Trajectory Kernel for Bayesian Optimization

Bachelor-Thesis von Sebastian Rinder aus Sindelfingen
Februar 2018

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Trajectory Kernel for Bayesian Optimization

Vorgelegte Bachelor-Thesis von Sebastian Rinder aus Sindelfingen

1. Gutachten: Prof. Dr. N. N.
2. Gutachten: Prof. Dr. N. N.
3. Gutachten: Prof. Dr. N. N.

Tag der Einreichung:

Please cite this document with:
URN: urn:nbn:de:tuda-tuprints-38321
URL: http://tuprints.ulb.tu-darmstadt.de/id/eprint/3832

Dieses Dokument wird bereitgestellt von tuprints,
E-Publishing-Service der TU Darmstadt
http://tuprints.ulb.tu-darmstadt.de
tuprints@ulb.tu-darmstadt.de

This puplication is licensed under the following Creative Commons License:
Attribution – NonCommercial – NoDerivatives 4.0 International
http://creativecommons.org/licenses/by-nc-nd/4.0/

For Thomas Hesse and Kevin Luck

Erklärung zur Bachelor-Thesis

Hiermit versichere ich, die vorliegende Bachelor-Thesis ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.
In der abgegebenen Thesis stimmen die schriftliche und elektronische Fassung überein.

Darmstadt, den 8. Februar 2018

(Sebastian Rinder)

Thesis Statement

I herewith formally declare that I 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.
In the submitted thesis the written copies and the electronic version are identical in content.

Darmstadt, February 8, 2018

(Sebastian Rinder)

Abstract

Reinforcement learning relies on policy gradient but the gradient is known only in expectation and most of the time stochastic policies. This leaves some room for zero order methods and BO can combine solving the problem and the exploration strategy from deterministic policies. We investigate in this paper how to integrate efficient exploration strategies stemming from Bayesian optimization for solving high dimensional reinforcement learning problems. We propose a novel optimization algorithm that is able to scale Bayesian optimization to such high dimensional tasks by restricting the search to the local vicinity of a search distribution and by proposing kernels capturing similarity in behavior rather than parameter. We show in the experiments that our approach can be very useful for applications such as robotics.

Zusammenfassung

Das Ziel im bestärkten Lernen ist das Finden einer Strategie, welche die erhaltene Belohnung eines Agenten maximiert. Da der Suchraum für mögliche Strategien sehr groß sein kann, verwenden wir Bayesian optimization, um die Anzahl der Evaluierungen durch den Agenten zu minimieren. Das hat den Vorteil, dass zeit- und kostenaufwändige Abläufe, wie beispielsweise das Bewegen eines Roboterarms, reduziert werden. Die effektivität der Suche wird maßgeblich von der Wahl des Kernels beeinflusst. Standardkernel in der Bayesian optimization vergleichen die Parameter von Strategien um eine Vorhersage über bisher nicht evaluierte Strategien zu treffen.

Der Trajectorykernel vergleicht statt der Parameter, die aus den jeweiligen Strategien resultierenden Verhaltensweisen. Dadurch werden unterschiedliche Strategien mit ähnlichem Resultat von der Suche weniger priorisiert.

Wir zeigen die Überlegenheit des verhaltensbasierten Kernels gegenüber dem parameterbasierten anhand von Robotersteruerungssimulationen.

Acknowledgments

Contents

Figures and Tables

List of Figures

List of Tables

Symbols and Notation

Matrices and constants are noted as capital letters, and vectors and scalars as lower case letters. Vectors are assumed as column vectors. The first dimension of a matrix indicates its row number, the second dimension its column number.

