Position Control of an Omnidirectional Mobile Robot

Jannik Abbenseth
TU Darmstadt

Nicolai Ommer
TU Darmstadt

Abstract

Moving robots in the RoboCup Small Size League with fast reaction and low latency is crucial for the overall performance of a team. A good movement control is important to overcome the frictions of the wheels of the omnidirectional robots. Because modeling the friction is quite complex, we applied a learning algorithm based on Relative Entropy Policy Search for learning an optimal movement policy. Experiments in the simulation and with a real robot show that we are able to learn a policy that leads us to a desired target state.

1 Introduction

The Small Size League is the fastest league at the RoboCup with real robots, because the robots are small and build for omnidirectional fast movement. The challenge of this league in comparison to other leagues is a smart gameplay among the six robots in each team. It is important for the robots to move fast and precise, because scoring possibilities can be closed fast by defending robots. Kicking the ball must be one fluent movement, regardless of the current position.

Moving fast and precise depends on the model that is used to generate motor torques. Calculating a model analytically is complex because of different frictions, so we want to learn the model instead. The learning algorithm should find an optimal policy for a given distance to a target position and orientation.

We want to use Relative Entropy Policy Search (REPS) for learning the policy because of its fast convergence and limited loss of information.

Preliminary work. Under review by AISTATS 2012. Do not distribute.

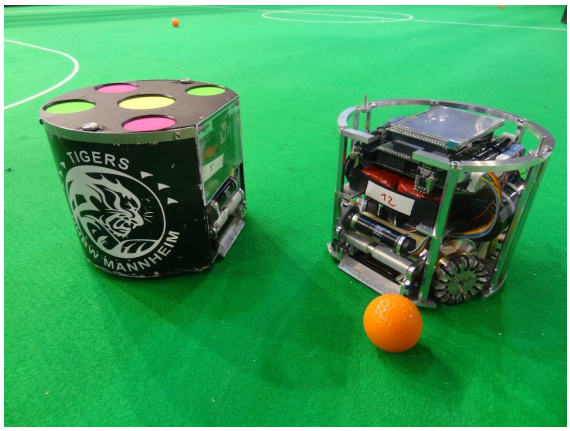

Figure 1: Two robots from Tigers Mannheim. The left robot has a colored pattern attached that is used to track the robots position, orientation and id.

2 Robot Design and Limitations

The robots that we are focusing on are from Tigers Mannheim[1], a RoboCup Small Size League team, and are shown in figure 1. The robots have a circular base with four symmetrically attached and powered wheels as shown in figure 2. Each wheel has small cross-wheels which enable the robot to move in any direction regardless of the current orientation. The drawback of this setup is a hard to model friction that occurs between the wheels and the ground.

The robots are controlled by an external computer that makes all decisions and continuously sends commands to all six robots. Both robots and computer receive global positions from an independent vision computer that tracks all objects in the field with cameras, attached over the field. The robots have colored patterns on the top as can be seen in figure 1. This means that robots know their global field position and can merge this information with their inertial sensors. With the global position, the robot can be controlled by only sending it a target position and orientation. This will reduce latencies, because positions on the robot are more current. Figure 3 shows this high level informa-

[1] http://www.tigers-mannheim.de

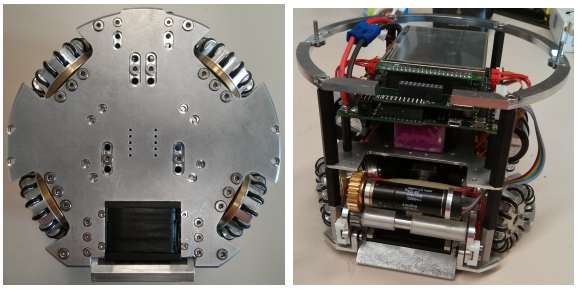

Figure 2: Left: A robots ground base with all four wheels visible. Right: The robot version that is used at the RoboCup 2014 in Brasil.

