Learning to Approach a Moving Ball with a Simulated Two-Wheeled Robot

Felix Flentge

Department of Computer Science
Johannes Gutenberg-University Mainz
D-55099 Mainz, Germany

Abstract. We show how a two-wheeled robot can learn to approach a moving ball using *Reinforcement Learning*. The robot is controlled by setting the velocities of its two wheels. It has to reach the ball under certain conditions to be able to kick it towards a given target. In order to kick, the ball has to be in front of the robot. The robot also has to reach the ball at a certain angle in relation to the target, because the ball is always kicked in the direction from the center of the robot to the ball. The robot learns which velocity differences should be applied to the wheels: one of the wheels is set to the maximum velocity, the other one according to this difference. We apply a *REINFORCE* algorithm [1] in combination with some kind of extended *Growing Neural Gas* (GNG) [2] to learn these continuous actions. The resulting algorithm, called *ReinforceGNG*, is tested in a simulated environment with and without noise.

1 Introduction

In this paper we propose using a reinforcement-based learning method to learn the actions of a simulated two-wheeled robot to approach a moving ball. The ball has to be reached under certain conditions in order to kick it towards a given target. First, the ball has to be in front of the robot. Second, since the ball is always kicked in the direction from the center of the robot to the ball, the angle between the vector from the robot to the ball and the vector from the robot to the target has to be sufficiently small (as sketched in Fig. 4).

While there are algorithms to find optimal trajectories for two-wheeled robots to reach a given static target [3], the problem gets significantly harder if the target is moving and has to be reached under certain conditions. Although the basic principles of these algorithms of either turning around or moving forward may be applicable in this case, they are far from optimal if there is noise in perception or in the execution of actions. Under these circumstances there have to be small corrections every step. There is no straightforward way to design a good control strategy. We would have to simulate the ball's movement and possible movements of the robot to determine the point where we could meet the ball under the right conditions.

As far as we know our work is the first attempt to learn continuous actions for a two-wheeled robot in a problem as complex as sketched above. There have been some limited attempts with learning algorithms on similar problems. For example

A. Bredenfeld et al. (Eds.): RoboCup 2005, LNAI 4020, pp. 106–117, 2006.

in [4] a robot arm learns to grasp a rolling ball, but only with a fixed initial situation and discrete actions. In [5] a simulated autonomous underwater vehicle with controls very similar to a two-wheeled robot is used in a two-dimensional environment. While the vehicle learns the continuous actions to reach the given static targets, the resulting trajectories do not look particulary time-optimal.

For learning we use a *REINFORCE* algorithm [1] together with an extended *Growing Neural Gas* (GNG) [2] in an actor-critic architecture [6]. The GNG is a self-organizing map which is able to adapt to the local dimension and the local density of an unknown possibly high-dimensional input distribution. This is particulary useful in the context of action learning because the input distribution is generally not known in advance and changes during training. Also, the data often stems from some lower-dimensional structures in the input space due to implicit dependencies between the data. The GNG is able to detect and to adapt to these structures. Using a REINFORCE algorithm to learn continuous actions should lead to smooth trajectories and should faciliate learning because similar situations should require similar actions.

In the next section we give a short description of the GNG and of the necessary extensions for function approximation. Since the data in our experiments have a specific order, i.e. they stem from trajectories, there have to be some further modifications to the original GNG. After describing these, we give a short introduction to REINFORCE algorithms and show how to use the extended GNG as a function approximator for these kind of algorithms. In section 4 we explain the simulation environment and show some results.

2 Growing Neural Gas for Function Approximation

2.1 Growing Neural Gas

The *Growing Neural Gas* (GNG) [2] is a self-organizing map without a fixed global network dimensionality, i.e. its topological structure adapts to the local dimensionality of the input data. It consists of a set of neurons $\{c_1, c_2, \ldots, c_N\}$ where each neuron c_i is associated with a position $w_i \in \mathbb{R}^d$ in input space. Neurons may be connected by undirected edges. If there is an edge between the neurons c_i and c_j, we denote it by $e_{i,j} = e_{j,i}$. The neighbors of a neuron c_i (i.e. the neurons having a common edge with c_i) will also be indexed $c_{i,j}, j = 0, \ldots, N_i$, where N_i is the number of neighbors of c_i and $c_{i,0} = c_i$ by convention. We use this notation for other variables as well, e.g. $w_{i,j}$ is the position of the j-th neighbor of c_i.

