \infty$ and $\sum_t \alpha_t^2 < \infty$ [20]. As pointed out earlier, large state spaces introduce severe issues and it is thus highly beneficial to introduce some kind of generalization [17]. Amongst many others (for detailed overviews see [19] or [4]), one approach to deal with large state spaces is the use of *state space abstractions*. Following [19], we define an abstract state space as follows:

Definition 1 (State Space Abstraction) *Let* $\mathrm{M} = (S, A, \mathrm{T}, r)$ *be a deterministic Markov decision process. We define the corresponding* abstracted MDP $\widehat{\mathrm{M}} = (\widehat{\mathcal{S}}, A, \mathrm{T}, r)$ *where* $\widehat{\mathcal{S}}$ *is a partition of the actual state space* S *and usually* $|\widehat{\mathcal{S}}| \ll |S|$ *holds. Each abstract state* $\widehat{\mathrm{s}} \in \widehat{\mathcal{S}}$ *is defined as a set* $\widehat{\mathrm{s}} := \{s \mid \psi(s) = \widehat{\mathrm{s}},\ s \in S\}$ *where the* abstraction-function ψ *is a mapping* $\psi : S \to \widehat{\mathcal{S}}$ *that maps each state of* S *to one of the states of the abstracted state space* $\widehat{\mathcal{S}}$*. Thus,* ψ *provides a partition of* S *with* $\bigcup_{\widehat{\mathrm{s}} \in \widehat{\mathcal{S}}} \widehat{\mathrm{s}} = S$ *and* $\widehat{\mathrm{s}}_1 \cap \widehat{\mathrm{s}}_2 = \varnothing,\ \forall\, \widehat{\mathrm{s}}_1 \neq \widehat{\mathrm{s}}_2 \in \widehat{\mathcal{S}}$.

The value functions in RL for an abstract MDP $\widehat{\mathrm{M}}$ can be learned from interactions with the original MDP M: The agent observes a state $s_t \in S$ and performs action a_t that takes it to the subsequent state $s_{t+1} = \mathrm{T}(s_t, a_t)$ and results in a reward $r(s_t, a_t)$. This information can be used e.g. in a Q-Learning update for the abstracted MDP [19]:

$$\begin{aligned}\widehat{Q}_{t+1}(\psi(s_t), a_t) &:= (1 - \alpha_t)\widehat{Q}_t(\psi(s_t), a_t) \\ &\quad + \alpha_t\Big[r(s_t, a_t) + \gamma \max_{a' \in A} \widehat{Q}_t(\psi(s_{t+1}), a')\Big] \qquad (1)\end{aligned}$$

Note, that the update for one abstract state $\widehat{\mathrm{s}}$ affects all states $s \in S$ that are abstracted to $\widehat{\mathrm{s}}$, i.e. all states s for which $\psi(s) = \widehat{\mathrm{s}}$ hold. This is a major advantage as one update affects several states and each (maybe unseen) state is treated equally as any other state abstracted to the same abstract state.

3.2 Growing Neural Gas State Quantizer

One approach to learn an approximation of the state space while performing reinforcement learning was presented in [1]. This approach learns the behavior and its representation in parallel: An adaptive vector quantizer (growing neural gas (GNG), [9]) is used to approximate the state space and in each learning step, Q-Learning is executed on the current approximation. The goal is to find a partition of the state space in regions such that each region contains states that can be treated equally. *GNG-Q* (summarized in Alg. 1) uses information collected during learning to adjust the approximation. Thus, the abstraction function ψ and the agent's approximation $\widehat{Q}$ for the abstracted MDP are learned in parallel.

GNG-Q aggregates states to so called *state regions* that are similar regarding some measure and require the same behavior. All states in one state region are treated equally and share one Q-vector. These regions are built using a growing neural gas: A set of units $n \in N$ called *neurons* are positioned in the state space and a Voronoi tessellation is created by a nearest neighbor rule that assigns any state $s \in S$ to the nearest neuron $nn(s) = \operatorname{argmin}_{n' \in N} \mathrm{d}(s, n')$ using some distance measure d. The abstraction function ψ is thus defined by the set N_t of neurons at time t and the nearest neighbor rule.

In each learning step, the nearest and second nearest neurons n_1, n_2 to the current state s_t are determined and both neurons are connected with a *neighborhood connection*. These connections are equipped with an age that is used to remove outdated connections. We call such connected neurons n_1 and n_2 *topological neighbors*.

The goal of the *refinement* is to relieve regions with incompatible states by splitting them. In the growing neural gas approach, a local *error* is introduced for each neuron. In the *GNG-Q* approach, the error is a counter for the changes in the policy in the respective region. Initially, the approximation has a very coarse resolution and in each learning step, Q-Learning is applied to the current approximation. Regions that need refinement are identified by monitoring changes in the policy learned so far: Every time, a Q-update causes that $\operatorname{argmax}_a \widehat{Q}_t(n_1, a) \neq \operatorname{argmax}_a \widehat{Q}_{t+1}(n_1, a)$ holds, the error for the current state's region is increased as this means, that the agent would now prefer a different action for this region. The approximation is periodically refined by adding a new neuron in the region with the highest error because this is evidence that the region consists of states that have to be treated separately to obtain a good policy.

In the generic growing neural gas approach, neurons are *moved* in order to adapt to the probability distribution by which the samples of the input space are drawn. In RL one has to deal with sequences of samples as the agent interacts with the environment in such a way that it iteratively transitions from one state to a subsequent state. Thus, the network in *GNG-Q* would try to follow the trajectories of the agent. In order to prohibit this behavior and to supply a static approximation during each episode, an additional set is introduced for each region: The so-called *regional states* R_n store the states, the agent visited in n's region during the current episode. After each episode, each neuron n is moved a small portion ϵ_b towards the centroid of R_n. Additionally, each topological neighbor of n is moved a much smaller portion $\epsilon_n \ll \epsilon_b$ towards the centroid of R_n. Thus, the positions of the neurons adapt to states visited during the last episode.

4 Revised Growing Neural Gas Q-Learning

This section argues that non-determinism may occur in the abstract state space although the original MDP is deterministic. We argue how to better deal with this non-determinism, show how to add eligibility traces to speed up learning, investigate the role of neighborhood connections and introduce criteria for the movement and the refinement as well as a revised refinement method. Finally, we sum up the new algorithm called *GNG-Q*$^+$ in Alg. 2.

Algorithm 1: GNG-Q (the numbers in braces indicate changes made in *GNG-Q*$^+$)

```
foreach episode do
    while episode not finished do
        observe $s_t$, perform $a_t$, observe $s_{t+1}$ and $r(s_t, a_t)$
        use current approximation to compute $\psi(s_t)$, $\psi(s_{t+1})$
        update Q-estimation according to Eq. 1                          (1)
        update neighbor connections and error values                    (2)
        if network still adapts then insert a new neuron every
        $\lambda$'th iteration                                          (3)
    if network still adapts then
        adapt $n$ and $n$'s topological neighbors to centroid of $n$'s
        regional states $R_n$                                           (4)
```

4.1 Dealing with Induced Non-Determinism

Although we consider deterministic environments—the state transition function as well as the reward function is deterministic—the aggregation of states can introduce non-determinism in the abstracted MDP: Consider the situation in a shortest path scenario depicted in Fig. 1: If the agent performs the action "go right" in one of the states abstracted by the region $\widehat{\mathrm{s}}_1$, then, depending on its location in this region, the subsequent state can be in region $\widehat{\mathrm{s}}_2$ or $\widehat{\mathrm{s}}_3$. As the agent updates its estimates of $\widehat{Q}(\widehat{\mathrm{s}}_1, \rightarrow)$ depending on the Q-vector of the succeeding state, the Q-values are prone to oscillate. This problem occurs, if there are at least two states s_1, s_2 in one region $\widehat{\mathrm{s}}$ that result in states that are abstracted by different abstract states after performing the same action, formally: $\exists s_1 \neq s_2 \in \widehat{\mathrm{s}}, \exists a \in A : \psi(\mathrm{T}(s_1, a)) \neq \psi(\mathrm{T}(s_2, a))$. This kind of non-determinism can be caused by irregular shaped regions (as in the *GNG-Q* approach, cf. Fig. 1(a)) but may also occur whenever transitions between differently sized abstract states are possible (cf. Fig. 1(b)). In [8], this non-determinism is also called "the loss of the Markov property" as the subsequent state $\widehat{\mathrm{s}}_{t+1}$ now not only depends on $\widehat{\mathrm{s}}_t$ but also on the states visited before time t.

To improve the behavior with this non-determinism in the abstract MDP, we equip *GNG-Q*$^+$ with a learning rate α_t that depends on the time step t such that $\sum_t \alpha_t = \infty$ and $\sum_t \alpha_t^2 < \infty$ hold. Note, that this is the same condition on the learning rate as given in [20] for the convergence of Q-Learning. Following the idea of [7], such a learning rate can be constructed as $\alpha_t = \frac{1}{1+\mathrm{visits}(\widehat{\mathrm{s}},a)^\omega}$ where $\mathrm{visits}(\widehat{\mathrm{s}}, a)$ is the number of executions of action a in the abstracted state $\widehat{\mathrm{s}}$ and ω is a constant to regulate the decrease of the learning rate over time. To fulfill the condition above, $0.5 < \omega \leq 1$ has to hold. This learning rate decreases the influence of updates of each $(\widehat{\mathrm{s}}, a)$ over time and helps to reduce the oscillation of the Q-values. The original *GNG-Q* did not consider this non-determinism and used a constant learning rate. Thus, the Q-update *(1)* in *GNG-Q*$^+$ is adapted to use a learning rate α_t as described.

Abstracting states transforms the learning problem in something close to a partially observable MDP (POMDP) [15, 19]. Unfortunately, this improvement does not provide a solution for the POMDP, but it introduces more stability to the learning process.

4.2 Adding Eligibility Traces

One way to speed up learning in reinforcement learning problems is the use of eligibility traces [6]. They offer a means to distribute immediate reward to all state-action pairs (s, a) that have been visited

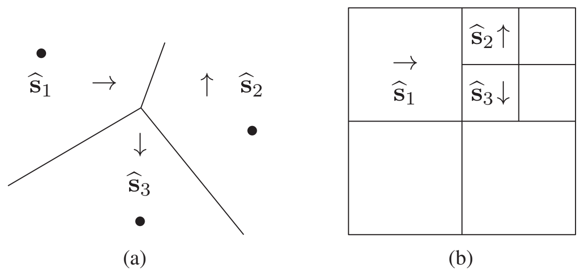


Figure 1. Induced non-determinism in different approximation schemes: The action $\rightarrow$ in $\widehat{s}_1$ may lead to different succeeding states depending on the actual state that is abstracted to $\widehat{s}_1$.

during the current episode according to their eligibility $e(s, a)$. This counter is increased by 1 every time the action a_t is performed in the current state s_t and if a_t is the action with the highest Q-value for state s_t. If a_t is not the maximal action, the eligibility traces for all state-action pairs are reset to zero. Additionally, each $e(s, a)$ is decayed by a factor $\lambda \in [0, 1]$ for all state-action pairs:

$$e_{t+1}(s,a) = \begin{cases} \gamma\lambda(e_t(s,a)+1) & \text{if } s = s_t \text{ and } a = a_t = a^\star \\ 0 & \text{if } a_t \neq a^\star \\ \gamma\lambda e_t(s,a) & \text{if } s \neq s_t \text{ or } a \neq a_t \end{cases}$$

with $a^\star = \text{argmax}_{a'} \widehat{Q}_t(s, a')$. Thus, reward or punishment can be credited for all state-action pairs that were "responsible" for it. If the agent performs an exploratory action (i.e. an action that has not the highest Q-value for the current state), the eligibility traces are cut off.

In every update, the temporal difference error δ_t is computed as $\delta_t = r - \widehat{Q}_t(s_t, a_t) + \gamma \max_{a'} \widehat{Q}_t(s'_t, a')$ between the current state s_t and the succeeding state s_{t+1}. This value is added to the Q-value of every state-action pair:

$$\widehat{Q}_{t+1}(s,a) = \widehat{Q}_t(s,a) + \alpha_t \delta_t e_t(s,a), \; \forall s \in S, \forall a \in A$$

The method used here is called *Watkins's Q(λ)* [17]; for a comparison of other approaches see [17, 6]. The transformation of this approach to neurons is straightforward: For each neuron n, we use $e_t(n, a)$ to express the eligibility for this neuron-action pair (compare Alg. 2).

4.3 Neighborhood Connections

In each update, the nearest neuron n_1 to the current state s and the nearest neuron n'_1 to the subsequent state are connected if $n_1 \neq n'_1$. Thus, the neighborhood connections abstract the transitions of the original MDP to the abstracted MDP: Each connection between two abstract states $\widehat{s}_1$ and $\widehat{s}_2$ implies that an action performed in a state s_1 with $\psi(s_1) = \widehat{s}_1$ resulted in a state s_2 with $\psi(s_2) = \widehat{s}_2$ (or vice versa as the connections are undirected). Every time, a neuron n_1 is the nearest neuron to the current state, the age of all connections (n_1, n') connecting n_1 with its neighbors n' is increased. Furthermore, a new connection (n_1, n'_1) is created with n'_1 representing the nearest neuron of s_2. If this connection already exists, its age is reset to zero. If the age of any connection exceeds age_{max}, the connection is removed as this is evidence, that this connection is outdated: Consider the latest age_{max} neuron-action pairs (n, a) then there was no action that lead from n_1 to n'_1 (or vice versa). Thus it is save to assume, that the approximation changed in a way that there is no action that leads from any state abstracted by n_1 to any state abstracted by n'_1 (or vice versa). If the deletion of connections results in isolated neurons, these may be removed as well, as they tend to be unreachable following the same argumentation.

Contrary to *GNG-Q* and the generic growing neural gas algorithm that use the neighborhood connections to adapt the nearest neuron and all its topological neighbors, *GNG-Q*$^+$uses the neighborhood connections to determine "dead" abstract states, i.e. abstract states that have not been visited for a long time (cf. Alg.1, *(2)*). Reasons for this could be changes in the environment or changes in the approximation due to adaptations or refinements. Additionally, the neighbor connections in *GNG-Q*$^+$ could be used to model an abstract transition function $\widehat{\mathrm{T}} : \widehat{\mathcal{S}} \times A \rightarrow \widehat{\mathcal{S}}$ for the abstracted MDP $\widehat{\mathrm{M}}$.

4.4 Adjusting the Approximation

After each episode, the approximation can be adjusted by two operations: The *adaptation* moves the neurons in such a way that they represent the state space as good as possible and the *refinement* is used to split regions that contain incompatible states. In the following, we will present changes to these operations and state new conditions for their application.

In the *GNG-Q*$^+$ approach, each neuron is moved by ϵ_b in the direction of the centroid of its regional states. Contrary to the *GNG-Q* approach, the positions of a neuron's topological neighbors (cf. Alg. 1, *(4)*) are not changed in order to increase the stability of the approximation learned so far. Thus, in *GNG-Q*$^+$ states visited in one region only affect the center of this particular region.

Especially, we only move a neuron n if its associated error value $\text{error}(n)$ is larger than a small threshold Δ (e.g. $\Delta = 1$). The intention is that we do not want to move a neuron, which is well positioned and has a useful Q-vector. It is intuitive to consider the error of a neuron for this purpose as this value is increased every time the policy in its region changes. Thus, the performance of neurons with high error values may increase by repositioning them whereas neurons whose local policy has not changed often recently shall keep their position. This behavior can be seen as parameter exploration as discussed e.g. in [14].

As each region is only assigned one Q-vector for all its contained states, it is important, that only compatible states are aggregated. During the learning steps, the policy is monitored and every time, the local policy in one region changes, the associated error is increased. In [1], after each λ_{insert} steps, the region of the neuron with the highest error is refined, unless a specific stopping condition is met.

In our approach, we refine the approximation *after* an episode, if $\sum_{n \in N_t} \text{error}(n) > |N_t|$ holds and at least λ_{insert} episodes have passed since the last refinement. Thus, the refined approximation can be adapted for some time and Q-vectors on the new approximation can be learned accordingly. Of course, one could refine the approximation whenever the sum of all errors is larger than zero. However, this might cause a too fine approximation, as sometimes a change in the policy is inevitable. The motivation for the condition above is, that on average each neuron is "allowed" to change its policy once per episode.

The refinement is done by cloning the neuron n_e with the highest error and perturbate n_e and the new neuron n^+ by a small amount to ensure that their initial positions differ slightly. The new neuron n^+ is initialized with the Q-vector of n_e and connected to the same topological neighbors. After this refinement, the error values of all neurons are reset to zero, to reflect the fact that the abstraction function ψ has changed. In *GNG-Q*$^+$ the insertion (*(3)* in Alg. 1) is performed after one episode.

The condition stated above implicitly provides a *stopping crite-*

rion for adjustments of the abstracted state space: If the errors of all neurons are small, this is evidence, that the overall policy has not changed often since the last insertion and the current policy can be expressed sufficiently with the current resolution. The *GNG-Q* uses an external measure to decide, when the approximation is fine enough, whereas *GNG-Q*$^+$ uses the above criteria on the error to decide when the approximation should be refined or moved.

As we change the number and the positions of the neurons over time, we also change the abstraction function because ψ is defined by the positions of the neurons and the used distance measure. After moving the neurons, one state s may be abstracted by a different neuron than before because it may now be in a different region. Additionally, if a new neuron is added, the number of regions is changed and thus, states may be in a different region after the refinement, too. The refinement also changes the domain of the estimated Q-function but the influence is rather low as the Q-vector in the two new regions is the same as before. Dead regions that are deleted also change the domain of $\widehat{Q}$ but this does not influence the approximation as these regions were not visited for a long time.

4.5 Complete Algorithm

This section presents the complete algorithm of our approach (cf. Alg. 2). To deal with different sized dimensions, it is useful to scale the values of the states to be from the same interval. A common approach is to normalize a value $x \in [x_{min}, x_{max}]$ to a value $x_{scaled} \in [x_{scaled}^{min}, x_{scaled}^{max}]$ such that

$$x_{scaled} = \frac{x - x_{min}}{x_{max} - x_{min}} \cdot (x_{scaled}^{max} - x_{scaled}^{min}) + x_{scaled}^{min}.$$

Thus, the distance function employed in the nearest neighbor rule weights all dimensions equally and no dimension will be favored just because its values are from a larger scale. In our approach we used a normalization to the interval $[0, 1]$.

Algorithm 2: *GNG-Q*$^+$

```
foreach episode do
    initialize state $s$
    initialize regional states $R_n = \varnothing,\ \forall n \in N$
    initialize eligibility traces: $e(n, a) = 0,\ \forall n \in N,\ \forall a \in A$
    while episode not finished do
            /* interaction with environment */
        observe current state $s_t$
        determine nearest neuron $n_1 = \mathrm{nn}(s_t)$ to current state
        select and perform action $a_t$
        observe subsequent state $s_{t+1}$
        determine nearest neuron $n'_1 = \mathrm{nn}(s_{t+1})$ to $s_{t+1}$
                /* update neurons */
        $\mathrm{visits}(n_1, a_t) \leftarrow \mathrm{visits}(n_1, a_t) + 1$
        store $s_t$ in $n_1$'s regional states: $R_{n_1} \leftarrow R_{n_1} \cup \{s_t\}$
        discount errors for all neurons
        connect neurons $n_1, n'_1$
        increase age of all neighborhood connections of $n_1$
                /* update $\widehat{Q}$ */
        $\alpha_t = \frac{1}{\mathrm{visits}(n_1, a_t)^\omega}$
        $\delta_t = r - \widehat{Q}_t(n_1, a_t) + \gamma \max_{a'} \widehat{Q}_t(n'_1, a')$
        $e_{t+1}(n, a) \leftarrow e_t(n, a) + 1$
        foreach neuron $n \in N$ do
            foreach action $a \in A$ do
                $\widehat{Q}_{t+1}(n, a) \leftarrow \widehat{Q}_t(n, a) + \alpha_t \delta_t e_t(n, a)$
                if $a_t = \mathrm{argmax}_{a'} \widehat{Q}_t(s_t, a')$ then
                    $e_{t+1}(n, a) \leftarrow \gamma \lambda e_t(n, a)$
                else
                    $e_{t+1}(n, a) \leftarrow 0$
            /* Monitor changes in policy */
        if $\mathrm{argmax}_a \widehat{Q}_t(n_1, a) \neq \mathrm{argmax}_a \widehat{Q}_{t+1}(n_1, a)$ then
            increase $\mathrm{error}(n_1)$
        /* Adaptation of approximation */
    foreach neuron $n \in N$ do
        if $\mathrm{error}(n) > \Delta$ then
            compute centroid $\overline{s_n}$ of regional states for neuron $n$:
                $\overline{s_n} = \frac{1}{|R_n|} \sum_{s_r \in R_n} s_r$
            adaptation of neuron $n$ to $\overline{s_n}$:
                $w_n \leftarrow w_n + \epsilon_b \cdot (\overline{s_n} - w_n)$
        /* Refinement of approximation */
    if $\sum_{n \in N} \mathrm{error}(n) > |N|$ then
        insert new neuron in most erroneous region
```

5 Evaluation

In this section, we experimentally evaluate the *GNG-Q*$^+$ approach and compare the results with those of other approaches from literature. At first, the different problem domains are described, followed by a description of the evaluation setup and the default parameter setting. Then *GNG-Q*$^+$ and *GNG-Q* are compared in a 2-dimensional continuous world. After that, *GNG-Q*$^+$ and the Fourier approach [10] are evaluated in a d-dimensional continuous world and *GNG-Q*$^+$ is used to solve the acrobot swing up control problem. We compare our results in the acrobot domain to a baseline approach and additionally to several approaches from literature.

5.1 Problem Domains

Here, we describe the two different problem domains that we chose to evaluate the performance of *GNG-Q*$^+$: The *d-dimensional continuous world* is defined and the *acrobot swing up* problem from [16] is introduced.

5.1.1 The d-Dimensional Continuous World

To test the performance of *GNG-Q*$^+$ in higher dimensional spaces, we use an extension of the continuous world employed in [1]: In the *d-dimensional continuous world* the coordinates are from $[0, 1]^d \subset \mathbb{R}^d$ and the agent has to learn the shortest path from all positions to the goal located in $s_{goal} := (1, 1, \ldots, 1) \in \mathbb{R}^d$. The agent can perform actions a_i^+ and a_i^- for each dimension $0 \leq i < d$ that increase or decrease the value of the i-th component by $0 < s_{step} \leq 1$, i.e. it takes a step along dimension i. Thus, the agent's action set A consists of $2 \cdot d$ actions and the state space is $S = \{(x, y) \mid x, y \in [0, 1]^d\}$. At the beginning of each episode, the agent is randomly placed inside the world and if it tries to leave the world in any dimension it is positioned on the border of this dimension. To relax the goal condition, the goal is modeled as a hypercube with the edge length equal to s_{step}. For the action that leads the agent to the goal, a reward of 0 is awarded, for all other action, the reward is -1. Clearly, for $d = 2$ this environment reduces to the environment from [2] and Sec. 5.4.

5.1.2 The Acrobot Swing Up Control Problem

As additional environment we considerd the "acrobot swing up" (sometimes called double pendulum swing up) control problem as it is defined in [16].

The acrobot is a two-link under-actuated robot with the goal to move the second link above a given height. The state space consists of four continuous dimensions, namely the angles $\theta_1, \theta_2 \in [-\pi, \pi]$ and the corresponding angular velocities $\dot{\theta}_1 \in [-4\pi, 4\pi]$ and $\dot{\theta}_2 \in [-9\pi, 9\pi]$. A sample state is shown in Fig. 2. The lengths of the

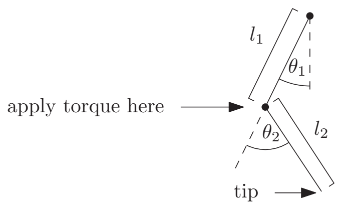


Figure 2. The acrobot from [16].

links ($l_1 = l_2 = 1$), their masses ($m_1 = m_2 = 1$), the gravity ($g = 9.8$), the goal height ($h = 1$) and the torque applied to the link are parameters that we chose in accordance to [16]. The behavior of the acrobot is calculated via formulas which can be found in [16], too. For our experiments we used a library[1] and adjusted it such that it fits the dynamics in [16].

5.2 Experimental Setup

In all experiments, one setting with a specific method and a fixed setup was simulated 100 times with different random seeds and averaged afterwards. Furthermore, the reward for the agent is always the same. If the agent reaches the goal state, it receives a reward of 0 and -1 in every other step. In each experiment, we initialized the Q-tables with zero for every state-action pair.

We divided the evaluation in a learning and a test-phase. In the learning phase we used an ε-greedy approach and learned for 10 episodes. In the following test phase, we tested the learned policy for 250 episodes in which the agent always chose the action with the highest Q-value. If the agent did not reach the goal after a fixed number of steps, this test-episode was stopped. During the evaluation phase, the agent was not allowed to learn.

For all our experiments in the continuous worlds, we chose the step size of the agent as $s_{step} = 0.05$ and the maximal number of trials in one episode as $\left(\frac{1}{s_{step}}\right)^d$. In the acrobot domain, we allowed the agent a maximal number of 5000 steps to learn and the evaluation was finished if either the goal was reached or 3000 steps have passed.

5.3 Parameter Values for Our Approach

To define useful basis values for our approach, we evaluated *GNG-Q*$^+$ for several parameter values on the 2-dimensional continuous world. In contrast to the other experiments, we only report on averages of 30 runs.