Symbol	Meaning
S	Matrix of states
A	Matrix of actions
p	state transitioning probabilitiy
p_0	probabilitiy of starting in state s_0
r	immediate reward
$\bar{r}$	cumulative reward, sum of all rewards from one trajectory
M	count of initial sample points before starting the Bayesian optimization
N	total number of Bayesian optimization steps
n	number of training points present
n_*	number of sample test points
d	number of dimensions in the problem
x	training point vector of length d
x_*	test point vector of length d
X	$n \times d$ matrix of n points $x^\top$
X_*	$n_* \times d$ matrix of m test points $x_*^\top$
y	vector of n evaluated objective function values
$K(X,X) = K$	$n \times n$ 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
$\circ$	Hadamard product, element-wise product
$\mathrm{diag}(V)$	returns the diagonal elements of the square matrix V as a vector

1 Introduction

2 Motivation

TODO

3 Foundations

3.1 Markov decision process

In our reinforcement learning problem we use a Markov decision process model to describe possible decisions as probabilites. Our model consists of a tuple (S, A, p, p_0, r), holding all states $s \in S$, all actions $a \in A$, all state transitioning probabilities p, and all corresponding rewards r. Assume an agent which executes a policy x for T time steps, producing a trajectory ξ, and therefore receiving the final reward

$$\bar{r}(\xi) = \sum_{t=1}^{T} r(s_{t-1}, a_{t-1}, s_t),$$

which is the sum of all immediate rewards, given by an rewarding function $r(s_{t-1}, a_{t-1}, s_t)$. This rewarding function depends on the given environment. For example it rewards a state we want to achieve by returning a value greater than zero and can penalize states we do not want our agent to be in by returning negative values. The cumulative reward stemming from a policy is what we want to maximize with as few evaluations as feasible.

3.2 Global Bayesian optimization

To find the maximum reward of our expensivly to evaluate agent we use Bayesian optimization. It makes the search process more efficient by incorporating a Gaussian process model to anticipate the agents behaviour. This model, containing estimates of the agent returns depending on policies, is used by a so called acquisition function to guide the exploration for promising policies. Each newly found policy is evaluated by the agent and the results are included in the Gaussian process model. Optimally these steps are repeated until the cumulative rewards converge at the maximum possible cumulative reward. In reality (Algorithm 1) we iterate over N steps and analyse the results.
To meet the mathematical context we will also refer to policies as points. And we will also refer to cumulative reward depending on a policy as objective function depending on a point.

```
input  : $X$: $M$ uniformly random samples from the search space
         $N$: number of Bayesian optimization iterations
output: $x_{M+N}$
$y$ = evaluations of the objective at the points $X$
for $n = M$ to $M + N$ do
    compute $K(X, X)$
    $x_{n+1}$ = point at the optimum of the acquisition function
    $y_{n+1}$ = evaluation of the objective at the point $x_{n+1}$
    $X = \{X, x_{n+1}\}$
    $y = \{y, y_{n+1}\}$
```

Algorithm 1: Global Bayesian optimization

During the acquisition function optimization $K(X, X)$, originally contained by the Gaussian process, does not change. So it is precomputed.

3.3 Acquisition function

The core of the Bayesian optimization consists of selecting the next evaluation point in our search space. We get that point by optimizing an acquisition function, which depends on the incumbent Gaussian process model. We choose expected improvement during the global Bayesian optimization and in the local optimization (chapter 4) we use Thompson sampling. Both acquisition functions take the mean und 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.

3.3.1 Expected improvement

To get an expected improvement function value at a test point x_* we need the mean value $\mu(x_*)$, and the standard deviation value $\sigma(x_*) = \sqrt{v(x_*)}$ 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 and the probability density function from the Gaussian distribution we write $\Phi(.)$ and $\phi(.)$ respectively. They both have zero mean and unit variance. We adopt the expected improvement function

$$\mathrm{EI}(x_*) = \begin{cases} (\mu(x_*) - y_{max} - \tau)\Phi(\mathrm{z}(x_*)) + \sigma(x_*)\phi(\mathrm{z}(x_*)) & \text{if } \sigma(x_*) > 0 \\ 0 & \text{if } \sigma(x_*) = 0 \end{cases}$$

where

$$\mathrm{z}(x_*) = \begin{cases} \frac{(\mu(x_*) - y_{max} - \tau)}{\sigma(x_*)} & \text{if } \sigma(x_*) > 0 \\ 0 & \text{if } \sigma(x) = 0 \end{cases}$$

as suggested in [1]. The trade-off parameter τ is set to 0.01 accordingly. To get the next evaluation point x_{n+1} the expected improvement function is optimized over the whole search space, such that:

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

3.3.2 Thompson sampling

For the Thompson sampling we sample one function from the Gaussian process posterior,

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

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 optimzer. These mean and variance represent our current search space. To draw function values we need the mean vector μ 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 g, which consists of independent Gaussian distributed valus with zero mean and unit variance. Finally we get a vector TS of sampled values:

$$\mathrm{TS}(x_*) = \mu + L_V g.$$

We take the one with the highest value such that,

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

to get the next evaluation point.

