Trajectory Kernel for Bayesian Optimization

Trajektoriekernel für Bayes'sche Optimierung
Bachelor-Thesis von Sebastian Rinder aus Sindelfingen
Februar 2018

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Trajectory Kernel for Bayesian Optimization
Trajektoriekernel für Bayes'sche Optimierung

Vorgelegte Bachelor-Thesis von Sebastian Rinder aus Sindelfingen

1. Gutachten: Riad Akrour, Ph.D.
2. Gutachten: Jan Peters, professor
3. Gutachten:

Tag der Einreichung:

Erklärung zur Abschlussarbeit gemäß § 23 Abs. 7 APB der TU Darmstadt

Hiermit versichere ich, Sebastian Rinder, die vorliegende Bachelor-Thesis ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Mir ist bekannt, dass im Fall eines Plagiats (§38 Abs. 2 APB) ein Täuschungsversuch vorliegt, der dazu führt, dass die Arbeit mit 5,0 bewertet und damit ein Prüfungsversuch verbraucht wird. Abschlussarbeiten dürfen nur einmal wiederholt werden.

Bei der abgegebenen Thesis stimmen die schriftliche und die zur Archivierung eingereichte elektronische Fassung überein.

English translation for information purposes only:

Thesis Statement pursuant to § 23 paragraph 7 of APB TU Darmstadt

I herewith formally declare that I, Sebastian Rinder, have written the submitted thesis independently. I did not use any outside support except for the quoted literature and other sources mentioned in the paper. I clearly marked and separately listed all of the literature and all of the other sources which I employed when producing this academic work, either literally or in content. This thesis has not been handed in or published before in the same or similar form.

I am aware, that in case of an attempt at deception based on plagiarism (§38 Abs. 2 APB), the thesis would be graded with 5,0 and counted as one failed examination attempt. The thesis may only be repeated once.

In the submitted thesis the written copies and the electronic version for archiving are identical in content.

Datum / Date: ______________________ Unterschrift / Signature: ______________________

Abstract

In this thesis we investigate the performance of two trajectory kernels used by Bayesian optimization to solve robotic reinforcement learning tasks. Furthermore, we contribute a trajectory kernel to a Bayesian optimization algorithm, which optimizes over a local search space. Bayesian optimization has proven to be particularly effective in the area of reinforcement learning tasks in recent years [1], but also has some disadvantages. When it comes to higher dimensional problems, Bayesian optimization does not scale well due to the search space that needs to be optimized globally. In addition, commonly used standard kernels in Bayesian optimization only detect similarities in policy parameters, but not in behaviour patterns. For this reason, we implement trajectory kernels to the Gaussian process used by Bayesian optimization. These kernels exploit trajectory data generated by the reinforcement learning agent, to derive a more precise measure than policy parameter values.

To compensate for the global nature of Bayesian optimization, we use the algorithm proposed by [2], which restricts the search space to a local area. In this thesis we show that restricting the search space enhances the learning performance in the reinforcement learning tasks of Cart Pole, Acrobot and Mountain Car. Based on our results, our trajectory kernel performs slightly better at its peak performance level compared to the standard kernels squared exponential and Matérn 5/2. The trajectory kernel adapted from [1] performs worse than the standard kernels.

Acknowledgments

First of all, I would like to thank my supervisor Raid Akrour for his guidance and support with an excellent expertise.
I would also like to thank Jan Peters, who facilitated a pleasant start at the department of Intelligent Autonomous Systems.

I thank all my friends and family for their encouragement. Especially, I would like to thank Sven Czermak, Lukas Jedrosek and my girlfriend Christina Rieth, who provided support and inspiration throughout my thesis.

Contents

Figures and Tables

List of Figures

List of Tables

Symbols and Notation

Matrices are noted as capital letters, and vectors as bold lower case letters. Scalars and constants are denoted by lower case letters.

Symbol	Meaning
t	time step of the simulation
$t_{\max}$	maximum time step of the simulation
$\mathbf{s}_t \in S$	states
$a_t \in \mathbf{a}$	actions
$p_t \in \mathbf{p}$	state transitioning probabilities
$r_t \in \mathbf{r}$	rewards
$\bar{r}$	cumulative reward, sum of all rewards from one trajectory
n_s	number of initial sample points before starting the Bayesian optimization
n_{BO}	number of total Bayesian optimization steps
n	number of training points currently present
n_*	number of sample test points
d	number of dimensions of the problem
$\mathbf{x}$	training point vector of length d
$\mathbf{x}_*$	test point vector of length d
X	$n \times d$ matrix of n points $\mathbf{x}^\top$
X_*	$n_* \times d$ matrix of n_* test points $\mathbf{x}_*^\top$
$\mathbf{y}$	vector of n evaluated objective function values
$\mathbf{y}'$	vector of standardized values of $\mathbf{y}$
$K(X,X) = K$	$n \times n$ covariance matrix
$K(X,X_*) = K_*$	$n \times n_*$ covariance between training and test points
$K(X_*,X) = K_*^\top$	same as $K(X,X_*)^\top$
$K(X_*,X_*) = K_{**}$	$n_* \times n_*$ covariance matrix
K_n	noise regarding prior covariance matrix
$\circ$	Hadamard product, element-wise product
$\mathrm{diag}(V)$	returns the diagonal elements of the square matrix V as a vector
$\mathcal{O}(n)$	big O notation - linear algorithmic performance
$\mathcal{O}(n^2)$	big O notation - quadratic algorithmic performance
σ_n^2	noise variance
σ_f^2	signal variance hyper parameter
σ_l^2	length scale hyper parameter
σ_a^2	action selection variance for continuous policies

Symbol	Meaning
ϵ	Gaussian distributed error with noise variance σ_n^2
f	objective black-box function
I	Identity matrix
$\mathbf{m}$, m	mean values of the Gaussian process from test points X_*
V	whole variance matrix of test points X_*
$\mathbf{v}$, v	variances from test points X_*
$\mathbf{s}_{\text{dv}}$	vector of standard deviations from test points X_*
$\mathbf{g}$	vector of random Gaussian distributed values with zero mean and unit variance

1 Introduction

The science of machine learning aims at creating well-functioning systems that are programmed to learn from data. This enables such systems to continuously improve their performance autonomously in order to accomplish a certain task. Machine learning is particularly important for tasks where manual design shows poor performance, such as image or speech recognition, effective web search or realization of self-driving cars. Considering the huge amount of data to be processed, machine learning algorithms easily surpass deterministic design. The advantage stems from generalizing learned examples. Accordingly, patterns in the learning data can be recognized and transferred to future situations without human supervision.

The goal of reinforcement learning, a sub-area of machine learning, is to teach an agent how to improve its behaviour [3]. In order to achieve a certain result, behaviour that contributes to the desired result is reinforced and behaviour that is not needed to achieve this result will be discouraged. The following example illustrates this. As an agent let us assume a robot arm with a table tennis racket and a ball. If our goal is to balance the ball for as long as possible, the agent will be penalized if it drops the ball, and rewarded for each successful time step of balancing. At the end of an evaluation episode all rewards received are summarized to get the cumulative reward as a measurement of performance.

The agent's behaviour can be defined by a set of policy parameters. These parameters tell the agent which action to perform depending on its current state. State parameters of the table tennis agent, for example, would contain the angle of the racket and the position of the ball. A well-tuned policy could tell the robot arm when and to what extent it should rotate in order to prevent the ball from falling. Finding such a well performing policy is the goal of our reinforcement learning task.

One approach of maximizing the performance of an agent could try random policies until finding a sufficient one. Unfortunately, this would take a very long time because it is highly unlikely to find suitable policy parameters by accident. Especially in more complex environments with higher dimensional policies. Besides the huge number of evaluations necessary, we would have to deal with wear and the limited movement speed of our robotic equipment.

For this reason, we need a more efficient search process to find properly functioning policies. A search process, which can learn from collected data, and extrapolate future behaviour. In recent years the Bayesian optimization approach has been shown to be very efficient in this field [4, 5, 6]. It tries to find an optimum of a black box function in as few steps as possible. In our reinforcement learning task the black box function would be the cumulative reward of an episode generated by a specific policy. In this case the optimum would be the policy with the highest cumulative reward achievable.

The modelling of the black box function and thus the prediction of future behaviour is based on measuring the distance between already evaluated policies with known results. The outcome of a new policy is then estimated by applying the learned distance metric. Out of many new policies Bayesian optimization selects one with the most promising enhancement. A good enhancement measure will include policies of uncharted areas in the search space and policies with high expected cumulative rewards as well. This balancing act is known as the exploration exploitation trade-off [4]. Only exploiting policies with a high expectation value can lead to a local optimum and thus to a result not as good as the best possible one. Whereas excessive exploration will cover the entire search space, but maybe will not find an optimum at all.

When using standard kernels, the distance between policies is usually measured by calculating the Euclidean distance between corresponding policy parameters [7]. A more significant measure would compare the resulting behaviour patterns stemming from certain policies instead of just comparing the parameters of the policies. Such a behaviour measurement is utilised by the trajectory kernel [1], which uses the trajectory data generated by our agent during an evaluation process. This data, consisting of state and action values, relate the respective policies for modelling the black box function. The relation of behavioural patterns has the advantage that different policies with similar results are recognized, and therefore less prioritized by the search. As a consequence, the trajectory kernel can make the search process more efficient, but also has the disadvantage that the effort for kernel computations is greatly increased.

When it comes to higher-dimensional problems, Bayesian optimization tends to explore too much because of its focus on global optimization. As a result, the optimum may not be found. Therefore, we use Bayesian optimization with a locally restricted search area [2] in addition to the commonly used global Bayesian optimization. The restriction of the search area is adjusted throughout the search process to cover the most promising parts of the search space. Consequently, we only have to optimize locally, which is more robust against high-dimensional problems and also computationally less demanding than global optimization. In order to perform this local optimization proposed by [2], a suitable trajectory kernel is also needed. Therefore we will contribute an additional trajectory kernel to the one we adapted from [1].

Accordingly, we will compare the performance of trajectory kernels with standard kernels in both global Bayesian optimization and Bayesian optimization in a local context. The robotic reinforcement learning environments will be given by the classic control tasks of Cart Pole, Mountain Car and Acrobot [3].

2 Foundations

Our reinforcement problem is described by a black-box function that only provides discrete values at the evaluation points. Bayesian optimization is well suited to solve such an optimization problem that does not supply any derivatives.