[1] `http://library.rl-community.org`

For the exploration strategy we used ε-greedy and experimented with $\varepsilon \in \{0.01, 0.05, 0.1, 0.15, 0.2, 0.3, 0.4\}$ and obtained the best results for $\varepsilon \in \{0.01, 0.05\}$ with $\varepsilon = 0.01$ being slightly better for all settings. The strengths of the adaptation towards the centroid of the regional states were chosen from $\{0.01, 0.025, 0.05\}$ and $\epsilon_b = 0.05$ performed best. We experimented with several exponents for the learning rate and chose $\omega \in \{0.51, 0.55, 0.6, 0.65, 0.7\}$ from which $\omega \in \{0.55, 0.6, 0.65\}$ performed best. As example, we investigated the influence of the number of episodes λ_{insert} between two refinements of the approximation and chose $\lambda_{insert} \in \{10, 20, 30, 40, 50\}$. The results are plotted in Fig. 3(a) and it can be seen, that $\lambda_{insert} \in \{30, 40, 50\}$ performs best while smaller values result in slower convergence. Note that after around 2000 episodes, all graphs reach their minimal values.

Based on theses experiments, we chose the following basic values for *GNG-Q*$^+$ (variations are mentioned correspondingly):

- decay factor for eligibility traces $\lambda = 0.9$
- number of episodes between two insertions $\lambda_{insert} = 40$
- maximal age of neighbor connections age$_{max} = 300$
- discount factor $\gamma = 0.9$
- exploration probability $\varepsilon = 0.01$
- exponent for the time dependent learning rate $\omega = 0.55$
- adaptation strength $\epsilon_b = 0.05$
- error decay $\beta = 0.9999$

5.4 Comparison between *GNG-Q* and *GNG-Q*$^+$

For the comparison of *GNG-Q* and *GNG-Q*$^+$ we employed the same scenario and the same parameter values as in [1] and thus, we used exploration rate $\varepsilon = 0.05$ and discount factor $\gamma = 0.95$ for both approaches, learning rate $\alpha = 0.1$, $\epsilon_b = 0.05$, $\epsilon_n = 0.0006$ and $\lambda_{insert} = 1000$ for *GNG-Q*. For the new approach, we used $\lambda_{insert} = 40$, $\epsilon_b = 0.05$ and for the eligibility traces $\lambda = 0.9$.

As we can see from Fig. 4, *GNG-Q* finds a good policy slightly faster in the beginning. After around 1000 episodes, *GNG-Q*$^+$ is as good as *GNG-Q* and remains more stable without high peeks, which can be seen in the zoomed section. Additionally, *GNG-Q*$^+$ needs less neurons than *GNG-Q* to represent the learned policy. Fig. 5 shows the number of neurons needed to represent the learned policy for which the number steps are shown in Fig. 4. *GNG-Q*$^+$ does not need more than 20 neurons on average with their number remaining stable whereas *GNG-Q* needs more than 100 neurons after 10000 episodes without stabilization of the number of neurons. Obviously, learning with *GNG-Q*$^+$ compared to *GNG-Q* results in a better policy while less neurons and thus less memory is needed.

Fig. 6 shows the approximation computed by one arbitrary run of *GNG-Q*. It can be seen in the left part, that the state regions are rather small and that the density of regions increases towards the goal. Although the policy (depicted in the right part of the figure) in combination with the computed approximation leads to an optimal behavior, the right part of this figure shows, that there are many redundant regions, e.g. all neighboring regions that point in the same direction. In Fig. 7(a), the state space approximation computed by one run of *GNG-Q*$^+$ together with the learned policy is shown. This result is clearly better than the one in Fig. 6 as the learned behavior for both approximation is identical but the approximation computed by *GNG-Q*$^+$ is much smaller.

Of course, the size of an approximation is not the only metric that has to be considered. However, for equal performances, a smaller approximation in number of states is definitely better than a larger one as this saves computing time and memory.

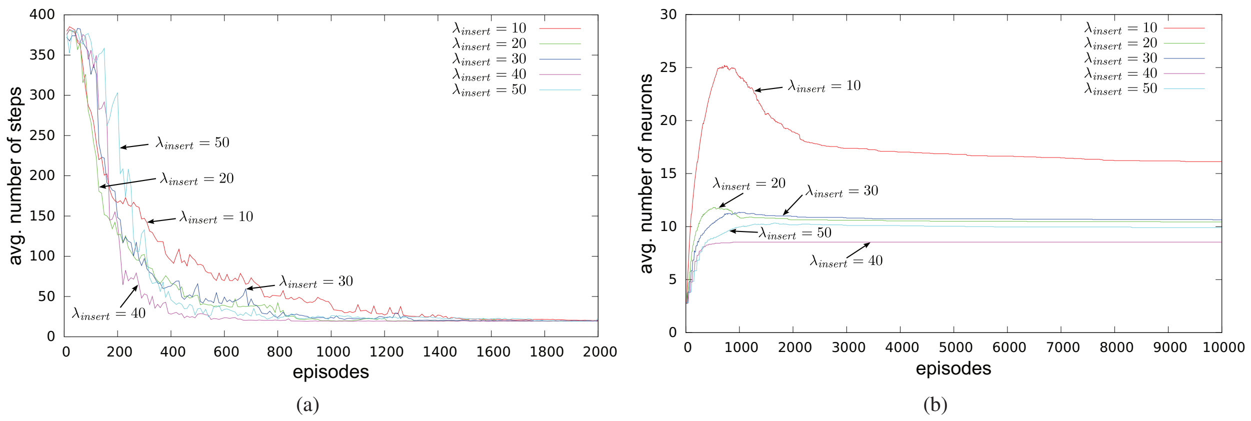


Figure 3. Average number of steps to reach the goal *(a)* and average number of neurons *(b)* with different values for λ_{insert} using *GNG-Q*$^+$.

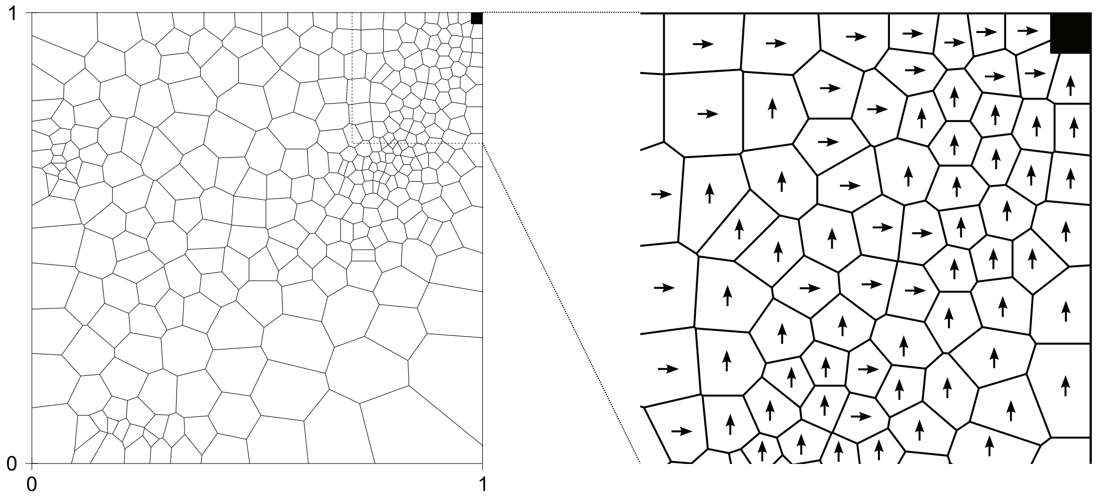


Figure 6. Approximation computed by one run of *GNG-Q*. The left part shows the abstracted state space and the right part depicts the learned policy for the dashed area.

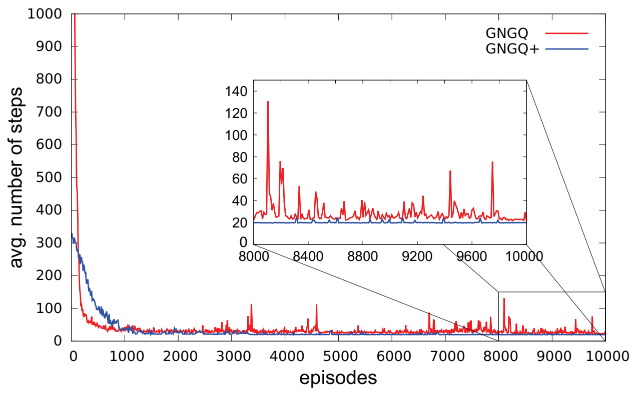


Figure 4. Average number of steps with *GNG-Q* and *GNG-Q*$^+$ in the scenario from [1].

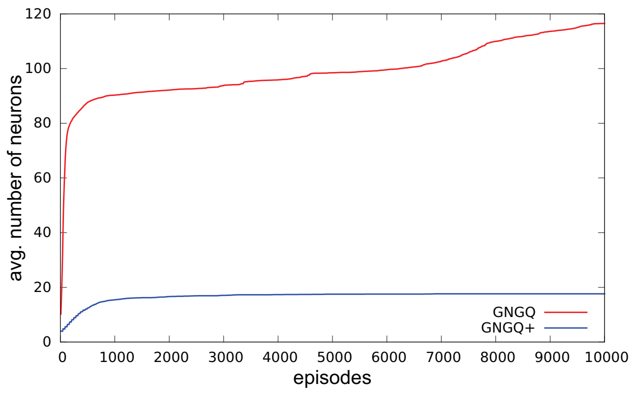


Figure 5. Size of the approximation in number of needed neurons with *GNG-Q* and *GNG-Q*$^+$ in the same scenario as in Fig. 4.

For the *GNG-Q*$^+$ approach, we investigated the distribution of the neurons in the 2-dimensional continuous world. For this, we collected the positions of all neurons from all evaluations and plotted them into the environment. The heat map in Fig. 7(b) gives an overview of the average distribution of the neurons: The opacity indicates the relation of number of neurons in this tile to the maximal number of neurons over all tiles ("the darker the tile, the more neurons fell into this particular tile").

We can see that the density of neurons increases towards the goal and that no neurons are positioned near $x = 0$ or $y = 0$. This stems from the fact, that the neurons are moved towards the centroid of all positions visited in the respective region. The distribution of neurons

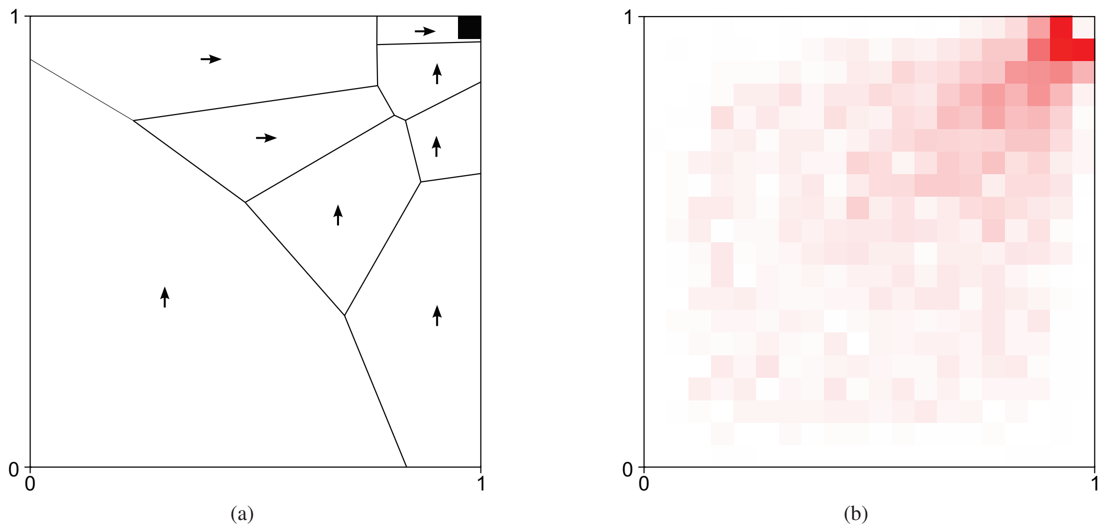


Figure 7. Exemplary state space with 8 states computed by one arbitrary run of *GNG-Q*$^+$ in (a) and heat map showing the density of average positions of all neurons from all evaluations in the 2-dimensional continuous world in (b).

in the old *GNG-Q* approach is similar but in *GNG-Q*$^+$, the number of neurons is drastically reduced (compare Fig. 5).

5.5 Experiments in Multidimensional Spaces

In this section we investigate the performance of *GNG-Q*$^+$ in two multi-dimensional environments and compare our results to approaches from literature.

5.5.1 The d-Dimensional Continuous World

Fig. 8 shows the comparison of *GNG-Q*$^+$ and the Fourier approach from [10] in a 3-dimensional continuous world with six actions. The parameters for the Fourier approach are taken from [10] for a similar environment called the “Discontinuous Room” (Fourier order = 5; $\lambda = 0.9$; $\alpha = 0.001$; $\gamma = 0.9$; $\epsilon = 0$). For the first 1000 episodes the Fourier approach outperforms *GNG-Q*$^+$. After that *GNG-Q*$^+$ remains more stable than the Fourier approach and needs less steps to reach the goal. Note that the implementation of the Fourier approach we used[2] was about 10-times slower than *GNG-Q*$^+$ (regarding computation time).

5.5.2 The Acrobot Swing Up Control Problem

We evaluated the performance of the acrobot for torque $\in \{1, 2, 5\}$ with our *GNG-Q*$^+$ approach and started from a down-hanging position, i.e. $\theta_1 = \theta_2 = \dot{\theta}_1 = \dot{\theta}_2 = 0$.

Fig. 9 shows the learning curves for the average number of steps for torque $\in \{1, 2, 5\}$. We can see that it takes about 12000 episodes to find a good policy for torque $= 1$. For torque $\in \{2, 5\}$, the learning curve does not improve significantly after 5000 episodes. After 40000 episodes the policies found by *GNG-Q*$^+$result in about 118 steps for torque $= 1$, 34 steps for torque $= 2$ and 17 steps for torque $= 5$ on average.

Fig. 10 shows the number of needed neurons for torque $\in \{1, 2, 5\}$. After 40000 episodes *GNG-Q*$^+$ uses around 60 neurons on average to approximate the state space for torque $= 1$, 30 neurons

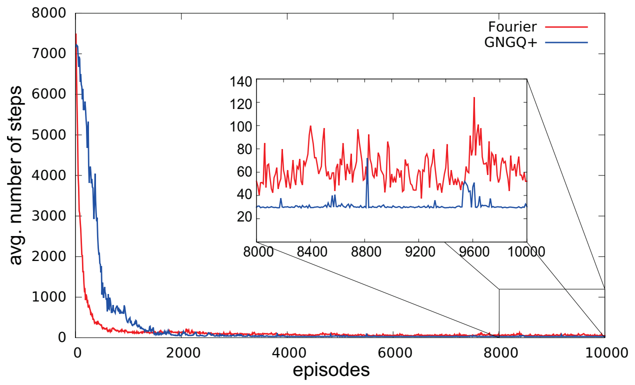


Figure 8. Average number of steps with *GNG-Q*$^+$ and Fourier based approach from [10] in a 3-dimensional continuous world.

for torque $= 2$ and 20 neurons for torque $= 5$. It seems that *GNG-Q*$^+$ would have inserted more neurons without an obvious benefit if the learning proceeded e.g. caused by exploration. As seen in Fig. 9 the policies for torque $\in \{2, 5\}$ do not improve after 5000 episodes but still, new neurons are inserted. This fact will be investigated in further research. Nevertheless, *GNG-Q*$^+$ can represent the four-dimensional state space of the acrobot problem in a very compact way with only 60 neurons on average for the hardest task.

For comparison, we employed the following baseline algorithm: We use Q-Learning on a predefined uniform discretization with a similar state space size as computed by *GNG-Q*$^+$ at convergence. Without any knowledge on the problem at hand, it is advisable to use the same resolution for each dimension. Of course, this may not be optimal as some dimensions might require a finer resolution than others. As the *GNG-Q*$^+$ approach needs approx. 61.2 neurons for torque=1, approx. 30 neurons for torque=2, and approx. 18.2 neurons for torque=5, we decided to split each dimension in 3 equal portions and thus the resulting state space has $3^4 = 81$ states. We also ran experiments for larger state spaces where the performance of this baseline approach improved but still was not satisfying. Nevertheless, this experiment should only serve as a rough comparison.

Fig. 11 shows the learning curves of the baseline approach for the

[2] http://library.rl-community.org/wiki/Sarsa_Lambda_Fourier_Basis_(Java)

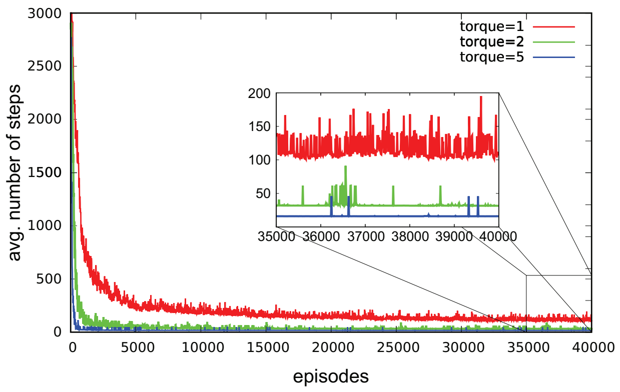


Figure 9. Comparison of the average number of steps for *GNG-Q*$^+$ in the acrobot domain with torque $\in \{1, 2, 5\}$.

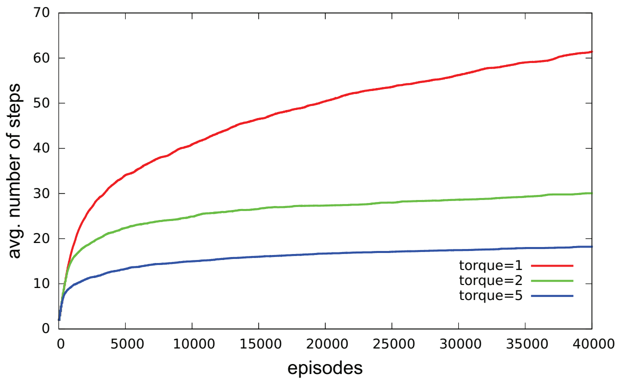


Figure 10. Comparison of the approximation size in numbers of neurons for *GNG-Q*$^+$ in the acrobot domain with torque $\in \{1, 2, 5\}$.

average number of steps with torque $\in \{1, 2, 5\}$. The learning curve for torque $= 1$ oscillates heavily, but the oscillation decreases with increasing torque. Table 1 shows the comparison in terms of mean values (μ) and standard deviation (σ) of *GNG-Q*$^+$ and the baseline approach for the last 5000 episodes. The values correspond to the learning curves from Fig. 9 and 11. Except for torque $= 5$, *GNG-Q*$^+$ outperforms the baseline approach considering the average number of steps needed to reach the goal by a factor of at least two. As already seen in Figs. 9 and 11 the learning curves of the baseline approach oscillate more than *GNG-Q*$^+$ which results in a higher standard deviation. For torque $= 5$ the baseline approach is slightly better than *GNG-Q*$^+$ on average over the last 5000 episodes, but the baseline approach needs approx. 15000 episodes to learn the good policy while *GNG-Q*$^+$ needs only 1000 episodes. Hence, *GNG-Q*$^+$ performs better than the baseline approach in most cases while needing less states. It seems that *GNG-Q*$^+$ places more neurons where a high density is necessary and only few neurons where a coarser resolution suffices.

Table 1. Comparison (mean value μ and standard deviation σ) of *GNG-Q*$^+$ and the baseline approach over the last 5000 episodes with torque $\in \{1, 2, 5\}$.

	GNG-Q$^+$		Baseline	
Torque	μ	σ	μ	σ
1	**117.69**	**16.53**	465.55	809.3
2	**33.18**	**5.98**	79.62	31.81
5	16.17	**2.66**	**15.25**	5.16

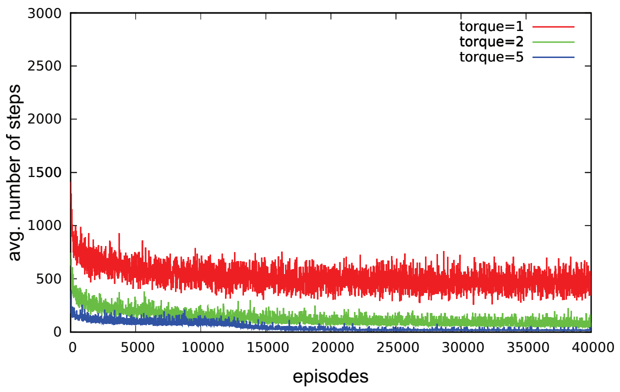


Figure 11. Comparison of the average number of steps for the baseline approach in the acrobot domain with torque $\in \{1, 2, 5\}$.

We also compared our results to others found in literature but it was not always possible to obtain precise values for the performance[3]. Furthermore, it was sometimes unclear which parameters values were used in the presented experiments. The *GNG-Q*$^+$ approach needs on average about 118 steps to reach the goal state from a down-hanging position. Always starting from that position the *best* policy found by *GNG-Q*$^+$ results in 72 steps. The policy found in [10] needs around 100 steps to reach the goal. Unfortunately, the start position and the torque applied are not mentioned for these experiments. The method described in [5] needs about 250 steps from the down-hanging position with torque $= 1$ whereas [3] needs at least 87 steps for the same setting. In [16], their CMAC approach needed around 85–90 steps.

5.6 Conclusion of Evaluation

In this section we compared *GNG-Q*$^+$ with *GNG-Q*, the Fourier approach, a baseline approach and other approaches from literature. The changes made to improve *GNG-Q* to *GNG-Q*$^+$ increased the performance regarding stabilization and needed neurons. Then we showed that *GNG-Q*$^+$ needs less steps at convergence than the Fourier approach. After that we investigated the difference between *GNG-Q*$^+$ and a baseline approach, resulting in more stable and in most cases better performance of *GNG-Q*$^+$. Finally, we presented results from comparable approaches.

6 Conclusion & Future Work

In this paper, we analyzed *GNG-Q*, an approach that uses a combination of Q-Learning and growing neural gas to learn a policy in parallel with a state space abstraction and proposed *GNG-Q*$^+$ that improves the former approach. The use of a non-deterministic Q-update, the incorporation of eligibility traces and the formulation of criteria for adjustments of the state space clearly improved the performance. Our evaluation showed that *GNG-Q*$^+$ is capable of learning compact state space representations in parallel with a (nearly) optimal policy in several continuous reinforcement learning problems with up to four dimensions and eight actions. Its performance is well competitive with other approaches from literature without the need of knowing the considered reinforcement learning problem beforehand.

For the future, we plan to incorporate a merging strategy to deal with the fact that Q-vectors of neighboring state regions may become

[3] We are aware that the actual results for the mentioned literature may be better than reported here and we do not want to discredit them. The values here should only show that our approach is comparable to existing methods.

similar during learning. The next step is to theoretically investigate the adapted method and to analyze convergence and optimality questions. Additionally, we will investigate how the number of neurons may be bounded further to avoid the increase caused e.g. by taking exploratory actions. Furthermore, the approach will be employed in the multi-agent reinforcement learning context. There, it will be investigated, how the proposed approach can deal with partial observability.

REFERENCES

[1] M. Baumann and H. Kleine Büning, 'State aggregation by growing neural gas for reinforcement learning in continuous state spaces', in *The Tenth International Conference on Machine Learning and Applications (ICMLA 2011)*, pp. 430–435. IEEE Computer Society, (2011).

[2] J. A. Boyan and A. W. Moore, 'Generalization in reinforcement learning: Safely approximating the value function', in *Neural Information Processing Systems 7*, pp. 369–376, (1995).

[3] K. A. Bush, *An Echo State Model of Non-Markovian Reinforcement Learning*, Ph.D. dissertation, Colorado State University, 2008.

[4] L. Buşoniu, D. Ernst, B. De Schutter, and R. Babuška, 'Approximate reinforcement learning: An overview', in *Proceedings of the IEEE Symposium on Adaptive Dynamic Programming and Reinforcement Learning (ADPRL 2011)*, pp. 1–8, (2011).

[5] A. da Motta Salles Barreto and C. W. Anderson, 'Restricted gradient-descent algorithm for value-function approximation in reinforcement learning', *Artif. Intell.*, **172**(4-5), 454–482, (2007).

[6] P. Dayan and T. J. Sejnowski, 'Td(lambda) converges with probability 1', *Machine Learning*, **14**(1), 295–301, (1994).

[7] E. Even-Dar and Y. Mansour, 'Learning rates for q-learning', *Journal of Machine Learning Research*, **5**, 1–25, (2003).

[8] F. Fernández and D. Borrajo, 'Two steps reinforcement learning', *International Journal of Intelligent Systems*, **23**, 213–245, (2008).

[9] B. Fritzke, 'A growing neural gas network learns topologies', in *Advances in Neural Information Processing Systems (NIPS) 7*, 625–632, MIT Press, (1995).

[10] G. Konidaris, S. Osentoski, and P. S. Thomas, 'Value function approximation in reinforcement learning using the fourier basis', in *Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence*, (2011).

[11] I. S. Lee and H. Y. Lau, 'Adaptive state space partitioning for reinforcement learning', *Engineering Applications of Artificial Intelligence*, **17**, 577–588, (2004).