Whenever some input data point $x \in \mathbb{R}^d$ is presented to the network during training, the idea is to move the best matching (i.e. the closest) neuron c_b by some fraction α_b and its neighbors $c_{b,i}$ by some smaller fraction α_n towards x. If we have different x we will also write $c_{b(x)}$ to make clear which x is meant. Edges are always created between the closest neuron and the second-closest one, which leads to the induced Delaunay triangulation [7]. As the neurons move around, some aging scheme has to be used to delete edges which are no longer part of the

induced Delaunay triangulation. Therefore, each edge $e_{i,j}$ has a counter named $age_{i,j}$, and edges with an age $age_{i,j}$ exceeding a certain age_{max} are deleted. New neurons are inserted from time to time in regions with a high quantization error until a maximum number N_{max} of neurons is reached. To estimate the local quantization error, every neuron is equipped with an error variable err_i which is updated with the current quantization error every time the respective neuron is the best matching neuron. Figure 1 shows the GNG algorithm.

Repeat (until some stopping criterion fulfilled):

1. Observe $x \in \mathbb{R}^d$
2. Find neurons c_b and $c_{b'}$ with smallest euclidean distances to x:
   $\|x - w_b\| \leq \|x - w_{b'}\| \leq \|x - w_i\| \, \forall i \neq b, b', \; b \neq b'$
3. Increase the age $age_{b,i}$ of all edges $e_{b,i}$ emanating from c_b by 1
4. If edge $e_{b,b'}$ does not exist, create it
5. Reset $age_{b,b'}$ to 0
6. Update error variables:
   $\Delta err_b = \|w_b - x\|^2$
   $\Delta err_i = -\delta_{err} \cdot err_i, i = 1, \ldots, N$ with constant $0 < \delta_{err} < 1$
7. Update position vectors:
   $\Delta w_b = \alpha_b \, (x - w_b)$
   $\Delta w_{b,i} = \alpha_n \, (x - w_{b,i})$ for all neighbors $c_{b,i}, i = 1, \ldots, N_b$ of c_b
8. Remove edges with $age_{i,j} > age_{max}$ and neurons without neighbors
9. If $N \leq N_{max}$, add a new neuron c_s every N_{steps} steps as follows:
   - Find the neuron c_q with the largest error err_q
   - Choose its neighbor $c_{q,r}$ with the largest error $err_{q,r}$, $r = 1, \ldots, N_q$
   - Insert a new neuron c_s between c_q and $c_{q,r}$
   - Reduce the errors err_q and $err_{q,r}$ and set the error err_s of the new neuron:
     $\Delta err_q = -\delta_{new} \cdot err_q$
     $\Delta err_{q,r} = -\delta_{new} \cdot err_{q,r}$
     $err_s = 0.5 \cdot (err_q + err_{q,r})$

Fig. 1. The GNG algorithm

2.2 Growing Neural Gas for Function Approximation

The easiest way to approximate a function $f : \mathbb{R}^d \to \mathbb{R}$ from training examples (x, y) with a self-organizing map is to associate each neuron c_i with a value $v_i \in \mathbb{R}$ and to have a local constant approximation by assigning the same value v_b to all input vectors x with $\|x - w_b\| < \|x - w_i\|$ for all $i \neq b$.

In order to improve the approximation quality of such a network we do not only use the best matching neuron's value v_b but also the values $v_{b,i}$ of its neighbors $c_{b,i}$. We then calculate normalized weights $m_{b,i}(x)$ for these values using Gaussian functions as it is done in *Radial Basis Function Networks* (RBF). The radii of the Gaussians used for calculating these weights are set to the average lengths of all edges emanating from $c_{b,i}$ following the suggestion in [8].