2.1 Bayesian optimization

To find the maximum point of a costly black box target function, we use Bayesian optimization [4, 5, 6]. It makes the search process more efficient by incorporating a Gaussian process model to anticipate the objective function's behaviour. This model, containing estimates of the function's returns depending on points, is used by a so called acquisition function to guide the exploration for promising new points. Each selected point is evaluated by the objective function and the results are included in the Gaussian process model. Optimally these steps are repeated until found function values converge at the maximum possible function value. In reality (algorithm 1) we iterate over n_{BO} steps and analyse the results. The change of the Gaussian process model affecting the acquisition function during Bayesian optimization iterations is illustrated in figure 2.1.

```
input  : $X = n_s$ uniformly random samples from the search space
         $n_{BO}$ = number of Bayesian optimization iterations
         set hyper parameters to 1
output: $\mathbf{x}_{n_{BO}+n_s}$
$\mathbf{y}$ = evaluations of the objective at the training points $X$
for $n = n_s$ to $n_{BO} + n_s$ do
    optimize hyper parameters (optional)
    prepare a Gaussian process model depending on $X$, $\mathbf{y}$ and hyper parameters
    $\mathbf{x}_{n+1}$ = point at the optimum of the acquisition function that uses the Gaussian process model
    $\mathbf{y}_{n+1}$ = evaluation of the objective at the point $\mathbf{x}_{n+1}$
    $X = \{X, \mathbf{x}_{n+1}\}$
    $y = \{y, \mathbf{y}_{n+1}\}$
```

Algorithm 1: Global Bayesian optimization

The optimization of hyper parameters is optional and depends on the Gaussian process kernel as mentioned in chapter 6. Also the noise level parameter σ_n (2.1) is not considered as hyper parameter in this context.

2.1.1 Gaussian Process Regression

The Gaussian process finds a prior distribution over the possible functions that are consistent with the training data. From the regression we get a posterior mean and variance, which describe our model of the objective function [7]. The mean represents a prediction of the true objective at a given point and the variance represents the uncertainty at that point. The more training points our model incorporates the smaller the variance, and the more precise the predictions, in the proximity around training points.

In real world applications we always have some noise in the objective observations. Therefore a Gaussian distributed error term,

$$\epsilon \sim \mathcal{N}(0, \sigma_n^2),$$

with zero mean and σ_n^2 variance is added to the function value. Ergo the observed target

$$\mathbf{y} = f(\mathbf{x}) + \epsilon$$

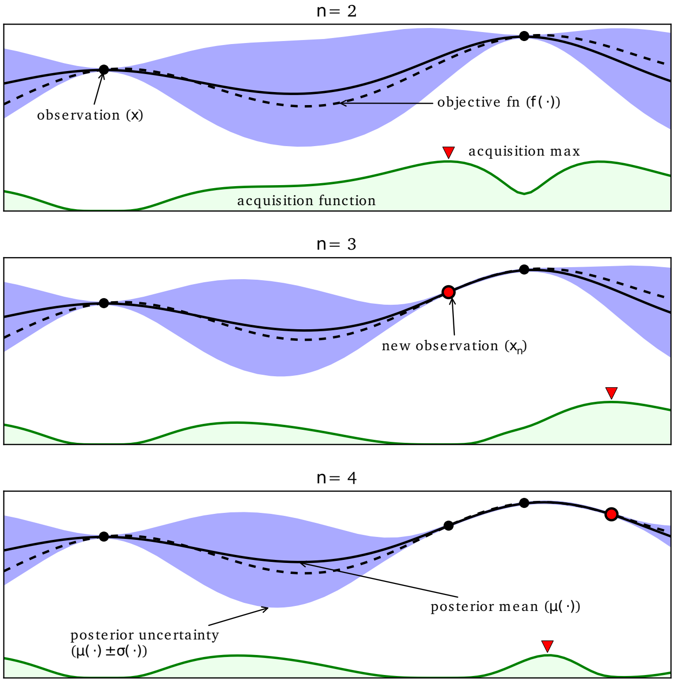


Figure 2.1: Visualization of the change in the Gaussian process model and the resulting acquisition function [4]. The x-axis contains the search space over a one-dimensional example problem. With every Bayesian optimization iteration step a new observation point at the maximum of the acquisition function is added to the training data. This new observation and its evaluation update the Gaussian process model. Thus the model becomes increasingly accurate in the vicinity of each observation. The number of observations is equal the number of iterations and is denoted by n.

regards this noise [4, 5, 6, 7]. The knowledge our training data provides is represented by the covariance matrix $K(X,X)$. With the matrix of test points X_* we get the joint distribution of the target values and the estimated function values at the test locations [7]:

$$\begin{bmatrix} \mathbf{y} \\ f_* \end{bmatrix} \sim \mathcal{N}\left(0, \begin{bmatrix} K(X,X)+\sigma_n^2 I & K(X,X_*) \\ K(X_*,X) & K(X_*,X_*) \end{bmatrix}\right).$$

For further simplification we define $K = K(X,X)$, $K_* = K(X,X_*)$, $K_*^\top = K(X,X_*)^\top = K(X_*,X)$, and $K_{**} = K(X_*,X_*)$. Now we can calculate the posterior mean and variance at given test points X_*:

$$K_n = K + \sigma_n^2 I \tag{2.1}$$

$$\mathbf{m} = K_* K_n^{-1} \mathbf{y} \tag{2.2}$$

$$V = K_{**} - K_* K_n^{-1} K_*^\top \tag{2.3}$$

$$\mathbf{v} = \mathrm{diag}(V) \tag{2.4}$$

$$\mathbf{s}_{\mathrm{dv}} = \sqrt{\mathbf{v}}. \tag{2.5}$$

The resulting vectors $\mathbf{m}$ and $\mathbf{v}$ contain the means and variances for all corresponding test points. Also we get the whole covariance matrix V.

2.1.1.1 Standard kernels

The similarity between points is measured by the covariance function. The better this covariance function is suited for our objective function the more precise is the resulting model. We use two standard kernels [7] to compare them to the trajectory kernel (section 2.2.2). In those standard kernels the distance metric for two points is given by the Euclidean distance:

$$D(\mathbf{x}_i, \mathbf{x}_j) = (\mathbf{x}_i - \mathbf{x}_j)^\top (\mathbf{x}_i - \mathbf{x}_j).$$

Which we then use in the **squared exponential kernel**,

$$K(\mathbf{x}_i, \mathbf{x}_j, \sigma_f, \sigma_l) = \sigma_f^2 \exp\left(-\frac{D(\mathbf{x}_i, \mathbf{x}_j)}{2\sigma_l^2}\right),$$

and the **Matérn 5/2 kernel**,

$$K(\mathbf{x}_i, \mathbf{x}_j, \sigma_f, \sigma_l) = \sigma_f^2 \left(1 + \frac{\sqrt{5D}}{\sigma_l} + \frac{5D}{3\sigma_l^2}\right) \exp\left(-\frac{\sqrt{5D}}{\sigma_l}\right),$$

where σ_f denote the signal standard deviation and σ_l the characteristic length scale. These so called hyper parameters can be tuned by hand or set with hyper parameter optimization.

2.1.2 Hyper parameter optimization

Selecting proper hyper parameters for the Gaussian process regression can reduce the number of objective function evaluations necessary.

The likelihood function of the Gaussian process describes the probability of the Gaussian process model being suited for the black box function. To find an optimum for the signal deviation hyper parameter σ_f and the length scale hyper parameter σ_l we maximize the log likelihood function depending on those parameters [6]:

$$\log p(\mathbf{y} = f \,|X, \sigma_f, \sigma_l) = -\frac{1}{2}\mathbf{y}^\top K_n^{-1}\mathbf{y} - \frac{1}{2}\log|K_n| - \frac{n}{2}\log 2\pi, \tag{2.6}$$

The number of observations is n, X is the $d \times n$ dataset of input points and K_n is the covariance matrix for the noisy target $\mathbf{y}$.

2.1.3 Acquisition function

The basis of the Bayesian optimization consists of selecting the next evaluation point in our search space. This point is at the optimum of the acquisition function, which depends on the current Gaussian process model. We choose expected improvement [4] during the global Bayesian optimization and in the local optimization (section 3.1) we use Thompson sampling [2]. Both acquisition functions take the mean and the variance generated by the Gaussian process model into account to guide the exploration process. The difficulty lies in avoiding excessive exploration or exploitation. Exploration seeks points with a high variance and exploitation selects points with a high mean instead. The latter one would result in a local optimum whereas too much exploration may not improve at all. An exemplary comparison of acquisition function is shown in figure 2.2.

2.1.3.1 Expected improvement

To get an expected improvement function value at a test point $\mathbf{x}_*$, we need the mean value $m(\mathbf{x}_*)$, and the standard deviation value $\mathrm{s}_{\mathrm{dv}}(\mathbf{x}_*) = \sqrt{v(\mathbf{x}_*)}$ from the Gaussian process model. Also we need the maximum of all observations $y_{\max}$ and a trade-off parameter τ. For the cumulative distribution function, we write $\Phi(.)$ and for the probability density function we write $\phi(.)$. They are both Gaussian with zero mean and unit variance. We adopt the expected improvement function

$$\mathrm{EI}(\mathbf{x}_*) = \begin{cases} (m(\mathbf{x}_*) - y_{\max} - \tau)\Phi(\mathrm{z}(\mathbf{x}_*)) + \mathrm{s}_{\mathrm{dv}}(\mathbf{x}_*)\phi(\mathrm{z}(\mathbf{x}_*)) & \text{if } \mathrm{s}_{\mathrm{dv}}(\mathbf{x}_*) > 0 \\ 0 & \text{if } \mathrm{s}_{\mathrm{dv}}(\mathbf{x}_*) = 0 \end{cases}$$

where

$$\mathrm{z}(\mathbf{x}_*) = \begin{cases} \frac{(m(\mathbf{x}_*) - y_{\max} - \tau)}{\mathrm{s}_{\mathrm{dv}}(\mathbf{x}_*)} & \text{if } \mathrm{s}_{\mathrm{dv}}(\mathbf{x}_*) > 0 \\ 0 & \text{if } \mathrm{s}_{\mathrm{dv}}(\mathbf{x}_*) = 0 \end{cases}$$

as suggested in [4]. The trade-off parameter τ can be set to zero or above to gain more exploration. To get the next evaluation point,

$$\mathbf{x}_{n+1} = \arg\max_{\mathbf{x}_*} \mathrm{EI}(\mathbf{x}_*),$$

we optimize the expected improvement function over the whole search space.

