English Title

Deutscher Titel
Bachelor-Thesis von Sebastian Rinder aus Sindelfingen
Februar 2018

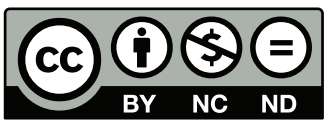
TECHNISCHE
UNIVERSITÄT
DARMSTADT

English Title
Deutscher Titel

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:



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 2. 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 2, 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 are noted as capital letters and vectors 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
$init$	count of initial sample points before starting the bayesian optimization
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	kernel function of the Gaussian process
$K(X,X)$	$n \times n$ covariance matrix
$K(X,X_*)$	$n \times n_*$ covariance between training and test points
$K(X_*,X)$	same as $K(X,X_*)^\top$
$K(X_*,X_*)$	$n_* \times n_*$ covariance matrix
$\circ$	Hadamard product (element-wise product)

1 Introduction

2 Motivation

TODO

3 Foundations

3.1 Global Bayesian optimization

To find the maximum of our expensive black box function we use Bayesian optimization. It makes the search process more efficient by incorporating a model of the unknown function. This model is used to guide the exploration for new points.

$init = 10$
$X = init$ uniformly random samples from the search space
$y =$ evaluations of the objective at the points X
for $n = init$ **to** $200 + init$ **do**
  compute $K(X,X)$
  Optimize hyper parameters (optional)
  $x_{n+1} =$ point at the optimum of the acquisition function
  $y_{n+1} =$ evaluation of the objective at the point x_{n+1}
  $X = \{X, x_{n+1}\}$
  $y = \{y, y_{n+1}\}$

During acquisition function optimization $K(X,X)$ from the Gaussian process does not change, so it is precomputed. The values for the initial sample count and the iteration count are not fixed, but for simplification set to 10 and 200 here.

3.2 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 [1]. 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 variance 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

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

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

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 a so called acquisition function. It depends on the incumbent Gaussian process model of the true objective. We choose expected improvement during the global bayesian optimization and in the local optimization we use Thompson sampling. Both acquisition functions take the mean und the variance from the gaussian process model into account to guide the exploration process. The difficulty lies in avoiding excessive exploration or exploitation. Exploration looks for 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 [2]. The trade-off parameter τ is set to 0.01 accordingly. The expected improvement function is then optimized over the whole search space to give us the next evaluation point

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

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 compute 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

Since we want to estimate an objective function in a machine learning environment we elect to use Gaussian process regression. It fits a multivariate gaussian distribution over our prior 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 points our model incorporates the smaller the variance, and the preciser the predictions, in the proximity around prior 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. The observed target

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

regards this noise. Before doing regression we transform our observations to zero mean and uniform variance:

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

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 normalized target values and the function values at the test locations:

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

Now we can calculate the posterior mean and variance at given test points X_*.

$$K_n = K(X,X) + \sigma_n^2 I \tag{3.3}$$
$$\mu = K(X_*,X)K_n(X,X)^{-1}y \tag{3.4}$$
$$V = K(X_*,X_*) - K(X_*,X)K_n(X,X)^{-1}K(X_*,X)^\top \tag{3.5}$$
$$\sigma^2 = v = \text{diag}(K(X_*,X_*) - K(X_*,X)K_n(X,X)^{-1}K(X_*,X)^\top) \tag{3.6}$$

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

3.5 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, and all corresponding rewards. Assume an agent which executes a policy x for T time steps receiving a final reward $\bar{R}(\xi)$. This reward depending on a policy is what we want to maximize.
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 $x \in \mathbb{R}^D$ 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. So we receive a final reward for a sampled trajectory,

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

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.