3.4 Gaussian Process Regression

The Gaussian process fits a multivariate gaussian distribution over our training data. From the regression we get a posterior mean and variance, which describe our model of the objective function. 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 preciser the predictions, in the proximity around training points.
In real world applications we always have some noise in our 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

$$y = f(x) + \epsilon$$

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

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

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_*:

$$K_n = K + \sigma_n^2 I \tag{3.1}$$

$$\mu = K_* K_n^{-1} y \tag{3.2}$$

$$V = K_{**} - K_* K_n^{-1} K_*^\top \tag{3.3}$$

$$v = \mathrm{diag}(V). \tag{3.4}$$

The resulting vectors μ and v contain the means and variances for all corresponding test points. Also we get the whole covariance matrix V.

3.4.1 Kernel for Gaussian process

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 to compare them to the trajectory kernel (section 3.5). In those standard kernels the distance metric for two points is given by:

$$D(x_i, x_j) = (x_i - x_j)^\top (x_i - x_j).$$

Which we then use in the squared exponential kernel,

$$K(x_i, x_j, \sigma) = \sigma_f^2 \exp\left(-\frac{D(x_i, x_j)}{2\sigma_l^2}\right),$$

and the Matern 5/2 kernel,

$$K(x_i, x_j, \sigma) = \sigma_f^2 \left(1 + \frac{\sqrt{5D}}{\sigma_l} + \frac{5D}{3\sigma_l^2}\right) \exp\left(-\frac{\sqrt{5D}}{\sigma_l}\right),$$

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. (CHAP HYPOPT)

3.5 Trajectory kernel

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

3.5.1 Behaviour based kernel

For relating policies to their resulting behavior we use the state transitioning probabilities from the Markov decision process. We formulate the conditional probability of observing trajectory ξ given policy x by

$$p(\xi|x) = p_0(s_0) \prod_{t=1}^{T} p(s_t|s_{t-1}, a_{t-1}) p_\pi(a_{t-1}|s_{t-1}, x)$$

in which trajectory $\xi = (s_0, a_0, ..., s_{T-1}, a_{T-1}, s_T)$ contains the sequence of state, action tuples and x a set of d policy parameters. $p_0(s_0)$ is the probability of starting in the initial state s_0. $p(s_t|s_{t-1}, a_{t-1})$ is the probability of transitioning from state s_{t-1} to s_t when action a_{t-1} is executed. The stochastic mapping $p_\pi(a_{t-1}|s_{t-1}, x)$ is the probability for selecting the action a_{t-1} when in state s_{t-1} and executing the parametric policy x.

3.5.2 Distance metric

To examine the difference between two policies x_i and x_j the discrete Kullback Leibler divergence

$$D_{\mathrm{KL}}(P(\xi|x_i)||P(\xi|x_j)) = \sum_i P(\xi|x_i)\, \log \frac{P(\xi|x_i)}{P(\xi|x_j)}.$$

is applied to the policy-trajectory mapping probabilities $P(\xi|x_i)$ and $P(\xi|x_j)$. It measures how the two distributions diverge from another.