$$m_{b,i}(x) = \frac{\exp\left(-\frac{\|x-w_{b,i}\|^2}{l_{b,i}^2}\right)}{\sum_{j=0}^{N_b} \exp\left(-\frac{\|x-w_{b,j}\|^2}{l_{b,j}^2}\right)} \quad i = 0, \ldots, N_b \tag{1}$$

$$l_{b,j} = \text{average length of all edges emanating from neuron } c_{b,j}$$

The final approximation $\widetilde{F}(x)$ is then calculated as a weighted sum:

$$\widetilde{F}(x) = \sum_{i=0}^{N_b} m_{b,i}(x) v_{b,i} \tag{2}$$

Please note that we only use the values of the best matching neuron and its neighbors for calculating the final approximation $\widetilde{F}(x)$ instead of using all values as it is done in the common RBF approach or in [8]. There are two reasons for this: first, it is faster since we do not have to calculate weights for all neurons, and second, using all values did not work well in the learning experiments described below. We think that is because the concept of "distance" is quite problematic if different input dimensions have different meanings (e.g. distances, angles and angular velocities). This problem is eased using (2) because neurons which are close to each other but are not direct neighbors have no influence on the approximation[1].

Whenever we observe some training data (x, y), the values of the best matching neuron and its neighbors $v_{b,i}$ are trained by gradient descent with learning rate α_v:

$$\Delta v_{b,i} = \alpha_v m_{b,i}(x) \left(y - \widetilde{F}(x)\right) \quad i = 0, \ldots, N_b. \tag{3}$$

2.3 Learning from Trajectories

An important difference to pure function approximation tasks is that in action learning the data stem from trajectories through the state space running from an initial state x_0 to some terminal state x_T. If we use the standard GNG algorithm, these data would lead to strange results because the neurons would concentrate at the ends of the trajectories. Therefore, we keep the neurons' positions fixed during an episode (i.e. the time span between $t = 0$ and $t = T$) and accumulate the changes in auxiliary variables $\widehat{w}_i$ attached to each neuron. We keep track of how often we changed these auxiliary variables by a counter z_i for each neuron. At the end of an episode we use these variables to change the positions of the neurons according to the averages of all changes:

$$\Delta w_i = \frac{\widehat{w}_i}{z_i} \text{ for all neurons } i = 1, \ldots, N \text{ with } z_i > 0 \tag{4}$$

Furthermore, we postpone the insertion of new neurons and edges as well as the deletion of edges until the end of an episode (steps 4, 5, 8, and 9 of the

[1] Of course, in contrast to RBF networks the resulting approximation is not continuous anymore, but has some points of discontinuity where the best matching neuron changes.

algorithm in Fig. 1). Therefore, we keep track of all pairs of neurons closest and second-closest to the input vectors x_t occurring during an epsiode in a list P. Figure 2 provides an overview of the complete algorithm for training. For look-up one has to perform only steps 1 and 2 as well as the steps 9 and 10 of the first part.

During an episode:

1. Get input pair (x, y)
2. Find the two nearest neurons c_b and $c_{b'}$ (Fig. 1 step 2)
3. Increase the age $age_{b,i}$ of all edges emanating from c_b by 1
4. If the pair $(c_b, c_{b'})$ is not in list P, add it
5. Update quantization error variables (Fig. 1 step 6)
6. Update auxiliary position variables:
   $\Delta\widehat{w}_b = \alpha_b\,(x - w_b)$
   $\Delta\widehat{w}_{b,i} = \alpha_n\,(x - w_{b,i})$ for all neighbors $c_{b,i}, i = 1, \ldots, N_b$ of c_b
7. Increase counter variables z_b and $z_{b,i}$ of c_b and of all of its neighbors $c_{b,i}$ by 1
8. If $N + N_{insert} < N_{max}$, increase insert counter N_{insert} every N_{steps} steps by 1
9. Calculate the weights $m_{b,i}(x)$ according to (1)
10. Calculate the final approximation $\widetilde{F}(x)$ according to (2)
11. Train the values $v_{b,i}$ according to (3)

At the end of an episode:

1. Update the weights w_i according to (4)
2. If there are any pairs (c_i, c_j) in P without edges $e_{i,j}$, create them
3. Set the ages $a_{i,j}$ of edges $e_{i,j}$ with a pair (c_i, c_j) in P to 0 and delete P
4. Remove edges with $age_{i,j} > age_{max}$ and neurons without neighbors
5. Insert N_{insert} new neurons (analogous to Fig. 1 step 9) and set the values of the new neurons according to the current approximations at their positions:
   $v_s = \widetilde{F}(w_s)$
6. Set all $\widehat{w}_i$, counters z_i and N_{insert} to 0

Fig. 2. The extended GNG algorithm for data from trajectories

3 ReinforceGNG

In reinforcement learning problems there is an agent which perceives the states x_t of an environment and chooses some action a_t at each time step t. The agent also receives some reward $r_t \in \mathbb{R}$ which rates the situation x_t it is in and the action a_{t-1} it has chosen the last time step. The goal of the agent is to maximize the sum of rewards it receives over time. Transitions between states and rewards may be stochastic and the rewards may also be delayed.