3.6 Kernel for Gaussian process

3.6.1 Squared exponential kernel

$$D(x_i,x_j) = x_i - x_j$$

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

3.6.2 Matern 5/2 kernel

$$D(x_i, x_j) = x_i - x_j$$

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

3.6.3 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. 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 respective policy-trajectory mappings $P(\xi|x_i)$ and $P(\xi|x_j)$. It measures how the two probability 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[3]. Therefore we exponentiate the negative of our distance matrix D. We also apply the hyper parameters σ_f to compensate for the signal variance and σ_l to adjust for signal scale. This gives us the covariance matrix

$$K(x_i, x_j, \sigma_f, \sigma_l) = \sigma_f \exp(-\sigma_l D(x_i, x_j)), \tag{3.7}$$

for the gaussian process.

3.7 Hyper parameter optimization

Selecting proper hyper parameters for the Gaussian process regression enhances the efficiency of our learning algorithm. Also it helps avoiding numerical problems. To find an optimum for the signal variance hyperparameter σ_f and the length scale hyperparameter σ_l we maximize the log marginal likelihood function

$$\log p(y|x, \sigma_f, \sigma_l) = -\frac{1}{2} y^\top K_n^{-1} y - \frac{1}{2} \log |K_n| - \frac{n}{2} \log 2\pi, \tag{3.8}$$

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

4 Experiments

4.1 Implementation

4.1.1 Optimizer

In global optimizing we used the `GlobalSearch` Toolbox from MATLAB first. But the cluster lacked of free licences for this toolbox. So we use the local optimizer `fmincon` on 10000 random starting points. In experiments that method performed almost as good as `GlobalSearch`.

4.1.2 Normalization constant

4.1.3 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

4.1.4 OpenAI Gym in Python

To use the simulations provided by the OpenAI Gym we prepared a python module which could be imported to MATLAB. After importing the python module with `py.importlib.import_module(moduleName)` we can call every method it contains. The main difficulty was the correct data type conversion when receiving data from the python module. The performance difference in the cart pole experiment is comparable. (measure)

4.1.5 Gaussian Process Regression

(difference between full covariance and cov vector)
Instead of calculating the inverse of K_n in (3.4) we use the lower Cholesky decomposed matrix:

$$LL^\top = K_n$$

This is considered faster and numerically more stable [4]. The mean vector μ is then computed as follows:

$$\mu = K_n^{-1}\, y = (L\, L^T)^{-1}\, y = (L^{-T}\, L^{-1})\, y = L^{-T}\, (L^{-1}\, y) = L^T \setminus (L \setminus y). \tag{4.1}$$

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 (3.3) 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 [5] and reduces the computational effort drastically.
For the whole covariance matrix we also avoid calculating the matrix inverse:

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

4.1.6 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 (3.7), we avoid getting a zero kernel matrix K.

When calculating $\log(|K_n|)$ for the hyperparameter optimization (3.8), 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 (3.8) is done by the same method we already use in the Gaussian process (4.1).

4.1.7 Estimation of Trajectory Kernel Function Values

We estimate the divergence values, because the computational effort will be greatly reduced[3]. 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 whole product.

4.1.8 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:

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

Therefore the resulting probability density of the action selection,

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

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

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

The function $f_s(s)$, depending on the state, computes our d dimensional state feature vector.
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.