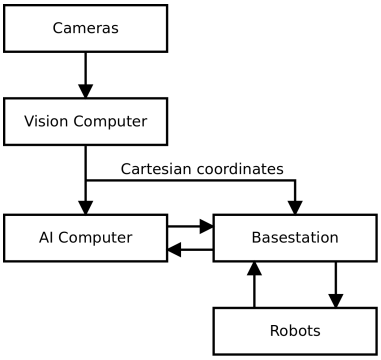

Figure 3: Data control flow in the RoboCup Small Size League. An external vision computer processes camera images and sends positions and orientations of all robots and the ball to both teams. Tigers Mannheim also send the positions to the robots via a basestation to reduce latencies.

tion flow from cameras to robot, while figure 4 shows the information flow within the robot and the possible input commands.

The robots only move on a plane 2D surface and they can move in any direction regardless of the current orientation. Additionally, they can rotate around their center. Robots can thus simply be controlled with Cartesian velocity $\dot{x}$ and $\dot{y}$, namely the position on the field, and rotation $\dot{\theta}$. Wheel velocities can be calculated from Cartesian velocities with a simple matrix multiplication as proposed in [1].

Positions and velocities are considered as the state in the learning problem. As we want to be independent of the global position, we use delta positions that indicate the distance to the desired position and rotation in the robot frame. This is visualized in figure 5. It leads us to the state vector

$$(\Delta x, \Delta y, \Delta\theta, \dot{x}, \dot{y}, \dot{\theta})^T \quad (1)$$

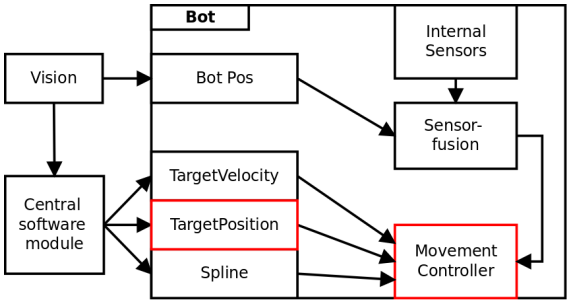

Figure 4: Information flow on the robot. The robot receives global positions from vision computer and merges it with local sensor data. It accepts velocities, positions and splines as input from the external software. The current approach is to use a PD-controller in the Movement Controller with the TargetPosition.

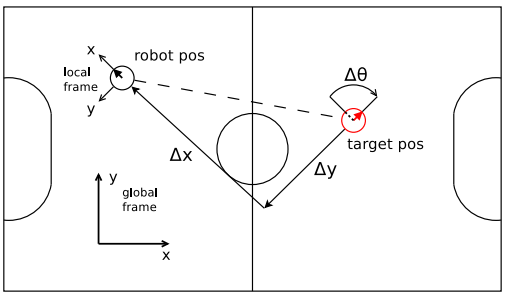

Figure 5: Visualization of the delta positions of x, y and θ and the different frames, global and local to the robot.

The policy that we want to optimize is

$$\varphi((\Delta x, \Delta y, \Delta\theta, \dot{x}, \dot{y}, \dot{\theta})^T) = (\dot{x}_d, \dot{y}_d, \dot{\theta}_d)^T \quad (2)$$

where $\dot{x}_d$,$\dot{y}_d$ and $\dot{\theta}_d$ are the desired velocities that are passed to the low level controllers on the robot after they were translated to wheel velocities as mentioned above.

The currently implemented approach is a PD-controller for each action with manually tuned parameters which are not optimal and result in overshooting and overcontrol. The aim of a learned policy is to be more precise while still staying fast and stable.

3 Optimal Policy using REPS

Our learning problem is a typical problem for Reinforcement Learning methods. For every state action pair it is possible to compute a reward value. Reinforcement Learning Methods allow us to learn a policy from random distributed state samples. Furthermore, by adding additional constraints to the basic Reinforcement Learning equations we can learn a sequence of actions instead of just one action for each

state. In the following section we will describe the Reinforcement Learning method we used for our solution of the problem and how we generalize for states that have not been covered during the learning process.

3.1 Relative Entropy Policy Search (REPS)

Compared to other Reinforcement Learning Methods the advantages of REPS are a fast convergence and limited loss of information.The limited loss of information acts as an upper bound on the policy update. The knowledge presented in the previous policy is not rejected for the succeeding policy, but is still present. This upper bound on the policy update is acquired by not only maximizing the reward but also limiting the relative entropy between two succeeding policies $\pi(a)$ and $q(a)$ for the bandit case[3].