[12] S. Lin and R. Wright, 'Evolutionary tile coding: An automated state abstraction algorithm for reinforcement learning', in *AAAI Workshops: Workshop on Abstraction, Reformulation, and Approximation (WARA-2010)*, (2010).

[13] B. Ratitch and D. Precup, 'Sparse distributed memories for on-line value-based reinforcement learning', in *Proceedings of the European Conference on Machine Learning (ECML)*, pp. 347–358, (2004).

[14] T. Rückstieß, F. Sehnke, T. Schaul, D. Wierstra, S. Yi, and J. Schmidhuber, 'Exploring parameter space in reinforcement learning', *Paladyn Journal of Behavioral Robotics*, **1**(1), 14–24, (2010).

[15] S. P. Singh, T. Jaakkola, and M. I. Jordan, 'Reinforcement learning with soft state aggregation', in *Advances in Neural Information Processing Systems 7*, (1995).

[16] R. S. Sutton, 'Generalization in reinforcement learning: Successful examples using sparse coarse coding', in *Neural Information Processing Systems 8*, pp. 1038–1044, (1996).

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

[18] C. Szepesvári, *Algorithms for Reinforcement Learning*, Morgan and Claypool, 2010.

[19] M. van Otterlo, *The Logic of Adaptive Behavior*, IOS Press, Amsterdam, 2009.

[20] C. J. C. H. Watkins and P. Dayan, 'Q-learning', *Machine Learning*, **8**, 272–292, (1992).

[21] C. J. C. H. Watkins, *Learning from Delayed Rewards*, Ph.D. dissertation, Cambridge University, 1989.

[22] S. Whiteson, M. E. Taylor, and P. Stone, 'Adaptive tile coding for value function approximation', Technical report, University of Texas at Austin, (2007).

5th International Workshop on Evolutionary and Reinforcement Learning for Autonomous Robot Systems (ERLARS 2012)
Montpellier, France, August 28 2012

Realizing Target-Directed Throwing With a Real Robot Using Machine Learning Techniques

Malte Wirkus[1] and **José de Gea Fernández**[2] and **Yohannes Kassahun**[3]

Abstract. This paper presents a practical application of machine learning techniques in real-world robotics. Our goal was to make a anthropomorphic robot throw a ball into a bin that is placed at an arbitrary position in front of the robot. We use evolutionary machine learning to optimize a cost function based on a simulation model to aim at the target and generate the necessary motion to throw the ball at the position that is estimated by the simulation model. In order to compensate for the error in the simulation model, we trained an artificial neural network based on data from real-world task executions. We show that a simple simulation model can already result in good throwing performance if machine learning is applied to compensate for the resulting simulation error.

1 Introduction

When machine learning is applied to robotic manipulation tasks, a common approach is to perform learning in a simulation environment. The simulation models can be very different in terms of complexity and accuracy in modeling real world behavior, ranging from very reduced models to very complex dynamic simulations. Especially for simpler models, but also for the more complex ones, it comes to problems when learned tasks are transferred to the real robot. These occur from restrictions of the real-world robot hardware that have not been considered in the simulation, or from inaccuracies that results from imprecise modeling of the robot, the task, physical phenomena and so on. Building a perfect simulation in a way that a seaming less transfer to a real robot is possible is very difficult and hardly possible for complex tasks that for example include dynamic interaction with the environment.

Different approaches to improve the transfer from simulation to the real system are given in literature. In [8] this problem is approached by constraining the search space to regions where the simulation is more likely to be correct. To realize this it is proposed to define a metric for the similarity of behaviors to compare simulation with real-world performance. An estimation of the likeliness for a successful transfer of a simulated to a real-world behavior is derived from this. An evolutionary algorithm is then used to optimizes multiple objectives that on the one hand optimize the primary target function in simulation, and on the other hand prefers solutions that are expected to perform well on the real system.

In [2], a forward model is learned that maps motor commands of a controller to expected sensor data (e.g. a compass) in simulation. During real-world execution, a simulation error is composed based on the forward model and actual sensor data. A correction function that modifies the behavior is then learned online to compensate for the simulation error.

In [3] walking optimization of a small two-legged robot was performed with no external simulation environment, instead a surrogate optimization scheme is used. By performing real-world experiments, data is collected to learn a smooth surrogate function that is used as a replacement for a simulation environment. Simultaneously the current estimation of the surrogate function is used to optimize the main objective (walking distance) in order to generate a new parameter set to be evaluated on the robot hardware. Although the surrogate function is not able to approximate the real-world performance of the robot appropriate on the whole domain, the number of experiments to be performed is rather low (≈ 50) to learn a good performing gait.

On the control area, learning strategies have been long using neural networks to implement dynamic controllers. At first they were used for learning the whole inverse dynamic model of the robot which was found later on to be too complex. Therefore, the use of neural networks was in recent years mainly to support an adaptive control scheme rather than trying to approximate the whole inverse dynamics. That is, there is available a certain inaccurate inverse model which is used by the main dynamic controller and the learning strategy utilizes neural networks to compensate for inaccuracies or unforeseen changes of that dynamic model [5], an idea we adopt in

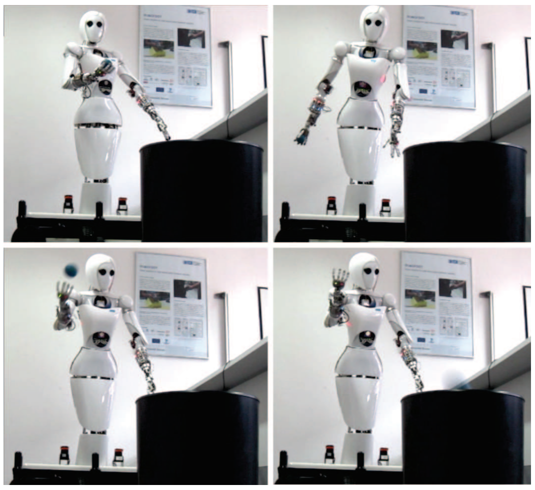


Figure 1. We used the robotic platform *AILA* in our experiments. This image sequence shows the execution of the throwing task.

[1] Robotics Innovation Center, DFKI Bremen, Germany, email: Malte.Wirkus@dfki.de
[2] Robotics Innovation Center, DFKI Bremen, Germany, email: Jose.de_Gea_Fernandez@dfki.de
[3] Universität Bremen, Germany, email: Kassahun@informatik.uni-bremen.de

our work in order to compensate for an imprecise simulation model.

In this paper we validate this approach on the problem of making a real robot throw a ball into a bin that is placed at an arbitrary position in a bounded area in front of the robot. An incomplete physical formulation simulates the problem of predicting how far a ball, placed in the hand of the anthropomorphic robot (see Figure 1), would be thrown, when executing a trajectory. A cost function is designed and minimized using an evolutionary learning technique to find trajectories and joint configurations to throw the ball at the desired target. Shortcomings in the simulation are compensated by supervised training of an artificial neural network in order to enable for real world application which is also trained using evolutionary learning. To increase stability of the approach a library of parameter presets for the initialization of the function minimization algorithm is used. An overview of the framework is given in Figure 2.

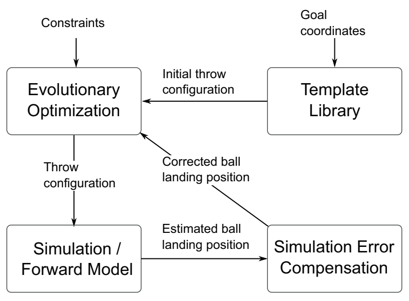

Constraints
Goal coordinates
Initial throw configuration
Evolutionary Optimization
Template Library
Throw configuration
Corrected ball landing position
Estimated ball landing position
Simulation / Forward Model
Simulation Error Compensation

Figure 2. The proposed learning framework.

The ball throwing is performed on a human-scaled anthropomorphic robot, equipped with a five-fingered tendon driven hand which is used to hold the ball and release it while performing the throw movement. The given task is challenging and difficult to simulate precisely. Beside an accurate calibration of the robots kinematics and description of the dynamics of its movements, also the dynamic interaction between the ball and the multi-fingered, inherently compliant hand must be modeled to fully describe the simulation physically. We show that an incomplete simulation in terms of accuracy of the modeled quantities and the fact that there are unmodeled dynamic phenomena can already lead to good results. The inaccuracies of the simulation can be compensated by a function that maps the simulation results to corrected values, so that the task can be successfully fulfilled on real robot hardware.

In Section 2 we show how the robot's throw movements are represented. This forms the *Simulation* or *Forward Model* (Section 3) that is used to estimate the expected position where the ball will land. To find the proper movement to throw the ball at an arbitrary target, we formulate a parameter optimization problem that is described in Section 4. In Section 5 we discuss the shortcomings of the forward model and introduce a simple method to overcome these. We apply a correction function that was learned using supervised learning using data collected from multiple throw performances. We made a number of experiments in order to investigate the problem we are dealing with on the robot, collected training data and evaluated the results from learning the simulation error compensation function. In Section 7 we present these experiments along with their result. A discussion of the results and concluding comments are given in Section 8.

2 Throw configuration

To be able to throw the ball to arbitrary positions, we need the possibility for a computer program to modify the movements the robot should execute. A parametric description of movements provides this possibility and will be discussed in this section. While in Section 4, we will go more into detail on the modification of the throw parameters, here we only want to state that for parameter optimization the number of adjustable parameters plays a crucial role in the optimization performance. For this reason we where looking for a compact parametric representation of joint movement. Also, the joint movement should satisfy some criteria:

- The trajectory is bounded within defined joint limits.
- It should be smooth and non-oscillating.
- It should start and end with zero velocity.

To keep the number of parameters in a reasonable size, we decided to distinguish between joints that move (*active joints*) and joint that keep their position during the throw but are still used for aiming (*inactive joints*). The combination of both build the set of adjustable parameters to make the robot throw at different targets. We call this set of parameters *throw configuration*.

As a representation for joint movements we adopted a concept from the computer graphics community, the Bézier curve that is used to encode a sequence of joint angles. A Bézier curve

$$\mathbf{q}(u) = \sum_{i=0}^{n-1} \mathbf{p}_i B_i(u) \tag{1}$$

is described in terms of a parameter u, that varies in the interval of 0 to 1. The parameter is used to blend a set of control points $\mathbf{p}_i = [p_i, t_i],\ i \in \mathbb{Z} | 0 \leq i < n$ by evaluating n basis functions B dependent on u.

The basis functions are the Bernstein polynomials, a set functions that are all always positive and together add up to 1 at each point in the interval of u. The resultant curve is a spline of degree $n-1$. Using fifth degree Bernstein polynomials as we do in this paper, Equation 1 can be written as

$$\begin{aligned} \mathbf{q}(u) &= [\mathbf{p}_0\ \mathbf{p}_1\ \mathbf{p}_2\ \mathbf{p}_3\ \mathbf{p}_4\ \mathbf{p}_5] \begin{bmatrix} (1-u)^5 \\ 5u(1-u)^4 \\ 10u^2(1-u)^3 \\ 10u^3(1-u)^2 \\ 5u^4(1-u) \\ u^5 \end{bmatrix} \\ &= \mathbf{P} \cdot \mathbf{b} \end{aligned} \tag{2}$$

The resulting curve starts at p_0, ends in p_5 and is always enclosed within the convex hull of the control polygon that is spanned between the control points, so no strong oscillations occur. By constraining the p-coordinate of the control points to the operating range of the corresponding joint, it is ensured that the trajectory is within the boundaries of the joint limits.

It can be shown that the first derivative at the first and last control point is given by

$$\dot{\mathbf{q}}(0) = n * (\mathbf{p}_1 - \mathbf{p}_0) \tag{3}$$

and

$$\dot{\mathbf{q}}(1) = n * (\mathbf{p}_{n-2} - \mathbf{p}_{n-1}). \tag{4}$$

To ensure zero velocity at the start and end of the trajectory we set $p_0 = p_1, p_{n-2} = p_{n-1}, t_0 \neq t_1$ and $t_{n-2} \neq t_{n-1}$ [10]. An example trajectory is shown in Figure 3.

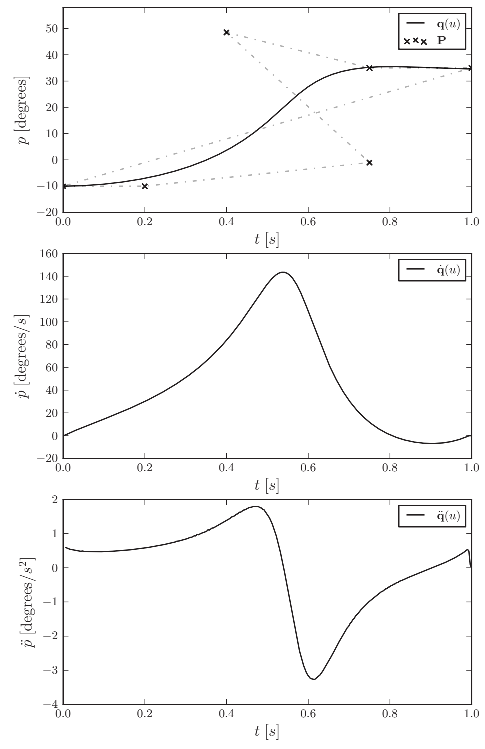


Figure 3. Joint trajectories are modeled as Bézier curves. By modifying the placement of the control points $\mathbf{P}$ the curve changes. The control polygon is shows as plotted line. Also joint velocity and acceleration are shown.

It is worth to mention that the parameter for the basis functions u lies within the range but is not equal to t. To sample the curve in discrete steps along the t-axis (which represents a scaled time axis), we sample $\dot{\mathbf{q}}(u)$ with a rate 4 times higher than the desired control frequency for the trajectory and linearly interpolate for the actual time steps.

3 Simulation / Forward Model

For our ball throwing task it is relevant, how the joint movements of the robot influence the movement of the ball, or stated differently: Given a throw configuration, where will the ball hit the ground. Actually, for throwing inside the bin we need to know when the ball passes a specified height, which is the height of the bin. We call a function that provides this information the forward model.

For a precise simulation, there are factors involved that are difficult to model. It starts with the question how the robot will be able to follow the joint space trajectories, which is dependent on the low-level controllers and mechanical parts like the motor, gears and so on. Also dynamic properties of each link have to be modeled if you want to be precise. In our case, the ball is held by the robot in a robotic hand. Here dynamic properties, involving the friction between hand and ball, as well as the exact shape of the fingers, should be considered. We decided not to try to provide a physical model that is very precise, but rather to use one that captures only the most relevant properties.

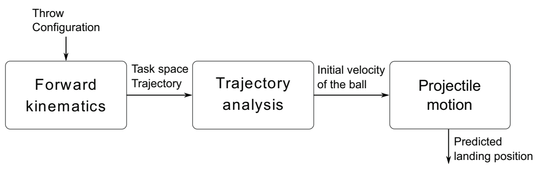


Figure 4. The Simulation / Forward model.

For the prediction where the ball will land we are especially interested in the movement of the robotic hand through space (*task space trajectory*) as the result from the throw configuration. Evaluating the task space trajectory then leads to an estimation of the ball trajectory in an ideal way. Figure 4 shows an overview about our forward model and will be explained in the following paragraphs.

The throw configuration contains fixed joint positions together with joint trajectory parameters that can be evaluated to joint space trajectories using Equation (2). That means all information from a throw configuration can be represented as a temporal sequence of joint configurations, when the fixed joint positions are considered as a trajectory of zero velocity. The task space trajectory results from the joint movements and the kinematic structure that connects the joint. The kinematic structure of the robot can for example be represented using the well-known Denavit-Hartenberg convention. Calculation of the forward kinematics (given the joint configuration at a particular time point) leads to the determination of the configuration of the end effector at this time. By repeating this for every sample from the joint trajectories we get the task space trajectory.

During the execution of the throw movement (i.e. the task space trajectory) we need to release the ball. The moment where the ball gets released should be well timed, so that it results in a nice trajectory of the ball through the air. The trajectory of the ball can be calculated using equations for projectile motion as shown in Equation (5). They depend on the initial velocity of the projectile (in our case the ball, which we assume has the velocity of the end-effector in direction of hand aperture) $\dot{\mathbf{p}}_0 = [\dot{x}_0 \; \dot{y}_0 \; \dot{z}_0]$, the gravity g, and the displacement from the origin frame at the moment of launch $\mathbf{p}_0 = [x_0 \; y_0 \; z_0]$. The origin frame $\mathbf{R}$ is placed centered to the robot on the ground (see Figure 5(b)).

$$\begin{aligned} x(t) &= x_0 + \dot{x}_0 t \\ y(t) &= y_0 + \dot{y}_0 t \\ z(t) &= z_0 + \dot{z}_0 t - 0.5 g t^2 \end{aligned} \tag{5}$$

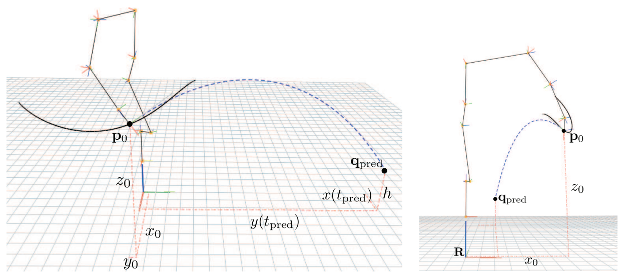
We are interested in the point where the ball passes a specified height h (the height of the bin). So we modify the last term from Equation (5) to include the height of the bin and solve the resulting quadratic equation for t

$$\begin{aligned} 0 &= z_0 - h + \dot{z}_0 t - 0.5 g t^2 \\ t_{\text{pred}} &= \frac{(\sqrt{2gz_0 + \dot{z}_0^2 - 2gh} - \dot{z}_0)}{g}. \end{aligned} \tag{6}$$

Now that we have the *time of flight* t_{pred}, we can insert the value in Equation (5) to calculate the predicted ball landing position

$$\mathbf{q}_{\mathrm{pred}} = \begin{bmatrix} x(t_{\mathrm{pred}}) \\ y(t_{\mathrm{pred}}) \end{bmatrix}. \tag{7}$$

To determine the end-effector velocity at each time point, we differentiate the task space trajectory with respect to time. We look for the point in the end-effector trajectory that has the highest velocity and use this for estimating where the ball will land using the Equations (6) and (7). The ball can leave the hand only in direction of the hand aperture, so the velocity along this vector is considered. In Figure 5(a) and 5(b) screenshots of a visualization of the simulation model are given.



(a) Side view (b) Back view

Figure 5. Two screenshots taken from the simulation for the same throw configuration from different views. The kinematic model is shown as black lines. The projectile trajectory is the dashed line. The end-effector trajectory is shown in thick black curve. We superimposed some of the quantities from the Equations (5 - 7).

It is clear that using this formulation alone as forward model will deliver faulty predictions. But in this paper we claim that an incomplete simulation can already be good enough to accomplish tasks in real world, if the error in the simulation is compensated as we will describe in Section 5.

4 Parameter optimization

In Section 2 we described how movement of the robot is modeled and in Section 3 how an estimation for the resulting ball landing position is calculated given a throw configuration. A throw configuration is a set of adjustable parameters and varying these will result in a different throw movement in terms of dynamics, position and orientation of the hand in task space. This finally results in a different estimated landing position of the ball. Modifying the values in the parametric description allows to aim for different targets to throw at.

Summarized the adjustable parameters $\mathbf{c}$ from a throw configuration are:

- The start and end position of each active joint represented by p_0 and p_5 (cf. Figure 3 and Equation (2)).
- The 2D-positions of the inner control points $p_1, \ldots, p_4$ for each active joint trajectory (also Equation (2)).
- The time for each active joint trajectory, which is the factor we scale the t axis from Figure 3 with.
- The angles of the inactive joints.

The problem to solve is to find the set of parameters, for which the result from the forward model corresponds to a given target position. We formulate this as a parameter optimization problem

$$\operatorname*{argmin}_{\mathbf{c}} f(\mathbf{c}). \tag{8}$$

The predicted ball landing position is given by Equation (7) of the forward model described in Section 3, which is a function $g(\mathbf{c})$ dependent on the throw configuration. To describe the quality of one particular throw configuration, we measure the squared Euclidean distance from the target $\mathbf{d}$:

$$f_{\mathrm{pos}}(\mathbf{c}) = \|g(\mathbf{c}) - \mathbf{d}\|^2 . \tag{9}$$

Working on actual hardware we need to generate feasible trajectories that are executable on the robot. This requirement constraints the joint limits and the maximum velocity per joint, that is limited by the dynamic properties of the robot, its actuators and the low-level controllers. Also we want to enforce a principle characteristic in the throw movement, i.e. we have an acceleration and deceleration phase in the end-effector movement. By constraining the position of the inner control points of the joint space trajectory parameters this can be achieved. To induce these constraints $\mathbf{k}$, we formulate the objective function from Equation (8) as

$$f(\mathbf{c}) = f_{\mathrm{pos}}(\mathbf{c}) + f_{\mathrm{penalty}}(\mathbf{c}, \mathbf{k}, \alpha), \tag{10}$$

where $f_{\mathrm{penalty}}(\mathbf{c}, \mathbf{k}, \alpha)$ calculates the absolute distance of each parameter value to the nearest boundary of the valid range if it exceeds it. For each parameter the constraint violation is weighted by a corresponding factor from the weight vector α.

From the different methods to choose for parameter optimization, we decided to use Covariance Matrix Adaptation Evolution Strategy (CMA-ES) [1]. This method allows for minimization of non-linear functions without the need of a derivative. The algorithm already showed very good results in many different problem domains[4].

5 Simulation error compensation

Due to our rather simple simulation model, the estimation of the throwing distance lacks precision. For example, the point on the trajectory where the ball leaves the hand and the direction are only rough estimations. Friction between the ball and the robotic hand or collisions of the ball with the fingers is not considered at all. Also the kinematic model of the robot is not precisely calibrated and the response behavior of the low-level controllers is not modeled in the simulation.

A classical way would be to try to improve the simulation to model all involved phenomena using physical equations and to establish a complete dynamic simulation of the problem. The resulting software would be much more complex and slower, and also the success of this approach can be questioned, since most simulation software show a discrepancy as compared to real-world system it simulates [6]. Instead we decided to accept that the simulation is not perfect but gives an estimation that only reflects the most important characteristics of the problem.

We wanted to overcome these shortcomings by means of supervised machine learning and decided to train a feed-forward neural network that acts as an error compensation function. As depicted in Figure 6, the neural network maps from the predicted position by

[4] A list of applications of the algorithm can be found on the website of the author of CMA-ES (http://www.lri.fr/ hansen/cmaesintro.html)

the forward model to a corrected position and was trained on data collected in real world experiments (see Section 7 for a detailed description of the experiments).

We used a feed-forward artificial neural network with one hidden layer that consists of 5 nodes activated by tanh functions. The two output nodes are also implemented using tanh activation functions.

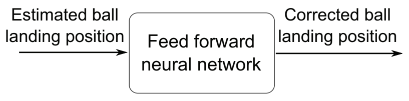

Estimated ball landing position
Feed forward neural network
Corrected ball landing position

Figure 6. Simulation Error Compensation: A feed-forward artificial neural network is used to correct the simulation error.

Learning is performed using the RPROP algorithm [9] to minimize the mean square error of the output of the network activated by the predicted ball landing position from Equation (10) to the measured throwing distances $\mathbf{m} = [m_x \ m_y]$:

$$\underset{\mathbf{x}}{\operatorname{argmin}} \ \frac{1}{2k}\sum_{i=0}^{k-1}\left(n(\mathbf{q}_{\text{pred}}^{(i)}, \mathbf{x})_x - m_x^{(i)}\right)^2 + \left(n(\mathbf{q}_{\text{pred}}^{(i)}, \mathbf{x})_y - m_y^{(i)}\right)^2 \tag{11}$$

Here, $n(\mathbf{q}_{\text{pred}}^{(i)}, \ \mathbf{x})_{x,y}$ denotes the activation level of the two output nodes of the artifical neural network as a function of the estimated throw position for the ith out of k samples, given the weight vector $\mathbf{x}$. As a result the corrected estimation is now given by

$$\mathbf{q}_{\text{corr}} = n(\mathbf{q}_{\text{pred}}, \mathbf{x}_{\min}), \tag{12}$$

where $\mathbf{x}_{\min}$ are the weights for the neural network obtained from training.

6 Template library

Our simulation error compensation function maps from expected ball landing positions to corrected ones. This approach completely neglects special characteristics in the resulting ball trajectory due to distinct parameter constellations in the throw configurations. So the correction function is likely to work only for throw configurations that are very similar to each other.

To improve the results, we decided to not strictly use one source configuration as initial parameter set for the generation of new throw configurations, but created a database from the training data. We then used the configuration that results in the closest ball landing position to the given target (determined by nearest neighbor search) as initial parameter set.

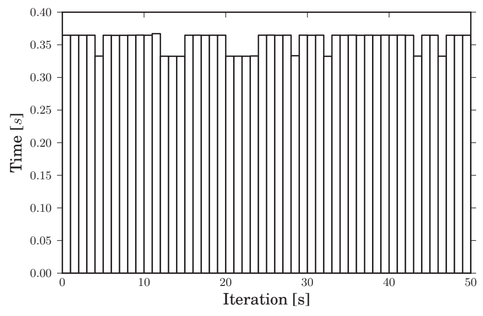
We expected to increase the performance of the algorithm by providing template throw configurations. But it is clear that it does not really solve the problem which arises from the fact that different throw configurations with the same target estimated by the forward model, might actually result in different ball landing positions.