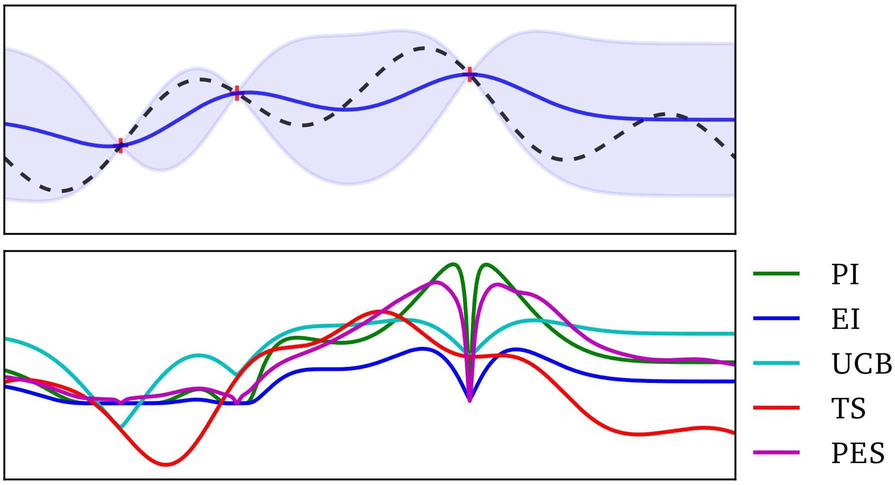


Figure 2.2: Visualization of Gaussian process posterior values corresponding to different acquisition functions [5]. The x-axis covers the search space over a one-dimensional example problem. In the upper frame the posterior mean m and double standard deviation $2\mathbf{s}_{\mathrm{dv}}$ is marked as the blue line with the shaded area. The black dashed line shows the true objective and the red crosses mark already evaluated points. The lower frame compares the plots of the resulting acquisition functions probability of improvement, expected improvement, upper confidence bounds, Thompson sampling and predictive entropy search. The peaks of probability of improvement next to values with high mean illustrates its exploitive behaviour compared to expected improvement, which takes the variance more into account. Thompson sampling is the one with the most explorative peak due to its random nature.

2.1.3.2 Thompson sampling

For Thompson sampling acquisition values we sample one function from the Gaussian process posterior [5],

$$\mathrm{TS} \sim \mathrm{GP}(0, K(X, X_*)),$$

where X is the dataset of already evaluated points, and X_* is a randomly Gaussian distributed set of points with mean and variance given by the local optimizer. These mean and variance represent our current search space. To draw function values, we need the mean vector $\mathbf{m}$ and the full covariance matrix V from the Gaussian process model. First we take the lower Cholesky decomposite of V such that $L_V L_V^\top = V$. Then we generate a vector $\mathbf{g}$, which consists of independent Gaussian distributed valus with zero mean and unit variance. Finally we get a vector TS of sampled values:

$$\mathrm{TS}(X_*) = \mathbf{m} + L_V \mathbf{g}.$$

We take the one with the highest value such that,

$$\mathbf{x}_{n+1} = \arg\max_{\mathbf{x}_*} \mathrm{TS}(\mathbf{x}_*),$$

to get the next evaluation point.

2.2 Reinforcement learning

In our reinforcement learning task we want to maximize the cumulative reward stemming from a policy with as few evaluations as feasible. We achieve this maximization by applying Bayesian optimization, treating the cumulative reward as the objective function and policies as points. The objective function evaluation consists of the simulation of a robotic agent.

2.2.1 Markov decision process

We use the Markov decision process model to describe the agent's decisions making [3]. This model consists of a tuple $(S, A, \mathbf{p}, \mathbf{r}, \gamma)$, holding all states $\mathbf{s}_t \in S$, all actions $a_t \in \mathbf{a}$, all state transitioning probabilities $p_t \in \mathbf{p}$, and all corresponding rewards $r_t \in \mathbf{r}$. The discount factor γ is set to 1 because we have a finite horizon Markov decision process where the timing of the reward is irrelevant. Otherwise γ would have a value between 0 and 1 to reduce the rewards with increasing time steps. Since the transitioning probabilities and the rewards are unknown, they are accessed through simulation runs of our environment. Assume the agent executing a policy $\mathbf{x}$ for $t_{\max}$ time steps, producing a trajectory ξ, and therefore receiving the final reward

$$\bar{r}(\xi) = \sum_{t=1}^{t_{\max}} \gamma^t r(\mathbf{s}_{t-1}, \mathbf{a}_{t-1}, \mathbf{s}_t),$$

as the sum of all immediate rewards given by a rewarding function $r(\mathbf{s}_{t-1}, \mathbf{a}_{t-1}, \mathbf{s}_t)$. The explicit probabilities corresponding to actions and state transitionings are also obtained (see 4.5) throughout the simulation run.

2.2.2 Trajectory kernel

Standard kernels like the squared exponential kernel, relate policies by measuring the difference between policy parameter values. Therefore policies with similar behaviour but different parameters are not compared adequately. The behaviour based trajectory kernel fixes this, by relating policies to their resulting behaviour [1]. This makes our policy search more efficient, since we avoid redundant search of different policies with similar behaviour.

2.2.2.1 Behaviour based measurement

For relating policies to their resulting behaviour we use the Markov decision process transitioning probabilities. We formulate the conditional probability of observing trajectory ξ given policy $\mathbf{x}$ as proposed in [1]:

$$p(\xi|\mathbf{x}) = p_0(\mathbf{s}_0) \prod_{t=1}^{t_{\max}} p(\mathbf{s}_t|\mathbf{s}_{t-1}, \mathbf{a}_{t-1}) p_\pi(\mathbf{a}_{t-1}|\mathbf{s}_{t-1}, \mathbf{x}).$$

Trajectory $\xi = (\mathbf{s}_0, \mathbf{a}_0, ..., \mathbf{s}_{t_{\max}-1}, \mathbf{a}_{t_{\max}-1}, \mathbf{s}_{t_{\max}})$ contains the sequence of state, action tuples and policy $\mathbf{x}$ a set of d parameters. The probability of starting in the initial state $\mathbf{s}_0$ is denoted by $p_0(\mathbf{s}_0)$, and $p(\mathbf{s}_t|\mathbf{s}_{t-1}, \mathbf{a}_{t-1})$ is the probability of transitioning from state $\mathbf{s}_{t-1}$ to $\mathbf{s}_t$ when action $\mathbf{a}_{t-1}$ is executed. The stochastic mapping $p_\pi(\mathbf{a}_{t-1}|\mathbf{s}_{t-1}, \mathbf{x})$ is the probability for selecting the action $\mathbf{a}_{t-1}$ when in state $\mathbf{s}_{t-1}$ and executing the parametric policy $\mathbf{x}$.

2.2.2.2 Distance metric

To examine the difference between two policies $\mathbf{x}_i$ and $\mathbf{x}_j$ the discrete Kullback-Leibler divergence

$$\mathrm{KL}(p(\xi|\mathbf{x}_i)||p(\xi|\mathbf{x}_j)) = \sum_{\xi in \xi_i} p(\xi|\mathbf{x}_i) \log \frac{p(\xi|\mathbf{x}_i)}{p(\xi|\mathbf{x}_j)}.$$

is applied to the policy-trajectory mapping probabilities $p(\xi|\mathbf{x}_i)$ and $p(\xi|\mathbf{x}_j)$ [1]. The Kullback-Leibler divergence measures how the two distributions diverge from another.

In general $\mathrm{KL}(p(\xi|\mathbf{x}_i)||p(\xi|\mathbf{x}_j))$ is not equal to $\mathrm{KL}(p(\xi|\mathbf{x}_j)||p(\xi|\mathbf{x}_i))$. But we need a symmetric distance measure. So we sum up the two divergences

$$D(\mathbf{x}_i, \mathbf{x}_j) = \mathrm{KL}(p(\xi|\mathbf{x}_i)||p(\xi|\mathbf{x}_j)) + \mathrm{KL}(p(\xi|\mathbf{x}_j)||p(\xi|\mathbf{x}_i)),$$

to achieve that $D(\mathbf{x}_i, \mathbf{x}_j) = D(\mathbf{x}_j, \mathbf{x}_i)$. An additional requirement for the kernel is the resulting matrix to be positive semi-definite and scalable[1]. Therefore we exponentiate the negative of our distance matrix D. We also apply the hyper parameters σ_f^2 to compensate for the signal variance and σ_l^2 to adjust for signal length scale. This gives us the covariance function

$$K(\mathbf{x}_i, \mathbf{x}_j, \sigma_f, \sigma_l) = \sigma_f^2 \exp\left(-\frac{D(\mathbf{x}_i, \mathbf{x}_j)}{2\sigma_l^2}\right). \tag{2.7}$$

In addition to the distance metric proposed by [1] we included the signal variance hyper parameter σ_f^2.

2.2.2.3 Estimation of Trajectory Kernel Values

To reduce the computational effort an estimation of kernel values is applied. The approximation with the Monte-Carlo estimate

$$\hat{D}(\mathbf{x}_i, \mathbf{x}_j) = \sum_{\xi \in \xi_i} \log\left(\frac{p(\xi|\mathbf{x}_i)}{p(\xi|\mathbf{x}_j)}\right) + \sum_{\xi \in \xi_j} \log\left(\frac{p(\xi|\mathbf{x}_j)}{p(\xi|\mathbf{x}_i)}\right) \tag{2.8}$$

is sufficient to measure the divergences between policies with already sampled trajectories [1]. ξ_i is the set of trajectories generated by policy $\mathbf{x}_i$. For our Gaussian process regression, we also need a distance measure between a policy with known trajectories and new policies with unknown trajectories. Since there is no closed form solution to this we use the importance sampled divergence

$$\hat{D}(\mathbf{x}_*, \mathbf{x}_j) = \sum_{\xi \in \xi_j} \left[\frac{p(\xi|\mathbf{x}_*)}{p(\xi|\mathbf{x}_j)} \log\left(\frac{p(\xi|\mathbf{x}_*)}{p(\xi|\mathbf{x}_j)}\right) + \log\left(\frac{p(\xi|\mathbf{x}_j)}{p(\xi|\mathbf{x}_*)}\right)\right]$$

to estimate the divergence between the new policy $\mathbf{x}_*$ and the policy $\mathbf{x}_j$ with already sampled trajectories ξ_j [1].
Since we only have a ratio of transitioning probabilities present in our trajectory kernel we can reduce the logarithmic term to:

$$\begin{aligned}
\log\left(\frac{p(\xi|\mathbf{x}_i)}{p(\xi|\mathbf{x}_j)}\right) &= \log\left(\frac{p_0(\mathbf{s}_0)\prod_{t=1}^{t_{\max}} p_s(\mathbf{s}_t|\mathbf{s}_{t-1}, \mathbf{a}_{t-1}) p_\pi(\mathbf{a}_{t-1}|\mathbf{s}_{t-1}, \mathbf{x}_i)}{p_0(\mathbf{s}_0)\prod_{t=1}^{t_{\max}} p_s(\mathbf{s}_t|\mathbf{s}_{t-1}, \mathbf{a}_{t-1}) p_\pi(\mathbf{a}_{t-1}|\mathbf{s}_{t-1}, \mathbf{x}_j)}\right) \\
&= \log\left(\prod_{t=1}^{t_{\max}} \frac{p_\pi(\mathbf{a}_{t-1}|\mathbf{s}_{t-1}, \mathbf{x}_i)}{p_\pi(\mathbf{a}_{t-1}|\mathbf{s}_{t-1}, \mathbf{x}_j)}\right) \\
&= \sum_{t=1}^{t_{\max}} \log\left(\frac{p_\pi(\mathbf{a}_{t-1}|\mathbf{s}_{t-1}, \mathbf{x}_i)}{p_\pi(\mathbf{a}_{t-1}|\mathbf{s}_{t-1}, \mathbf{x}_j)}\right).
\end{aligned}$$

Summing up the logarithms of the probability ratios in the end is also numerically more stable than computing the logarithm of the products.