The basic equations for policy search problems are given in (3). For limiting the relative entropy and thereby the policy update a new constraint (4) is introduced.

$$\max_{\pi} J(\pi) = \sum_{a} \pi(a) R_a, \tag{3}$$

$$s.t.\ 1 = \sum_{a} \pi(a),$$

$$\epsilon \geq \sum_{a} \pi(a) \log \frac{\pi(a)}{q(a)} \tag{4}$$

The given equations (3) and (4) only apply for problems where the system is always in the same state, or the state does not matter for the performance.

For our problem the reward of an action determined by the policy depends not only on the action itself, but also on the rewards for all future state and actions the robot will encounter. Therefore REPS maximizes the expected reward for $p(s,a) = \mu_\pi(s)\pi(a|s)$. To accomplish this infinite horizon for the reward the state distribution $\mu_\pi(s)$ must follow the steady state constraint. Therefore the probability to be in one state has to be the same as the probability to be in any other state and get to this specific state. This is expressed in (8). This limitation leads to consistency over the state distribution and the fourth constraint for REPS. Furthermore the policy to optimize is now given by $\pi(a|s)$, the transition probability by $P(s'|s,a)$ and the equations for REPS are thereby extended to (5), (6), (7) and (8).

$$\max_{p} J(p) = \sum_{s,a} p(s,a) R_s^a, \tag{5}$$

$$s.t.\ 1 = \sum_{s,a} p(s,a), \tag{6}$$

$$\epsilon \geq \sum_{s,a} p(s,a) \log \frac{p(s,a)}{q(s,a)}, \tag{7}$$

$$\forall s' :\ \mu_\pi(s') = \sum_{s,a} \mu_\pi(s)\pi(a|s)P(s'|s,a) \tag{8}$$

The values for $p(s,a)$ which are the result of REPS can be interpreted as a weighting on how good it is to execute action a in state s.

As shown in [5] the equations (5), (6), (7) and (8) for REPS for the infinite horizon case can be transcribed in a Lagrangian optimization problem

$$p(s_i,a_i) \propto q(s_i,a_i) \exp\left(\frac{h(s_i,a_i)}{\eta}\right) \tag{9}$$

with

$$h(s_i,a_i) = R_s^a + \mathbb{E}_{s_i'}\left[V(s_i')|s_i,a_i\right] - V(s_i) \tag{10}$$

Where the Lagrangian parameters $V(s)$ and η can be retrieved by minimizing the dual function (11).

$$g(\eta,V) = \eta \log\left(\frac{1}{n}\sum_{i=1}^{n} \exp\left(\frac{h(s_i,a_i)}{\eta}\right)\right) + \eta\epsilon \tag{11}$$

Equation (8) implies, that we need one Lagrangian parameter for each s which we assume is computed by

$$V(s) = \sum_i \alpha_i k(s_i,s) \tag{12}$$

where $k(\cdot,\cdot)$ is a valid kernel to compute the features of s'.

The expectation $\mathbb{E}_{s_i'}\left[V(s_i')|s_i,a_i\right]$ can be written as

$$\sum_i \left(\alpha_i \mathbb{E}_{s'}\left[k(s_i,s')|s,a\right]\right) \tag{13}$$

by substituting $V(s')$. The parameters which can be retrieved by (11) are α and η.

3.2 Model Learner

To calculate the expectation $\mathbb{E}_{s'}\left[k(s_i,s')|s,a\right]$ we need some transition model of the robot. This model cannot be created analytically due to unknown physical parameters of the robot. Nevertheless a reasonable model to predict the outcome of a given state and action can be learned. The model needs to predict the

outcome state s' given an input state s and an action a. The prediction has to be suitable for all possible combinations of s and a, regardless if the combination was explored or not. In our case the model function is computed by a ridge kernel regression[4]. An advantage of kernel ridge regression over linear ridge regression is that we don't need to know the features $\varphi(s')$ and $\psi(s,a)$ explicitly, but it is sufficient to know some basic properties of the model. In our case we assume the model to be continuous and thereby regression with a large number of Gaussian features will be able to approximate the real model. Ridge kernel regression uses the fact that every inner product of features $\varphi(x)^T\varphi(x)$ can be computed by a valid kernel $k(x,y)$, where we assume $k(\cdot,\cdot)$ to be a Gaussian kernel due to the continuity assumption. Another advantage of kernel ridge regression is its complexity, which is determined by the number of given data points instead of the number of features. The mapping of an input $\mathbf{x}$ to an output $\mathbf{y}$ is computed by (14), where $y_i = k(s_i, s')$, $x_i = k_1(s_i,s)k_2(a_i,a)$ and $K_{sa|ij} = k_1(s_i,s_j)k_2(a_i,a_j)$.