In general $D_{\mathrm{KL}}(P(\xi|x_i)||P(\xi|x_j))$ is not equal to $D_{\mathrm{KL}}(P(\xi|x_j)||P(\xi|x_i))$. But we need a symmetric distance measure. So we sum up the two divergences

$$D(x_i, x_j) = D_{\mathrm{KL}}(P(\xi|x_i)||P(\xi|x_j)) + D_{\mathrm{KL}}(P(\xi|x_j)||P(\xi|x_i)),$$

to achieve that $D(x_i, x_j) = D(x_j, x_i)$. An additional requirement for the kernel is the resulting matrix to be positive semi-definite and scalable[2]. 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 scale. This gives us the covariance function

$$K(x_i, x_j, \sigma_f, \sigma_l) = \sigma_f^2 \exp(-\sigma_l^2 D(x_i, x_j)), \tag{3.5}$$

3.6 Hyper parameter optimization

Selecting proper hyper parameters for the Gaussian process regression can reduce the number of objective function evaluations necessary. Also it helps avoiding numerical problems. To find an optimum for the signal deviation hyperparameter σ_f and the length scale hyperparameter σ_l we maximize the log marginal likelihood function

$$\log p(y|X, \sigma_f, \sigma_l) = -\frac{1}{2} y^\top K_n^{-1} y - \frac{1}{2} \log |K_n| - \frac{n}{2} \log 2\pi, \tag{3.6}$$

of our Gaussian process. The number of observations is n, X is the $d \times n$ dataset of inputs and K_n is the covariance matrix for the noisy target y.

4 Contributions

- including trajectory kernel to local BO – fmincon local search on distance covariance –TS trajectory kernel — Mixing up kernel metrics (scaling factor) —- random initial samples well distributed in search space
- debug plotting – mixed up kernel metrics – acq func plots – hyper opt plots — select figure by name
- global BO – random initial samples well distributed in search space - Ei use max means
- simultaion implementation of cart pole, acrobot and mountain car – also visualization, but not maintained (YET)
- sampling of actions from probabilities

Before doing regression we transform our observations to zero mean and uniform variance:

$$y = \frac{y_n - \text{mean}(y_n)}{\text{std}(y_n)}.$$

4.1 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 [3]. 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 center and the exploration reach respectively. To update that mean and variance properly we minimize the Kullback-Leibler divergence between the incumbent 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 we constrain the minimization with the hyper parameters α and β. Therefore our optimization problem is given by

$$\begin{aligned} &\arg\min_{\pi} && \text{KL}(\pi || p_n^\star), && \\ &\text{subject to} && \text{KL}(\pi || \pi_n) && \leq \alpha, \quad (4.1) \\ & && \mathscr{H}(\pi_n) - \mathscr{H}(\pi) && \leq \beta, \quad (4.2) \end{aligned}$$

where $\text{KL}(p||q) = \int p(\boldsymbol{x}) \log \frac{p(x)}{q(x)} d\boldsymbol{x}$ is the KL divergence between p and q and $\mathscr{H}(p) = -\int p(\boldsymbol{x}) \log(p(\boldsymbol{x})) d\boldsymbol{x}$ is the entropy of p.

4.2 Efficiency and robustness

We vectorize every suited operation to speed up calculations in Matlab. Additionally we tune the code to run on a parallel pool.

Sometimes numerical instabilities lead to negative values for covariances. To avoid getting complex numbers we filter negative values before applying the square root on covariances when computing the expected improvement. Negative values also occur for the distance between two trajectories, and are filtered too.

Depending on the acquisition function we calculate either the covariance matrix or the covariance vector from the Gaussian process regression. In case of expected improvement this saves a lot of computation time, since we only need the vector of covariances.

Before each bayesian optimization iteration step we precompute K because it only depends on the set of training points which won't change during optimization. Then $alpha$ and L are derived from K. REF

For every matrix inverse calculation we use the lower Cholesky decomposition instead. Therefore the matrix to decompose must be positive definite. We achieve this by doubling the noise variance diagonal added to the original matrix as shown in algorithm 2.