7 Experiments and Results

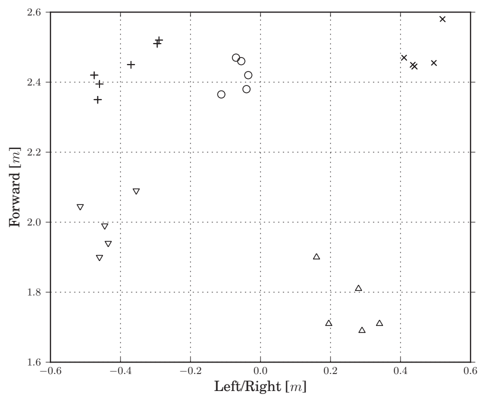
7.1 Initial tests of the robot hardware

As robotic platform to implement the framework, we use a human scaled mobile dual-arm robot named AILA (shown in Figure 1). It consists of a complete anthropomorphic upper body mounted on a mobile platform. The upper body consists of two arms, each of them with seven degrees of freedom, a torso with four joints, and a head with two degrees of freedom. Each torso and arm joint is equipped with position and velocity sensors. To the right arm a five fingered hand is mounted. The hand is underactuated which means that several of the 19 joints are coupled and only actuated by one motor. In total there are 9 active degrees of freedom for the hand. The design of the hand makes it in parts inherently compliant. The head of the robot is equipped with two cameras.

The task requires the robot hand to grasp and hold the ball and release it during the trajectory after accelerating at the time of highest task space velocity. We use predefined hold and release configurations that are position controlled by a PID-controller. To synchronize the hand and body movements, we determined the time for changing from the hold to the release configuration empirically.



(a)



(b)

Figure 7. (a) Difference in opening time of the hand over multiple repetitions. (b) Resulting ball landing position after execution of the same trajectory multiple times. Each symbol represents a different trajectory.

Beside other factors like the capabilities of the low-level controllers in the body and arms of the robot to follow a given trajectory in a similar way over multiple repetitions, the repeatability of the hand opening movement was identified as an important factor to influence the overall ball throwing performance. To evaluate this, we brought the hand into an open configuration, placed the ball and set the hold configuration as reference for the position controllers of the hand joints. Now we set the release configuration as reference and measured the time until the movement is completed using software timers. This was performed 50 times (see Figure 7 (a)). The mean time needed for the trajectory was 0.357s with a standard deviation in the opening time is 0.014s.

Together with unmodeled dynamics inside the system, the repeatability of complete throw performances, measured as the mean standard deviation in the ball landing position is 0.08m. This was found out by performing the resulting trajectory from one throw configuration multiple times. For each trial the position where the ball landed was marked and measured. This resulted in the distributions as shown in Figure 7 (b).

7.2 Visual detection

In this work we did not focus on computer vision algorithms, but found using the measuring tape cumbersome. A simple method to detect the bin visually and estimate its position using the camera in the head of the robot was implemented. We detect the bin by finding and filtering blobs (according to size and position constrains) that result from the difference of the current camera image to a reference image taken earlier in an initialization step. For the reconstruction of the position of the bin we search for the pixel in the detected blob that has the highest y-value, since this is a relatively strong feature that in most poses corresponds to the same point on the rotational invariant bin (eg. the centre point of the frontal arc of the bins bottom). The resulting image coordinates point is reprojected into 3D space and the intersection with a virtual ground plane is calculated that was determined for a fixed pose of the torso and head in a calibration procedure.

In the next sections we will evaluate the overall performance of the system while using the vision system.

7.3 Training the error compensation function

To train the compensation function, first we designed a exemplary throw configuration (source configuration). We defined an operating range (the range of possible targets) of 1-2.5m to the front of the robot and 1m to the left and right. The throwing distance of the source configuration was approximately 2m straight to the front. This source configuration defines the initial parameter set for the parameter optimization that is performed in order to generate a throw configuration for a new target as described in Section 4).

Now, we randomly defined new targets near the predicted distance of the source throw configuration and generated new throw configurations for these targets. We performed the throws for each target several times to find good optimal position to place the bin for this throw configuration. Finally we took the distance for the bin position using the computer vision system. In total data from 17 throws has been collected for training.

To train the feed forward neural network described in Section 5, each weight in the neural network was initialized with a small random value. Using the normalized predicted distances as input and the also normalized measurements from the vision system as reference output, after approximately 500 iterations of RPROP[5] we stopped optimization. Figure 8 shows the training data for the error compensation function (bigger arrows) as well as the correction function sampled in discrete steps. Here we can see that the internal model, as expected, gives a rather high deviance from the measured distances of 0.388m (RMS error) what implies the necessity of the correction function since with such a prediction error it's very unlikely to hit the target. Using the neural network the error is reduced a lot to 0.088m.

[5] For training we used the implementation given by the FANN library: http://leenissen.dk/fann/wp/

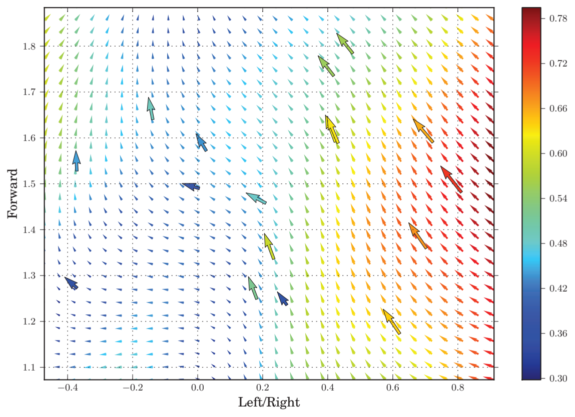


Figure 8. The learned correction function. The tail of the arrows show the predicted position $\mathbf{q}_{\mathrm{pred}}$. The direction points towards the corrected position $\mathbf{q}_{\mathrm{corr}}$ with the color encoding the magnitude of $\mathbf{q}_{\mathrm{corr}} - \mathbf{q}_{\mathrm{pred}}$.

Running the RPROP algorithm for more iterations could further decrease the error, but leads to overfitting to the few data points available what decreases the overall performance which is evaluated below.

7.4 Overall performance evaluation

For evaluating the overall performance, we placed the bin at 5 random positions inside the operating range. The position of the bin was measured using the vision system and for the resulting coordinates a throw configuration was generated with the error compensation enabled. Each throw was then performed 10 times. For every throw it was counted whether the ball landed inside the bin or missed it. Table 1 shows the results of this experiment, resulting in a final success rate of 0.46. The bin that was used in this experiment was cylinder shaped and has a height of 34cm and a diameter of 28cm.

Target	1	2	3	4	5
Success	7	6	1	4	6
Fail	4	4	9	6	4
Success rate	0.64	0.6	0.1	0.4	0.6
Avg. success rate			0.46		

Table 1. Evaluation of the overall performance. First, the bin was detected visually. A throw configuration was generated afterwards and executed multiple times. This was done for five different bin positions. *Success* is the number that indicates how often the ball landed inside the basked and *fail* the number how often it missed.

Table 1 shows a summary of the experiment. The success rate ($\frac{\text{Success}}{\text{Fail+Success}}$) shown there is influenced by the repeatability of executing one particular throw configuration similarly (cf. Section 7 and Figure 7(b)) together with the position error due to the visual bin detection and errors in the simulation error compensation. We can see in Table 1, that for target 3, there was only one successful throw out of 10 trials, what is an indication that the at this point the error compensation failed.

8 Conclusion and Future Work

In this paper we showed an approach to make a anthropomorphic robot throw a ball into a bin that is placed randomly in a region in front of the robot. Instead of completely modeling the problem with an exact physical simulation we proposed a different approach, to use a simple simulation model and compensate the resulting error using machine learning techniques.

To accomplish this we represented parametric joint space trajectories as fifth order polynomials based on Bernstein basis functions. The joint space trajectories are analyzed to identify the task space motion. From the task space motions, based on an incomplete simulation model, an estimation of the position where the ball will land is given. By minimizing a constrained error function, throw movements can be generated, that lead to arbitrary ball landing positions. The deviation between simulation and the real world is compensated using a neural network that maps the estimated landing position of the ball to a corrected one that corresponds to real world positions. Dynamics that are unmodeled in the simulation and calibration errors are compensated by this function. Providing a mapping from estimated to corrected throw positions cascades the underlying parameters to generate the trajectories, but this error compensation only works reliably if the resulting movements are similar. We build a database of parameter configurations that stores example trajectories for distinct target positions. In order to generate similar trajectories, the function minimization is initialized using template configuration that results in a nearby target.

Our solution gives a simple and practical approach to the given problem, but suffers especially from one weakness. Different throw configurations that are predicted by the forward model to have the same ball landing position, but in reality result in different positions cannot be handled by the simulation error compensation in its current form. The correction is solely be made on the predicted target coordinates from the forward model. The underlying parametric configuration of the throw movement is not considered. Instead of learning a correction function, learning the forward model from examples would solve this. Even better would be to learn the inverse model, i.e. a function that directly maps goal coordinates to a proper throw configuration. We want to look at this issue and try different learning methods on this problem. Another possibility is to use reinforcement learning methods, so that the estimation can get gradually better over time with each trial. In [7] reinforcement learning was used to improve an initial control policy given by imitation learning [4] to solve the challenging *ball in a cup* task. One concern with this approach might be, that while with the approach shown in this paper only 17 training samples where it is likely that control policy improvement with reinforcement learning would need more data.

9 Acknowledgements

This work was performed within the iStruct project funded by the German Space Agency (DLR) with federal funds of the Federal Ministry of Economics and Technology (BMWi) in accordance with the parliamentary resolution of the German Parliament, grant no. 50RA1013 and grant no. 50RA1014.

REFERENCES

[1] N. Hansen and A. Ostermeier, ‘Completely derandomized self-adaptation in evolution strategies.’, *Evolutionary computation*, **9**(2), 159–95, (January 2001).

[2] Cedric Hartland and Nicolas Bredeche, ‘Evolutionary Robotics , Anticipation and the Reality Gap’, in *Proceedings of the 2006 IEEE Conference on Robotics and Biomimetics*, pp. 1640–1645, (2006).

[3] T. Hemker, M. Stelzer, O. von Stryk, and H. Sakamoto, ‘Efficient Walking Speed Optimization of a Humanoid Robot’, *The International Journal of Robotics Research*, **28**(2), 303–314, (February 2009).

[4] A. J. Ijspeert, J. Nakanishi, and S. Schaal, ‘Movement imitation with nonlinear dynamical systems in humanoid robots’, in *Proceedings 2002 IEEE International Conference on Robotics and Automation (Cat. No.02CH37292)*, pp. 1398–1403. Ieee, (2002).

[5] Akio Ishiguro, Takeshi Furuhashi, Shigeru Okuma, and Yoshiki Uchikawa, ‘A Neural Network Compensator for Uncertainties of Robotics’, *Industrial Electronics, IEEE Transactions on*, **39**(6), 565–570, (1992).

[6] N. Jakobi, ‘Evolutionary Robotics and the Radical Envelope-of-Noise Hypothesis’, *Adaptive Behavior*, **6**(2), 325–368, (September 1997).

[7] Jens Kober, Betty J. Mohler, and Jan Peters, ‘Imitation and reinforcement learning for motor primitives with perceptual coupling’, in *From Motor Learning to Interaction Learning in Robots*, 209–225, (2010).

[8] Sylvain Koos, Jean-Baptiste Mouret, and Stephane Doncieux, ‘Crossing the Reality Gap in Evolutionary Robotics by Promoting Transferable Controllers’, in *In Proc . of GECCO*, pp. 119–126, (2010).

[9] Martin Riedmiller and Heinrich Braun, ‘A direct adaptive method for faster backpropagation learning: the RPROP algorithm’, in *IEEE International Conference on Neural Networks*, pp. 586–591. Ieee, (1993).

[10] Alan Watt, *3D Computer Graphics*, Pearson Education, Essex, 3. edn., 2000.

Architecture of an Erlang-Based Learning System for Mobile Robot Control

Łukasz Bęben and **Bartłomiej Śnieżyński** and **Wojciech Turek** and **Krzysztof Cetnarowicz** [1]

Abstract. Machine learning methods have proven usability in many complex problems concerning mobile robots control. The integration of a learning algorithm with a mobile robot control systems is not a trivial task. Limited resources of a mobile robots and heterogeneous technologies used for creating robot's controllers causes significant difficulties in applying complex learning methods. In this paper an Erlang-based architecture of a mobile robot control system is described. It creates a virtual environment, which can host a novel, Erlang-Based Distributed Learning Library. The Library can use remote computational resources as well as robots on-board computers to learn from collective experience of several robots.

1 INTRODUCTION

Many contemporary problems concerning mobile robots can be solved using different learning algorithms. The most common learning technique in robot systems is reinforcement learning [13]. It allows to generate robot's policy (mapping between a world state and actions) using feedback from the environment. There are many examples of applications like soccer, prey-pursuing or target observations.

Other learning strategies can be also applied. Good survey on learning from demonstration is presented in [2]. In this approach robot's policy is learned from examples provided by a teacher. Various strategies may be applied to generate the mapping, with most popular reinforcement learning and supervised learning (both regression and classification). Also plan learning can be applied.

Similarly, in multi-agent systems broad range of learning strategies are applied [7, 9]. Here also reinforcement learning seems to be the most popular technique. However, other strategies can be also applied and agents may learn from its experience [6, 10, 1].

Real systems are often too complex for traditional learning techniques [2]. Complexity problem may be solved by learning distribution. Also using supervised learning instead of reinforcement learning can give better results [11]. This leads us to the need of machine learning framework for mobile robot control.

The integration of a learning algorithm with a mobile robot controller is not a straightforward task. Typically robot on-board computer has limited resources and computational power, which reduces learning capabilities. The issue becomes more significant when a group of robots is considered. The robots could collectively gather data and learn using the common knowledge / experience. However, this would require a proper management architecture of a leaning system.

In this paper an Erlang-Based management and learning architecture for multi-robot systems is presented. The architecture is based on the assumptions of an agent-based dual-space control paradigm [12]. The learning infrastructure utilizes a novel Distributed Learning Library created in Erlang technology.

2 ERLANG IN MULTI-ROBOT SYSTEMS

The approach described in this paper is based on a technology, which has been created for completely different applications. In this section a brief overview of the Erlang technology is presented to justify usability of the technology in robot control applications. Later on an architecture of and Erlang-Based Robot Control System is described.

2.1 Erlang Language and Technology

Erlang is a programming language designed by Ericsson in early nineties [3]. It has been developed as a platform for creating concurrent and distributed systems. Fundamental features provided by Erlang technology are: high availability, fault tolerance, concurrency and communication, soft real-time and ease of maintenance.

Erlang is a a functional programming language, which is compiled to a byte code and executed in a virtual machine. The Erlang Virtual Machine (EVM) can be run at almost every modern computer as well as on on embedded devices or mobile phones. This feature is crucial for using Erlang in robot control.

Several EVMs (called Erlang nodes) can connect with each other creating a distributed runtime environment. Applications working in a distributed Erlang machine can almost transparently use all available hardware and communicate using messages.

Erlang platform creates a distributed environment which does not have a single point of failure. It is achieved by equating all nodes and creating complete network of connections between nodes. This feature (illustrated in Figure 1) is crucial when control of robot groups is considered.

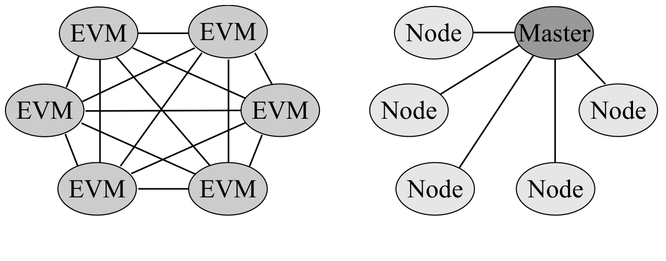


Figure 1. Distributed virtual machine architecture. On the left: Erlang approach with the complete network of connections; on the right: approach with a selected master node.

[1] AGH University of Science and Technology, Krakow, Poland

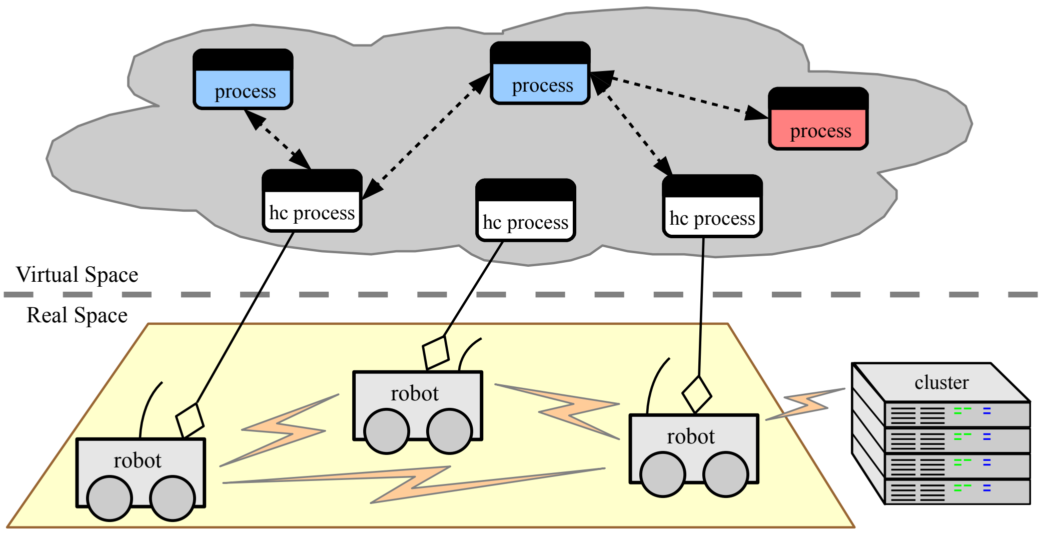


Figure 2. Logical separation of Erlang virtual machine and hardware hosting Erland nodes. The hardware control processes (hc) are executed on robots on-board computers. Other processes can be executed on any node of the EVM.

Failure of one or a few robots (one or a few nodes) cannot cause failure of the whole platform. Moreover, in this approach messages routing is much faster due to direct connections available.

Each Erlang program is composed of processes, which never share memory. Processes communicate by sending and receiving asynchronous messages. Erlang does not use threads of the underlying operating system to implement concurrency. It implements internal light-weight processes which can be run in millions on a single computer.

Another useful feature of EVM is the mechanism of hot code loading. Source code can be changed on runtime, without stopping the application. Even the internal state of a process can be preserved during code modification.

It is worth mentioning, that the technology is not experimental. It has proven its usability in many industrial applications. Several large corporations use Erlang-based solutions for providing, large scale, high-availability services.

The features of Erlang technology seem suitable for creating robot control programs and systems for multi-robot group management.

2.2 Architecture of an Erlang-Based Robot Control System

The assumptions of Erlang technology are very similar to the assumptions of the software agent paradigm. Both are based on the principle of autonomous, proactive pieces of software, which can communicate using message passing concurrency. Therefore basic ideas used in agent-based architectures for robot management can be adopted for Erlang-based robot control systems.

Agent-based architectures has proven suitability for this type of applications. Very important non-functional features of control systems can be easily provided by proper use of this paradigm. An example of architecture designed for providing scalability, extensibility and durability has been described in [12]. The dual-space approach defines a virtual space for agents, which use hardware robots as tools. This separation makes it possible to separate various functions of the system in different, autonomous agents.

The biggest problem with real-world applications of agent systems was caused by a technology used for creating software agent platforms. The platforms are typically implemented in imperative languages using shared-memory concurrency. This creates significant limitations in number of agents and messaging performance. Use of Erlang technology could help overcoming this issue.

Erlang can create a virtual machine interconnecting several computers. Connected computers can be heterogeneous, therefore one platform can include hardware robot controllers and high performance servers. The basic architecture of an Erlang-based robot control system is shown in Figure 2.

The architecture assumes, that each robot can host a node of the Erlang virtual machine. Basic hardware operations, like reading data from sensors or motor control are executed on this computer by dedicated hardware control processes. The processes responsible for controlling sensors implement publish-subscribe design pattern. This allows several other processes to be automatically notified about new sensor readings.

The processes, which perform complex computations or access remote resources (large data structures, databases), can be executed on remote computers. The can easily access all other processes and simultaneously make use of efficient hardware.

This approach can be very useful in terms of machine learning. A system can use several robots to collect data in the real environment and execute learning algorithms on available servers. This approach can make good use of a parallelized learning algorithms, which will be described in next section.

3 LEARNING SYSTEM IN ERLANG

Erlang Machine Learning Library (EMLL) provides machine learning and data mining algorithms in Erlang. The library is designed to take an advantage of distributed processing in order to process large datasets in parallel manner.

Since robot agents and the library are both implemented using Erlang, the library can easily use both computer cluster or robots computing resources as computing nodes. In other words, robots, which are collecting the data, can use powerful cluster of computers to learn from the gathered data. Robots can also cooperate and share their computational resources to process the data concurrently (see fig. 4). The library can process well known machine learning formats: Arff,

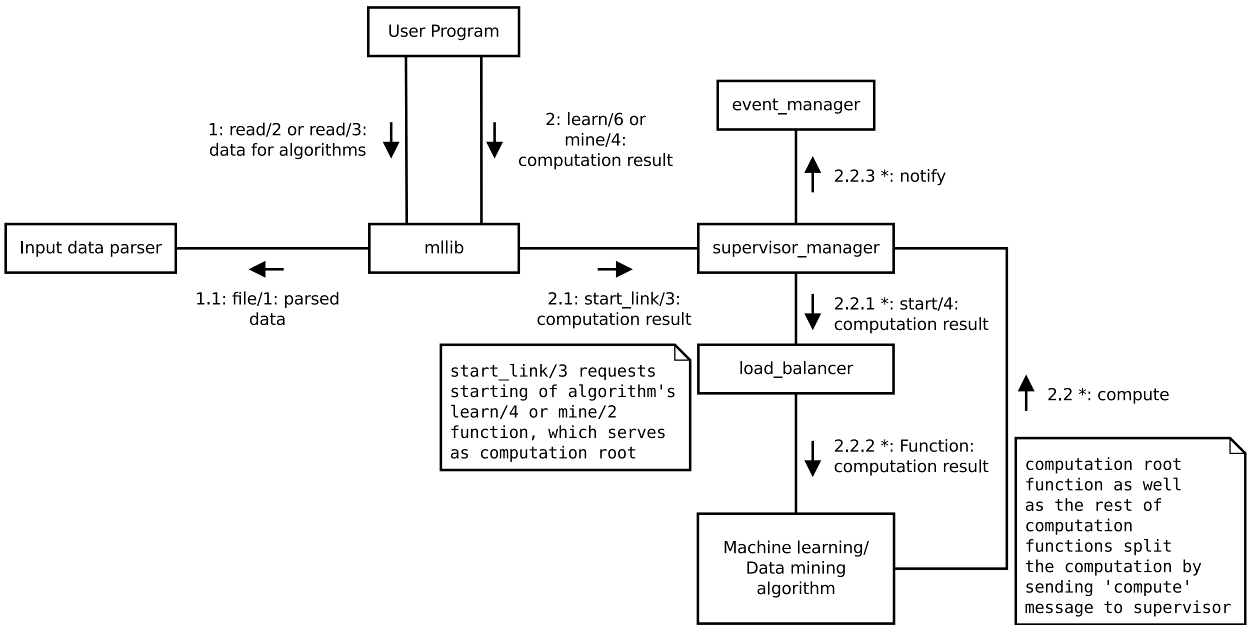

User Program
1: read/2 or read/3: data for algorithms
2: learn/6 or mine/4: computation result
event_manager
2.2.3 *: notify
Input data parser
mllib
supervisor_manager
1.1: file/1: parsed data
2.1: start_link/3: computation result
2.2.1 *: start/4: computation result
start_link/3 requests starting of algorithm's learn/4 or mine/2 function, which serves as computation root
load_balancer
2.2 *: compute
2.2.2 *: Function: computation result
computation root function as well as the rest of computation functions split the computation by sending 'compute' message to supervisor
Machine learning/ Data mining algorithm

Figure 3. The Machine Learning Library - communication diagram.

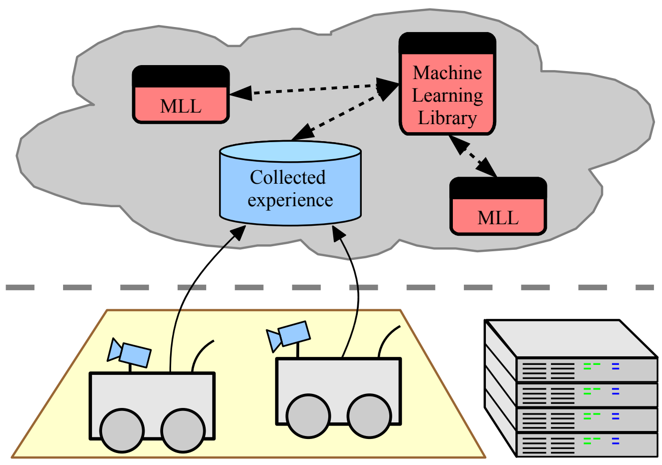


Figure 4. Erlang Machine Learning Library can use robots' resources, a computer cluster or both as computation nodes.

C4.5. EMLL can also perform some basic data preprocessing like discretization.