3.1 REINFORCE Algorithms

The reinforcement learning component of ReinforceGNG is based on the class of REINFORCE algorithms introduced by Williams in 1992 [1]. The general

setting for REINFORCE algorithms is a network of (stochastic) units u_i which propagate an input x through the net to produce an output a. This output leads to a direct scalar reinforcement signal R that is used by the units in the net to adjust their weights w_i. A stochastic unit u_i draws its output a_i from a probability distribution depending on its input x_i and its weight vector w_i. In the continuous case the probability distribution is given by a density function $g_i(\xi, x_i, w_i)$.

REINFORCE algorithms try to maximize the expectation value of the immediate reinforcement values R over time. Under certain stationary and independence conditions on the environment's choice of inputs and reinforcement signals, this expectation value only depends on the units' weight vectors w_i. Therefore, we will assume a weight matrix W consisting of all weight vectors w_i and will write $E\{R|W\}$ for this expectation value. Williams shows that with an update of the form

$$\Delta w_i = \alpha_i (R - b_i) e_i \tag{5}$$

the average update vector lies in a direction where this performance measure $E\{R|W\}$ is increasing. In this update equation $\alpha_i > 0$ is some learning rate (depending at most on the time t and w_i), b_i is the reinforcement baseline (just some value conditionally independent of the output a_i given W and x_i) and

$$e_i = \frac{\partial \ln g_i(a_i, x_i, w_i)}{\partial w_i} \tag{6}$$

is the characteristic eligibility of w_i.

In the following we assume that we have only one unit u and omit all the indices i. An interesting possibility is to draw the output a from a normal distribution $N(\mu(x), \sigma(x))$ with input dependent means $\mu(x)$ and standard deviations $\sigma(x)$ as weights. The standard deviations $\sigma(x)$ can be used to control the behavior of the algorithm: large $\sigma(x)$ lead to an explorative behavior while small $\sigma(x)$ lead to the exploition of the behavior learned so far. Using the density of the normal distribution we get the following characteristic eligibilities:

$$\frac{\partial \ln g\left(a, \mu(x), \sigma(x)\right)}{\partial \mu(x)} = \frac{a - \mu(x)}{\sigma(x)^2} \tag{7}$$

$$\frac{\partial \ln g\left(a, \mu(x), \sigma(x)\right)}{\partial \sigma(x)} = \frac{\left(a - \mu(x)\right)^2 - \sigma(x)^2}{\sigma(x)^3} \tag{8}$$

Williams suggests setting the learning rates to $\alpha\sigma(x)^2$ and to choose the reinforcement baseline $b(x)$ according to a reinforcement comparison scheme, i.e. to an estimate of the upcoming reinforcement based on the experience so far. This leads to the update equations:

$$\Delta\mu(x) = \alpha(R - b(x))(a - \mu(x)) \tag{9}$$

$$\Delta\sigma(x) = \alpha(R - b(x)) \frac{\left(a - \mu(x)\right)^2 - \sigma(x)^2}{\sigma(x)} \tag{10}$$

These equations give some insight in how the algorithm works: if the agent gets an reinforcement signal R better than the reinforcement baseline $b(x)$, $\mu(x)$ is moved in the direction of the output a, if R is worse than $b(x)$, $\mu(x)$ is moved in the opposite direction. In (10) the standard deviation $\sigma(x)$ is changed to make the occurance of a more likely if R is better than the reinforcement baseline $b(x)$, and vice versa. E.g. if $R > b(x)$ and the distance $|a - \mu(x)|$ is greater than the standard deviation $\sigma(x)$, $\sigma(x)$ is increased.