3 Contributions

As we will see in the Experiments, the local Bayesian optimization technique is more efficient than the commonly used global one. We implement this local Bayesian algorithm developed by [2] and we also contribute a trajectory kernel suited for the Thompson sampling acquisition function used by the proposed algorithm.

3.1 Local Bayesian optimization

Modelling the objective function for a higher dimensional search space is challenging. Also global Bayesian optimization tends to over-explore. To perform a more robust optimization we use local Bayesian optimization as stated in [2]. It restricts the search space of the acquisition function to a local area which is moved, resized, and rotated between iterations. This local area is defined by a Gaussian distribution in which the mean and variance represent the centre and the exploration reach respectively. To update that mean and variance properly we minimize the Kullback-Leibler divergence between the current search distribution π_n and the probability $p_n^* = p(\boldsymbol{x} = \boldsymbol{x}^* | \mathcal{D}_n)$ of $\boldsymbol{x}^*$ being optimal. This results in a search area which neglects poorly performing regions.

To prevent the mean from moving too fast from the initial point and to avoid the variance becoming too small quickly the minimization is constrained with the hyper parameters α and β. Therefore our optimization problem is given by

$$\begin{aligned} &\underset{\pi}{\arg\min} && \mathrm{KL}(\pi || p_n^\star), && \\ &\text{subject to} && \mathrm{KL}(\pi || \pi_n) && \leq \alpha, \quad (3.1) \\ & && \mathcal{H}(\pi_n) - \mathcal{H}(\pi) && \leq \beta, \quad (3.2) \end{aligned}$$

where KL(.||.) denotes the Kullback-Leibler divergence $\mathcal{H}(p) = -\int p(\boldsymbol{x}) \log(p(\boldsymbol{x})) d\boldsymbol{x}$ is the entropy of p. We implement the local Bayesian optimization as suggested in [2].

3.1.1 Constraint Thompson sampling

To perform Thompson sampling on the variable search distribution we first sample a number of points from the current search distribution (algorithm 2). Then we compute the Mahalanobis distance for each test point to discard samples, which are at the outer edge of the search space. The Mahalanobis distance is used to regard the covariance of the current search distribution in distance computations.

```
input  : $X = n$ already evaluated points
         $X_* =$ random samples from the search distribution
         $d_t =$ distance threshold
output: $\mathbf{x}_{n+1}$: next evaluation point

compute the Mahalanobis distance from test points to current search distribution
keep samples, which are inside the distance threshold $d_t$ of the distribution density
get mean and covariance matrix from the Gaussian process
get values from Thompson sampling
$\mathbf{x}_{n+1} =$ sample at the maximum of Thompson sampled values
```

Algorithm 2: Thompson sampling acquisition for local Bayesian optimization

3.1.2 Trajectory kernel for local Bayesian optimization

If using Thompson sampling as the acquisition function, the whole covariance matrix V of test points is needed. To compute this covariance matrix, we need a distance measure between unknown trajectories. The kernel proposed by [1] does not support a measurement between unevaluated policies. We therefore implement our own distance metric based on states, which are already present in the training data. First we filter a random subset from the training data and save it to S_{sub}. For the continuous action space we then calculate each mean

$$m_i = f_s(S_{\text{sub}})^\top \mathbf{x}_i \qquad \text{and} \qquad m_j = f_s(S_{\text{sub}})^\top \mathbf{x}_j$$

for the given policies $\mathbf{x}_i$ and $\mathbf{x}_j$. The difference between the two policies is measured by the squared Euclidean distance between the means:

$$D(\mathbf{x}_i, \mathbf{x}_j) = (m_i - m_j)^\top (m_i - m_j).$$

For the discrete action space we sample the actions $\mathbf{a}$ from $p(\mathbf{a}|S_{\text{sub}}, x)$ for the given policies $\mathbf{x}_i$ and $\mathbf{x}_j$ from (4.3). The sampled actions then select the associated probability values. We compare the sets of probability values for each policy by applying the discrete Kullback-Leibler divergence

$$D(\mathbf{x}_i, \mathbf{x}_j) = \sum p(\mathbf{a}_i) \log \frac{p(\mathbf{a}_i)}{p(\mathbf{a}_j)} + \sum p(\mathbf{a}_j) \log \frac{p(\mathbf{a}_j)}{p(\mathbf{a}_i)},$$

where $\mathbf{a}_i$ denotes the actions sampled from the probabilities $p(\mathbf{a}|S_{\text{sub}}, \mathbf{x}_i)$ and $p(\mathbf{a}_i)$ the corresponding probabilities. As in the other trajektory kernel (section 2.2.2) we add the both Kullback-Leibler divergences to get a symmetric distance measure.

3.2 Numerical stability and efficiency

Due to numerical instabilities, errors may occur during calculations. These errors are caused by the discrete implementation of continuous mathematical formulas. Sometimes numerical instabilities lead to negative values for covariances. To avoid getting complex numbers we assume negative values as zero before applying the square root on covariances when computing the standard deviation from the Gaussian process model (2.5). Negative values can also occur for the distance between two trajectories, and are thresholded to zero, too.

When computing the symmetrical distance matrix $\hat{D}(X,X)$ (2.8) only the upper triangle matrix $\hat{D}_u$ is calculated. That halves the computational effort. We obtain $\hat{D} = \hat{D}_u + \hat{D}_u^\top$, since the diagonal elements of $\hat{D}$ are zero and $\hat{D}(\mathbf{x}_i, \mathbf{x}_j) = \hat{D}(\mathbf{x}_j, \mathbf{x}_i)$.

To gain robustness we use the lower Cholesky decomposition instead doing matrix inverse calculation. Therefore, the matrix to decompose must be positive definite. We achieve this by doubling the noise variance added to the diagonal elements of the original matrix as shown in algorithm 3.

```
input  : $K$, $\sigma_n$
output: $L$
$K_n = K + \sigma_n^2 I$
while $K_n$ not positive definite do
    double $\sigma_n^2$
    $K_n = K + \sigma_n^2 I$
$L$ = lower Cholesky of $K_n$
```

Algorithm 3: Lower Cholesky with variance doubling

When doing hyper parameter optimization, we do not double the noise variance. Instead our log marginal likelihood function (2.6) returns negative infinity for hyper parameters, which produce a non positive definite matrix K_n. Therefore, such hyper parameters do not come into consideration when maximizing.

3.2.1 Gaussian process regression

Since the prior covariance matrix K_n (and therefore its Cholesky decomposite L) does not change during the acquisition function optimization it can be pre-computed to make the Gaussian process regression more efficient. Additionally we customize the computation of the variance vector to lower the computational effort.

3.2.1.1 Inverse of prior covariance matrix

Instead of calculating the inverse of K_n in (2.2) we use the Cholesky decomposed matrix:

$$LL^\top = K_n$$

This is considered faster and numerically more stable [7]. The mean vector m is then computed as follows:

$$\mathbf{m} = K_n^{-1} y = (L\,L^\top)^{-1}\mathbf{y} = (L^{-\top} L^{-1})\mathbf{y} = L^{-\top}(L^{-1}\mathbf{y}) = L^\top \setminus (L \setminus \mathbf{y}). \tag{3.3}$$

The backslash operator denotes the matrix left division, so the solution $\mathbf{x} = A \setminus \mathbf{b}$ satisfies the system of linear equations $A\mathbf{x} = \mathbf{b}$. Matrix K_n must be positive definite for the Cholesky decomposition. So we double the noise variance hyper parameter σ_n^2 as shown in algorithm 3.

3.2.1.2 Variances

For the expected improvement function, we only need the vector of variances. Instead of calculating the whole covariance matrix V and taking the diagonal elements (2.4) we can take a shortcut. All elements on the diagonal of $K(X_*, X_*)$ equal σ_f^2 because the difference between one $\mathbf{x}_*$ and the same $\mathbf{x}_*$ is zero. Therefore we can write:

$$L_k = L \setminus K(X_*, X) \tag{3.4}$$

$$v = \sigma_f - \sum_{\text{rows}} (L_k \circ L_k). \tag{3.5}$$

This adaptation is inspired by [8] and reduces the computational effort drastically from $\mathcal{O}(n^2)$ to $\mathcal{O}(n)$.
For the whole covariance matrix for Thompson sampling we also avoid calculating the matrix inverse:

$$V = K(X_*, X_*) - (L_k^\top L_k)^\top$$

3.2.2 Action selection

In continuous action space our stochastic policy is Gaussian distributed (4.1). Therefore the resulting probability density of the action selection,

$$P_\pi(\mathbf{a}|\mathbf{s}, \mathbf{x}) = \frac{1}{\sqrt{2\pi\sigma_a^2}} \exp\left(-\frac{(\mathbf{a} - f_s(\mathbf{s})x)^2}{2\sigma_a^2}\right),$$

allows us to do the computations of the logarithm of the probability ratios in the trajectory kernel more efficient:

$$\begin{aligned}
\sum_{t=1}^{t_{\max}} \log\left(\frac{P_\pi(a_t|s_t, \mathbf{x}_i)}{P_\pi(a_t|s_t, \mathbf{x}_j)}\right) &= \sum_{t=1}^{t_{\max}} \log\left(\frac{\frac{1}{\sqrt{2\pi\sigma_a^2}} \exp\left(-\frac{(a_t - f_s(\mathbf{s}_t)\mathbf{x}_i)^2}{2\sigma_a^2}\right)}{\frac{1}{\sqrt{2\pi\sigma_a^2}} \exp\left(-\frac{(a_t - f_s(\mathbf{s}_t)\mathbf{x}_j)^2}{2\sigma_a^2}\right)}\right) \\
&= \sum_{t=1}^{t_{\max}} \log\left(\exp\left(-\frac{(a_t - f_s(\mathbf{s}_t)\mathbf{x}_i)^2}{2\sigma_a^2} - \left(-\frac{(a_t - f_s(\mathbf{s}_t)\mathbf{x}_j)^2}{2\sigma_a^2}\right)\right)\right) \\
&= \frac{1}{2\sigma_a^2} \sum_{t=10}^{t_{\max}} \left((a_t - f_s(\mathbf{s}_t)\mathbf{x}_j)^2 - (a_t - f_s(\mathbf{s}_t)\mathbf{x}_i)^2\right).
\end{aligned}$$

3.2.3 Hyper parameter optimization

Especially for the trajectory kernel we want a hyper parameter optimization, because all the values of the distance matrix D may get very big. When dividing by a well tuned hyper parameter σ_l before applying the exponential function (2.7), we avoid getting a diagonal matrix K or a zero matrix K_*.