When doing hyper parameter optimization we do not double the noise variance. Instead our log marginal likelihood function returns -infinity for hyper parameters which produce a non positive definit matrix.

```
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 2: Lower Cholesky with variance doubling

4.3 Normalization constant

4.4 Action selection

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

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

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

$$\begin{aligned}
\sum_{t=0}^{T-1} \log\left(\frac{P_\pi(a_t|s_t,x_i)}{P_\pi(a_t|s_t,x_j)}\right) &= \sum_{t=0}^{T-1} \log\left(\frac{\frac{1}{\sqrt{2\pi\epsilon_a^2}} \exp\left(-\frac{(a_t - f_s(s_t)x_i)^2}{2\epsilon_a^2}\right)}{\frac{1}{\sqrt{2\pi\epsilon_a^2}} \exp\left(-\frac{(a_t - f_s(s_t)x_j)^2}{2\epsilon_a^2}\right)}\right) \\
&= \sum_{t=0}^{T-1} \log\left(\exp\left(-\frac{(a_t - f_s(s_t)x_i)^2}{2\epsilon_a^2} - \left(-\frac{(a_t - f_s(s_t)x_j)^2}{2\epsilon_a^2}\right)\right)\right) \\
&= \frac{1}{2\epsilon_a^2} \sum_{t=0}^{T-1} \left((a_t - f_s(s_t)x_j)^2 - (a_t - f_s(s_t)x_i)^2\right).
\end{aligned}$$

4.5 Global optimization

We select the initial samples set 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.

When optimizing the expected improvement function or the hyper parameter space we use the `GlobalSearch` Toolbox from MATLAB. But if we run the experiments on the cluster we use the local optimizer `fmincon` on 10000 random starting points to work around the lack of free global optimization toolbox licences.

4.6 Numerical stability

in the end -> 1. explain what we want to achieve (no log of small values for example) 2. cholesky 3. hyp param matrix

4.7 Hyper parameter optimization

4.7.1 Optimization starting point

When optimizing the log marginal likelihood of the gaussian process we are confronted with undefinded areas in the hyper parameter space because K is not positive semi definit in those areas. To start the optimization properly we randomly select points until finding a defined one.

4.7.2 Adaptation of the search space

We start our search space with a generous boundary. For example $[\exp(-10), \exp(10)]$. After finding the first set of hyper parameters we change that boundary to a smaller one to make the following optimizations more efficient. Also the center of the search space is adjusted to always match the latest set of hyper parameters.