3.2 ReinforceGNG

Since the task we want to learn involves delayed rewards, but REINFORCE algorithms learn only from direct reinforcement signals[2], we first have to think about how we can get a direct reinforcement value from delayed rewards. The standard way to do this is to use some kind of actor-critic architecture [6]. The actor chooses an action a_t for a state x_t and receives a direct reinforcement value R_{t+1} from the critic one time step later. This reinforcement value is used to improve the policy (i.e. the mapping from situations to actions) π used by the actor. The critic learns an approximation $\widetilde{V}(x)$ of the value function $V^\pi(x)$ for the current policy π of the actor. The value $V^\pi(x_t)$ of a state x_t for a certain policy π is given by the expectation value of the discounted sum of rewards gained when starting in this state and following the policy π until the last time step T.

$$V^\pi(x) = E_\pi \left\{ \sum_{k=0}^{T} \gamma^k r_{t+k+1} \;\middle|\; x_t = x \right\} \tag{11}$$

The future rewards r_{t+k+1} are discounted by γ to decrease the influence of later rewards in favor of more immediate ones. Usually, the value function cannot be calculated exactly, because the transition probabilities between states and the expectation values of the rewards are not known. Reinforcement learning methods often try to approximate this function. One way to learn an approximate value function, known as *Temporal Difference* (TD) learning, is to base the estimation of the value of a state x_t on the sum of the immediate rewards and the discounted approximated value of the next state $r_{t+1} + \gamma\widetilde{V}(x_{t+1})$. This sum can also be used as a direct reinforcement signal $R_{t+1}$for the actor rating the last action a_t. Since r_{t+1} and $\widetilde{V}(x_{t+1})$ are only known in the following timestep, the updates for time t are performed at time $t + 1$.

The extended GNG is used to learn the approximate value function $\widetilde{V}(x)$, the means $\widetilde{\mu}(x)$, and the variances $\widetilde{\sigma}(x)$ used for determing the actions. The neurons' values v_i are used for approximating the value function $\widetilde{V}(x)$. They are trained as in (3) with the data pairs $\left(x_t, r_{t+1} + \gamma\widetilde{V}(x_{t+1})\right)$. Two additional variables are added to each neuron: μ_i and σ_i. These are used to calculate the approximations $\widetilde{\mu}(x)$ and $\widetilde{\sigma}(x)$ as weighted sums analogous to (2). They are trained according

[2] There is one exception for episodic tasks with only a single reward that is delivered at the end of an episode.

to (9) and (10) taking the weights $m_{b(x_t),i}(x)$ into account. With $\widetilde{V}(x_t)$ as the reinforcement baseline we get[3]:

$$\Delta\mu_{b(x_t),i} = \alpha_\mu m_{b(x_t),i}(x_t)\left(r_{t+1} + \gamma\widetilde{V}(x_{t+1}) - \widetilde{V}(x_t)\right)(a_t - \widetilde{\mu}(x_t)) \tag{12}$$

$$\Delta\sigma_{b(x_t),i} = \alpha_\sigma m_{b(x_t),i}(x_t)\left(r_{t+1} + \gamma\widetilde{V}(x_{t+1}) - \widetilde{V}(x_t)\right)\frac{(a_t - \widetilde{\mu}(x_t))^2 - \widetilde{\sigma}(x_t)^2}{\widetilde{\sigma}(x_t)} \tag{13}$$

If there are minimum and maximum values for actions a_{min} and a_{max}, it has to be assured that the resulting μ_i are within these bounds by setting them to their respective limits. The same holds for the σ_i with minimum σ_{min} and maximum σ_{max} to ensure proper exploration behavior. Figure 3 gives an overview of the complete ReinforceGNG algorithm. For look-up only the approximate mean $\mu(x_{t+1})$ has to be calculated and is then used as action a_{t+1}.

1. Receive situation x_{t+1} and reward r_{t+1}
2. Calculate $\widetilde{V}(x_{t+1})$, $\widetilde{\mu}(x_{t+1})$ and $\widetilde{\sigma}(x_{t+1})$ as weighted sums according to (2)
3. Train the extended GNG with $\left(x_t, r_{t+1} + \gamma\widetilde{V}(x_{t+1})\right)$ according to the extended GNG algorithm as described in Fig. 2
4. Train the $\mu_{b(x_t),i}$ of last time step's best matching neuron $c_{b(x_t)}$ and its neighbors $c_{b(x_t),i}$ according to (12)
5. Train the $\sigma_{b(x_t),i}$ of last time step's best matching neuron $c_{b(x_t)}$ and its neighbors $c_{b(x_t),i}$ according to (13)
6. Determine action $a_{t+1} \sim N(\widetilde{\mu}(x_{t+1}), \widetilde{\sigma}(x_{t+1}))$
   if $a_{t+1} < a_{min} \Rightarrow a_{t+1} = a_{min}$; if $a_{t+1} > a_{max} \Rightarrow a_{t+1} = a_{max}$