When calculating $\log(|K_n|)$ for the hyper parameter optimization (2.6), again we use the Cholesky decomposition of K. Thus the determinant transforms to

$$|K_n| = |L\,L^T| = |L|\,|L^T| = |L|\,|L| = |L|^2.$$

Since the determinant of the Cholesky decomposed matrix,

$$|L| = \prod_i L_{ii},$$

is the product of its diagonal elements, we can transform this into a numerically more stable version:

$$\log(|K_n|) = \log(|L|^2) = 2\,\log(|L|) = 2\,\log(\prod_i L_{ii}) = 2\sum_i \log(L_{ii}).$$

The computation of $K_y^{-1}\mathbf{y}$ in (2.6) is done by the same method we already use in the Gaussian process (3.3).
For debugging purposes and for estimating the quality of the hyperparameter optimization, we plotted a grid of values generated by the log marginal likelihood function during development and testing. Figure 3.1 contains a very stable example of the hyper parameter space.

3.2.3.1 Independent log-normal prior

The log marginal likelihood maximization will sometimes succeed at the borders of our search space resulting in very high high or very low hyper parameters. We prevent this by adding an independent log-normal prior term as introduced in [6]:

$$\sum_{i=f,l}\left(\frac{-\log(\sigma_i)^2}{2\cdot 10^2} - \log\left(10\sqrt{2\pi}\right)\right).$$

Since the suggested prior is centred at the point of origin with standard deviation 10, but we want our search space to adapt between iterations, we make the mean and the standard deviation adjustable. We write:

$$\sum_{i=1,2}\left(\frac{-(h_i-c_i)^2}{2(b_{u_i}-b_{l_i})^2} - \log\left((b_{u_i}-b_{l_i})\sqrt{2\pi}\right)\right),$$

where b_l and b_u denote the lower and upper bounds of the search space and c its center. And since we optimize over the logarithmic space $h_1 = \log\sigma_f$ and $h_2 = \log\sigma_l$.

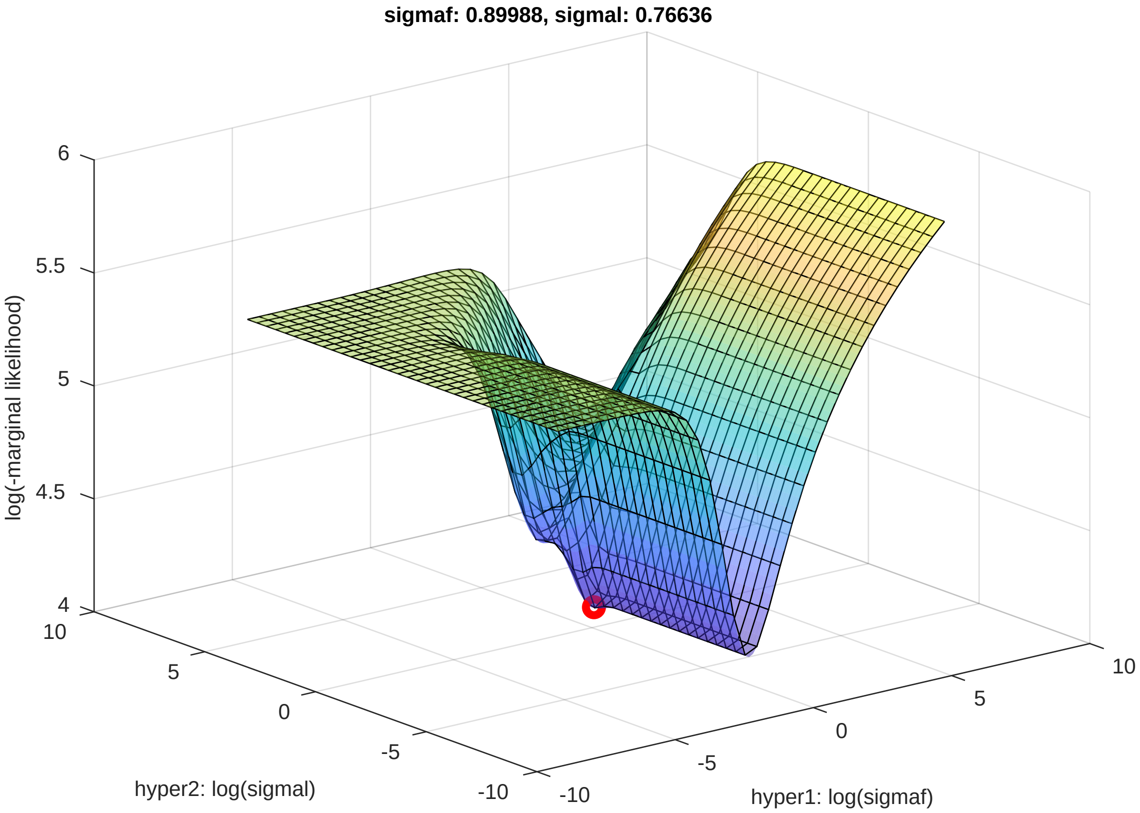


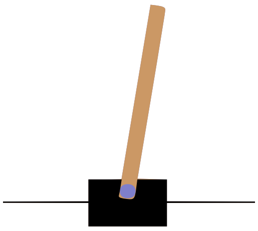
Figure 3.1: Exemplary hyper parameter minimization plot with the optimum marked as the red circle.

4 Experiments

Each of the classic control environments has its unique features described below. For the MATLAB Cart Pole implementation we use the starting conditions, the rewarding function and boundary parameters from the OpenAI Gym [9].

4.1 Cart Pole

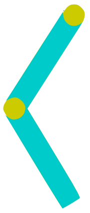
The Cart Pole environment consists of a cart with a pole attached to its top. The cart is accelerated to the left or the right to balance the pole for as long as possible. An episode starts with small random state values and it ends when the maximum of time steps or a specific state is reached. The state values contain the position and the velocity of the cart, and the angle and the angular velocity of the pole. Each of these four state values are set uniformly random between -0.05 and 0.05 at the beginning. The episode ends after 200 time steps or if the pole falls below 12 degrees relative to the vertical. It also ends when the cart is more than 2.4 meters away from the center. For each completed time step a reward of 1 is returned. The cart pole implementation from OpenAI Gym ('CartPole-v0') accepts a discrete action value, -1 or 1, for either applying force to the left or to the right. With the four state values and an additional bias value we have five dimensions for each discrete action resulting in a total of ten dimensions. In our own cart pole simulation, we only use the four state parameters for calculating an continuous action value between -1 and 1. It turned out that adding a bias value would make the policy search slightly more difficult here. Therefore, this policy has only four dimensions. Also we let an episode end after a maximum of 1000 timesteps.



OpenAI Gym Cart Pole

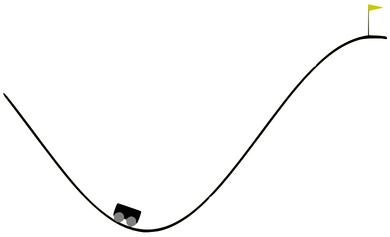
4.2 Acrobot

The Acrobot consists of two joints and two pendulums. The upper joint has a fixed position. It connects to the first pendulum, which is attached to the second joint. A torque can be applied to that second joint, which controls the second pendulum. The pendulums start in equilibrium position and the goal is to gain enough momentum and swing the end of the second pendulum above a certain mark. If the Acrobot passes that mark that is one pendulum length above the upper joint, the episode ends. A reward of -1 is received for every timestep needed. The angle between vertical and the first pendulum and the angle between the two pendulums form the six state parameters. Four parameters are the sine and the cosine of these two angles each, whereas the last two parameters are the angular velocities 4.1. The discrete action can have the values -1, 0 or 1. Each value stands for the amount of torque applied to the joint between the two pendulum links. With the six state values, an additional bias value, and three discrete action possibilities we get 21 dimensions for the Acrobot policy. We use the OpenAI Gym implementation of Acrobot ('Acrobot-v1').



OpenAI Gym Acrobot

4.3 Mountain Car



OpenAI Gym Mountain Car

During the Mountain Car task, an underpowered car tries to drive uphill. It can only reach the goal on the right side if it uses both hills to gain momentum. The end of the left hill is an inelastic wall, so if the car hits it, the velocity is set to zero. The reward for the discrete version of Mountain Car is -1 for every time step to the goal. In the continuous setup the reward starts at 100. After an episode the squared count of performed actions is subtracted. Since the hills are formed by a sine curve, it is sufficient that the position of the car is only one-dimensional. The vertical position of the vehicle can be easily derived from the horizontal position with the sine function. The horizontal position and the car's velocity are cubically expanded (table 4.1) to get nine state values. Also a bias parameter is added as a tenth state value. In the continuous setting, we therefore receive ten dimensions. In the discrete setting we have three action values for either applying acceleration to the left or right (-1,1) or doing nothing (0). In this case we receive 30 dimensions. We use the OpenAI Gym implementation of the continuous Mountain Car task ('MountainCarContinuous-v0') and the discrete one ('MountainCar-v0').