The library uses a central administering process called Supervisor Manager to handle the computations. Supervisor Manager is spawned at the beginning of the computation. Supervisor Manager is responsible for managing computation requests, putting results altogether and sending them back to the requesting node. It is also responsible for monitoring remote nodes. If any node goes down during computations, Supervisor Manager can redirect computations to other node and repeat lost computations. As Supervisor Manager receives a computation request, it passes the request to the Load Balancer running in separate process and a computation is spawned on the node chosen by the Load Balancer so as to optimize available resources usage. Spawned task is identifiable and Supervisor Manager can pass additional messages to the task. Tasks can send computation requests to the Supervisor Manager in a way similar to spawning local threads. The data is stored on nodes in ETS tables (O(1) access synchronized hash tables), which means that data can be sent to each node only once and then be used by many computing threads spawned on the node. Since Erlang is not suitable for massive computations, most computationally demanding parts of algorithms can be implemented in other technology (for example in C) and easily integrated. Supervisor Manager can also manage events produced during computations. Events are used mainly to profile computations.

Figure 3 presents a communication diagram which shows the actual computation flow. A user program interacts mainly with "mllib" module. The user program can collect data from file (1) and a specific format parser is used (1.1). User program can then invoke a specified supervised learning or a data mining algorithm (2). "Mllib" module spawns a Supervisor Manager (2.1), which creates Event Manager and Load Balancer processes. At this point, all specified computations nodes are health checked and all input data is sent to ETS tables on each node(if requested). First computation task creation request is sent to the Load Balancer (2.2, 2.2.1) which spawns a new computation node (2.2.2). When an algorithm can split its computations into parallel tasks, it sends a "compute" request to the Supervisor Manager (2.2) and a new computation task is created using the Load Balancer (2.2.1, 2.2.2). When the computation task finishes, an event containing that task's execution time is generated, and a result is sent back to the requesting node (2.2.2). When the last root computation task finishes, a result is returned to the User Program.

Currently, the library contains two supervised learning algorithm implementations: C4.5, Naïve Bayes and a Data Mining algorithm: Apriori. C4.5 was originally proposed by Robert Quinlan [8]. It uses decision trees to represent the classifier. The tree is built using top-down recursive approach. Naïve Bayes is a basic algorithm, that uses the probability theory and Bayes theorem to classify data. The Data

Mining Apriori algorithm is able to produce both Association rules describing the data or frequent sequences that can be found in the dataset.

4 EXPERIMENTS AND RESULTS

We have performed several experiments in order to test the Machine Learning Library's capabilities. We used datasets from a popular datasets repository - UCI Machine Learning Repository [4]. Below we present learning results for Wall-Following Robot Navigation Data Set [5]. The task associated with the dataset is a classification. The goal is to learn a robot's move that should be performed in order to avoid collisions with walls. The decision is to be made based on readings from 24 ultrasound sensors. The whole dataset consists of about 5500 labelled examples. The actual available moves and class distributions are presented in Table 1.

Move	Samples in dataset
Move-Forward	2205 samples (40.41%)
Slight-Right-Turn	826 samples (15.13%)
Sharp-Right-Turn	2097 samples (38.43%)
Slight-Left-Turn	328 samples (6.01%)

Table 1. Characteristics of Wall-Following Robot Navigation Data Set

The dataset was split in a way, that 60% of samples from each class' category were included in a training set, and the other 40% formed a test set. C4.5 algorithm was used in a supervised learning process. The classifier performed well, assigning the correct category to 97.5% of samples from the test set.

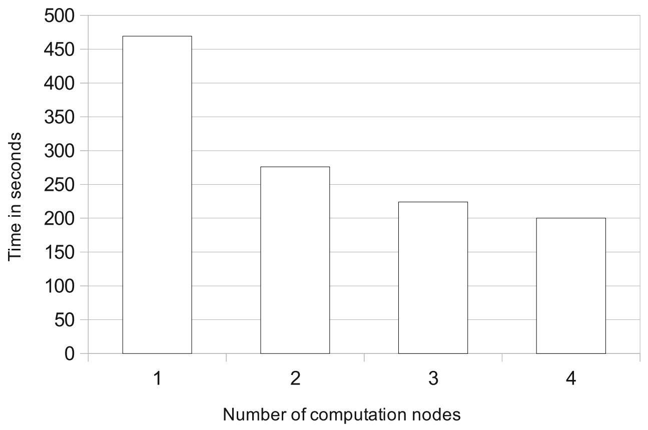


Figure 5. Learning time for 1-4 computing nodes

Learning was performed on a single 2-Core 4-Threads machine using a couple of Erlang virtual machines. Each virtual machine's resources were restricted to one system thread only. Machines were communicating using the network. Learning task was repeated using a different number of computing nodes. Results are presented in Figure 4.

We can observe a significant reduction of execution times as the number of computing nodes increases. At the beginning (when two nodes are used instead of one), the calculation time reduction is very big. At the end, the time reduction is minor because the costs of transmitting data exceed paralelization benefits. It should be noted that the overall execution time is high, because in a tested version all the computations were performed in Erlang. However, our goal was to test correctness and paralelization speed-up and the overall speed was not crucial.

5 CONCLUSIONS AND FURTHER WORK

The Erlang technology is a suitable solution for creating complex robot management systems. It provides distributed processing and efficient message passing concurrency.

The Erlang Machine Learning library described in this paper seems a promising solution for implementing learning algorithms for mobile robot groups. It can integrate experience collected by the group and parallelize learning process, utilizing robots on-board computers and available servers.

Further research will focus on development of an experimental hardware platform. The group of wheeled rovers will be used for evaluating real-life learning scenarios. Another task is to re-implement computationally expensive parts of algorithms in a low level language, like C, which can be easily integrated with Erlang virtual machine.

ACKNOWLEDGMENT

The research presented in this paper has received founding from Polish National Science Centre under the grant no. UMO-2011/01/D/ST6/06146.

REFERENCES

[1] Stéphane Airiau, Lin Padham, Sebastian Sardina, and Sandip Sen, 'Incorporating learning in bdi agents', in *Proceedings of the ALA-MAS+ALAg Workshop*, (May 2008).
[2] Brenna D. Argall, Sonia Chernova, Manuela Veloso, and Brett Browning, 'A survey of robot learning from demonstration', *Robotics and Autonomous Systems*, **57**(5), 469 – 483, (2009).
[3] Francesca Cesarini and Simon Thompson, *Erlang Programming - A Concurrent Approach to Software Development*, O'Reilly, 2009.
[4] A. Frank and A. Asuncion. UCI machine learning repository, 2012.
[5] Ananda Freire, Marcus Veloso, and Guilherme Barreto. Wall-following robot navigation data data set, 2012.
[6] Dimitar Kazakov and Daniel Kudenko, 'Machine learning and inductive logic programming for multi-agent systems', in *Multi-Agent Systems and Applications*, pp. 246–270. Springer, (2001).
[7] Liviu Panait and Sean Luke, 'Cooperative multi-agent learning: The state of the art', *Autonomous Agents and Multi-Agent Systems*, **11**, 2005, (2005).
[8] J.R. Quinlan, *C4.5: Programs for Machine Learning*, Morgan Kaufmann, 1993.
[9] S. Sen and G. Weiss, *Learning in multiagent systems*, 259–298, MIT Press, Cambridge, MA, USA, 1999.
[10] B. Śnieżyński, 'Agent strategy generation by rule induction', *In Press at Computing and Informatics*, **32**, (2013).
[11] B. Śnieżyński and J. Dajda, 'Comparison of strategy learning methods in farmer-pest problem for various complexity environments without delays', *In Press at Journal of Computational Science*, (2012).
[12] Wojciech Turek, 'Extensible multi-robot system', in *Proceedings of the 8th international conference on Computational Science, Part III*, ICCS '08, pp. 574–583, Berlin, Heidelberg, (2008). Springer-Verlag.
[13] Erfu Yang and Dongbing Gu, 'Multiagent reinforcement learning for multi-robot systems: A survey', Technical report, Department of Computer Science, University of Essex, (2004).

5th International Workshop on Evolutionary and Reinforcement Learning for Autonomous Robot Systems (ERLARS 2012)
Montpellier, France, August 28 2012

Task Variability in Autonomous Robots: Offline Learning for Online Performance

Majd Hawasly and **Subramanian Ramamoorthy**[1]

Abstract. A problem faced by autonomous robots is that of achieving quick, efficient operation in unseen variations of their tasks after experiencing a subset of these variations sampled offline at training time. We model the task variability in terms of a family of MDPs differing in transition dynamics and reward processes. In the case when it is not possible to experiment in the new world, e.g., in real-time situations, a policy for novel instances may be defined by averaging over the policies of the offline instances. This would be suboptimal in the general case, and for this we propose an alternate model that draws on the methodology of hierarchical reinforcement learning, wherein we learn partial policies for partial goals (subtasks) in the offline MDPs, in the form of options, and we treat solving a novel MDP as one of sequential composition of partial policies. Our procedure utilises a modified version of option interruption for control switching where the interruption signal is acquired from offline experience. We also show that desirable performance advantages can be attained in situations where the task can be decomposed into concurrent subtasks, allowing us to devise an alternate control structure that emphasises flexible switching and concurrent use of policy fragments. We demonstrate the utility of these ideas using example grid-world domains with variability in task.

1 Introduction

In this paper we address the autonomous agent's problem of performing a specific task in a world drawn from a family of related worlds with *no opportunity to experiment afresh*. Modelling these as MDPs, we consider that all the processes have the same state-action space, but differ in dynamics and/or reward processes. The agent, typically a robot with limited knowledge of context, may have no knowledge of the *type* of the new MDP; but it is asked to perform the task *as well as possible* and in *real-time*, without further experimentation in this new world.

1.1 Motivation

A robot is considered autonomous when it is capable of achieving relatively sophisticated tasks in changing worlds and over an extended deployment time. A characteristic of these changing worlds is that they are arbitrarily rich and may continuously change. In practice, the robot has only limited time in any newly-assigned task to act efficiently. This *real-time* requirement is due to the change in the world or the expiry of the task.

Consider, as an example, the case of a self-driving car in an urban area. Even though the task is structured and probably well known, the vehicle has to interact with the unmodelled dynamics emerging from the existence of other vehicles and pedestrians. The agent would have been trained in many different situations, but given the size and the richness of the problem, no specific instance would ever happen twice. The vehicle should achieve its task, of navigating toward a goal location, in real-time under the possibly unseen dynamics, and with no chance to repeat the same interaction again as learning algorithms usually require. The issue is more pronounced when the task is inherently unstructured, like the navigation problem for field robotics, or in the domain of disaster response.

1.2 Rationale

The question we are tackling then is: *how should the* offline *experience be utilised to enable the robot to survive the* online*, real-time task?* Here, utilisation is interpreted as determining which behaviours, representations and control structures should be learnt and how they should be used afterwards.

A standard reinforcement learning approach to deal with the variability might be to treat it as uncertainty generated by a big, latent stochastic process. That is, to consider the family of MDPs as one giant MDP. Then, a policy may be learnt for that stochastic process if given enough time and experience. This form of learning *across* the set of MDPs experienced in the offline phase would yield a suboptimal *averaging policy*.

In the control theory literature, where the goal is typically to handle disturbances and dynamics abnormalities by devising large basins of attraction, it is known that it is very hard to find suitably robust large-basin controllers for many complex but realistic systems. So, a collection of controllers is devised instead, with each controller specialised to stabilise a certain context, which then can be sequenced in a way that achieves the desired robustness in the complete task [3, 21]. Composing controllers in robotics has been an issue of interest recently, and, besides control, the question of reasoning about generic robot capabilities to generate viable plans that adhere to task specification has been investigated, e.g., using symbolic reasoning [2].

Here, we take a similar approach but posing the question using reinforcement learning. In [3], Burridge et al. employ a backchaining mechanism on a set of hand-designed feedback controllers with overlapping domains of attraction and goal sets, creating a hierarchy of 'funnels', to produce robust trajectories that lead to the goal starting from a large applicability domain (specifically, the union of individual controllers' domains of attraction). Here, we learn a set of policy fragments from the offline instances, we organise these policies in an appropriate control hierarchy, and we employ a switching paradigm that promotes reactivity to the environment 'feedback', ex-

[1] School of Informatics, The University of Edinburgh, 10 Crichton Street, Edinburgh, United Kingdom, EH8 9AB, email: m.hawasly@sms.ed.ac.uk

tracted from the changes caused by the unmodelled dynamics.

1.3 Approach

Reinforcement Learning (RL) is the classical technique for learning from online interaction [18] giving automatic guarantees for stationary environments, but careful thought and design effort are needed to make it work properly in situations with levels of change that cannot be described as slight parameter drifts. Besides, RL methods require many training episodes to achieve good performance in any task, which is problematic with our real-time 'one-shot' requirement. Model-based methods can achieve acceptable performance faster than model-free methods but need a good model of the environment, which is a 'moving target' that we assume we cannot identify fully. Hence, our focus is on the problem of structuring the RL problem so as to exploit the structure in the world and the task to achieve the stated requirements.

We propose to decompose the task into a collection of subtasks, then learn policies for these subtasks from the offline instances. We claim that factoring the variability into these components is beneficial to the quality of the produced policies. After that, we learn to compose these subtasks for novel instances, producing a policy that takes into account the new instance through an indirect feedback mechanism.

We argue that decomposing the task into subtasks, in the spirit of hierarchical reinforcement learning [1], and learning component-wise policies from the experienced MDPs may be beneficial to online performance. The intuition comes from the benefits of decomposition as a standard approach to learning in intricately complex stochastic systems, such as the process that generates all the variation in our problem, by factoring the variation to multiple subtasks and how they are put together, reducing the blurring effect induced by policy averaging. Our proposed method applies to any domain where the task has internal structure that supports such a decomposition, and many practical domains of interests satisfy this requirement.

We require that the robot has sufficient offline training time to learn in a few samples of the MDP family. In practice, this needs not be a separate phase, but all the experience accumulated in past interactions can be included to handle the new instance (cf. lifelong learning [22]). In the training phase, the robot may develop a set of *capabilities* - generic, reusable controllers that target relevant subtasks across the family of experienced worlds. To interpret these in the language of Hierarchical Reinforcement Learning (HRL), one may consider them to be options [19]: temporally-abstracted actions. If the task supports a set of variation-persistent subtasks, we can utilise the offline phase to develop a hierarchical model describing the task. A key aspect of our proposed approach is that through such a decomposition, a hierarchical model of offline-developed capabilities might outperform the alternative of an averaging policy learnt across the set of unstructured MDPs. In particular, we will show that this enables the agent to 'jump start' in terms of performance in a novel instance, without having to learn afresh.

A caveat that must be associated with any usage of hierarchical models in RL is that the best policy that can be achieved, in the general case, may be suboptimal. The notion of optimality in HRL is known as *hierarchical optimality* which refers to the goodness of policies in the hierarchy-induced policy subspace [5]. Nonetheless, the true optimality can be approached by modifications to the hierarchy or its induced control [7, 5].

One such modification in the options framework is known as interrupting options [19], which is a means that plays on option termination conditions to improve global performance in a specific MDP. The termination condition of an option is not restricted anymore to reaching a terminal state, but also in cases when a better option at some intermediate step can be invoked. This would loosen the constraints on the policy space, allowing policies that are chains of subtrajectories rather than chains of complete option-generated trajectories. To generate the interruption signal, the knowledge of all option values at all states in the specific MDP is usually assumed. This requirement is strong if the world is not known *a priori*. We propose that this can be relaxed by using values that are not immediately from the current instance, but rather statistics from the offline MDPs.

Another improvement can come from the hierarchical decomposition itself. Usually, subtasks are chosen to represent different objectives that the agent may need to achieve while seeking its goal. This decomposition does not often produce truly 'orthogonal' subtasks. The resulting policies, in many cases, share the state-action space and may have overlapping or non-compatible reward signals. Nonetheless, the hierarchical model eventually handles them as if they are truly independent, just as the standard RL framework handles primitive actions. To tackle that, we propose to include in the hierarchy, along with the original subtasks, a collection of *composed subtasks*: policies that achieve the goals of some subtasks concurrently, optimising their overlap.

In this paper, we will use the options framework to build a hierarchy to organise a set of policies (and compositions) learnt from extended offline experience, and plan with a reactive interruption mechanism allowing flexible sequencing in response to changes in the environment. Also, we relax the decomposition boundaries induced by the hierarchy by learning to achieve multiple subtasks at once for subtasks that support concurrency, and propose an alternate control hierarchy around that. We demonstrate these techniques using two gridworld tasks: a navigation task with changing, unpredictable wind, and a resource gathering task with partial observability and adversaries.

2 Setup

2.1 Markov decision process

We assume that the task of the robot can be modelled as a discrete-time Markov decision process (MDP). An MDP m is the tuple (S, A, T, R), where S is a finite state space, A is a finite action space, $T : S \times A \times S \to [0, 1]$ is the (stationary) dynamics of the world, and $R : S \times A \times S \to \mathbf{R}$ is the (stationary) reward process that encodes the goal of the task.

A (Markov) policy for an MDP is a (stochastic) mapping from states to actions, $\pi : S \times A \to [0, 1]$, and the optimal policy π^* is the policy that maximises expected cumulative reward. The cumulative reward is summarised using a state-action value function: $Q^\pi(s, a) = \mathbf{E}^\pi\{r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \ldots | s, a\}$ for the future rewards $\{r_t\}$ and the discounting factor γ. The optimal action value function is $Q^*(s, a) = \max_\pi Q^\pi(s, a)$ for all pairs (s, a).

2.2 Family of MDPs

We model the variability in a task using a family of related Markov decision processes $\mathcal{M}$ with the same goal. The family shares the state-action space $S \times A$, but the dynamics $T_i : S \times A \times S \to [0, 1]$ and the reward process $R_i : S \times A \times S \to \mathbf{R}$ may be different for each MDP in the family $m_i \in \mathcal{M}$. We assume that the variability model, formally defined as $\{\mathcal{T}, \mathcal{R}\}$ where $T_i = \mathcal{T}(i)$ and

$R_i = \mathcal{R}(i)$, to be *unknown* to the agent. The agent 'samples' from the variability model during the offline learning phase.

2.3 Semi-Markov decision process

In a semi-Markov decision process (SMDP), actions are allowed to extend over multiple time steps. This makes reasoning about time explicit in learning and planning. This kind of process emerges when dealing with temporally extended actions in MDPs, e.g. in hierarchical reinforcement learning.

2.4 Options and Interruption

The framework of options is one approach to hierarchical reinforcement learning [1, 19] using temporally-extended actions. An option is the three-tuple $\langle I, \pi, \beta \rangle$: $I \subseteq S$ is the initiation state set where the agent is allowed to invoke the option, π is a (Markov or semi-Markov) policy which be followed when the option is invoked, and β is a termination probability distribution over the state space, which encodes stochastically the termination condition of the option. Options are flexible objects that generalise primitive actions which can be considered as (trivial) 1-step options for planning purposes.

For a set of options O, an option switching policy $\mu : S \times O \rightarrow [0, 1]$ is a (stochastic) map from states to options. It is proven that sequencing a set of (Markov or semi-Markov) options from O defined over an MDP gives a well-defined semi-Markov decision process (SMDP), allowing planning and learning of μ via similar approaches to planning and learning in MDPs [19]. Q^μ is the option value function for μ.

Normally, an option selected by μ at some state continues to run until its termination condition is satisfied. On the other hand, option interruption is the process of switching control from the running option before its normal termination if its value at the current state s_t is inferior to the expected value of the policy at s_t: $Q^\mu(s_t, o) < \sum_{q \in O} \mu(s_t, q) Q^\mu(s_t, q)$. This is a kind of non-hierarchical execution in HRL, and it allows for performance improvement over the SMDP policy.

3 Handling task variability

3.1 Averaging policy

For a set of sampled MDPs $\mathcal{M}' \subseteq \mathcal{M}$, the *mean MDP* $\bar{m}$ is the process that has the same state-action space $S \times A$ as the members of $\mathcal{M}$ but has the dynamics and reward processes $\{\bar{T}, \bar{R}\} = \mathbf{E}_{\mathcal{M}'}\{\mathcal{T}, \mathcal{R}\}$.

The *averaging policy* $\bar{\pi}$ is the optimal policy for the mean MDP $\bar{m}$. Note that the value function of this policy averages sample returns generated from the models of sampled members in the family. If the variability in $\mathcal{M}$ is extensive, the performance of $\bar{\pi}$ would be poor in general. Intuitively, This is due to the policy trying to choose actions that suit the full spectrum of dynamics and rewards, and thus ending up with actions that are conservative for many of the task instances. Note that the specific averaging policy that will emerge from training on some $\mathcal{M}'$ will depend on the nature of the sampled MDPs as well as the sampled trajectories in them.

3.2 Capabilities: offline options

A *capability* is a skill that targets a specific subtask that occur frequently in the worlds that an agent has experienced. We model that subtask as an MDP that possibly lives in a subspace of $\mathcal{M}$'s state-action space, and which has a localised dynamics and a special reward function.

A capability can be described as an option $\langle I, \pi, \beta \rangle$, with I and β specifying the subtask (where the capability is viable, and when to stop it), and π being the policy learnt for the subtask across the set of offline instances that have been experienced. For this, we call a capability an *offline option*.

Because it is a smaller problem, we argue, as we show later in the experiments, that a capability will be less affected by variability compared to the full task. Intuitively, the initiation set $I \subseteq S$ and the policy π will limit the generated trajectories to a subset of all trajectories, which will reduce the dynamics variability effect on the option. Also, the capability's reward targets, by definition, a more persistent component of the complete task reward, making it less susceptible to reward variability.

Defining capabilities as options allows us to interpret the capability switching strategy as as an option switching policy μ that can be learnt over the SMDP induced from the set of offline options. Remember that in our case, however, the agent is unable to learn the switching policy μ for the online instance directly, due to the real-time requirement. If an averaging switching policy $\bar{\mu}$ is to be learnt to achieve the task from offline experience, this would only produce a hierarchically-optimal policy for the mean MDP, which will be suboptimal to the averaging policy $\bar{\pi}$ in the general case. To improve on that, we propose to incorporate a notion of option interruption into the process. This is described in the next section.

3.3 Offline interruption

Employing interruption in the options framework not only improves performance through non-hierarchical execution of hierarchical policies, but also adds an element of 'reactivity' in the control structure by making it sensitive to the state of interaction. This appears to be useful for handling unknown situations, but the interruption condition in the original concept [19] requires knowledge of all option values in the desired instance, which we do not have. We propose a modified notion of interruption that depends on values extracted form the offline experience.

Definition 1 (Offline interruption)**.** *A running option o in the unknown MDP m may be* offline-interrupted *at state s_t if the maximum value of o at that state in all instances under the averaging option policy, $\hat{Q}^{\bar{\mu}}(s_t, o) = \max_{m \in \mathcal{M}} Q^{m,\bar{\mu}}(s_t, o)$, is strictly less than the averaging value of interrupting o and selecting a new option according to the policy $\bar{\mu}$: $\hat{Q}^{\bar{\mu}}(s_t, o) < V^{\bar{\mu}}(s_t) = \sum_q \bar{\mu}(s_t, q) Q^{\bar{\mu}}(s_t, q)$.*

The intuition is that the choices of $\bar{\mu}$ would be conservative and 'safe' across the set of experienced MDPs, and following $\bar{\mu}$ would be suboptimal in general. When the best seen value of the running option goes below that stable safety threshold upon reaching some state, it is reasonable to follow the safe choice. Because the agent only observes a subset of all instances $\mathcal{M}' \subseteq \mathcal{M}$, we estimate the maximum value function from the seen instances: $\hat{Q}^{\bar{\mu}}(s_t, o) \approx \max_{m \in \mathcal{M}'} Q^{m,\bar{\mu}}(s_t, o)$. Then, there implicitly lies an assumption that the real value of the option at the current instance would not be higher than what have been seen so far in the offline instances.

Following the original theorem of option interruption [19], we give a condition for offline interruption soundness in the next theorem.

Theorem 1 (Offline interruption)**.** *For a family of MDPs $\mathcal{M}$, a set of options O defined over the family, and an averaging switching* *policy $\bar{\mu} : S \times O \to [0,1]$, define a new set of option O' with one-to-one mapping between the two option sets except for termination conditions which are defined as follows: $\beta = \beta'$ for all states but the ones in which $\hat{Q}^{\bar{\mu}}(s,o) < V^{\bar{\mu}}(s)$ – that is, the maximum value at s for an option o is less than the expected value of that state under the averaging policy – then we* may *make the termination condition $\beta'(s) = 1$. Let $\bar{\mu}'$ be the policy over O', $\bar{\mu} = \bar{\mu}'$. If the averaging policy is* pessimistic *with respect to values in an instance $m \in \mathcal{M}$, then $\bar{\mu}'$ is no worse than $\bar{\mu}$: $V^{m,\bar{\mu}'}(s) \geq V^{m,\bar{\mu}}(s)$ for all $s \in S$.*

Proof. We follow the proof in [19]. In some arbitrary MDP m, for $V^{m,\bar{\mu}'}(s) \geq V^{m,\bar{\mu}}(s)$ to be true for arbitrary s, it is enough to show that following μ' from s then continuing with μ is no worse than following μ all the time. So, it is enough to show that: $r_s^{o'} + \sum_{s'} p_{ss'}^{o'} V^{m,\bar{\mu}}(s') \geq r_s^o + \sum_{s'} p_{ss'}^o V^{m,\bar{\mu}}(s')$. The two sides are equal for any history ss' that is not interrupted (because the policies are the same in that case), therefore we shall only consider the expected value under interrupted histories. That is, we would like to prove that: $E\{r + \gamma^k V^{m,\bar{\mu}}(s')\} \geq E\{\beta(s')[r + \gamma^k V^{m,\bar{\mu}}(s')] + (1-\beta(s'))[r + \gamma^k Q^{m,\bar{\mu}}(s',o)]\}$, for the interrupted histories ss' under o' followed by o. This is true if $Q^{m,\bar{\mu}}(s',o) \leq V^{m,\bar{\mu}}(s')$, i.e. the return from the interrupted option o at interruption state s' *in the specific instance* m is less than the return of $\bar{\mu}$ in m.