We can do this because it is not expected that the hyper parameters change drastically between iterations.

4.8 Independent log-normal prior

The log marginal likelihood maximization will sometimes succeed at the borders of our search space. We prevent this by adding an independet log-normal prior term as introduced in [4]:

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

Since the suggested prior is centered 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:

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

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 $h_2 = \log \sigma_l$.
—write whole hyper opt with L and stuff

4.9 OpenAI Gym in Python

To use the simulations provided by the OpenAI Gym we prepare 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 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 python are made usable with `numpy.asarray()`.
Implementing the OpenAI Gym grants us 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[5]. This will facilitate future research on solving higher dimensional problems.

5 Experiments

- parallel runs on the cluster
- BO : We set the initial sample count $M = 10$ and the iteration count $N = 200$.

5.1 Implementation

5.1.1 Problem environment

In early test runs we used `Matlab` implementations of cart pole, acrobot, and mountain car adapted from [6]. But later we implemented OpenAI Gym to have a wide range of problem environments available and to have consistent rewarding functions. Otherwise, the rewarding function would depend on the source and therefore produce less comparable results.

5.1.2 Gaussian Process Regression

(difference between full covariance and cov vector)
Instead of calculating the inverse of K_n in (3.2) we use the lower Cholesky decomposed matrix:

$$LL^\top = K_n$$

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

$$\mu = K_n^{-1}\, y = (L\, L^T)^{-1}\, y = (L^{-T}\, L^{-1})\, y = L^{-T}\, (L^{-1}\, y) = L^T \setminus (L \setminus y). \tag{5.1}$$

The backslash operator denotes the matrix left division, so the solution $x = A \setminus b$ satisfies the system of linear equations $Ax = b$. Matrix K_n must be positive definite for the cholesky decomposition. So we double the noise variance hyperparameter σ_n^2 from (3.1) until positive definiteness is achieved.