$$\mathbf{y_i} = \mathbf{y}(K_{sa} + \lambda\mathbf{I})^{-1}\mathbf{x} \tag{14}$$

3.3 Gaussian Process Policy

REPS does not provide a continues policy, but only provides a weighting of samples from a state-action distribution, which can not be used directly to control a robot. The distribution may be sparse for some states or actions and would not perform well there, so we need to create a policy out of the distribution that generalizes to missing data.

We decided to use a Gaussian Process policy for our algorithm. GPs are non-parametric and thus does not need predetermined model complexity. We do not know the friction model and expect it to be non-linear, so the GP will support us with its flexibility.
We need to control three actions. x and y are probably not too much correlated, but the rotation θ may have some influences on the other two actions, especially with higher velocities.

The existing implementation of the GP policy in the used framework is capable of dealing with multiple actions. However, using a single GP also means to have the same kernel parameters, especially exploration rate and bandwidth, for all actions, because the variance only depends on the kernel and the input data. In order to have independent exploration rates for each action, we created a multidimensional GP that wraps the original GP, but splits up the problem into independent GPs for each action and merges the results. This is shown in figure 6.

A Gaussian Process is defined by a mean and covariance function. The covariance function can be defined

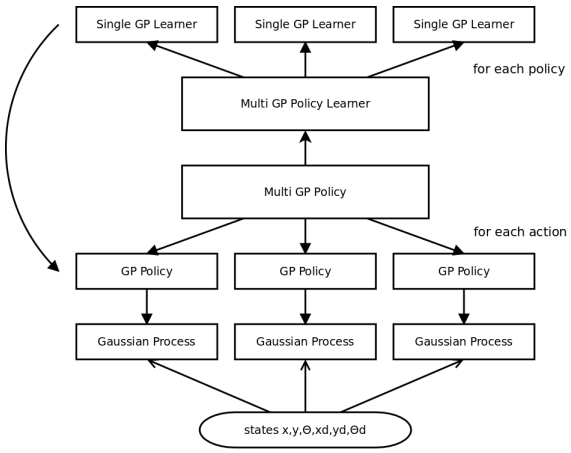

Figure 6: High level overview of the multiple independent Gaussian Process classes. The Multi-GP Policy class splits the policy up into independent single GP polices that output a single action, but get all states as input. The Multi-GP Learner class wraps single learners, so that all GP policies can be learned individually.

as $k(x_i,x_j) = \langle x_i, x_j\rangle = K_{ij}$. It can be an arbitrary kernel, so for $x, y, \dot{x}, \dot{y}$ and $\dot{\theta}$ we use an exponential quadratic kernel as defined in (15).

$$K_{expQuad} = k(x,x') = \theta_0 \exp\left(-\frac{\theta_1}{2}||x-x'||^2\right) \tag{15}$$

It uses two parameters that are to be learned. θ_0 can be seen as a scaling and θ_1 as inverse of the standard deviation σ. Instead of σ it is more intuitive to deal with a bandwidth, so in our case $\theta_1 = \frac{1}{bw^2}$.

The orientation θ is periodic, thus a periodic kernel is used for this part of the state, as defined in (16).

$$K_{periodic} = k(x,x') = \sigma^2 \exp \frac{-2\sin^2\left(\frac{\pi|x-x'|}{p}\right)}{l^2} \tag{16}$$

It uses a fixed period of $p = 2\pi$. Scale σ and bandwidth l can be learned in this case. The five exponential quadratic kernels and the periodic kernel are combined by a simple product kernel (17)

$$K_{product} = k(x,y,x',y') = k_x(x,x')k_y(y,y') \tag{17}$$

4 Kernel experiments

For the task of playing robot soccer it is important to achieve a high precision of the final position, while having a large radius of action. With exponential quadratic kernels neither the precision nor the generalization were sufficient. As figure 7 shows the policy does not fit the data well and its values decrease to zero the larger the distance to the target gets.