Fig. 3. The complete ReinforceGNG algorithm

4 Experiments

It was intended to use ReinforceGNG to learn an "intelligent" approach ball behavior for the 3D Simulation League. Unfortunately, simulated two-wheeled robots with an explicit kick-effector were not yet available at the time of our experiments. So we had to create our own simulation environment, but we tried to build the simulation meaningful with respect to the Simulation League.

4.1 Simulation Setup

We decided two use circular robots with a diameter of 0.44 m, which was the size of the robots in the 3D Simulation League in RoboCup 2004 and is also

[3] This can be regarded as using a reinforcement comparison scheme since the value of a situation is an estimate of the upcoming rewards based on the experience so far. The resulting expression $r_{t+1} + \gamma\widetilde{V}(x_{t+1}) - \widetilde{V}(x_t)$ is known as TD error.

about the size of a midsize robot. The maximum velocity of the robot is 2 $\frac{\text{m}}{\text{s}}$. We do not consider acceleration, braking or friction for the robot's movements and do not simulate collisions. The robot is able to change the velocities of its wheels every 100 milliseconds. The diameter of the ball is 0.22 m. The velocity of the ball decreases exponentially with $v^B_{t+1} = 0.995 v^B_t$ every simulation step of 10 milliseconds, i.e. every second the ball loses about 40 percent of its velocity.

The goal is to position the robot relative to the ball so it is able to kick the ball towards a given target. Figure 4 shows the situation. To kick the ball, the distance between the robot and the ball d^B_t has to be 0.07 m or smaller and the angle between the orientation of the robot and the ball α^B_t has to be no bigger than 22.5° in either direction. The ball is kicked in the direction of the vector from the center of the robot to the center of the ball. Therefore, to reach the target the angle α^T_t between this vector and the vector to the target should not be bigger than 12.25°. If these conditions are met, the trial is considered successful. A trial is canceled after 25 seconds of simulation time if the robot does not reach the ball under the right conditions.

We concentrate on the situation where the ball is on the ground and not too fast. We think that otherwise it would be almost impossible to approach the ball at the required angle due to noise in the perception of the ball and in the execution of the actions. Also, we assume that the ball is somewhere near the robot, because this is very easy to achieve with some hand-coded behavior. Therefore, at the beginning of a trial, the ball is set to an arbitrary point up to 5 m away from the robot with a velocity of up to 4 $\frac{\text{m}}{\text{s}}$. The target is set to an arbitrary point 5–15 m away from the ball.

The action we learn is a continuous value a between -1 and 1. It encodes the difference in velocity which should be applied to the wheels, i.e. if the action is

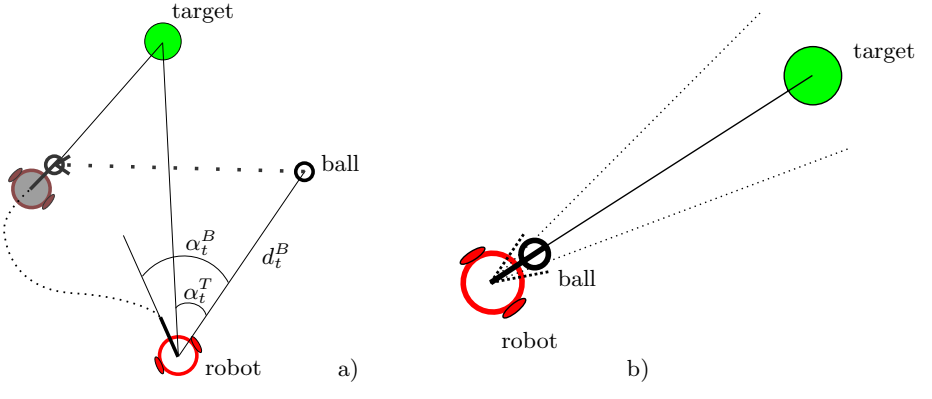

Fig. 4. Sketch of different situations with the relevant distance and angles. a) Some random initial situation with a possible solution trajectory. b) The terminal situation to reach in order to kick the ball towards the target. The dashed lines show the limits for the angle between the robot's orientation and the ball and for the angle between the kick direction and the target direction.

greater then zero, we apply a velocity of 1 to the left wheel and $1 - 2\,|a|$ to the right wheel and vice versa. So, the robot always tries to move as fast as possible.