We know that $Q^{m,\bar{\mu}}(s',o) \leq \hat{Q}^{\bar{\mu}}(s',o)$ (by the definition of $\hat{Q}^{\bar{\mu}}$), and that $\hat{Q}^{\bar{\mu}}(s',o) < V^{\bar{\mu}}(s')$ for all interruption states s' (by the definition of offline interruption). For that, the proof will be completed when $V^{\bar{\mu}}(s') \leq V^{m,\bar{\mu}}(s'), \forall s'$, i.e. the averaging policy pessimistically underestimates the values of the interruption states in the instance m. □

Although the premise would not be true always for any instance $m \in \mathcal{M}$, the method is still able to produce good results empirically. Algorithm 1 gives a simple procedure for decision making with offline options and offline interruption.

Algorithm 1 Decision making with Offline Interruption

```
Require: $O$: option set; $\bar{\mu}$: averaging option policy; $Q^{\bar{\mu}}$: value
    function of $\bar{\mu}$ over $O$; $\hat{Q}^{\bar{\mu}}$: maximum values of the option set
    $O$ under $\bar{\mu}$; $s_t$: current state.
 1: $o_{run} \leftarrow \phi$.
 2: for every time step $t$ do
 3:   if $o_{run}$ is empty then
 4:     $o_{run} \leftarrow \arg\max_{o \in \mathcal{O}} Q^{\bar{\mu}}(s_t, o)$.
 5:   else
 6:     if $\hat{Q}^{\bar{\mu}}(s_t, o_{run}) < \sum_q \bar{\mu}(s_t, q) Q^{\bar{\mu}}(s_t, q)$ then
 7:       $o \sim \bar{\mu}(s_t, .)$
 8:       $o_{run} \leftarrow o$.
 9:     end if
10:   end if
11:   $a_t \sim \pi^{o_{run}}(s_t, .)$.
12: end for
```

In the algorithm, $\bar{\mu}(s,.)$ is a probability distribution over options, corresponding to the option switching averaging policy, and $\pi^o(s,.)$ is a probability distribution over other options or primitive actions, corresponding to option o's policy.

3.3.1 Example - Windy gridworld

The aim of this experiment is to test offline options and offline interruption. A gridworld of 5×5 cells has an obstacle with two exits (Figure 1). Wind blows immediately before the exits. It has an unknown but fixed direction in any instance, and it blows strong gusts with an unknown but fixed probability throughout the episode, pushing the agent one cell at a time. The goal of the agent, starting from a cell in the the leftmost column (marked with 'S'), is to pass one of the exits to the right side (marked with 'G'). The agent gets -1 penalty for every action taken until the goal is reached or the episode elapsed (100 time steps). Moving towards a wall does not change the location of the agent, but it will cost it the usual penalty.

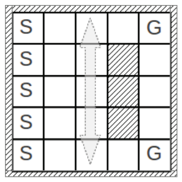


Figure 1. The Windy gridworld. The agent starts randomly in one of the cells marked with 'S' and is tasked with reaching any of the cells marked with 'G'. The arrow indicates the locations and possible directions of wind.

Each instance of this MDP family is characterised with two parameters (p, dir). $p \in [0,1]$ is the probability with which the wind will succeed in changing the position of the agent, while $dir \in \{North, South\}$ is the wind direction. The agent might be pushed one cell in the direction dir with the probability p while in the windy cells. Figure 1 shows the setup.

The agent experiences many instances of this family in the offline phase, and learns an averaging policy across all these instances. This is a policy over the primitive actions that reaches the goal via any of the two exits. At the same time, the agent is made to learn two options with handpicked goals, one for reaching the goal through each of the exits. The agent learns an averaging option switching policy as well, and estimates the maximum option values from these training instances using a Monte-Carlo learning method.

Figure 2 shows the result of 5 runs of a 10000-episode testing phase.

In each test episode, the agent is given 100 time steps in a new instance (p, dir), in which both the flat averaging policy and the offline-interrupted option policy are evaluated. The performance criterion is the accumulated reward in the episode (ranging from -100 to 0). We report in Figure 2 the difference in performance between the two methods, sorted in ascending order to ease interpretation.

As the figure shows, the offline-interrupted option policy is no worse than the flat averaging policy in almost 80% of all episodes. This can be justified by the ability of the leant partial-policies to capture delicate details about the interaction (e.g. the consistent correlation of the wind direction in the windy cells), in contrast to the flat averaging policy. Also, the interruption mechanism allowed for active intervention in the control process (sensed through the change in state) that put that knowledge into play. The results look similar if the setup is not identical in the learning and testing phases, e.g. having only *southerly* winds in the test phase.

3.4 Composition-based hierarchy

Although decomposition of a task is beneficial to manage variability and improve performance in an unknown world, the hierarchy does limit the producible policies to a policy subspace spanned by

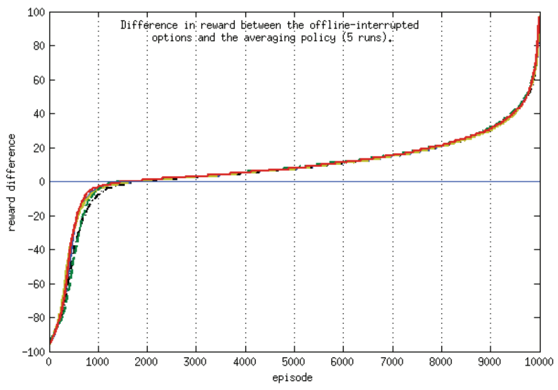


Figure 2. Difference in performance between the offline-interrupted option policy and the averaging policy in 5 runs. Values above zero are episodes where the offline interruption outperforms the averaging policy. Note that episodes are sorted in the figure by performance to facilitate interpretation.

sequences of (sub-)trajectories generated by the (interrupted) option policies. This is a known problem of hierarchical reinforcement learning that resulted in adopting various optimality criteria by HRL methods [5]. We propose next a representation for tasks that admit concurrent execution of subtasks.

3.4.1 Subtask composition

In robotic applications, subtasks are rarely fully independent in the way HRL frameworks handle them. That is, it is common that option policies share state-action space and may have conflicting or interacting goals. We propose to learn options that target the goals of more than one subtask at once, and we call these *composed subtasks*. This can be interpreted as a controlled kind of concurrency for policy synthesis.

Definition 2 (Composed subtask)**.** *If the domain of applicability of two offline options happen to intersect in a non-trivial way, a new option can be defined for that intersection where both the corresponding subtasks are tenable. For o_1 and o_2 being offline options with domains $S_1 \times A_1$ and $S_2 \times A_2$ respectively, a composed subtask $o_{1,2}$ is a capability (of order 2) defined for the state space $S_1 \cup S_2$ and the action space $A_1 \cup A_2$ that optimises the goals of the two subtasks simultaneously. For the composed subtask $o_{1,2}$, the subtasks o_1 and o_2 are called* components.

Because $o_{1,2}$ subsumes o_1 and o_2 in domain and reward, an action taken by a component subtask policy is an action of the composed subtask from a learning point of view. This leads to that every learning update of a component policy also triggers an update for its parent, composed subtask. Thus, almost all learning in the hierarchy can happen *off-policy* while learning the components at the leaves, and very little additional learning effort is ever needed.

Composition is not limited to order 2. Three subtasks may be composed to produce a higher-order subtask, but if the components are *pairwise-composable*. The composition relation defines a tree hierarchy with the primitive options at the leaves, and more complex subtasks at higher levels.

3.4.2 Model description

Composed subtasks can be used as any other option for the purpose of policy synthesis, along with their offline values and interruption. This will produce richer policies that are closer to true optimality. However, this 'unstructured' approach ignores the natural subsumption of these subtasks. We would like to see the composed subtask given the priority over its components when it is able to achieve their joint objectives. That is, an appropriate component should take charge only when the composed subtask is not tenable, but handles control immediately when it is. The generic options framework does not provide us with this flexibility.

To emphasise that, we propose a control hierarchy for concurrent, rather than sequential, tasks that we impose over our offline option implementation. In the *composition-based hierarchy*, control always starts with the highest-order composed subtask (which achieves the goals of all the subtasks underneath). Note that the policy of this subtask is the full task averaging policy. This process is allowed to run as long as it is able to achieve its objectives. Otherwise, it is interrupted and control is moved to an appropriate component. The component, which may be composed from other subtasks as well, is run in a similar fashion. The trick is that the status of the parent subtask is continuously checked, and when it is ready to run again, the running component is interrupted and the parent subtask regains control. In short, control is moved up and down the tree hierarchy, from the more general to the more specific and vice versa, in response to changes in the world captured by the offline interruption function. This can be seen as a special kind of *polling execution* of non-hierarchical execution of hierarchical policies [5].

3.4.3 Learning a model of subtask attainability

The mechanism of offline interruption is one way to learn control switching from seen instances, but it is not by any means the only one. Any function that has the quality of predicting the viability of subtasks can be considered a generalisation to the interruption mechanism. This, for example, is important if the variability in rewards is high in a way that makes relying on upper bounds less useful. *Subtask robustness* is one example of these generalisations.

Robustness is an offline estimate of goal attainability that does not depend immediately on the rewards accumulated by the different policies. Rather, it abstracts away from values and ask explicitly about the success or failure of a subtask. It utilises a level of aspiration as a threshold to control when to switch out from one policy.

One simple realisation of robustness can be in the form of a state-action value function, e.g. acquired using Q-learning. The value $\mathbf{R}(o, s, a)$ estimates the expectation that the option o will be able to achieve its goals from state s and action a. Switching can be controlled through a threshold τ that represents the level of reliability/safety the agent requires. Whenever the option fails to deliver robustness at least as high as the threshold, e.g. due to unobservable task parameters, it is deemed unsafe to use and suspended. This is a trade-off between performance and safety, as robustness prefers a policy that is safe when the world unexpectedly changes, but cannot in general guarantee performance. The robustness threshold τ can be used as a tunable parameter to control the risk attitude of the agent.

Note, however, that the notion of robustness should not be restricted to simple scalar value functions, but can have other forms that reflect the viability of capabilities at states in different settings. This may include, for example, a Pareto efficiency measure with a Pareto frontier as a switching threshold in multi-objective capabilities, or the utility of correlated equilibria with a minimax solution threshold in interactive, multi-agent setting.

3.4.4 Algorithm

Algorithm 2 shows a simple procedure for action selection by hierarchical search in a composition-based hierarchy $\mathcal{K}$. The running option o is used as long as it is able to produce satisfactory behaviour as defined by its robustness function (line 6). If the robustness drops below the threshold τ, o is suspended and one of its components are selected using a suitable metric, e.g. robustness (line 10). The new capability is used similarly (line 11) until the robustness of the parent is recovered above its threshold (line 3).

Algorithm 2 HierarchicalSearch

```
Require: $\mathcal{K}$: task hierarchy; $o_{run}$: running subtask; $p$: parent subtask; $\tau$: robustness threshold; $s_t$: current state.
 1: $a_p \sim \pi^p(s_t, .)$.
 2: if $\mathbf{R}(p, s_t, a_p) > \tau$ then
 3:     return $a_p$.
 4: else
 5:     $a \sim \pi^{o_{run}}(s_t, .)$.
 6:     if $o_{run}$ is primitive option or $\mathbf{R}(o_{run}, s_t, a) > \tau$ then
 7:         return $a$.
 8:     else
 9:         $O' \leftarrow$ components of $o_{run}$ in $\mathcal{K}$.
10:         $q^* \leftarrow \arg\max_{q \in O'} \mathbf{R}(q, s_t, a)$.
11:         return HierarchicalSearch($\mathcal{K}$, $q^*$, $o_{run}$, $\tau$, $s_t$).
12:     end if
13: end if
```

The first call to the algorithm takes the root of the hierarchy as the running subtask. $\pi^o(s, .)$ is a probability distribution over primitive actions related to the policy of option o at state s. The function $\mathbf{R}(o, s, a)$ returns the robustness of state-action pair (s, a) for the option o. Finally, the function **HierarchicalSearch** is a recursive call for the procedure itself.

3.5 Example - Wargus resource gathering task

Following the setup in [14], we implemented Wargus resource gathering task. On a gridworld of 32×32 cells, an agent has to collect items while avoiding an adversarial patrol agent. Agents can move in the eight compass directions, they can only see objects less than 8 cells away (but bearing is always observed), and the adversary can shoot for a maximum range of 5 cells. The objective of the agent is to collect as many items as possible in a specific duration. These items appear in the world one at a time and stay in place until picked by the agent, triggering a new item to appear in a random spot. The patrol agent navigates in a random walk most of the time, but when it sees the agent it uses the shortest path to it, then shoots once in range.

The variability in this task comes from the random (and, most often, unobservable) location of items, and the random (and, most often, unobservable) location of the adversary. A specific assignment of these two parameters produces one MDP from the possible family.

3.5.1 Experimental setup

No relearning is allowed for our agent, and hence every instance of the world is tried by the agent only once. We mix the training and testing phases in this experiment, such that new instances use all the knowledge gathered in all the previous episodes, an approach related to the notion of lifelong learning [22].

For this, the agent starts with *no prior knowledge* of any sort and learns everything (averaging policy, options, switching policies, robustness) from scratch.

An episode starts with a random positioning of the two agents and a single item, and it only ends when the agent is destroyed or when the episode elapses. Performance is measured by the number of items the agent manages to gather throughout the episode. A set of 30 episodes followed by a set of 5 nominal test episodes is called a trial. The test episodes have fixed parameters, but the agent is not allowed to learn in them. This is in order to test the improvement in performance as more experience is gathered. We report the scores achieved in the nominal test episodes.

The experiment is run for 500 trials, and repeated 5 times with the final scores averaged and smoothed.

3.5.2 Methods

We compare the composition-based hierarchy with 3 other methods: a simple averaging policy, offline options, and offline options with interruption:

- Averaging policy: Q-learning over the 8 primitive actions, trained across all the experienced instances.
- Offline options: three options, and their switching policy, are learnt. The options are: `ToGoal (TG)` for navigating towards the item, `FromEnemy (FE)` for navigating away from the patrol agent, and `TG+FE`, the composition of the two. These options are designed to start anywhere on the grid, and terminate upon the occurrence of an event such as seeing the adversary or losing sight of the goal item. The option policies and the switching policy are averaging policies over the experienced instances.
- Offline options with interruption: we added to the previous implementation an offline interruption mechanism that uses the offline values of the option policy. That is, the policy may switch between the three options based on their maximal historical values, rather than having to wait until normal termination.
- Composition-based hierarchy: the hierarchy in Figure 3 is implemented using the same options as above and the following robustness function: the robustness values for the `TG` is acquired using Q-learning with the reward $+1$ if an action leads the agent to where the goal is reachable (seen), -1 only when it leaves the reachability area, and 0 otherwise. For `FE`, a penalty of -1 is given as long as the agent is within the range of sight of the opponent, $+1$ once when it escapes it, and 0 otherwise. The composed subtask `TG+FE` learns using the sum of the two rewards. The robustness threshold is chosen empirically to be 2 for all capabilities.

3.5.3 Results

We compare the performance of the four methods for 500 trials. The results are shown in Figure 4.

As the results show, using the composition-based hierarchy is superior to the other three methods. It exploits the composed capability much more than the other methods, specifically in every state where the subgoals are not in conflict, producing the score difference. Notice that the composition-based hierarchy is approximating the optimal averaging policy, hence it cannot beat the averaging policy's Q-learning asymptotically if given enough time and experience. Still,

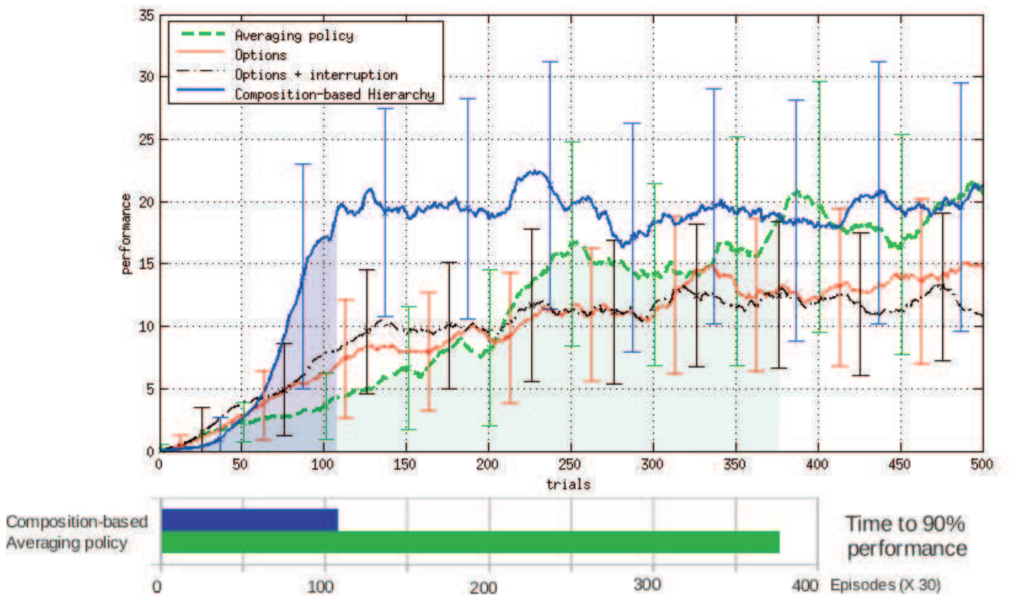


Figure 4. Performance in 500-trial Wargus resource gathering experiment. The curves show the average item count in the test phases of each trial. The curves are means of 5 repetitions, and the error bars show the standard deviation. The composition-based hierarchy (solid - blue) outperforms the other implementations in this task with a clear margin, and achieves the performance level of the averaging policy (green - dashed) in less than third the time. The bars underneath the plot show the time needed by these two methods to achieve 90% to their maximum performance.

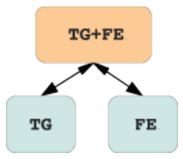


Figure 3. Composition-based hierarchy of the Wargus resource gathering experiment. The bottom nodes represent the subtasks of seeking the resource and escaping the adversary, respectively. The upper node is the composition of the two, with the goal of achieving both objectives at once (here, the full task). The arrows connecting the nodes suggest how control flows flexibly up and down the hierarchy in response to changes in the environment.

the performance head-start is evident as well as the persistent difference in performance for the first half of the experiment. The averaging policy approach needs around 15000 episodes to catch up. The shaded regions in the figure show the time required by the two methods to achieve 90% of their final performance. While the averaging policy needed more than 350 trials, the composition-based hierarchy achieved that in around 100 trials.

Our method needed more time than the other methods in the first few episodes to ascend performance. This can be justified by the need to learn a robustness function in addition to learning the option policies as required by the other methods as well. Remember that these policies and functions are to be readily learnt in the offline phase, preparing the robot to perform immediately in the real world.

4 Related work

The problem of dealing with task variations is of fundamental interest within autonomous robotics, and autonomous agent design in a more general setting. This problem has been studied from many different angles, with tools and techniques being developed to address aspects of the full problem.

One such thread, of relevance to the discussion in this paper, is transfer for reinforcement learning [20]. Techniques for transfer learning attempt to use an existing policy in a source task, typically as an exploration policy to bias learning in a novel target task, with the hope of achieving learning speed-up. When transfer works well, it is because of exploitation of structural properties of tasks which allows for reuse. This is the high-level goal of our approach as well. However, at the algorithmic level, in contrast to transfer as a bias that speeds up learning over numerous trials, we seek a policy that may be immediately applied in a novel instance (in some cases, as a sophisticated initialisation to a final learning step which allows for convergence to a true optimum, but we do not handle this in this paper). Another difference is that we aim to repeatedly apply that transferred policy in many different novel, unknown worlds, while the usual assumption in transfer is that there is a single (and usually known) target task. This is the motivation behind our definition of the problem in terms of policy fragments that are sequenced in different ways to achieve a novel policy.

Multi-task reinforcement learning (MTRL) [20] is a specific branch of the general problem of reinforcement learning transfer, in that it explicitly targets the problem of variation within a family of tasks. Typically, in the MTRL setting, tasks share the same state-action space, and the aim is to learn a policy from sampled task instances that appropriately utilises this experience of multiple contexts, for example, yielding an Average Value Function [13] which is similar to our averaging policy. Some MTRL methods (e.g. [12]) assume the observability of the *type* of the task, while our assumption that the agent is oblivious to that is more practically plausible. Others allow the agent many trials in the new world (e.g. [6]), while we require the agent to act promptly without time to learn afresh. Bayesian methods (e.g. [23]) assume explicit knowledge of the distribution describing the MDP family, whereas we do not.

In this paper, we assumed that the capability goals are chosen by the designer. However, learning these could have also been considered (cf. learning structure and subgoal discovery in HRL, e.g. [11, 17, 9, 8]).

Learning termination conditions for pre-acquired options is discussed in [4]. They use gradient descent procedure on a special functional encoding of the termination condition to learn the optimal switching points under certain requirements, not necessarily maximising accumulated reward. We aim for a more general and less constrained approach for interruption, using offline values and robustness functions.

Finally, Concurrency and composition in HRL is a problem with some history within the literature, e.g., concurrent ALisp [10] and concurrent options [15]. In [16], an explicit approach to composing policies is given. We alternatively opted for allowing the agent to learn the composed subtasks in the offline training phase, as it learns the other options. We believe this to be the better approach to capture the intricacies of different domains and subtask models in order to be able to deal with the problem of offline learning for improved online performance.

5 Conclusion

The main motivation for this paper is the problem of designing an autonomous robot or agent that is capable of quickly and efficiently solving a variety of different problems, drawn from some family defining the domain. This family of problems represents many real world effects, including incompleteness of knowledge arising from the arbitrary richness of the environment (e.g., factors outside the model that do have additional dynamics), or continual change (such as due to other agents in the environment). This way of phrasing the problems differs from the more traditional question of obtaining an optimal policy for a stochastic environment, although the issue is increasingly being considered in many different communities such as under the heading of transfer and multi-task learning.

We have presented a novel approach to policy design and learning, wherein we learn subtasks that make sense across the entire domain (for multiple task/environment settings) and associate with them models such as for interruption. This allows us to define novel policies in terms of compositions of policy fragments that are learnt offline. Our approach builds on existing methodologies such as hierarchical RL with options, but modifies them to address the more general problem identified above. With this, we demonstrate that we are able to achieve superior performance in an online setting, benefiting from problem structuring.

ACKNOWLEDGEMENTS

This work has taken place in the Robust Autonomy and Decisions group within the School of Informatics. Research of the RAD Group is supported by the UK Engineering and Physical Sciences Research Council (grant number EP/H012338/1) and the European Commission (TOMSY Grant Agreement 270436, under FP7-ICT-2009.2.1 Call 6).

REFERENCES

[1] A.G. Barto and S. Mahadevan, ‘Recent advances in hierarchical reinforcement learning’, *Discrete Event Dynamic Systems*, **13**(4), 341–379, (2003).

[2] C. Belta, A. Bicchi, M. Egerstedt, E. Frazzoli, E. Klavins, and G.J. Pappas, ‘Symbolic planning and control of robot motion [grand challenges of robotics]’, *Robotics & Automation Magazine, IEEE*, **14**(1), 61–70, (2007).

[3] R.R. Burridge, A.A. Rizzi, and D.E. Koditschek, ‘Sequential composition of dynamically dexterous robot behaviors’, *The International Journal of Robotics Research*, **18**(6), 534–555, (1999).

[4] G. Comanici and D. Precup, ‘Optimal policy switching algorithms for reinforcement learning’, in *Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems: volume 1-Volume 1*, pp. 709–714. International Foundation for Autonomous Agents and Multiagent Systems, (2010).

[5] T.G. Dietterich, ‘Hierarchical reinforcement learning with the maxq value function decomposition’, *Journal of Artificial Intelligence Research*, **13**(1), (1999).

[6] F. Fernández and M. Veloso, ‘Probabilistic policy reuse in a reinforcement learning agent’, in *Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems*, pp. 720–727. ACM, (2006).

[7] L.P. Kaelbling, ‘Hierarchical learning in stochastic domains: Preliminary results’, in *Proceedings of the Tenth International Conference on Machine Learning*, volume 951, pp. 167–173. Citeseer, (1993).

[8] S. Kazemitabar and H. Beigy, ‘Automatic discovery of subgoals in reinforcement learning using strongly connected components’, *Advances in Neuro-Information Processing*, 829–834, (2009).

[9] S. Mannor, I. Menache, A. Hoze, and U. Klein, ‘Dynamic abstraction in reinforcement learning via clustering’, in *Proceedings of the twenty-first international conference on Machine learning*, p. 71. ACM, (2004).

[10] B. Marthi, S. Russell, D. Latham, and C. Guestrin, ‘Concurrent hierarchical reinforcement learning’, in *Proceedings of the national conference on artificial intelligence*, volume 20, p. 1652. Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999, (2005).

[11] A. McGovern and A.G. Barto, ‘Automatic discovery of subgoals in reinforcement learning using diverse density’, *Computer Science Department Faculty Publication Series*, 8, (2001).