For the expected improvement function we only need the vector of variances. Instead of calculating the whole covariance matrix V we can take a shortcut. All elements on the diagonal of $K(X_*, X_*)$ equal σ_f because the difference between one x_* and the same x_* is zero. Therefore we write:

$$L_k = L \setminus K(X_*, X)$$

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

This adaptation is inspired by [8] and reduces the computational effort drastically.
For the whole covariance matrix we also avoid calculating the matrix inverse:

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

5.1.3 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 $sigma_l$ before applying the exponential function (3.5), we avoid getting a zero kernel matrix K.

When calculating $\log(|K_n|)$ for the hyperparameter optimization (3.6), 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 $K_y^{-1}y$ in (3.6) is done by the same method we already use in the Gaussian process (5.1).

5.1.4 Estimation of Trajectory Kernel Function Values

We estimate the divergence values, because the computational effort will be greatly reduced[2]. We use a Monte-Carlo estimate for the approximation

$$\hat{D}(x_i, x_j) = \sum_{\xi \in \xi_i} \log\left(\frac{P(\xi|x_i)}{P(\xi|x_j)}\right) + \sum_{\xi \in \xi_j} \log\left(\frac{P(\xi|x_j)}{P(\xi|x_i)}\right)$$

of the divergences between policies with already sampled trajectories. Here ξ_i is the set of trajectories generated by policy 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

$$\hat{D}(x_{new}, x_j) = \sum_{\xi \in \xi_j} \left[\frac{P(\xi|x_{new})}{P(\xi|x_j)} \log\left(\frac{P(\xi|x_{new})}{P(\xi|x_j)}\right) + \log\left(\frac{P(\xi|x_j)}{P(\xi|x_{new})}\right)\right]$$

to estimate the divergence between the new policy x_{new} and the policy x_j with already sampled trajectories ξ_j.
Since we only have a ratio of transitioning probabilities present in our trajectory kernel we can reduce the logarithmic term to:

$$\begin{aligned}
\log\left(\frac{P(\xi|x_i)}{P(\xi|x_j)}\right) &= \log\left(\frac{P_0(s_0)\prod_{t=1}^{T} P_s(s_t|s_{t-1}, a_{t-1})P_\pi(a_{t-1}|s_{t-1}, x_i)}{P_0(s_0)\prod_{t=1}^{T} P_s(s_t|s_{t-1}, a_{t-1})P_\pi(a_{t-1}|s_{t-1}, x_j)}\right) \\