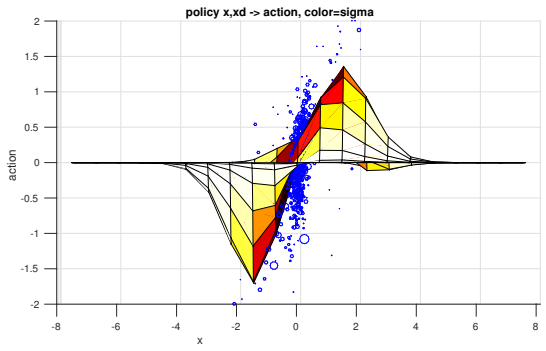

Figure 7: Policy depending on x and $\dot{x}$. The samples are shown by circles with varying diameter show the weights computed by REPS.

The activation of an exponential quadratic kernel as presented in (15) depends on the bandwidth bw and the distance between the two points x and x'. To decrease the distance of the point on which the robot settles to the target position and thereby increase the precision of the learned policy, more and narrower basis functions are needed around the target position. To get narrower basis functions one can easily decrease bw and to increase the number of basis functions one can increase the number of feature points for the policy. Both of these trivial solutions are not feasible in our case. Due to the fact that the bandwidth smooths the policy to be good for points other than the feature points, decreasing bw would lead to bad generalization. Furthermore one cannot use more feature points to compute the policy, because the policy has to be computed for each robot in real-time. Increasing the number of feature points leads to a higher dimensional matrix K and thereby to increased computation times. To solve this we came up with and evaluated two distinct solutions.

4.1 Projecting the linear state dimensions

One way to tackle the problem of increasing the precision while maintaining generality is to use some prior knowledge of the system and thinking of the problem in polar coordinates where each state is presented by a distance to the target $d = \sqrt{\Delta x^2 + \Delta y^2}$ and a direction we can assume fixed. The robots have a limited maximum velocity and we can assume that if there is a distance d_{cutoff} where the maximum action is perfect, for every distance $d \geq d_{cutoff}$ the maximum action is perfect as well. So all states where $d \geq d_{cutoff}$ can be treated the same. All other states however are still different concerning the action to choose. To implement this knowledge we proposed a projection of the position dimension by $\tilde{x} = \theta_0 \tanh(\theta_1 x)$. The similarity of two states for our problem is presented by $||\tilde{x}_1 - \tilde{x}_2||^2$. For two nearby unprojected states we can see that the projected value and thereby the similarity is inverse proportional to the gradient of the projection function. To keep the intuition of the bandwidth for exponential quadratic kernels in the area around the target, the gradient of the projection function has to be 1 in this area. It can be proven that this property is achieved by setting $\theta_1 = \theta_0^{-1}$ and thereby our projection function gets

$$\tilde{x} = \theta_0 \tanh(\theta_0^{-1} x) \tag{18}$$

Projection with (18) results in squashing the nonlinear state dimensions in an interval of $(-\theta_0, \theta_0)$.

Computing the kernel activation of an exponential quadratic kernel for a projected state and project the activation back to the real state we can see that the activation gets narrower the closer the state is to the target.

Projecting the states leads to a good generalization far from the target position, while keeping the number of feature points needed and the narrow activation close to $\mathbf{0}$.

4.2 Adding a linear part

Another approach is to decompose the policy in one subpolicy for generalization and one subpolicy to increase precision. For our task we propose a linear policy

$$K_{lin} = k(x, x') = \sigma_v^2(x)(x') + \sigma_b^2 \tag{19}$$

for generalization and a Gaussian policy for precision. To learn these two policies the data is first fitted by a plane and then only the differences between the data and the plane are modeled by Gaussian processes.

The final policy is computed by adding (19) and (15):

$$K_{sum} = K_{lin} + K_{expQuad} \tag{20}$$

σ_v represents a prior on the steepness of the kernel activation function, whereas σ_b is a prior on the offset.

A resulting policy is shown in figure 8. The policy fits the data better and for states far from the target where there is no data some reasonable action is computed.

5 Evaluation

For evaluation we first used a simple simulation in Matlab. It is described in the first section. Later we went to a physical simulator that transparently simulates the real system and finally we executed the whole system on the real system.