Environment	Description	Action	State feature	d	maximum time steps
Cart Pole	p: cart position θ: pole angle	continuous: [-1,1]	$(p, \dot{p}, \theta, \dot{\theta})$	4	1000
		discrete: {-1,1}	$(p, \dot{p}, \theta, \dot{\theta}, 1)$	10	200
Acrobot	θ_1: angle of pendulum 1 θ_2: angle between pendulum 1 and 2	discrete: {-1,0,1}	$(\cos(\theta_1), \sin(\theta_1), \cos(\theta_2), \sin(\theta_2), \dot{\theta}_1, \dot{\theta}_2, 1)$	21	500
Mountain Car	p: horizontal car position u: car velocity	continuous: [-1,1]	$(p, u, p^2, u^2, pu, p^2u, pu^2, p^3, u^3, 1)$	10	999
		discrete: {-1,0,1}	$(p, u, p^2, u^2, pu, p^2u, pu^2, p^3, u^3, 1)$	30	200

Table 4.1: Environment parameters of the experiments. Column d contains the number of dimensions of each problem.

4.4 Implementation of the environments

Because we developed the Bayesian optimization algorithm in MATLAB, we first implemented a MATLAB version of Cart Pole that was adapted from [10]. Then we also implemented the OpenAI Gym [9] to provide a wide range of problem environments. To use the simulations written in Python we prepared a python module which is imported to MATLAB. After loading the python module with `py.importlib.import_module(moduleName)` we can call every method it contains via the `py.moduleName.` prefix. It is vital to convert our MATLAB data correctly before calling the python subroutine with it. The MATLAB variables containing whole numbers are converted from `Double` to `Int`. Also all the vectors received by the python module have to be converted to an array through `numpy.asarray()`.

To cut the time for experiment runs we vectorized every suited operation to speed up calculations in MATLAB. In addition, we tuned the code to run on a parallel pool so we could run experiments on the cluster efficiently. Unfortunately, calling python modules does not work if we use the parallelization of MATLAB. So we used the OpenAI Gym environments only with the less computationally demanding local Bayesian optimization.

4.5 Action selection

In continuous action space we use a linear policy to action mapping

$$a = f_s(\mathbf{s})^\top \mathbf{x} + \sigma_a, \tag{4.1}$$

with a small Gaussian noise σ_a needed for stochastic policies. So the Gaussian distributed actions,

$$a \sim \mathcal{N}(f_s(\mathbf{s})\mathbf{x}, \sigma_a^2),$$

result in the probability measure by the Gaussian density:

$$p_\pi(a|\mathbf{s}, \mathbf{x}) = \frac{1}{\sqrt{2\pi\sigma_a^2}} \exp\left(-\frac{(a - f_s(\mathbf{s})\mathbf{x})^2}{2\sigma_a^2}\right). \tag{4.2}$$

In discrete action space environments we use a parametric soft-max action selection policy:

$$p(a|\mathbf{s}, \mathbf{x}_a) = \frac{\exp(f_s(\mathbf{s})^\top \mathbf{x}_a)}{\sum_{a \in A} \exp(f_s(\mathbf{s})^\top \mathbf{x}_a)}. \tag{4.3}$$

Again it consists of a linear mapping $f_s(\mathbf{s})^\top \mathbf{x}_a$. The resulting action is sampled from the probability of action a given state $\mathbf{s}$. The policy $\mathbf{x}_a$ for a discrete action a is a subset of parameters from the policy $\mathbf{x}$.
The state feature function $f_s(\mathbf{s})$ depends on the environment as listed in Table 4.1.

4.6 Initial settings

For the trajectory kernel, we proposed in section 3.1.2, a maximum of 500 random states as the subset S_{sub} are used.
For the continuous action selection the noise parameter σ_a is set to 10^{-3}. We adapted the action selection strategies from [1]. Unfortunately, they do not provide any parameters. Therefore this error deviation value was guessed after a few test runs.
The noise variance σ_n^2 is set to 10^{-8} for all performance tests.

4.6.1 Global context

In the global search context the policy parameter boundaries are set to -10 and 10 for each dimension resulting in a d dimensional hypercubic search space.
For the Bayesian optimization we select the initial samples set X_n from a bunch of sample sets such that the Euclidean distance between points of the selected set is maximized. This grants us a well covered search space as a starting condition. We also set the number of initial samples to $n_s = 10$ and the iteration count to $n_{BO} = 200$.
In the expected improvement function, the trade-off parameter τ is set to 0.01 as proposed by [4].
The hyper parameter optimization runs every fifth Bayesian optimization step or if the found maximum of the expected improvement function drops below 10^{-6}. In test runs this threshold near zero was a good indicator for poorly suited hyper parameters.

4.6.2 Local context

The Bayesian optimization in the local context starts with a hypersphere centred at the origin with radius 10 as the policy search space boundary. The starting point for the search is set to the origin accordingly.
We run the hyper parameter optimization every time the search space is adjusted. In total we carry out 400 Bayesian optimization steps, in which the search space is adapted every four steps.
Before doing Gaussian process regression we transform our observations to zero mean and uniform variance:

$$\mathbf{y}' := \frac{\mathbf{y} - \text{mean}(\mathbf{y})}{\text{std}(\mathbf{y})}.$$

This standardization proposed by [2] affects the Thompson sampling exploration exploitation trade-off. In algorithm 2 we only use 300 samples for the Thompson sampling due to its quadratic computational effort. The distance threshold d_t for the sample filtering during Thompson sampling is set to 80%.

5 Results

We plot the mean and the standard deviation of the cumulative reward for all kernel runs. One kernel run consists of a number of trials depending on the experiment. We use more than one trial per kernel to compensate for the variable result one run can produce. Also, due to high variations between iteration steps, all plots are smoothed by a moving mean of 15 steps. And the standard deviation values are divided by five to maintain readability when three or four different kernel plots are present in one graph. Each plot includes the mean of every kernel's outcome as a line and the corresponding standard deviation as a shaded area above and below. The green and blue graph show the two standard kernels, squared exponential and Matérn 5/2, and the magenta and red graph show the trajectory kernels. The trajectory kernel proposed by [1] is marked as the magenta one, whereas the trajectory kernel we come up with (Section 3.1.2) is marked as the red one. Also a table is provided for each experiment showing the benchmark data for corresponding kernels. The column **mean(time)** contains the mean value of the minutes needed for each trail and the column **std(time)** the associated standard deviation. The column **time performance** relates the time needed for the whole run to the time needed for the squared exponential kernel run. If the percentage is negative the associated kernel took more time than the squared exponential kernel. The last column, **learning performance**, compares the summarized cumulative rewards to those of the squared exponential kernel. If the percentage is positive then the overall learning performance is better than the squared exponential kernel ones. The discrete version of the OpenAI Gym Mountain Car ('MountainCar-v0') stayed at the lowest possible reward during all experiments, therefore we plot no results.

Environment	Optimizer	Action space	Kernels	Simulation
Cart Pole	globally	continuous	S, M, TW, TO	MATLAB (cluster)
Cart Pole	locally	continuous	S, M, TO	MATLAB (cluster)
Cart Pole	locally	discrete	S, M, TO	OpenAI Gym (laptop)
Acrobot	locally	discrete	S, M, TO	OpenAI Gym (laptop)
Mountain Car	locally	continuous	S, M, TO	OpenAI Gym (laptop)

Table 5.1: Overview of all performance experiments. The abbreviations in the column **kernels** stand for: S = squared exponential, M = Matérn 5/2, TW = trajectory (Wilson 2014), TO = trajectory (own work). The part of the TU Darmstadt cluster we used for parallel runs is equipped with Intel Xeon E5-4650 processors with 2.7 GHz each. The runs involving the OpenAI Gym were performed on my laptop's Intel Core i5-7200U processor operating at 2.5 GHz.

5.1 Learning performance

Here we plot the learning performance of each experiment. We also provide tables with corresponding benchmark data.

5.1.1 Cart pole

This subsection contains the results of the different Cart Pole setups.

5.1.1.1 MATLAB implementation of Cart pole on the global optimizer

In figure 5.1 we plot four different kernel runs of the MATLAB CartPole implementation in the global Bayesian optimization context. Our MATLAB CartPole implementation runs at most until 1000 time steps. For this implementation we use a continuous action selection policy with four dimensions. Ten trials were completed on each kernel to observe the average performance.

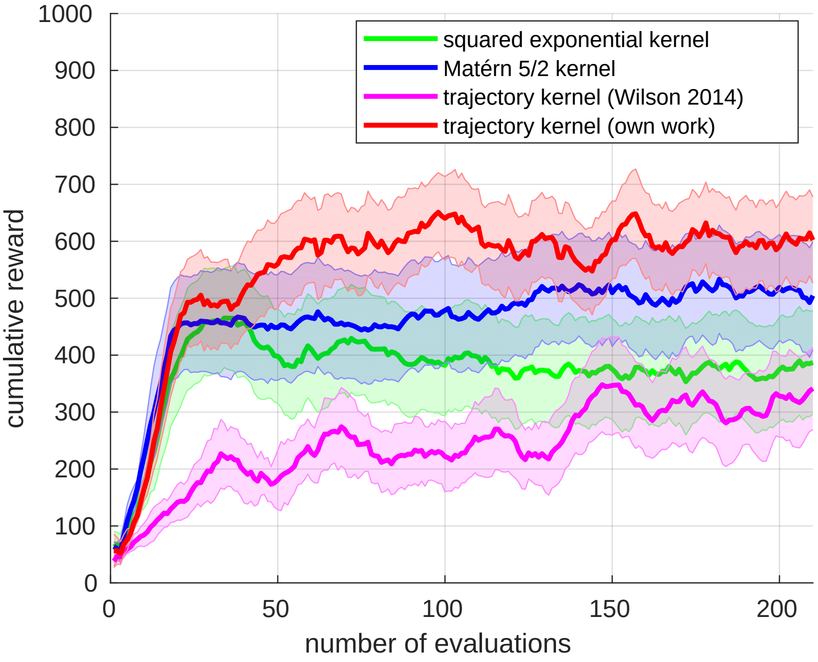


Figure 5.1: Results of the MATLAB Cart Pole implementation with a continuous action selection policy with the global optimization. Each kernel graph contains the average over 10 trials.

kernel	mean(time)	std(time)	time performance	learning performance
squared exponential	17 min	5 min	0 %	0 %
Matérn 5/2	18 min	4 min	-5 %	+23 %
trajectory (Wilson 2014)	150 min	48 min	-769 %	-35 %
trajectory (own work)	393 min	51 min	-2187 %	+47 %

Table 5.2: Performance of different kernels on the MATLAB Cart Pole implementation with the global optimizer.