4.2 Cart pole

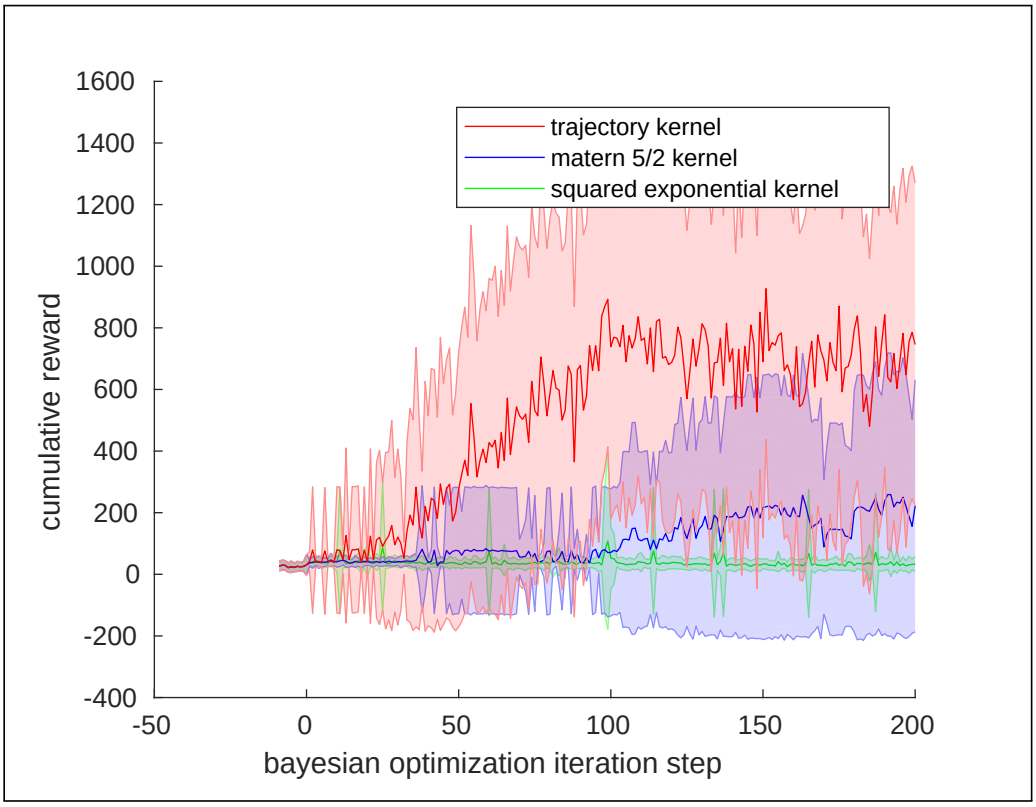


Figure 4.1.: Global opt, 4 dim. Mean and standard deviation from 32 trials of each kernel. Cartpole matlab implementation with old reward function, 1000 timesteps, and continuous action selection.