[12] N. Mehta, S. Natarajan, P. Tadepalli, and A. Fern, ‘Transfer in variable-reward hierarchical reinforcement learning’, *Machine Learning*, **73**(3), 289–312, (2008).

[13] T.J. Perkins and D. Precup, ‘Using options for knowledge transfer in reinforcement learning’, *University of Massachusetts, Amherst, MA, USA, Tech. Rep*, (1999).

[14] M. Ponsen, M. Taylor, and K. Tuyls, ‘Abstraction and generalization in reinforcement learning: A summary and framework’, *Adaptive and Learning Agents*, 1–32, (2010).

[15] K. Rohanimanesh and S. Mahadevan, ‘Decision-theoretic planning with concurrent temporally extended actions’, in *Proceedings of the Seventeenth Conference on Uncertainty in Artificial Intelligence*, pp. 472–479. Morgan Kaufmann Publishers Inc., (2001).

[16] K. Rohanimanesh and S. Mahadevan, ‘Coarticulation: An approach for generating concurrent plans in markov decision processes’, in *Proceedings of the 22nd international conference on Machine learning*, pp. 720–727. ACM, (2005).

[17] Ö. Şimşek and A.G. Barto, ‘Using relative novelty to identify useful temporal abstractions in reinforcement learning’, in *Proceedings of the twenty-first international conference on Machine learning*, p. 95. ACM, (2004).

[18] R.S. Sutton and A.G. Barto, *Reinforcement learning: An introduction*, volume 1, Cambridge Univ Press, 1998.

[19] R.S. Sutton, D. Precup, and S. Singh, ‘Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning’, *Artificial intelligence*, **112**(1), 181–211, (1999).

[20] M.E. Taylor and P. Stone, ‘Transfer learning for reinforcement learning domains: A survey’, *The Journal of Machine Learning Research*, **10**, 1633–1685, (2009).

[21] R.L. Tedrake et al., ‘Lqr-trees: Feedback motion planning on sparse randomized trees’, (2009).

[22] S. Thrun and T.M. Mitchell, ‘Lifelong robot learning’, *Robotics and autonomous systems*, **15**(1-2), 25–46, (1995).

[23] A. Wilson, A. Fern, S. Ray, and P. Tadepalli, ‘Multi-task reinforcement learning: a hierarchical bayesian approach’, in *Proceedings of the 24th international conference on Machine learning*, pp. 1015–1022. ACM, (2007).

5th International Workshop on Evolutionary and Reinforcement Learning for Autonomous Robot Systems (ERLARS 2012)
Montpellier, France, August 28 2012

Teaching a Helicopter to Fly with EANT2 – Initial Results

Nils T Siebel[1] and **Sven Grünewald**[2]

Abstract. Evolutionary algorithms are a popular tool to find good solutions for complex optimisation problems. However, they tend not to work well in reinforcement learning ("RL") scenarios. In this short paper we present results from initial experiments where our neuro-evolutionary method EANT2 learns to control a helicopter from the RL-Competition problem base for 2009. The results are promising, although areas for future work remain.

1 INTRODUCTION

Controlling a helicopter is a difficult task indeed. It is considered harder than to pilot an aircraft [7]. One of the most difficult tasks for a helicopter pilot is to hover in mid-air. Learning this is very difficult for human operators *(ibid.)*.

In 2009's RL-Competition (Reinforcement Learning Competition [13] a software environment was made available to learn hovering by reinforcement learning using the RL-Glue[3] framework in a simulation.

Using this framework, we have applied our neuro-evolution learning technique EANT2 [9] to learn this control in the reinforcement learning environment. In this short paper we will briefly show initial results from these experiments.

2 PRELIMINARIES AND RELATED WORK

2.1 Reinforcement Learning

Reinforcement learning [11] is a learning method based on the principle of trail-and-error. An agent, equipped with sensors, is acting in an environment. It is given a reward based on results of its actions, according to a given task unknown to the agent. The agent can learn to solve the task better by maximising the reward (minimising the error) without ever being told which action would be the best in a particular situation.

There is a range of standard "Reinforcement Learning" ("RL") methods set in this scenario, which can be found in the book cited above. In this article we will be using a different method – in the strict sense, no RL method – which nevertheless operates in the RL scenario, using rewards and never been given the correct action.

2.2 Neuro-Evolution

Up to the late 90s only small neural networks have been evolved by evolutionary algorithms [14]. 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 [15], GNARL [1] and NEAT [10]. 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 [2]). 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 [6]. 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 [2, chap. 9].

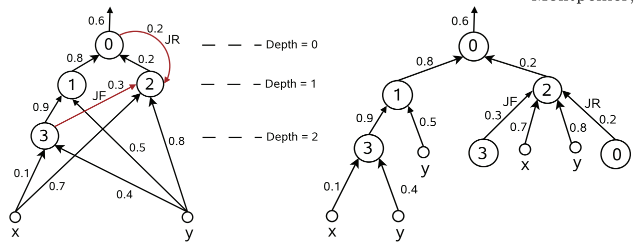
3 THE EANT2 ALGORITHM FOR NEURO-EVOLUTION

3.1 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 [8]. One main feature of this method, and a difference to methods like NEAT, is the separation of network parameter optimisation from the topology optimisation (details below).

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

[1] HTW University of Berlin, Germany, email: siebel@htw-berlin.de
[2] University of Kiel, Germany
[3] http://glue.rl-community.org/



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

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

(c) Corresponding Linear Genome

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

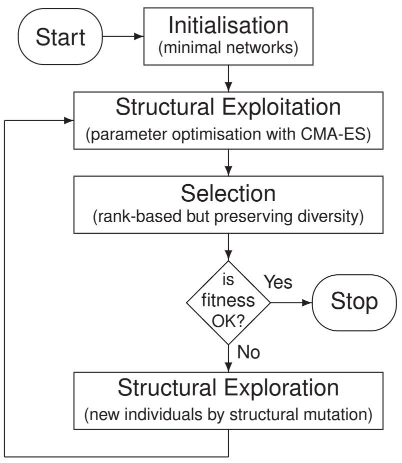


Figure 2. The EANT2 algorithm. Please note that CMA-ES has its own optimisation loop which creates a nested loop in EANT2.

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") [3]. 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 [4].

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.

3.2 Comparison with Other Methods

EANT2 is closely related to the methods described in the related work section above. One main difference to most other methods is the clear separation of structural exploration and structural exploitation. This means it can be called a "two loop" algorithm (many others just have one loop). 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 important 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 [9].

One feature of EANT2, as with all "two-loop" algorithms, 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.

4 EXPERIMENTS AND RESULTS

In the setup used the goal is to hover in a position for 6000 steps, each lasting .1 seconds. The EANT2 learning algorithm minimises the error (maximises the reward) for actions by each of its candidate solutions, i.e. neural networks with a current set of parameters. The feedback given to the algorithm is the distance to the desired position after a new control command it sends out is evaluated in the simulation. A very high error value (10^7) is returned to it on a helicopter crash, ending the current trial.

4.1 RL-Glue

RL-Glue is a framework for RL tasks [12]. There is a strong separation between the trainer, running the experiments, the environment, giving observations (sensor data) and the agent, the learner. The interface of the components is well-defined to enable control and communication between them. All communication in the system routed through RL-Glue.

4.2 Helicopter/Training Interface

Observation Space: 12 dimensional, continuous valued The interface to the learning consists of sensor (input) data, action/control (output) data and the reward, all continuous[4]:

Input from sensors, 12-dimensional:

- forward, sideways and downward velocity ($u_{err}, v_{err}, d_{err}$)
- helicopter position error; x, y and z coordinates ($x_{err}, y_{err}, z_{err}$)
- angular rate (rotation) around x, y and z axes ($p_{err}, q_{err}, r_{err}$)
- quaternion; x, y and z components ($qx_{err}, qy_{err}, qz_{err}$)

Output of controller, 4-dimensional:

- pitch angle of rotor, longitudinal and latitudinal (y_1, y_2)
- main rotor, collective pitch (y_3)
- tail rotor, collective pitch (y_4)

4.3 Weak Baseline Controller (for Comparison)

One method of control for the helicopter is the "weak baseline controller" by Pieter Abbeel, Adam Coates and Andrew Y. Ng of Stanford University, included in the software package. It determines the output as follows:

$$y_1 = -w_y * y_{err} - w_v * v_{err} - w_r * qx_{err} + w_a$$

$$y_2 = -w_x * x_{err} - w_u * u_{err} + w_p * qy_{err} + w_e$$

$$y_3 = -w_q * qz_{err}$$

$$y_4 = w_z * z_{err} + w_w * w_{err} + w_c$$

[4] from http://2009.rl-competition.org/helicopter.php

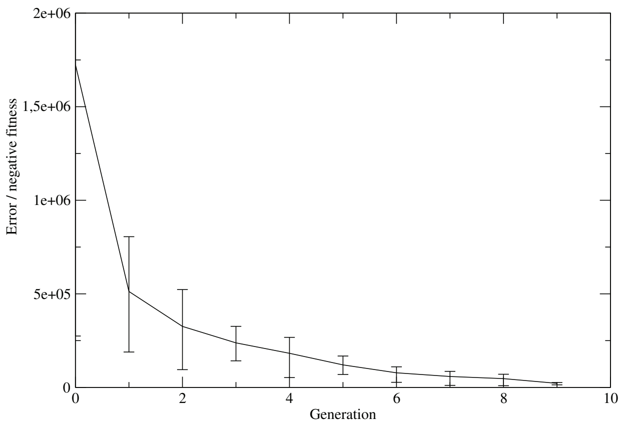


Figure 3. Error (negative fitness value) over EANT2 generations

with weights $w_a = 0.02$, $w_c = 0.23$, $w_e = 0$, $w_p = 0.7904$, $w_q = 0.1969$, $w_u = 0.0322$, $w_v = 0.0367$, $w_w = 0.1348$ $w_x = 0.0185$, $w_y = 0.0196$ and $w_z = 0.0513$.

This controller performs well, in 500 trials it has always kept the helicopter in the air, resulting in an error value of 6670.34.

4.4 Fitness Calculation for EANT2

The fitness used internally for maximisation by EANT2 is calculated once using the errors of the whole episode (all steps of controller, while in the air) as follows:

$$f = -\frac{\sum_{t=0}^{n-1} r_t}{n},$$

where r_t is the reward after time step t and n is the number of steps in the air. The use of the factor n helps to avoid the situation where a fast crash gives a better fitness than a slow crash, as would be the case when just using the sum.

The calculations of the reward after the episode means that *the reward is delayed* until the whole flight is finished. While this makes it harder for the algorithm (less data $\Rightarrow$ it needs more learning steps) than constant feedback it also helps to determine a good value before adapting the controller or its parameters.

4.5 Results and Discussion

EANT2 was used with 15 individuals (neural network structures) per EANT2 generation and a maximum number of 15,000 CMA-ES generations. Due to computational overhead (interfacing with RL-Glue through Java) and the complexity of the simulation 7 weeks of computational time resulted in only 9 EANT2 generations. This is the reason the experiment could only be run once, and these are initial results only.

The results are seen in Figure 3. The training of EANT2 shows a considerable improvement, reducing the error considerably in each generation. Given the delayed reward setup and difficult problem, we consider this promising. A agent with random weights crashes on average after 3.18 steps and reaches a fitness value of $3.3 \cdot 10^6$, which our networks beat after one training steps.

Network sizes are considerable, with 50–60 parameters (synaptic weights etc.) to be optimised by the algorithm, see Figure 4. It is encouraging that the fitness of individuals is still improving even with

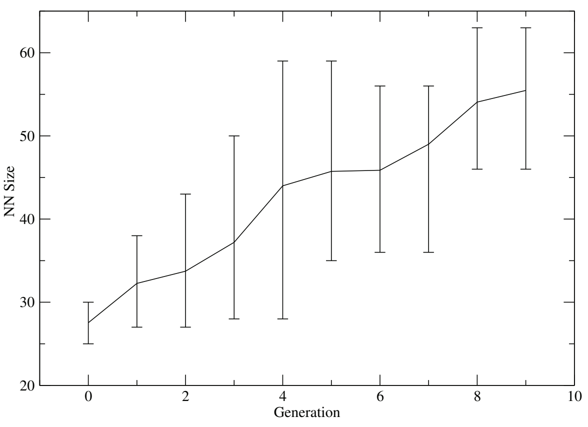


Figure 4. Size of networks (number of parameters) over EANT2 generations

these sizes, which means that EANT2 still works with these large networks.

One difficulty was that the simulation is non-deterministic. Therefore a result found after these 9 generations sometimes flies well for 6000 time steps, sometimes it crashes after 8 steps only. In order to develop controllers more robust to this noise they would need to be trained more, e.g. by running the simulation more than once per evaluation.

5 CONCLUSIONS

We have solved the helicopter flight problem from the Reinforcement Learning 2009 competition with a neuro-evolution algorithm. This is unusual and one might argue that this family of algorithms is not ideally suitable for these types of problem settings. However, the initial results presented here are promising and show that this can, in fact, work.

A remaining problem is still the computational complexity of the learning algorithm, although in these particular experiments a lot of CPU was lost because of the way our C++ code needed to interface via Java with the RL-Glue environment and the simulation, which is not designed to be used in such a way.

Future work will focus on faster training possibilities and hopefully be able to use a faster framework.

REFERENCES

[1] Peter J Angeline, Gregory M Saunders, and Jordan B Pollack, 'An evolutionary algorithm that constructs recurrent neural networks', *IEEE Transactions on Neural Networks*, **5**(1), 54–65, (1994).
[2] Ágoston E Eiben and James E Smith, *Introduction to Evolutionary Computing*, Springer Verlag, Berlin, Germany, 2003.
[3] Nikolaus Hansen and Andreas Ostermeier, 'Completely derandomized self-adaptation in evolution strategies', *Evolutionary Computation*, **9**(2), 159–195, (2001).
[4] Christian Igel, 'Neuroevolution for reinforcement learning using evolution strategies', in *Proceedings of the IEEE Congress on Evolutionary Computation (CEC 2003)*, 2588–2595, IEEE Press, (2003).
[5] Yohannes Kassahun and Gerald Sommer, 'Efficient reinforcement learning through evolutionary acquisition of neural topologies', in *Proceedings of the 13th European Symposium on Artificial Neural Networks (ESANN 2005)*, pp. 259–266, Bruges, Belgium, (April 2005).
[6] Scott Kirkpatrick, Charles Daniel Gelatt, and Mario P Vecchi, 'Optimization by simulated annealing', *Science*, **220**(4598), 671–680, (May 1983).
[7] J. Gordon Leishman, *Principles of helicopter aerodynamics*, Cambridge University Press, Cambridge, UK, 2000.
[8] Nils T Siebel and Yohannes 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*, p. 6 (4 pages). IEEE Computer Society, (December 2006). ISBN 0-7695-2662-4.
[9] Nils T Siebel and Gerald Sommer, 'Evolutionary reinforcement learning of artificial neural networks', *International Journal of Hybrid Intelligent Systems*, **4**(3), 171–183, (October 2007). ISSN 1448-5869.
[10] Kenneth Owen Stanley and Risto P Miikkulainen, 'Evolving neural networks through augmenting topologies', *Evolutionary Computation*, **10**(2), 99–127, (2002).
[11] Richard S Sutton and Andrew G Barto, *Reinforcement Learning: An Introduction*, MIT Press, Cambridge, USA, March 1998.
[12] Brian Tanner and Adam White, 'RL-Glue : Language-independent software for reinforcement-learning experiments', *Journal of Machine Learning Research*, **10**, 2133–2136, (September 2009).
[13] Shimon Whiteson, Brian Tanner, and Adam White. 2009 reinforcement learning competition. `http://rl-competition.org/`, 2009.
[14] Xin Yao, 'Evolving artificial neural networks', *Proceedings of the IEEE*, **87**(9), 1423–1447, (September 1999).
[15] Xin Yao and Yong Liu, 'A new evolutionary system for evolving artificial neural networks', *IEEE Transactions on Neural Networks*, **8**(3), 694–713, (May 1997).

Evolution of CPN Controllers for Multi-objective Robot Navigation in Various Environments

Amiram Moshaiov and **Michael Zadok**[1]

Abstract. This study explores evolutionary training of counter-propagation neural-networks. It concerns navigating a robot in an environment that differs from the trained one. The training of the counter-propagation networks is done using a two phase approach to achieve tuned weights for both classification of inputs and the control function. To diversify trained solutions, and to obtain controllers for various scenarios, a multi-objective evolutionary approach is used. The numerical simulations, which are reported here, demonstrate the ability of the proposed approach to cope with the gap between the trained and tested environment. The evolved controllers exhibit multi-objective characteristics in both the trained and tested environment. This is reflected by the different degrees of path safety and attraction to targets, as observed in the results, which are reported here

1 INTRODUCTION

Evolutionary Robotics (ER) is a general approach to the design of robots [1]. Its scope includes not only the evolution of controllers but potentially all other components of robots. In many ER studies either an Elman or a simple Feed-Forward Network (FFN) is used (e.g., [2]). Here we deviate from that trend by using a Counter-Propagation Network (CPN). CPN includes a self-organizing (instar) network of Kohonen [3] as a first layer and a Grossberg's outstar net [4] as the second one. Such an approach differs from the common methods, as it involves not only the training of a mapping from sensed information to actions, but also the organization of the sensed information into classes, based on a similarity measure. In CPN, The later aspect is handled by the Kohonen's self organizing layer, whereas the former is accomplished by the Grossberg second layer.

It is noted, surprisingly, that CPN has not been used in ER studies (to the best of our knowledge). To employ CPN in an evolutionary context we had to devise a special training approach. The purpose of the training is to find Neuro-Controllers (NCs) that can navigate the robot in a navigation problem and environment, which differ from the trained one.

In robot navigation, evolutionary training of NCs is often defined as a single objective problem (e.g., [2]). In contrast to multi-objective problems, solving a single-objective one typically produces one optimal behavior rather than diverse behaviors. In the last decade, with the availability of Multi-Objective Evolutionary Algorithms (MOEAs), e.g. [5], several Multi-Objective ER (MO-ER) studies employed MOEAs to obtain Pareto-optimal NCs based on contradicting objectives (see a review in [6]). The usefulness of diversity, as obtained by a multi-objective approach, has been recently demonstrated, in [7] and [8], for the bootstrap problem that is common in single objective ER. Such studies suggest that reaching diverse behaviors for one problem may produce useful (initial) solutions for another problem. The motivation to use a multi-objective approach is two-fold. First, as in [7] and [8], it provides diversity, which may help coping with numerical problems. The second, as in [6] and similar studies, it provides useful controllers for different scenarios. In particular, similar to [6], we use here the trade-off between safety and target-attraction to produce a diversity of controllers, with remarkable different behaviors.

To obtain NCs for diverse behaviors one may use various methods. With a repeated use of the common single objective approach, with a new objective at each independent search, one may expect to find a set of different NCs. However, due to the substantially different objectives, this approach is expected to produce behaviors which are drastically different from one another. Alternatively, one may employ weighted sum of objectives, with a gradual change of objective preferences, or a Pareto-approach. The later techniques can be used to simultaneously produce diverse solutions with gradual changes. Here we employ NSGA-II, a well known MOEA, [5], as the evolutionary search mechanism to obtain Pareto-based diversity. The obtained solutions are expected to support coping with future shifts from one environment/problem to the other. It is noted that due to the use of CPN for the NCs, the algorithm employs NSGA-II in two separated phases.

Our use of a CPN is motivated by a claim, which has been raised in [9]. It is argued there that for complex environments, such as dealt with in [9], a semi-manual modular approach is required. The aforementioned approach involves separate trainings of several simple neural networks on several simple environments. We postulate that the use of a CPN, backed with multi-objective search, may help reducing such a requirement. In the current study we do not attempt to substantiate such a claim. Rather, given the infancy of using CPN in ER, we restrict our focus to a description of the proposed multi-objective training of CPNs, and to demonstrating its applicability in dealing with particular environments, which are adopted from the study in [9].

The rest of this paper is organized as follows. The background, in section 2, provides some references and details relevant to the foundations of our approach. It is followed by a methodology, in section 3, which describes the details of the current search approach for the Pareto-optimal CPN-NCs. The methodology is continued with descriptions of the trained and tested environments. The results of the numerical simulations are presented in section 4. Finally, the conclusions from this study are provided.

[1] The Iby and Aladar Fleischman Faculty of Engineering, Tel-Aviv University, Israel, email: moshaiov@eng.tau.ac.il

2 BACKGROUND

2.1 General

Algorithms in ER frequently operate on populations of candidate controllers (or individual robots) that are initially selected randomly. This population is then repeatedly modified to reach optimality according to an objective function, or functions. To study the possible use of CPN in the context of ER, we followed the simulation details of the robot kinematics, and multi-objectives, of [6], while using environments as in [9]. We note that many ER researchers amend simulation with actual testing [1]. According to [1], there appear to be particular advantages in combining simulated, training phase evolution with lifelong adaptation by evolution on a physical robot. Here, we concentrate on a simulation-based study. Such a study is currently sufficient for demonstrating the applicability of CPN to the environment in question. By applicability we mean producing a simulation-based initial population of NCs. Such solutions may be considered as candidates for use, with adaptation, in actual testing, which is likely to be required for coping with un-modeled aspects of the simulation.

Most of the early and late studies in ER have not attempted to study contradicting objectives. There have been only a few such ER studies, which deal with the simultaneous optimization of separate objectives [6]. The following sub-section provides some relevant background to this type of optimization and its solution by MOEAs.

2.2 Pareto-optimality

Pareto-based search deals with finding the Pareto-optimal set, or its numerical approximation, using dominance relation (see below). The Pareto-optimal set includes non-dominated solutions from the feasible search space given no a-priori preferences on a finite set of objectives which are contradicting. In the following definition the symbol $\lhd$ stands for < and for > in minimization and in maximization, respectively. It denotes *better than* and is used between two solutions to denote that one solution is better than the other in a certain objective, $u \lhd v$ denotes that solution u is better than solution v in a particular objective.

Definition 1 (Dominance): *A solution $x^{(1)}$ is said to dominate solution $x^{(2)}$ ($x^{(1)} \preccurlyeq x^{(2)}$) if both conditions 1 and 2 are satisfied:*

1. *The solution $x^{(1)}$ is no worse than $x^{(2)}$ in all n objectives, or $f_j(x^{(1)}) \ntriangleright f_j(x^{(2)})$ for all j=1 ,2 ,..n*
2. *The solution $x^{(1)}$ is strictly better than $x^{(2)}$ in at least one objective, or $f_j(x^{(1)}) \lhd f_j(x^{(2)})$ for at least one $j \in \{1,2,\dots,n\}$*

Definition 2 (The global Pareto-optimal set): *Among the feasible solutions, the Pareto Set is the set of solutions* P' *which are not dominated by any other member of the feasible solution space.*

The Pareto-front is the set of performance vectors in objective space of all solutions of the Pareto-optimal set. For the Pareto-based evolution we have used NSGA-II [5]. Due to the permutation problems the use of a genetic algorithm is not recommended for the evolution of neural-networks [10]. Hence, as pointed out in [6], NSGA-II may not be the most optimal search algorithm for NCs, and a modified version, as used in [6], may be better. Yet NSGA-II proved to be useful for our current demonstration purposes.

2.3 Counter-Propagation Networks

CPNs are NNs which differs from FFNs as explained in the introduction. The original idea of mixing Kohonen and Grossberg layers is attributed to Hecht-Nielsen [11]. While a promising concept, their use is not as common as that of FFNs. With increasing interest in cognitive robotics, the type of training should shift, from simple behavior-based mappings of sensors to actuators, to more complex approaches. CPNs are one such possibility, which has not been investigated in the context of ER. The advantage of using CPNs is that, once trained, they provide knowledge about the environment in the form of input classes. In regular training of CPNs there are two phases. The first is to cluster the inputs, and the second is to create a mapping by the use of a supervised approach. The unsupervised learning, in the Kohonen self organizing layer, is commonly based on neighborhood functions. This means that weight adjustments are done not only for the winner neuron but also to its neighboring units [3]. Due to the lack of a supervisor in ER, and due to the learning by interactions with the environment, there is a need to re-examine existing CPNs learning algorithms. In particular, there is a need to investigate various alternatives to the evolutionary training of CPNs, and to compare it with other approaches. Here, we suggest and describe one possible pseudo-code that can be used either with a single objective ER or for MO-ER.

3 METHODOLOGY

3.1 Simulated Robot

The robot model is based on the miniature 5.5 cm diameter Khepera robot, as in [2], with the following modifications. A total of 16 simulated sensors is used. Eight simulated IR sensors identify obstacles (walls) and the others identify targets. The sensors are located, as pairs of an obstacle and a target sensor, at eight locations as shown in figure 1.

All simulated IR sensors have the same characteristics. The max range of any IR sensor is 5cm and its span angle is 6^0 (shown for the two front sensors). The target sensors have a narrower span of 30^0 as shown for one of them. The max range of the target sensor is simulated to be 100cm, which ensures target sensing anywhere in the maze. The sensors do not have a "blind range," and their output range is [0, 1]. The zero represents an object found at the max range, and the one is for the case when the sensed object is at the robot periphery. In the current implementation no noise is added to the simulated sensors, which is left for future research.