The situation is encoded in a five dimensional input vector with the following dimensions (see Fig. 4):

- the distance d_t^B from the robot to the ball
- the angle α_t^B between the current orientation of the robot and the direction towards the ball
- the angle α_t^T between the direction from the robot towards the target and the direction from the robot towards the ball
- the difference $\Delta\alpha_t^B = \alpha_t^B - \alpha_{t-1}^B$ in α_t^B between the last and the current time step
- the difference $\Delta\alpha_t^T = \alpha_t^T - \alpha_{t-1}^T$ in α_t^T between the last and the current time step

These dimension are normalized to lie roughly between 0 and 1. Every time step the robot does not reach the required conditions, it gets a penalty of $-0.01 d_t^B$. If the trial is successful, the robot gets a reward of 100. The small penalty on the distance makes successful trials more likely in the beginning because the robot tries to minimize the distance to the ball and will eventually meet the ball under the right conditions.

To choose the learning parameters we did some preliminary tests with different settings but they were not really optimized systematically. The maximum number of neurons is set to 1000, we started with $3^5 = 243$ neurons evenly distributed in the hypercube $[0, 1]^5$. New neurons are inserted every 100 steps. The learning rates for the movement of the neurons α_n and α_b are set to 0.01 and to 0.025. The maximum age of edges is 100. The neurons' errors are reduced with rate $\delta_{err} = 0.001$ every time step and with $\delta_{new} = 0.01$ if a new neuron is inserted. In the beginning all μ_i and v_i are initialized to 0, σ_i to 0.25. The minimum and maximum values of σ_i are set to $\sigma_{min} = 0.1$ and $\sigma_{max} = 0.25$. The learning rates α_μ and α_σ are set to 0.1 and the discount factor γ to 0.9.

4.2 Simulation Results

To assess how well the simulated robot has learned the task we perform a fixed set of predefined tests. In each test the ball is set 2 m away in front of the robot. We put targets at 10 m distance from the ball and at 8 different angles relative to the robot's orientation $(-180°, -135°, -90°, \ldots, 90°, 135°)$. The ball either moves at 4 $\frac{\text{m}}{\text{s}}$ in one of the four directions $-180°, -90°, 0°, 90°$, or does not move at all. So all in all we have 40 different tests.

We performed several experiments and the robot always learned to approach the rolling ball within the required angles for all test situations. In Fig. 5 one can see some of the learned trajectories, which look quite smooth. The targets are almost always hit with high precision.

We also did experiments with noise in perception and in the execution of actions. For perceptional noise, we added normal distributed noise of $N(0, 0.0965)$ to the distance information and normal distributed noise of $N(0, 0.1225)$ to angular information as it is done in the 3D Simulation League. For noise in the

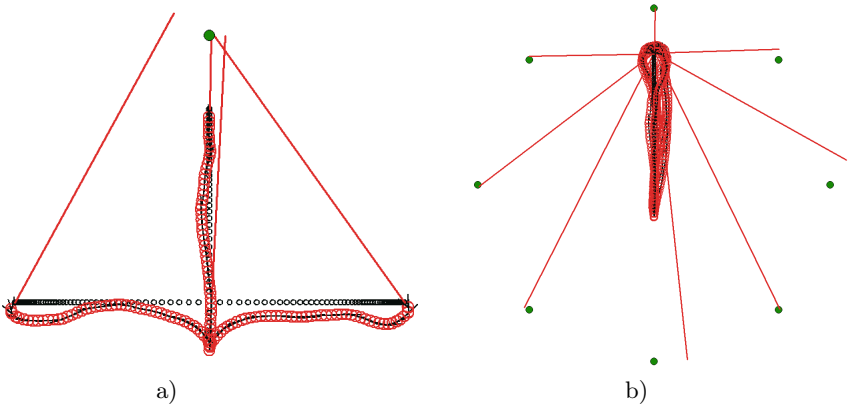

Fig. 5. Trajectories of the learned approach ball behavior. The robot is the bigger empty circle, the ball is the smaller empty circle and the target is the filled circle. The objects were drawn every 100 milliseconds. The straight lines show the direction the ball is kicked in at the end of each episode. a) The four different directions for the moving ball with one target. The ball moving towards the agent can hardly be seen because it is reached significantly faster than all other balls. b) Trajectories of the robot following an upward moving ball with shots to all eight different targets.