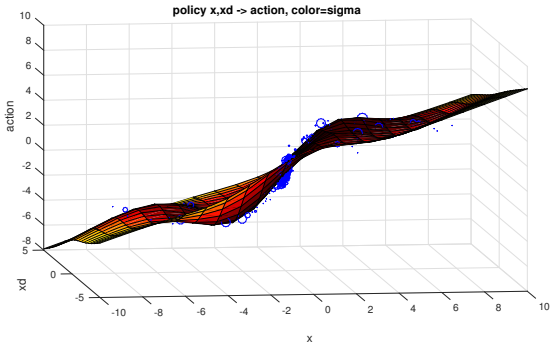

Figure 8: Policy composed of a linear and an exponential quadratic part depending on x and $\dot{x}$. The samples are shown by circles with varying diameter show the weights computed by REPS. The surface color indicates the standard deviation of the policy.

5.1 Matlab Simulation

We implemented a simulation for simple and fast evaluation of the learning methods. It uses a simplified model for the dynamics of the robots. To simulate the masses and inertias we decided to damp the actions resulting by the policy. A manually defined damping factor $\mathbf{k_p}$ reduces the occurring simulated accelerations. Each element k_{pii} damps the i-th action to consider different values for the mass and the inertia.

This results to (21).

$$\begin{pmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{pmatrix} = \begin{pmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{pmatrix} + \mathbf{k_p} \left(\begin{pmatrix} \dot{x}_d \\ \dot{y}_d \\ \dot{\theta}_d \end{pmatrix} - \begin{pmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{pmatrix} \right) \tag{21}$$

The first step of the position update is to calculate the new position vector in a straight forward way presented in (22).

$$\begin{pmatrix} x \\ y \\ \theta \end{pmatrix} = \begin{pmatrix} x \\ y \\ \theta \end{pmatrix} + dt \begin{pmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{pmatrix} \tag{22}$$

Thereafter the resulting values for x and y have to be updated according to the occurred rotation of the robot. This could be avoided by defining the state as positions and velocities of the robot represented in the target frame, which leads to increased difficulties in the first position update step due to the fact that the actions finally have to be presented in the robot frame, which is why we chose to represent the target in the robot frame. The second position update step is basically a rotation of the position vector around the rotated angle as displayed in (23) and visualized in figure 9.

$$\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} \cos(\Delta\theta) & -\sin(\Delta\theta) \\ \sin(\Delta\theta) & \cos(\Delta\theta) \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} \tag{23}$$

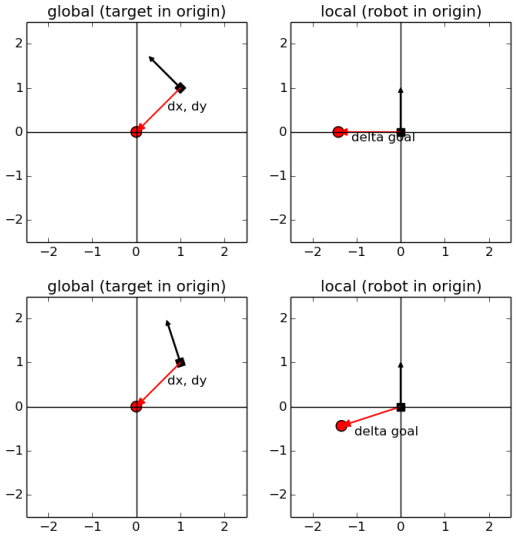

Figure 9: Schematics of robot in target frame in the left column and target in robot frame in the right column before a rotational action in the first and after the action in the second line

The simulation was implemented in Matlab and integrated into the policysearchtoolbox as provided by the IAS[2].

5.2 Real Robots

While we keep all the code for learning a policy in Matlab, we needed an interface to the software controlling the robot. As explained in section 2, all robots are connected to a central software running on an external computer. The software is roughly based on an STP model[6] where we have a skill that implements the Gaussian policy and a role (tactic) for performing sampling.