&= \log\left(\prod_{t=1}^{T} \frac{P_\pi(a_{t-1}|s_{t-1}, x_i)}{P_\pi(a_{t-1}|s_{t-1}, x_j)}\right) \\
&= \sum_{t=1}^{T} \log\left(\frac{P_\pi(a_{t-1}|s_{t-1}, x_i)}{P_\pi(a_{t-1}|s_{t-1}, x_j)}\right).
\end{aligned}$$

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

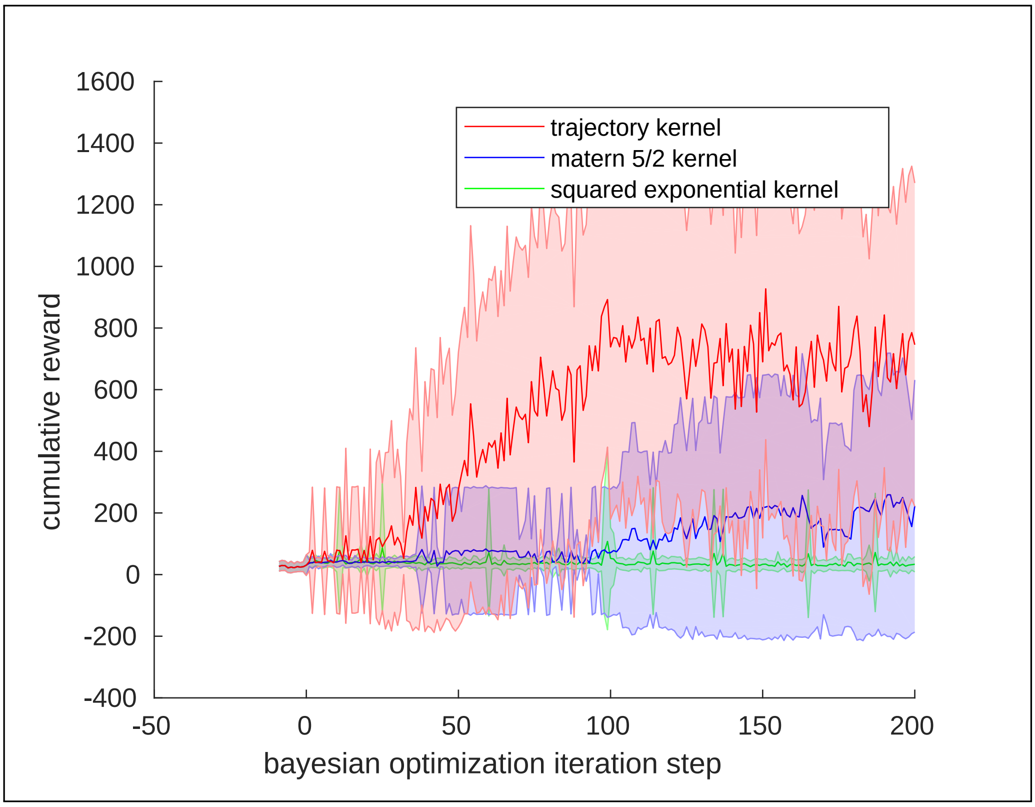

Figure 5.1.: Global opt, 4 dim. Mean and standard deviation from 32 trials of each kernel. Cartpole matlab implementation with old reward function, 1000 timesteps, and continuous action selection.

5.1.5 Action selection

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

$$a = f_s(s)^\top x + \epsilon_a,$$

with a small gaussian noise ϵ_a needed for stochastic policies. So the actions are Gaussian distributed:

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

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

$$P(a|s) = \frac{\exp(f_s(s)^\top x_a)}{\sum_{i \in A} \exp(f_s(s)^\top x_i)}.$$

Again it consists of the linear mapping $f_s(s)^\top x$, and A holds all possible actions. The resulting action is then sampled from the probability of action a given state s.
The state feature function $f_s(s)$ is defined for each environment as shown below.

5.2 Cart pole

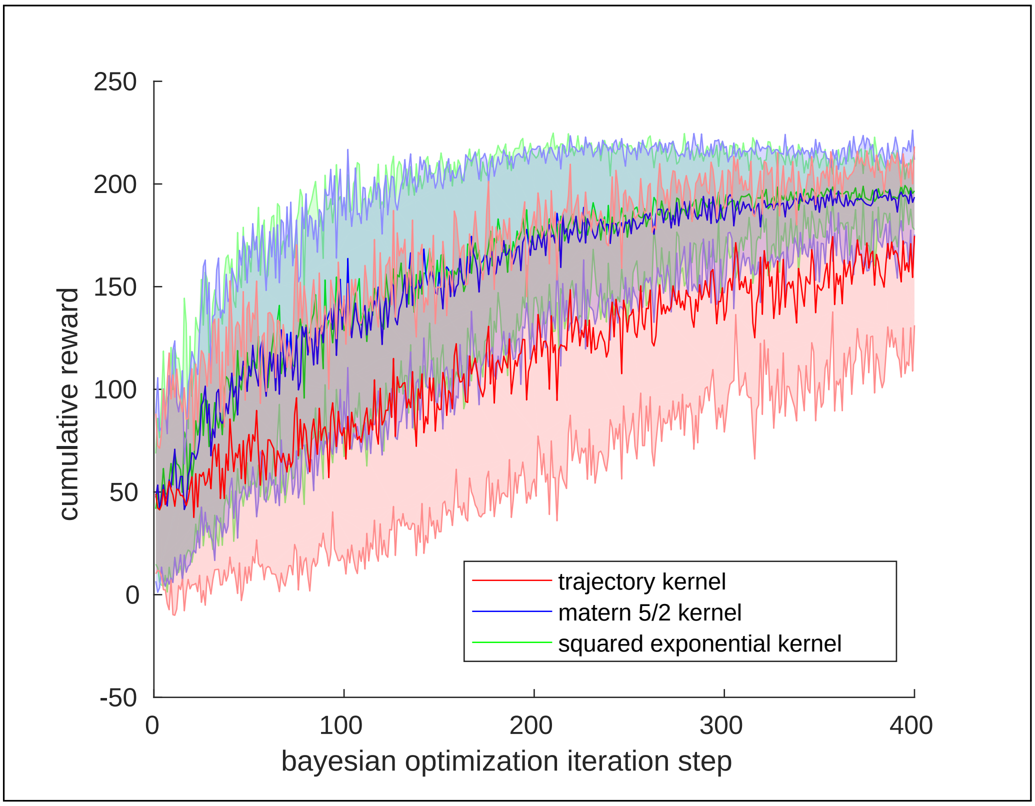

Figure 5.2.: Local opt, 4 dim. Mean and standard deviation from 32 trials of each kernel. Cartpole matlab implementation with new reward function, 200 timesteps, and continuous action selection.

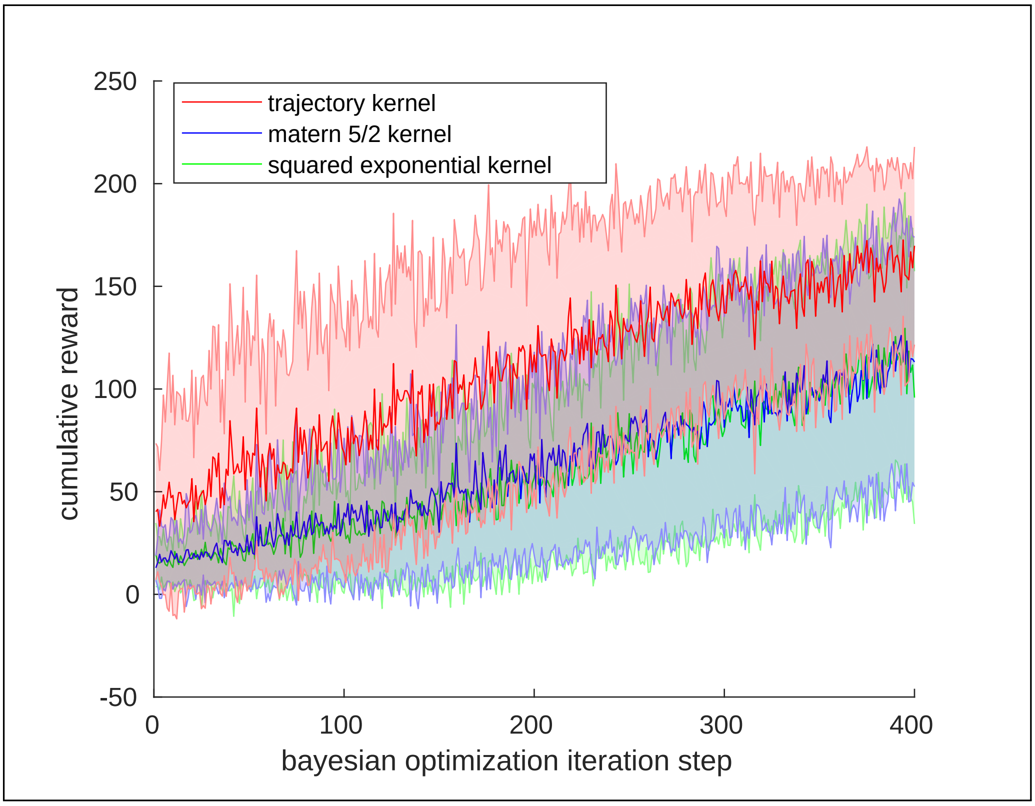

Figure 5.3.: Local opt, 10 dim. Mean and standard deviation from 32 trials of each kernel. Cartpole python gym implementation, 200 timesteps, and discrete action selection.

6 Results

TODO

7 Discussion

TODO

8 Outlook

- more than one dimensional actions - non-linear action selection policies - higher dimensional environment with traj kernel

Bibliography

[1] 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.

[2] 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.

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

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

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

[6] J. Antonio Martin H., "Matlab sarsa implementation."

[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] 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.

A Some Appendix

Use letters instead of numbers for the chapters.