5.1.1.2 MATLAB implementation of Cart pole on the local optimizer

In figure 5.2 we plot three different kernel runs of the MATLAB CartPole implementation in the local Bayesian optimization context. For this implementation we use a continuous action selection policy with four dimensions. Ten trials were completed on each kernel to observe the average performance.

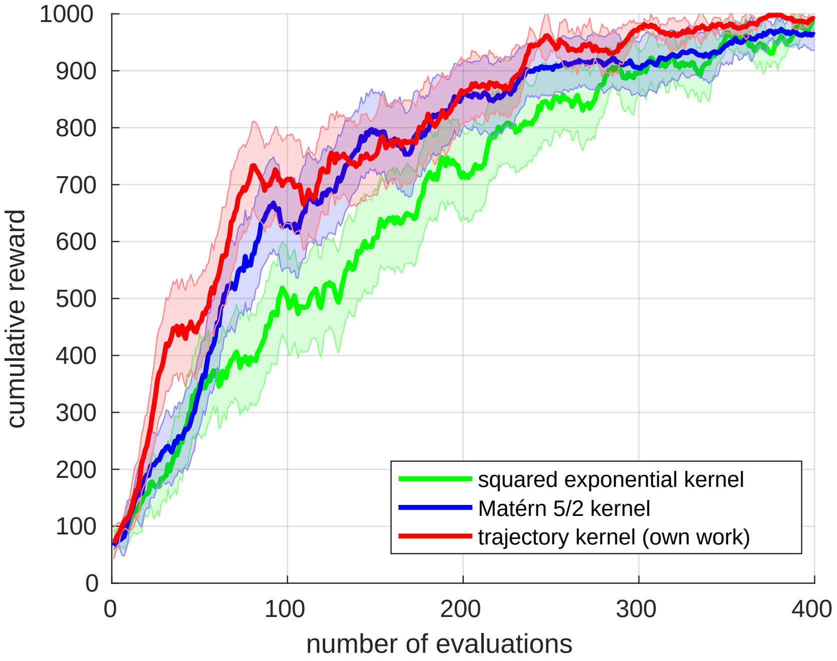


Figure 5.2: Results of the MATLAB Cart Pole implementation with a continuous action selection policy on the local optimizer. Each kernel graph contains the average over 10 trials.

kernel	mean(time)	std(time)	time performance	learning performance
squared exponential	9 min	1 min	0 %	0 %
Matérn 5/2	10 min	1 min	-13 %	+11 %
trajectory (own work)	30 min	1 min	-235 %	+18 %

Table 5.3: Performance of different kernels on the MATLAB Cart Pole implementation on the local Bayesian optimizer.

5.1.1.3 OpenAI Gym implementation of Cart pole on the local optimizer

In figure 5.3 we plot three different kernel runs of the OpenAI Gym CartPole implementation in the local Bayesian optimization context. For this implementation we use a discrete action selection policy with five dimensions for two possible actions resulting in a total of ten dimensions. Ten trials were completed on each kernel to observe the average performance.

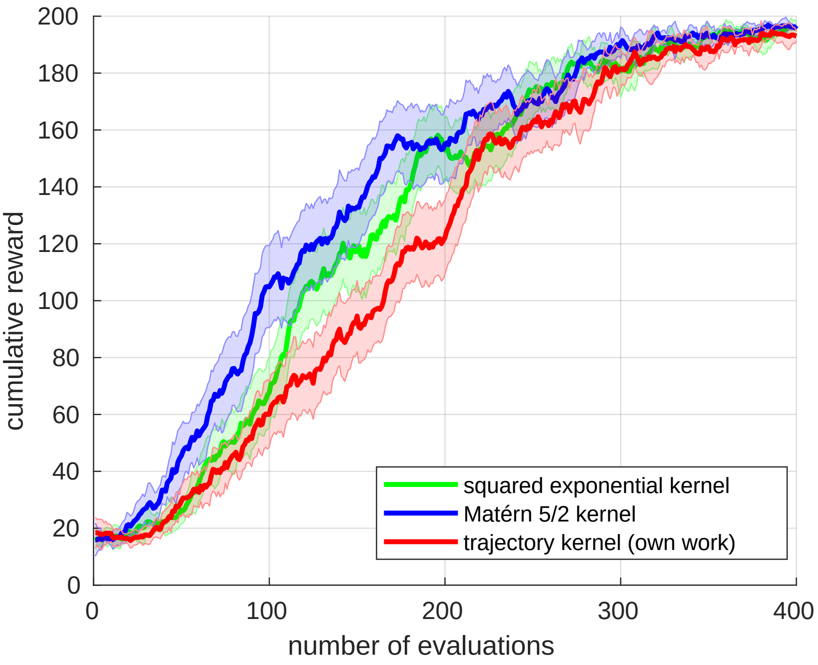


Figure 5.3: Results of the OpenAI Gym Cart Pole implementation with a discrete action selection policy on the local optimizer. Each kernel graph contains the average over 10 trials.

kernel	mean(time)	std(time)	time performance	learning performance
squared exponential	6 min	1 min	0 %	0 %
Matérn 5/2	9 min	1 min	-37 %	+7 %
trajectory (own work)	62 min	3 min	-861 %	-8 %

Table 5.4: Performance of different kernels on the OpenAI Cart Pole on the local optimizer.

5.1.2 Acrobot

In figure 5.4 we plot three different kernel runs of the OpenAI Gym Acrobot implementation in the local Bayesian optimization context. For this implementation we use a discrete action selection policy with seven dimensions for three possible actions resulting in a total of 21 dimensions. Ten trials were completed on each kernel to observe the average performance.

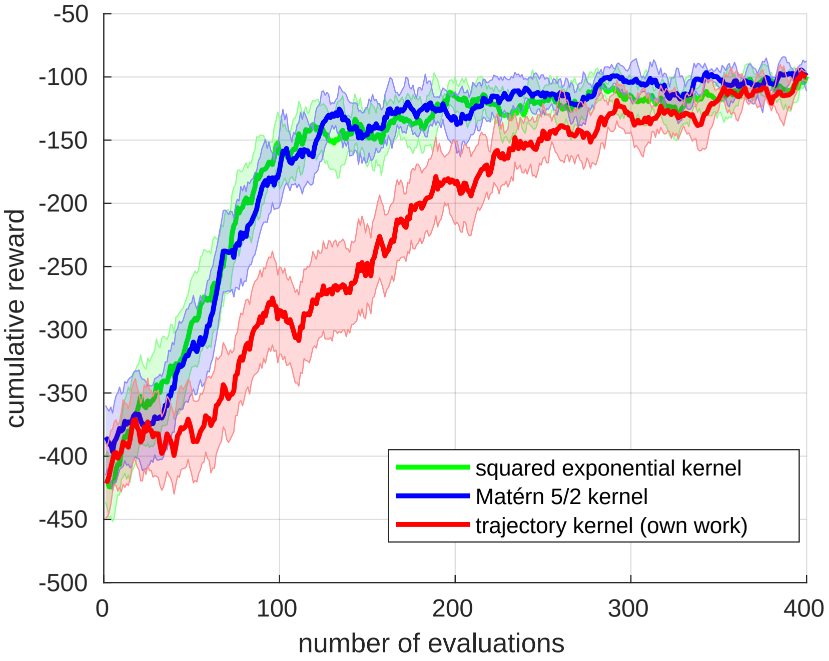


Figure 5.4: Results of the OpenAI Gym Acrobot implementation with a discrete action selection policy on the local optimizer. Each kernel graph contains the average over 10 trials.

kernel	mean(time)	std(time)	time performance	learning performance
squared exponential	7 min	0 min	0 %	0 %
Matérn 5/2	8 min	1 min	-12 %	+1 %
trajectory (own work)	70 min	5 min	-923 %	-32 %

Table 5.5: Performance of different kernels on the OpenAI Gym Acrobot on the local optimizer.

In figure 5.5 we plot three different kernel runs of the OpenAI Gym Mountain Car continuous implementation in the local Bayesian optimization context. For this implementation we use a continuous action selection policy with ten dimensions. Ten trials were completed on each kernel to observe the average performance.

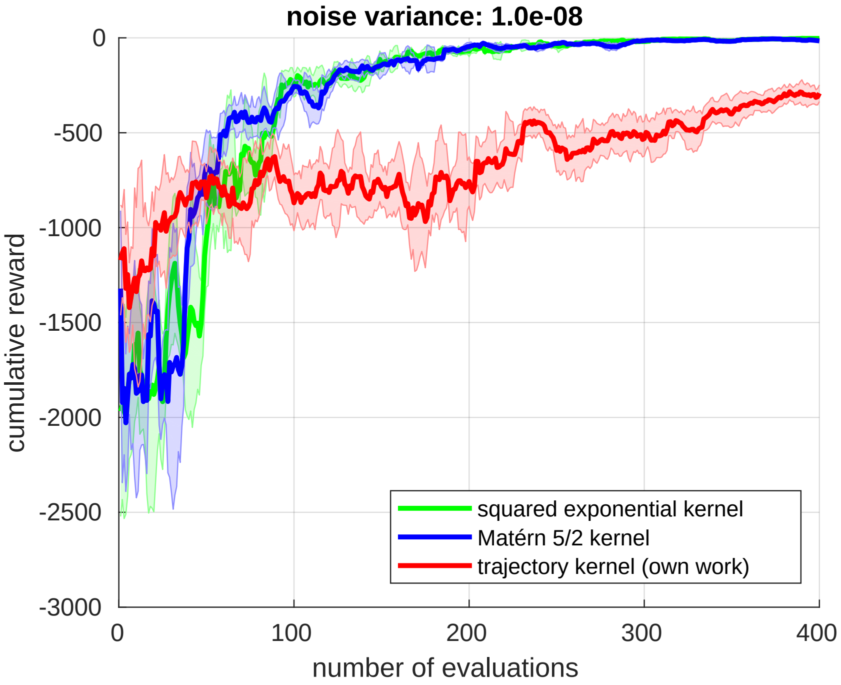


Figure 5.5: Results of the OpenAI Gym Mountain Car continuous implementation with a continuous action selection policy on the local optimizer. Each kernel graph contains the average over 10 trials.

kernel	mean(time)	std(time)	time performance	learning performance
squared exponential	1 min	0 min	0 %	0 %
Matérn 5/2	1 min	0 min	-25 %	+10 %
trajectory (own work)	101 min	6 min	-12728 %	-103 %

Table 5.6: Performance of different kernels on the OpenAI Gym Mountain Car continuous on the local optimizer.