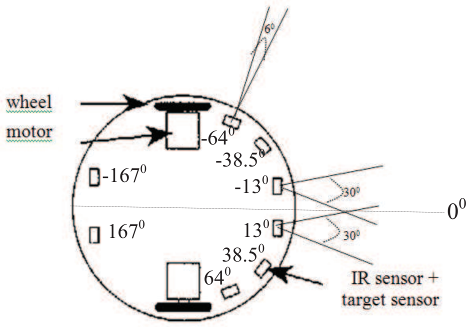


Figure 1. Robot and sensors

The simulated robot model converts motor commands on rotational speeds of the robot wheels from the outputs of the NNs into simulated robot motions. The range of the wheel speeds scaled into the range [-0.5, 0.5]. The wheels radius is taken as 1cm. The time-step of the simulation set to 5sec, and the robot moves 2.5cm per step at maximum speed).

3.2 Trained and tested environments

Both the trained and tested environments, which are depicted in figures 2 and 3 respectively, are based on environments used in [9].

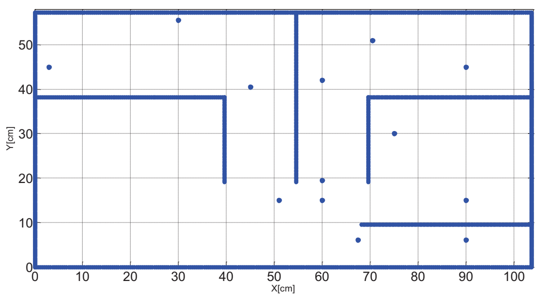


Figure 2. Trained environment

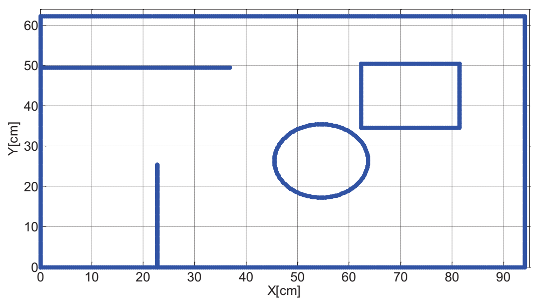


Figure 3. Tested environment

According to [9], the trained environment concerns nine different types of robot situations such as a wide corridor, a narrow corridor, the need to turn right/left, pass freely without walls, etc. In contrast to [9], our approach does not use separated simple environments for a semi-manual training. Rather, we use the complex environment directly for a non-manual evolutionary training.

For the learning process, we use several targets that the robot should reach. The targets are designed to specific positions including: (90.0, 6.0), (67.5, 6.0), (60.0, 15.0), (60.0, 42.0), (90.0, 45.0), (70.5, 51.0), (60.0, 19.5), (90.0, 15.0), (75.0, 30.0), (51.0, 15.0), (45.0, 40.5), (30.0, 55.5), (3.0, 45.0). These are shown, using dots, in figure 2. Spreading the targets aims to create an evolutionary pressure towards the different regions of the maze. In addition, we allocated a place in the maze with no targets. This supports simulating areas that are less desirable to be reached. For training we have used four different robot start-points located at (95, 5), (95, 45), (15, 5), (15, 45). In the two left points the robot is facing towards the right and vice versa.

In order to check the generality of the NCs, it is tested with an unknown environment. The tested environment, which is depicted in figure 3, is also based on [9]. For the testing case, the robot start point is at (5, 55) and it is facing right. The target point, for the tested case, is at (80, 10).

3.3 Simulated neuro-controllers

The simulated NCs maps sensors' information (input) into appropriate motor commands (output). The simulated CPN has two layers: the input layer connects the 16 sensors to a hidden layer. The hidden layer has 9 neurons that connect to two neurons in the output layer (motor commands). The reason of using a hidden layer of 9 neurons is that we try to compare it with the MNN (Modular-Neural-Network) of [9]. The 9 neurons follow the 9 classifications used in [9]. We use a 163 length vector to define each NC. This is based on: 16 weight inputs multiply by 9 neurons in the hidden layer + 9 weight neurons multiply by 2 outputs neurons. An additional weight is used to determine the slope of the sigmoid for the activation functions. No bias weights are used. In the current implementation of the CPN it has been observed that there is no need to adjust the weights of neighboring neurons, hence only the weights of the winner are adjusted.

3.4 Objective functions

Two fitness functions are used (marked as F1, F2). The details are similar to [6]. The first function F1, which is based on [2], aims at fast and straight motions with obstacle avoidance without any specific destination. F1 is given as follows:

$$F_1 = \frac{\sum_{j=1}^{act\_step} V_j (1-\sqrt{\Delta v})_j (1-I)_j}{\max\_step}$$

$$0 \le V \le 1$$
$$0 \le \Delta v \le 1$$
$$0 \le I \le 1$$

Where:

- V is the absolute value of the sum of the rotational speeds of wheels. V is high when the robot is moving fast (forward or backward).
- Δv is the absolute value of the difference between the wheel speeds. $1-\sqrt{\Delta v}$ is high when the robot is moving straight without making any turn during the step.
- I is the normalized activation value of the sensor with the highest value. I is high if the sensors perceive an obstacle.

F1 is calculated as an average over the maximum allowable number of steps of the accumulated score. The accumulation, however, is over the actual number of steps which are performed over a run of any particular NC. The function can have any value between 0 and 1, with the aim to be maximal. The second objective, F2, concerns reaching targets (e.g., food-targets). Similar to F1, F2 is based on averaging of performances over the maximum allowable number of steps of the process, and summing over the actual number of steps. This reduces scores to non-moving robots at the training phase. Once a target is touched by the robot, it is eliminated (consumed). After the robot finishes touching all targets, they re-appear. Then the process of reaching targets continues as long as the robot does not reach a maximum allowable number of steps. F2 is defined as follows:

$$F_2 = \sum_{j=1}^{act\_step} \frac{f_2}{\max\_step}; \quad f_2 = \begin{cases} H & if\ hit\ target \\ \dfrac{1}{1+d} & else \end{cases}$$

Where:

- d is the distance from our robot to the nearest target among the remaining targets at the current step.
- H is the score that the robot gets when it reach a target. It is set to 50 in the current implementation.

F1 and F2 follow [6], where it is shown that these are

$$F_1^{test} = \frac{\sum_{j=1}^{act\_step} V_j(1-\sqrt{\Delta v})_j(1-I)_j}{act\_step}$$

$$F_2^{test} = \sum_{j=1}^{act\_step} \frac{f_2}{act\_step}$$

contradicting. It is noted however, that the application in [6] is substantially different from the current one. In particular, in [6] there has been no separation between the trained and tested scenarios as done here, and no target sensor were used.

The above objective functions are good for measuring performance with infinite loop of targets that re-appear. Here there is a need to adapt the performance evaluations for the tested case. In that case, our interest is in reaching only one target.. We reformulated F1 and F2 for the tested case as follows:

3.5 Evolving CPN

Training a CPN requires special care due to the existence of two separated training issues. The first is to produce a classification of inputs, whereas the second is to produce an optimal model of mapping the classified input into optimal outputs. It is noted that the first issue is in essence independent from the second one. It constitutes an organization of expected inputs into classes regardless of the expected use of the classes.

In the current application, the optimality measures are integrative, and performance evaluations are based on sequences of interactions with the environment. Hence, the difficulty is three-fold. There is a need to handle both the two separated issues of the training as well as the integrative nature of the evaluation. To overcome these difficulties we propose a two-phase evolutionary search. The pseudo-code is schematically presented in figure 4, and described below.

The separation between the two search phases can easily be spotted in figure 4, where the 1st phase involves the left side and vice versa. The primary goal of the 1st phase is a step-wise updating of the Kohonen layer based on steps of interactions with the environment. Yet, it also contains the evolution of weights of the Grossberg layer as described under interact # 1a and # 1b. The primary goal of the 2nd phase is to further adjust the weights of the Grossberg layer while fixing the Kohonen layer as resulted in phase 1. The various blocks of the pseudo-code of figure 4 are described below.

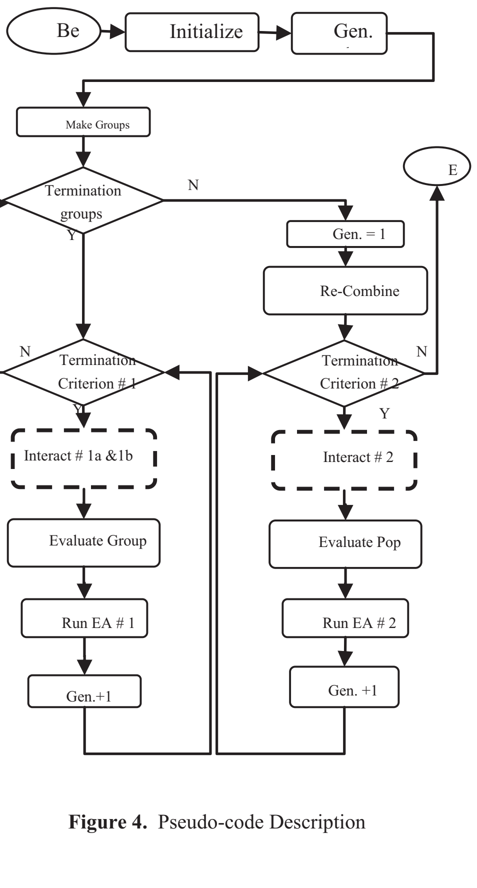


Figure 4. Pseudo-code Description

It is noted that the proposed pseudo-code can be used for both single and multi-objective problems. This depends on the type of evolutionary search algorithm used in the blocks "Run EA#1 and #2 of figure 4.

Initialization:

At the beginning of the search, a random population is initialized with N individuals. In our application each individual is a CPN-based NC with a fixed structure, were both weights of the Kohonen layer and weights of the Grossberg layer are sought.

Make Groups:

In our study we have found that dividing the population into groups is beneficial for the 1st phase. Four groups are used in the current study, corresponding to the four start-points of the trained environment, as described in the previous section. At each generation, individuals from the same group are trained from the same start-point. During the 1st phase individuals evolve only within the group. In our study we set N=56 to be divided into four groups, of 14 individuals each, for the 1st phase.

Termination Group:

This criterion is used to terminate the 1st phase of the algorithm after the completion of the evolution of the four groups. If there is a group that has not evolved, it is forward to Termination Criterion #1.

Termination Criterion # 1:

This criterion is used to terminate the evolution of a group. In our study maximal number of generations is used (50 generations per group). If the group evolution has not reached that number, the individuals of the group proceed to interact # 1a.

Interact # 1a and 1b:

At each generation each individual performs two consecutive sequences of interactions with the environment, both starting at the corresponding start-point of its group. The purpose of the 1st sequence (Interact # 1a) is to update the weights of the Kohonen layer. These updates are done at each step of the sequence. The updates form Interact#1a are used for the next sequence. In the 2nd sequence (Interact # 1b) no weight update is done during the sequence of interactions with the environment. The purpose of the 2nd sequence is to obtain the performances F1 and F2 of the individual NC based on the updated version of the Kohonen layer. Each robot finishes the interaction either due to an obstacle or by using a pre-defined number of steps (200 steps in the current implementation)

Evaluate Group:

This step performs the calculation of F1 and F2 for each individual NC based on the accumulated scoring during Interact # 1b.

Run EA # 1:

This part of the algorithm can be based on existing evolutionary algorithms. In the current implementation the search is for the Pareto-optimal set and front using NSGA-II based on [10]. The results include offspring population to be evaluated in the next iteration. During this evolutionary stage weights of the Grossberg layer are tuned, whereas the Kohonen layer is kept fixed (no crossover or mutation). For recombination in the Grossberg layer we used 100% probability. As typically depicted in figure 5, we employed: (Wyd',Wya')=SBX(Wyd,Wya); (Wzd',Wza') = SBX(Wzd,Wza). The mutation in the Grossberg layer is done with polynomial mutation.

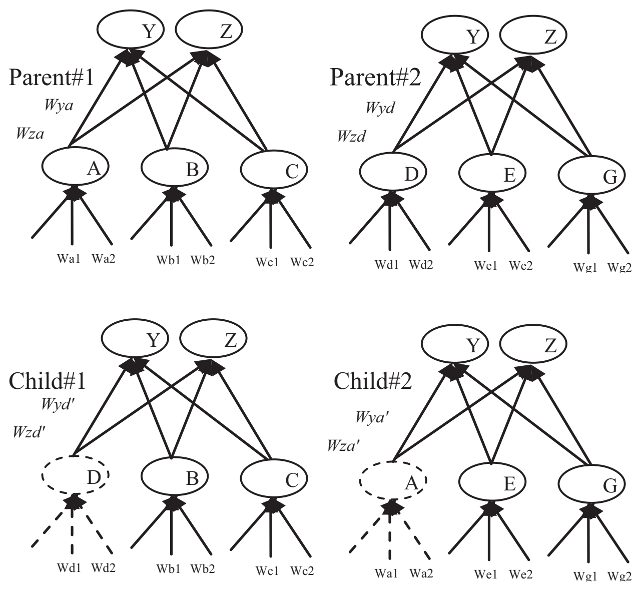


Figure 5. Mating and mutating CPNs

Once the 1st phase of the search is terminated, the 2nd one is starting with a new counting of the generation number, and with a new termination criterion. The 2nd phase includes:

Re-combine Groups:

The groups used in the 1st phase are no longer needed, and the entire population is allowed to mate with any individual regardless of the original groups.

Termination criterion # 2:

This criterion is used to terminate the second phase of the algorithm. In our study a maximal number of generations is used (currently 250).

Interact # 2:

In contrast to the corresponding function of the 1st phase, here no classification updates are done during the performance of the sequence of interactions (similar to Interact #1b).

Evaluate Pop:

This step performs the calculation of F1 and F2 for each individual NC from the entire population (merged groups) based on the accumulated scoring during Interact # 2.

*Run EA #*2:

This stage, which is depicted in figure 5, is similar to EA #1. However, the crossover operation used here is different from that used in EA#1. Here crossover includes a procedure that performs changes both to the Kohonen layer and to the Grossberg layer. The Kohonen layer of an offspring is based on a random selection of units from the parents. This means that the Kohonen weights are transferred "as are" with each inherited neuron. In addition the Grossberg weights of the offspring results from SBX crossover of the weights from the parents. With respect to figure 5, the recombination of the Kohonen layer involves switching unit A with unit D, etc. No mutation used here. In addition crossover is carried out in the Grossberg layer. For example: (Wyd',Wya')=SBX(Wyd,Wya); (Wzd',Wza')=SBX(Wzd,Wza). The mutation for the Grossberg layer is done with polynomial mutation.

4 EXPERIMENTAL STUDY

4.1 Trained solutions

Figure 6 depicts results from 30 different training runs. One front is shown for each run. Each front is depicted by using the same shape and color for all performance vectors of the front. The horizontal and vertical axes are F1 and F2 respectively. As clearly observed from the fronts there is a strong scatter, namely the repeatability of the fronts is poor. This phenomenon is due to the definition of the objective functions, and not to the training scheme. It appears as a local Pareto problem, but due to the large dimension of the search space it is hard to explore it in-depth. This means that a large number of runs would be required to increase the confidence on reaching the global front. Yet, due to lack of an analytical solution, the best front obtained is always just a guess. For the purpose of this study we are interested in satisficing solutions. Hence this phenomenon is acceptable here. In fact, the best front that is accumulated, using dominance relation among all 30 fronts of the different runs, has been sufficient for the purpose of this study. As described in the rest of this section, the obtained NCs, which are associated with the accumulated front, are satisfactory for the tested environment.

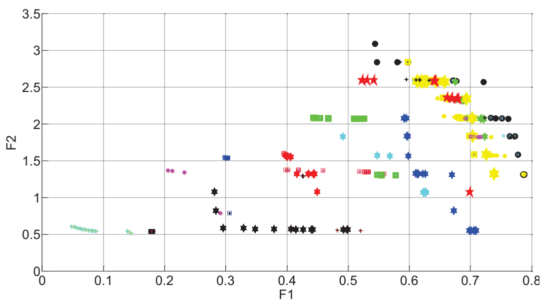


Figure 6. Pareto-fronts

Figure 7 shows the path, with 200 steps, which is associated with the NC with the best F1 performance. This has the performance vector of F1=0.7879 and F2= 1.3104. In contrast, figure 8 depicts the path using the controller with the best F2 (F1=0.5435, F2= 3.0884). When observing figure 7, recall that the concept behind this controller is to have a safe-straight moving robot with maximal speed, on the expense of a reduced target collection abilities. This means that the robot is expected to be attracted to spacey areas, where it can move straight and far from obstacles, rather than to the targets. The shown path clearly avoids narrow areas where most of the targets are. The best F1 controller "prefers" using the available steps on the larger, hence safer, yet empty room. Comparing the path of figure 8 with that of figure 7, it becomes evident that the best F2 controller is much less safe as it runs the robot into narrow places, while striving to reach all targets. As shown in the rest of this section, the nature of these controllers is observable also in the tested environment.

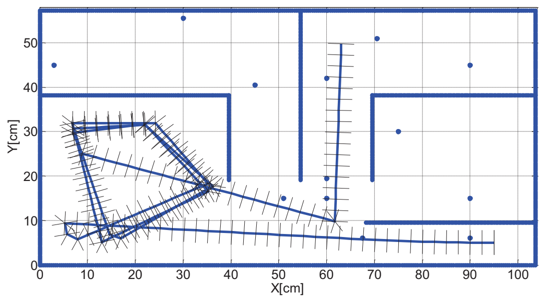


Figure 7. Path of the best F1 controller

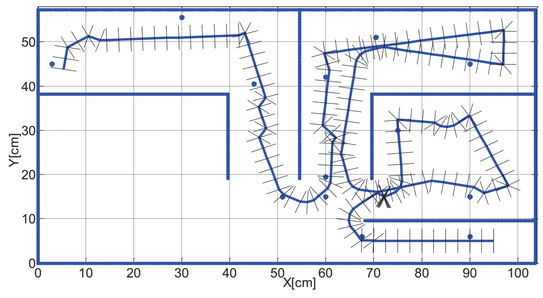


Figure 8. Path of the best F2 controller

4.2 Tested solutions

The tested environment is designed to be substantially different from the trained one. This can easily be observed from figures 2 and 3. It should be noted that the tested problem differs from the trained one not just by the structure and shape of the walls, but also by the target setting. Here we aim at reaching only one target from a far location. Figures 9 and 10 show the obtained paths using the controllers with best F1 and best F2 respectively.

Both paths start at the upper left point, inside the narrow corridor. Both controllers cope well with this challenge. The safest controller of figure 9 shows "less confidence," inside the corridor. It attempts to avoid walls (behave as safe as possible), while trying to be fast. Apparently, this results in an unstable behavior. The less safer and potentially slower controller, of figure 10, shows a more stable behavior inside the corridor. When reaching the end of the corridor both controllers keep moving in a similar fashion till facing with the rectangular obstacle. At that point the safer controller, of figure 9

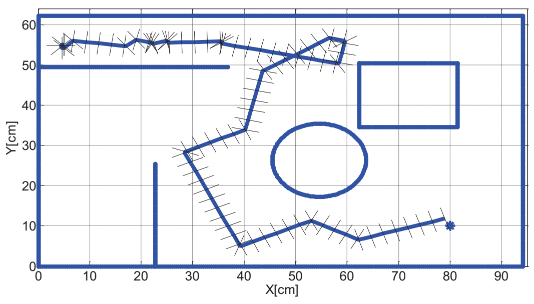


Figure 9. Path of the best F1 controller (test)

avoids entering the narrow space between the rectangle and the side wall. It turns into the relatively spacey zone but still is drawn towards the target which is located at the lower left side of the environment. Both controllers eventually reach the target. Yet the less safer one reached it in a much shorter, but less safer way, as compared with the other one.

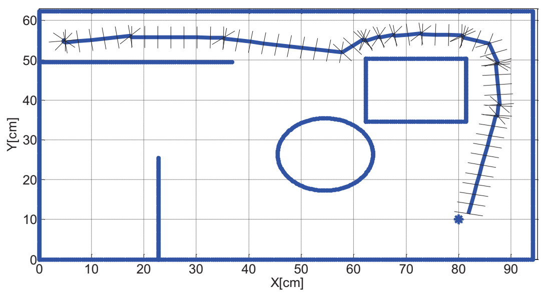


Figure 10. Path of the best F2 controller (test)

4.3 Performance vs. generation

In this section of the numerical studies we are interested in a demonstration of the benefit of having an initial population of NCs that have been trained on the 1st environment for the purpose of finding optimal solutions for the 2nd environment. For this purpose, we have used the accumulated Pareto-optimal set of the 1st environment, as opposed to a random population of the same size. The first population is termed pre-trained population and the second is termed random. Both include 46 individuals. Next we use these two populations as initial populations to be trained on the environment and problem of figure 3, with the target and start point as in the tested case of figures 9 and 10 (no division to groups). At each generation of each separated run we record the current best F1 and best F2. Figures 11 and 12 respectively show the F1 and F2 performance improvement over generation.

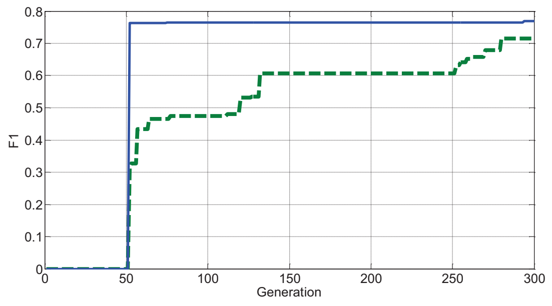


Figure 11. F1 performance over generations – Pre-trained vs. Random

Each figure compares the performance in one objective and contains two curves. The first, marked by the green (dashed) curve, is of the random NCs. The second, marked by the blue (continues) curve, is of the pre-trained NCs. As expected the pre-trained controllers are much superior. In fact, they appear to be optimal from the start, which is counted after the first phase of the training at 50 generations (the graphs present the 300 generations as summed from the 2 phases (50+250)).

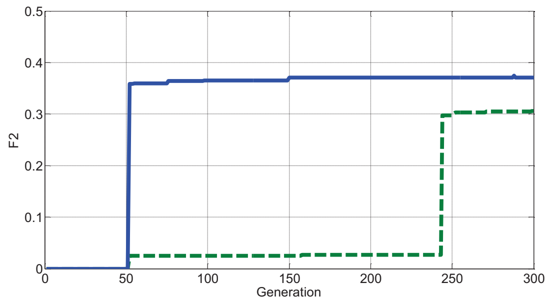


Figure 12. F2 performance over generations - Trained vs. non-trained

5 CONCLUSIONS

The main conclusion of this study is that counterpropagation networks coped well with the problem of evolving controllers for the multi-objective navigation problem used here. In particular, the ability to generalize from the trained environment to the tested one seems promising. This should be evaluated with respect to the semi-manual approach of [9], which has been used for the same environments.

As noted in the introduction, this study appears to be the first to employ CPNs for ER applications. As such, the primary focus is on the demonstration of the proposed algorithm, rather than on its optimization and comparison with other methods.

Much work is left for future studies on the methodology and computational aspects of evolving CPNs as compared with current approaches. Systematic study is still needed to explore the full power of the proposed method. Future studies should deal not only with a systematic comparison with current ER approach, but also include other issues such as: (a) exploitation of the proposed scheme for more challenging problems, (b) actual implementation in physical environments. Of a particular interest will be the understanding of the obtained classes of inputs and their meaning and influence on the results.

In addition to dealing with the evolution of CPN-based NCs, this preliminary study concerns the use of MOEAs in ER. The demonstrations provided here show that the multi-objective "nature" of the examined controllers is kept also in the tested problem. Future studies should explore the advantage of the proposed method both for single and multi-objective problems.

REFERENCES

[1] J. Walker, S. Garrett and M. Wilson, 'Evolving controllers for real robots: A survey of the literature', *Adaptive Behavior*, **11 (3)**, 179–203, (2003).

[2] D. Floreano and F. Mondada, 'Evolution of homing navigation in a real mobile robot,' *IEEE Transactions on Systems, Man, and Cybernetics, Part B,* **26 (3)**, 396-407, (1996).

[3] T. Kohonen, 'Self-Organizing Feature Maps and Abstractions', *Proc. of the Third Int. Conf. on Artificial Intelligence and Information-Control Systems of Robots,* 39-45, (1984).

[4] S. Grossberg, *Studies of mind and brain: neural principles of learning, perception, development, cognition, and motor control*, Dordrecht, D. Reidel Pulishing Company, 1982.

[5] K. Deb, A. Pratap, S. Agarwal and T. Meyarivan 'A fast and elitist multiobjective genetic algorithm: NSGA-II', *IEEE Trans Evol Comput*; **6(2)**, 182–97, (2002).

[6] A. Moshaiov and A. Ashram-Wittenberg, 'Multi-objective Evolution of Robot Neuro-Controllers', *IEEE Congress on Evolutionary Computation,* 1093 – 1100, CEC 2009.

[7] J. B. Mouret and S. Doncieux, 'Overcoming the bootstrap problem in evolutionary robotics using behavioral diversity', *IEEE Congress on Evolutionary Computation*, 1161 – 1168, CEC 2009.

[8] S. Israel and A. Moshaiov, 'Bootstrapping Aggregate Fitness Selection with Evolutionary Multi-objective Optimization', to appear in the *Proc. of PPSN 2012*.

[9] S.J. Han and S.Y. Oh, 'An optimized modular neural network controller based on environment classification and selective sensor usage for mobile robot reactive navigation,' *Neural Comput Appl*, **17(2)**, 161–173, (2008).

[10] X. Yao, 'Evolving artificial neural networks', *Proc. IEEE*, **87(9)**, 1423–1447, (1999).

[11] R. Hecht-Nielsen, 'Counterpropagation Networks', *Applied Optics*, **26 (23)**, 4979-4984 (1987).