Matlab will start the learning process and write a controller file with the definition of the policy and information on how to sample to shared memory. Then it will wait until it detects a resulting file created by the controlling software. The policy trainer role will read the controller file and create a skill with a new Gaussian policy controller. The skill will use the variance of the Gaussian process for exploration, which would be switched off if it should be executed outside of learning. The sampling process will generate random positions on the field and send the robot to this state. With a certain reset probability it will cancel the rollout and start a new one by generating a new point. When the

[2] http://www.ias.tu-darmstadt.de/

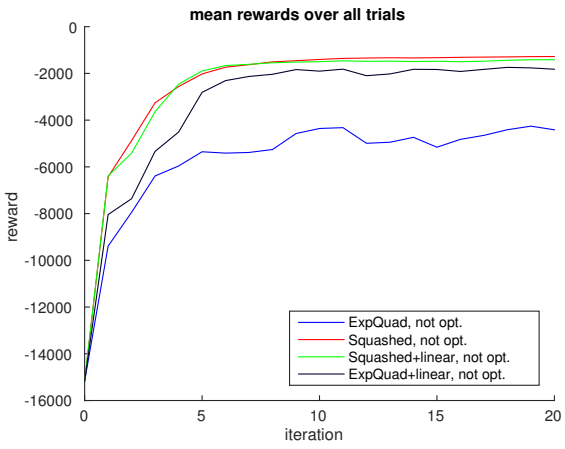

Figure 10: The mean reward over multiple trials with different initial random seeds for sampling for different evaluation configurations: exponential quadratic vs. linear state projection (squashing) and addition of these to the linear kernel. For all evaluations optimization of scale and bandwidth was disabled, only λ was optimized. Optimization lead to unstable and worse results.

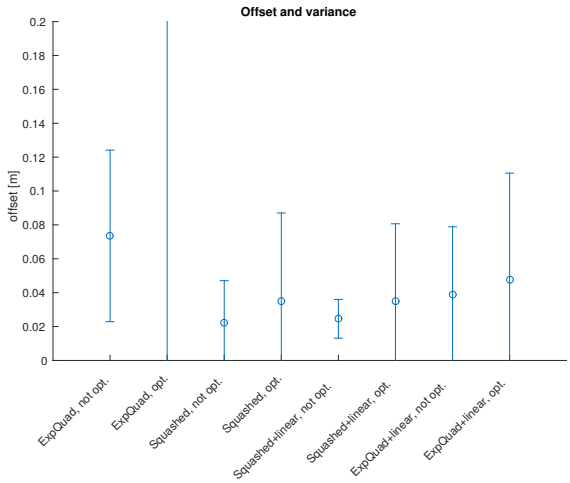

Figure 11: The plot shows the offset from the converged robot position after executing the policy to the actual target position for different learning configurations. The error bars indicate the 95% confidence interval. Learning with exponential quadratic kernels and optimizing the hyperparameters (scale and bandwidth) leads to an offset of around 2.6m.

robot reaches the field border, the rollout will also be reset and the robot is send to the center of the field with the default controller. This will be done for a given number of rollouts. States, next states and actions will be captured with a certain dt and stored in shared memory afterwards so that Matlab can proceed with the iteration.

5.3 Results from Evaluations

We evaluated the proposed changes mainly with the simulation and multiple trials with different initial random seeds to get a more reliable result. The best configuration was also evaluated on the robot. The sampling process takes much longer, so we only verified that the real system behaves similar to the simulation. Results showed that we receive comparable results with the real robot.

Figure 10 shows the rewards for evaluations with the standard exponential quadratic kernel and the squashed exponential quadratic kernel as well as both kernels in combination with a linear kernel. Each reward value is calculated by evaluating 100 rollouts that are independent from the training data and by summing up the rewards for each state/action pair. The rewards were additionally averaged over 6 trials with different initial random seeds. The figure shows that our proposed squashed kernel converges faster and results in an overall better final reward. The linear part that is added to the kernel will also increase the performance, especially for the standard exponential kernel.

While the reward is a common indicator for the performance of the learning method, it does not reflect the actual performance of the system in terms of optimality and precision. We were very interested in the precision of the policy, because as described in the first part of this paper, precision is important for robots in the RoboCup. Figure 11 therefore shows the final offset, which is the distance from the target position to the final position that the simulation converged to after executing the learned policy. The robot was not able to precisely reach the target state, regardless of the time. The figure shows the mean offset that results from simulated rollouts with random initial state and using policies from different trials. The error bars indicate the 95% confidence. The results correlate with the rewards. Using the squashed kernel results in the lowest mean offset followed by the combination with a linear kernel. The hyperparameter optimization that we used turned to be unstable for our purpose. It regularly results in large scales or bandwidth, resulting in policies that do not generalize well to new data.