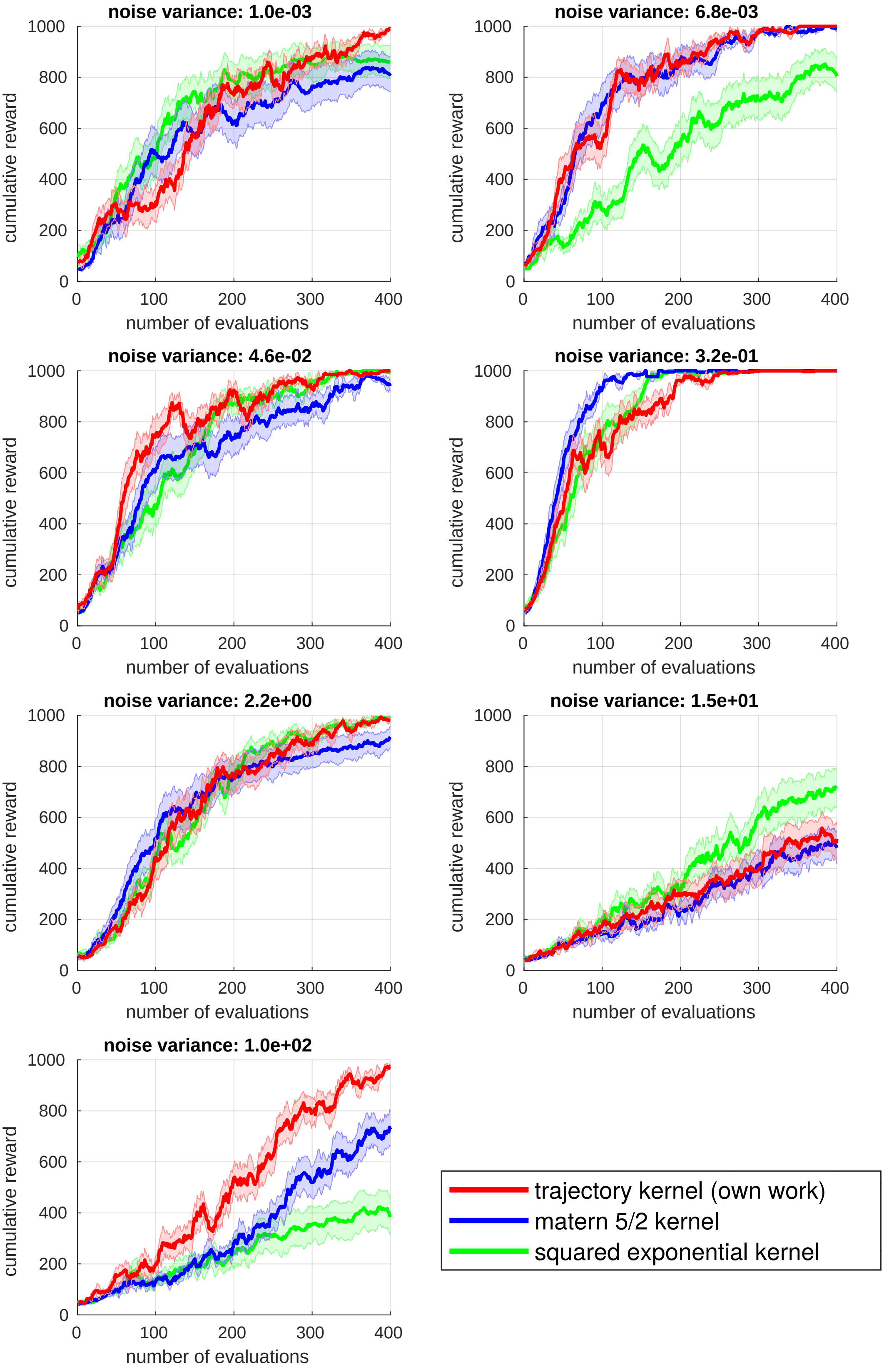


Figure 5.6: Noise level comparison on the MATLAB Cart Pole implementation with 1000 time steps on the local Bayesian optimizer. The noise levels are logarithmically spaced and sorted in ascending order from 10^{-3} to 10^{2}. Each kernel graph contains the average over 5 trials.

6 Discussion and Conclusion

6.1 Learning performance

The continuous Cart Pole MATLAB experiment on global Bayesian optimization (figure 5.1) shows a learning performance advantage of our trajectory kernel. It finds well performing policies faster than the other kernels. The Wilson kernel performed far worse than the others. The other kernels have a promising start, but after 20 iterations they stop improving constantly. Instead they find very good and very bad policies, resulting in quite constant mean with a high deviation. This could be an issue of the expected improvement acquisition function, exploring too much after a while. Unfortunately, none of the kernels converges during the 200 Bayesian optimization steps at the maximum possible cumulative reward of 1000. As listed in Table 5.2 a single trial for our trajectory kernel takes almost seven hours on the cluster on average. Therefore we only did 200 iteration steps for the very time consuming runs with the global Bayesian optimization.

The continuous Cart Pole MATLAB experiment on local Bayesian optimization (figure 5.2) shows a slight performance advantage of our trajectory kernel. Each kernel converges at the maximum after about 400 black-box function evaluations. Presumably due to the well suited Thompson sampling acquisition function in the local optimization context and the autonomously adjusting local search space. Comparing it with the same experiment (figure 5.1), the local optimization clearly outperforms the global one.

The discrete Cart Pole OpenAI Gym experiment on local Bayesian optimization (figure 5.3) shows a slight performance disadvantage of our trajectory kernel. In this experiment we have a maximum of 200 time steps per episode, and therefore the highest possible cumulative reward is 200.

The discrete Acrobot OpenAI Gym experiment on local Bayesian optimization (figure 5.4) converges at roughly -100. The Acrobot has no definition of solving, but reaching the goal in 100 time steps is fairly good. Our kernel performs worse than the standard kernel in the beginning but catches up towards the end.

The continuous Mountain Car OpenAI Gym experiment on local Bayesian optimization (figure 5.4) shows a major performance drawback compared to the standard kernels. But the result is not conclusive because the environment did not solve. The received cumulative rewards on the best runs converge at zero, which means that the agent learns to do nothing. This is due to the rewarding function that discourages any action taken. A successful continuous Mountain Car run would receive a reward around 90. In this experiment we also do not use any hyper parameter optimization, because it fails constantly. In this case we set σ_f and σ_l to 1.

6.2 Model parameters

Due to the high variations in the plots, we divide the standard deviations by five before visualizing them in order to maintain the clarity of the graph. Not only the variance between trials is high, but also between iteration steps. Therefore we apply a moving mean of 15 steps on each plot too enhance readability. Maybe these high variances are an indicator for poorly selected model parameters. As shown exemplary in figure 5.6 tuning of the noise level parameter σ_n can have a huge impact on the results. To show this impact we did some continuous Cart Pole runs with different noise level parameters.

When computing the continuous action space trajectory distance metric, proposed by [1], hyper parameter optimization is mandatory to compensate for high distance values. These values occur because of the Gaussian distribution density used as the probability measure for actions (4.2). Since we exponentiate the negative of the distance values, very high distance values will result in covariance values that are zero. The optimization of the scale length hyper parameter σ_l will prevent all-zero covariance matrices. Experiments have shown that tuning the signal deviation parameter σ_f is also helpful, because we simulate a policy only once for a result. Since a policy will not produce the exact same result after repeated evaluations, we have to assume a high variation on each result. This variation is regarded by σ_f. For consistency we do the hyper parameter optimization on each kernel.

6.3 Conclusion

The results show that trajectory kernels can have some advantage if computation time is not expensive compared to evaluations including robotic movement. Unfortunately, we were not able to reproduce the outstanding results with the trajectory kernel from [1]. Maybe because they do not provide any parameters, environment settings or rewarding functions. In our experiments the overall learning performance of the trajectory kernels was either worse or slightly better than the learning performance of standard kernels. If you take the computing efficiency into consideration they perform far worse than standard kernels. Depending on the setup, the trajectory kernels took between 3 and 22 times more computational time than the squared exponential kernel (the measured times of continuous Mountain Car are not considered, because it did not solve).

Maybe the bad learning performance of the trajectory kernel is a problem of poorly tuned hyper parameters σ_f and σ_l, the noise level parameter σ_n, the expected improvement trade-off parameter τ or other model constraints. Apart from that, the trajectory kernel would only be worthwhile on black box functions that are quite expensively to evaluate compared to the computational effort needed for trajectory kernel calculations.

7 Outlook

Future test runs could include a higher Bayesian optimization iteration count to receive more precise results. Also more than ten kernel trials per run would be helpful to get more expressive data.

Implementing the OpenAI Gym provides the classic control problems like Cart Pole, Mountain Car, and Acrobot. Furthermore a lot of more complex environments like a humanoid walker or some Atari games are provided [9]. This could facilitate future research if working with higher dimensional problems.
Dealing with more complex environments, will also include non-linear action mappings and more than one-dimensional actions. The trajectory kernels would have to be modified accordingly.

Sometimes the hyper parameter optimization fails to maximize the log marginal likelihood. To gain more robustness the maximization with partial derivatives could be implemented as proposed by [7, 6].

Computing the Gaussian distribution density for the continuous action selection probability measure has lead to numerical problems. Instead, a sigmoid function, for example the Gaussian cumulative distribution function, could be used for probabilities of continuous actions.

Bibliography

[1] A. Wilson, A. Fern, and P. Tadepalli, “Using trajectory data to improve bayesian optimization for reinforcement learning,” *The Journal of Machine Learning Research*, vol. 15, no. 1, pp. 253–282, 2014.

[2] R. Akrour, D. Sorokin, J. Peters, G. Neumann, *et al.*, “Local bayesian optimization of motor skills,” 2017.

[3] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*, vol. 1. MIT press Cambridge, 1998.

[4] E. Brochu, V. M. Cora, and N. De Freitas, “A tutorial on bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning,” *arXiv preprint arXiv:1012.2599*, 2010.

[5] B. Shahriari, K. Swersky, Z. Wang, R. P. Adams, and N. de Freitas, “Taking the human out of the loop: A review of bayesian optimization,” *Proceedings of the IEEE*, vol. 104, no. 1, pp. 148–175, 2016.

[6] D. J. Lizotte, *Practical bayesian optimization*. University of Alberta, 2008.

[7] C. E. Rasmussen and C. K. Williams, *Gaussian processes for machine learning*, vol. 1. MIT press Cambridge, 2006.

[8] N. de Freitas, “Python demo code for gp regression,” 2013.

[9] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, “Openai gym,” *CoRR*, vol. abs/1606.01540, 2016.

[10] J. Antonio Martin H., “Matlab sarsa implementations,” 2006.