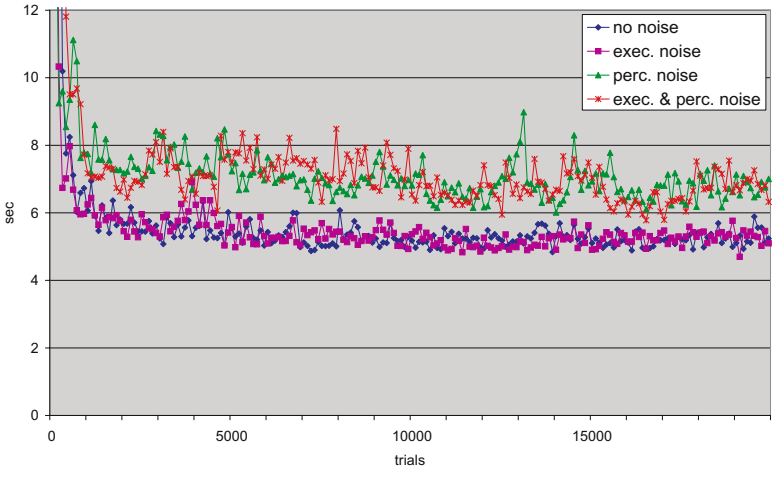

Fig. 6. Average simulation time needed per test from single runs without noise, with noise in the execution of the action, with noise in the perception, and both

execution of actions, we added normal distributed noise of $N(0, 0.05)$ to the velocities of the wheels every simulation step. ReinforceGNG was again able to learn to approach the rolling ball in all test situations. Noise in the execution of actions had practically no influence on the robot's performance. Figure 6 shows the development of the average simulation time needed per test. All 40 tests were

performed every 100 trials. The values shown are from single runs. All in all we did 10 individual runs for all kinds of noise and the results always look very similar to those in the figure. We see that the method learns relatively quickly but sometimes the performance gets worse during learning. Therefore, we keep track of the best average time so far and if we get a better value, we save the current network for later use.

5 Conclusion

We showed how a REINFORCE algorithm in combination with an extended GNG with some modifications regarding the learning from trajectories can successfully learn to approach a moving ball under certain conditions with respect to a given target. For the future we plan to use this method in our agent for the 3D Simulation League. We also plan to do experiments with moving targets and train networks for different requirements on the precision of the kick. Unprecise but fast kicks should be used in situations were opponents are near and precise kicks can be used if the agent has enough time. We also plan to incorporate the proposed learning method into a midsize robot.

References

1. Williams, R.J.: Simple statistical gradient-following algorithms for connectionist reinforcement learning. Machine Learning **8** (1992) 229–256
2. Fritzke, B.: A growing neural gas network learns topologies. In Tesauro, G., Touretzky, D., Leen, T., eds.: Advances in Neural Information Processing Systems 7, Cambridge MA (1995)
3. Balkcom, D., Mason, M.: Time optimal trajectories for bounded velocity differential drive vehicles. International Journal of Robotics Research **21** (2002) 199–217
4. Riedmiller, M., Janusz, B.: Using neural reinforcement controllers in robotics. In: Proc. 8th Australian Conference on Artificial Intelligence, Canberra (1995)
5. Gaskett, C., Wettergreen, D., Zelinsky, A.: Q-learning in continuous state and action spaces. In: Australian Joint Conference on Artificial Intelligence. (1999) 417–428
6. Sutton, R., Barto, A.: Reinforcement Learning. An Introduction. MIT Press, Cambridge, Massachusetts (2000)
7. Martinetz, T.: Competitive hebbian learning rule forms perfectly topology preserving maps. In: Proc. of ICANN'93, Springer (1993) 427–434
8. Fritzke, B.: Fast learning with incremental RBF networks. Neural Processing Letters **1** (1994) 2–5