One important topic with kernel policies is the hyperparameter optimization. The exponential quadratic kernel has a bandwidth and a scale as described in section 3.3. Additionally, we use a parameter for regularization λ. This parameter will increase the variance of the kernel at locations where we have less data. This is important to be able to explore in a larger range during the sample process. We want λ to be high in the beginning of the learning process and have it decreased

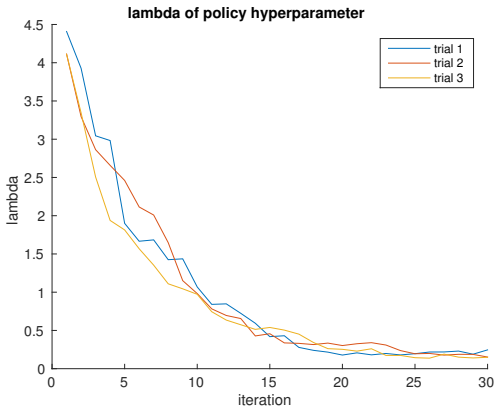

Figure 12: The typical development of the lambda hyperparameter over iterations.

to near zero over iterations. Figure 12 shows a typical development of this parameter. While it decreases over iterations as expected, it does not converge to zero though. This is an issue that we could not solve yet. It prevents us from getting more precise in a reasonable amount of iterations, because the global exploration rate is too high.

6 Conclusion and Future Work

Our learning method gives us a smooth movement policy for 2 dimensions. We learn a policy from a 4-dimensional state input to a 2-dimensional action output and the robot will approximately reach its target state. The rewards over iterations suggest that REPS works well in optimization the reward. However, comparisons with a hand-tuned linear PD-controller showed that the policy is not optimal yet, both in terms of the final offset as well as the overall execution time.

We proposed different methods to deal with these issues and could achieve some improvements. Both, the squashing kernel and the additional linear part improve the reward. We still see potential optimizations, though. The Gaussian policy generalizes well over the whole state space, but it should also fit more to the data near zero. The divergence from the data points will result in a large exploration rate for the next iteration. In order to get more precise, the exploration rate should decrease to a near zero value over iterations, but in our case it converges too early.

Executing the Gaussian policy on the robot through the central software worked without issues. The *dt* of the control cycle should not be too high to avoid excessive computation. Unfortunately, this will probably also influence the precision of the robot. If a Gaussian policy would be used productively for all robots, it could be considered to move the matrix operations to a GPU. The large matrices will be constant, so only few data transfers would be necessary. Implementing a Gaussian policy directly on the robot will most likely fail because of slow microprocessors on the robots.

Acknowledgement

We would like to thank Herke van Hoof for his patient help in understanding the topic and solving bugs in the used framework.

References

[1] R. Rojas and A. Gloye Förster. *Holonomic Control of a robot with an omnidirectional drive.* BöttcherIT Verlag, 2006.

[2] G. Neumann *Some Notes on Relative Entropy Policy Search.*

[3] J. Peters and K. Muelling and Y. Altun. *Relative Entropy Policy Search.* Proceedings of the Twenty-Fourth National Conference on Artificial Intelligence (AAAI), Physically Grounded AI Track, 2010.

[4] M. Welling. *Kernel ridge Regression* Department of Computer Science, University of Toronto, Canada.

[5] H. van Hoof and J. Peters and G. Neumann *Learning of Non-Parametric Control Policies with High-Dimensional State Features* Proceedings of the International Conference on Artificial Intelligence and Statistics (AISTATS), 2015.

[6] B. Browning and J. Bruce and M. Bowling and M. Veloso *STP: Skills, Tactics and Plays for Multi-Robot Control in Adversarial Environments* IEEE Journal of Control and Systems Engineering, 2004.

[7] D. Duvenaud *The Kernel Cookbook: Advice on Covariance functions* `http://mlg.eng.cam.ac.uk/duvenaud/cookbook/index.html`