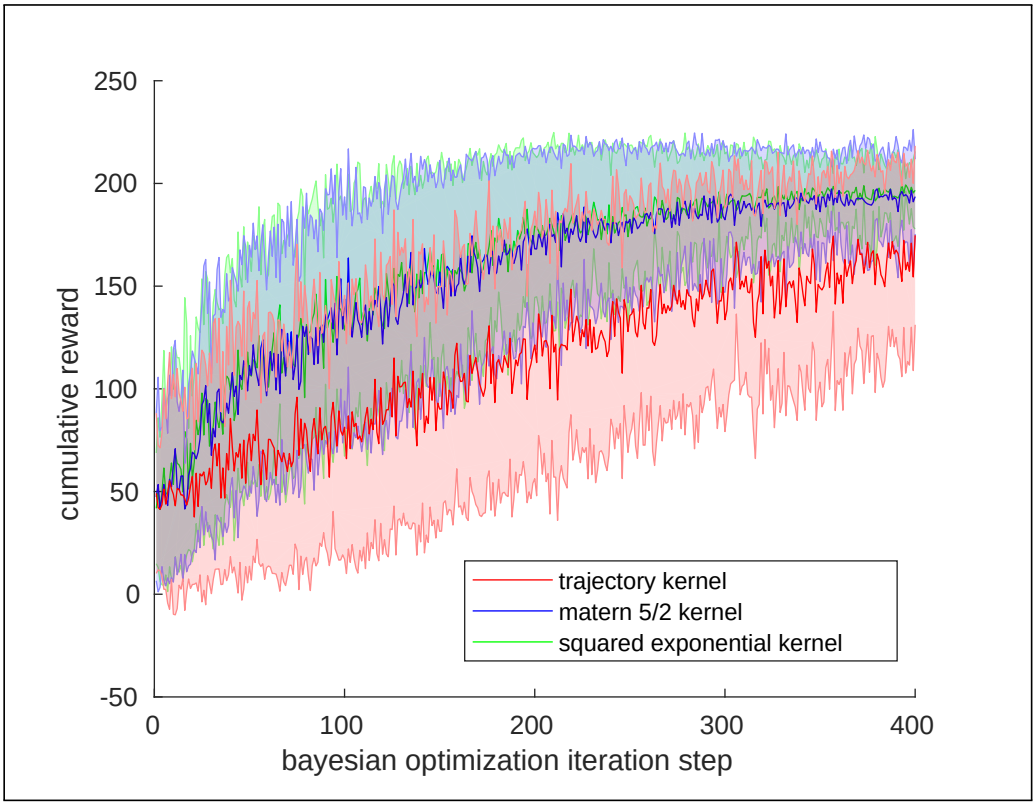


Figure 4.2.: Local opt, 4 dim. Mean and standard deviation from 32 trials of each kernel. Cartpole matlab implementation with new reward function, 200 timesteps, and continuous action selection.

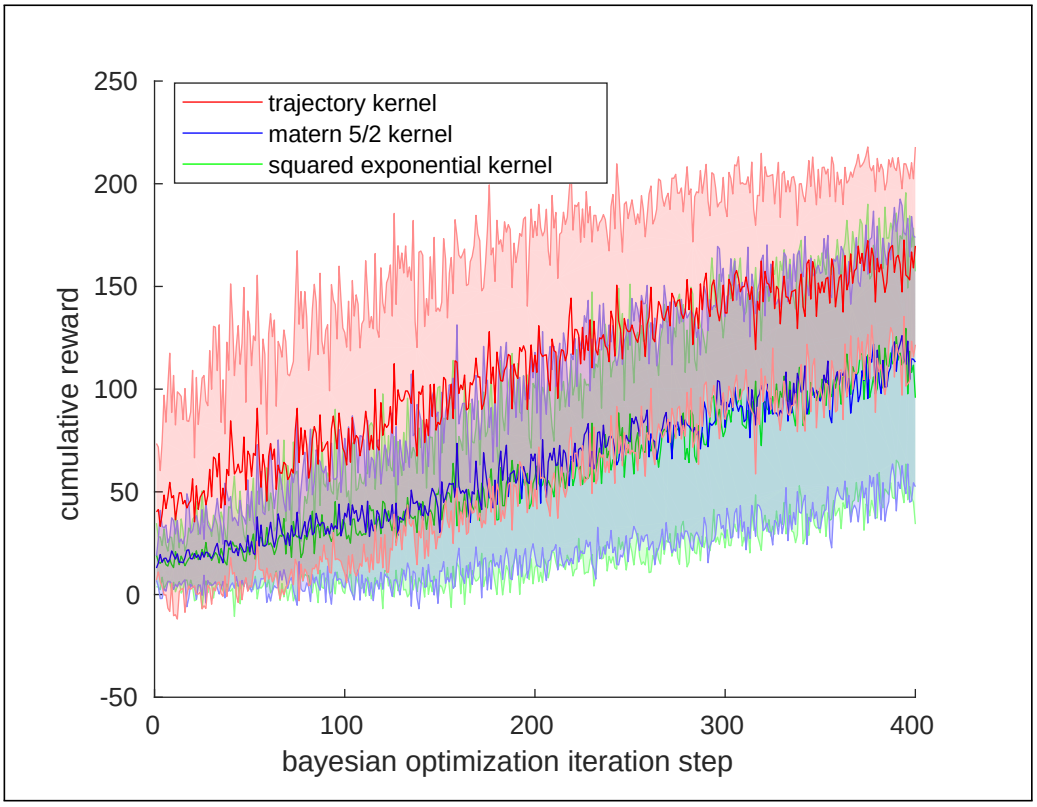


Figure 4.3.: Local opt, 10 dim. Mean and standard deviation from 32 trials of each kernel. Cartpole python gym implementation, 200 timesteps, and discrete action selection.

5 Results

TODO

6 Discussion

TODO

7 Outlook

TODO

Bibliography

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

[2] 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.

[3] 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.

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

[5] N. de Freitas, "Python demo code for gp regression," 2013.

[6] 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.