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:

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

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	number of initial sample points before starting the Bayesian optimization
N	number of total 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 n_* 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

In the reinforcement learning area of machine learning we teach an agent how to improve his behaviour. We want a specifc result from that agent, therefore we discourage results we do not seek and reinforce behaviour which leads to our goal. As an agent let us assume a robot arm with a table tennis racket and a ball. If our goal was to balance the ball for as long as possible, we would penalize the agent for letting the ball fall, and reward the agent for each successful time step of balancing. At the end of an evaluation episode we sum up all received rewards to get the cumulative reward as a measure of performance.

The agent's behaviour can be defined by a set of policy parameters. These parameters tell the agent how to act depending on his state. A state 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 arm when to rotate, and at which magnitude, to keep 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 since it is highly unlikely to find fitting policy parameters on accident. Especially in more complex environments the policy can easily have dozens of dimensions. Besides playing the lottery, we would have to deal with wear and tear, and the limited execution speed of our robotic equipment.

Therefore we require a more efficient search process for finding properly functioning policies. Optimally a search process, which learns from collected data, and extrapolates future behaviour. In recent years the Bayesian optimization approach has been shown to be very efficient in this field [1, 2, 3]. 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. And the optimum would be the policy with the highest cumulative reward achievable.

The prediction of future behaviour is based on measuring the distance between policies. The outcome of a new policy is then estimated by applying the learned distance metric. Out of many new policies BO selects one with the most promising coverage of the search space. A good coverage 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. Only exploiting policies with a high expectation can lead to a local optimum and thus to a result below the best possible one. Whereas excessive exploration will cover the entire search space, but maybe will not find an optimum.

When using standard kernels, the distance between policies is usually measured by calculating the Euclidean distance between corresponding policy parameters. A more meaningful measure would compare the resulting behaviours stemming from given policies instead comparing the policies' parameters themselves. We incorporate such a behaviour measure by using the trajectory data generated by our agent. Starting an evaluation episode, the agent is in a specific state. From this state and the given policy an action is sampled. This action determines the next state, from which an action is sampled again. The evaluation process continues until we reach a break condition. This data set generated during an episode is then used by the trajectory kernel. The kernel derives a behaviour measure from the state and action parameter sets, to relate policies accordingly. This relation has the advantage that different policies with similar results are recognized as redundant and are therefore less prioritized by the search. As a consequence the trajectory kernel makes the search process more efficient, but also has the disadvantage that the effort for kernel calculations is greatly increased.

When it comes to higher dimensional problems BO tends to over explore because of its focus on global optimization. This can lead to not finding an optimum at all. We therefore also use BO with a locally restriced search area [4]. This area is adjusted throughout the search process to cover the most promising parts of the entire search space. As a result, we only have to optimize locally. This method is more robust against high-dimensional problems and also computationally less demanding than global optimization.

So we will compare the trajectory kernel to standard kernels in the Bayesian optimization. We also do test runs with global Bayesian optimization and Bayesian optimization in a local context. The reinforcement learning environment will be given by the classic control tasks cart pole, mountain car and acrobot [5].

2 Motivation

inf/0

3 Foundations

3.1 Markov decision process

In our reinforcement learning problem we use the Markov decision process model to describe possible decisions as probabilites. This 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 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 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 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

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

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,

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

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

3.3.2 Thompson sampling

For the acquisition with 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 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

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

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_*:

$$K_n = K + \sigma_n^2 I \tag{3.1}$$
$$\mu = K_* K_n^{-1} y \tag{3.2}$$
$$V = K_{**} - K_* K_n^{-1} K_*^\top \tag{3.3}$$
$$v = \operatorname{diag}(V). \tag{3.4}$$

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[6]. 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

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

3.5.3 Estimation of Trajectory Kernel Values

We estimate the divergence values, because the computational effort will be greatly reduced[6]. 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.

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 the 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
- simulation implementation of cart pole, acrobot and mountain car – (also visualization)
- sampling of actions from probabilities
- standardize objective values for acquisition functions (Before doing regression we transform our observations to zero mean and uniform variance:

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

)

4.0.1 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:

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

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 (3.1) 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:

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

4.1 Local Bayesian optimization

- starting search dist - formula for search dist update
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 [4]. 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 $p_n^* = p(\boldsymbol{x} = \boldsymbol{x}^* | \mathscr{D}_n)$ 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

$$\begin{aligned} &\arg\min_{\pi} && \mathrm{KL}(\pi || p_n^\star), && \\ &\text{subject to} && \mathrm{KL}(\pi || \pi_n) && \leq \alpha, \quad (4.2) \\ & && \mathscr{H}(\pi_n) - \mathscr{H}(\pi) && \leq \beta, \quad (4.3) \end{aligned}$$

where $\mathrm{KL}(p||q) = \int p(\boldsymbol{x}) \log \frac{p(x)}{q(x)} d\boldsymbol{x}$ 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.

```
input  : $X$: $N$ already evaluated points
         $X_*$: $M$ random samples from the search distribution
output: $x_{n+1}$: next evaluation point
$m_d$ = mahalanobis distance from test points to incumbent search distribution
discard samples, which are outside 80% of the cumulative inverse-chi-squared distribution
get mean and covariance matrix from the Gaussian process
get values from Thompson sampling
$x_{n+1}$ = sample at the highest Thompson sampled value
```

Algorithm 2: Thompson Sampling acquisition for Local Bayesian optimization

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 (algorithm **??**).

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 3.
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.

4.3 Normalization constant

```
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

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 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 (4.1).

4.6.1 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.

4.6.2 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.

4.6.3 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 independet log-normal prior term as introduced in [3]:

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

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

4.7 Python OpenAI Gym from Matlab

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 the python module have to be converted to an array through `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[9]. This will facilitate future research working with higher dimensional problems.

5 Experiments

- 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 [10]. But later we implemented the 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 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.

5.2 Cart Pole

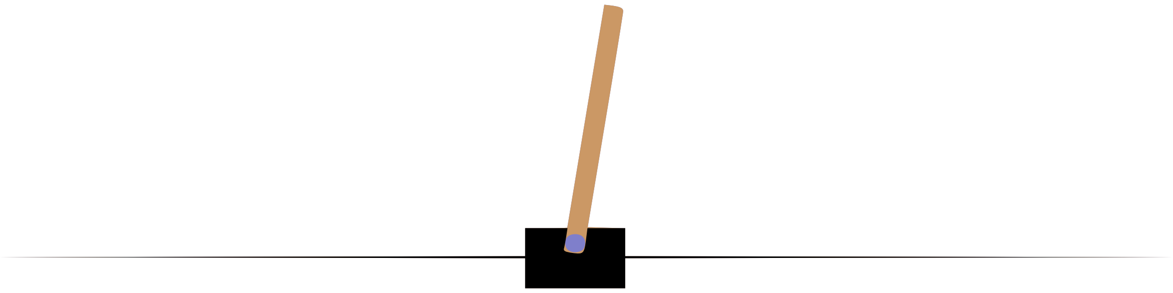

Figure 5.1.: Visualization of the Cart Pole rendered by the Open AI Gym

The Cart Pole environment consist 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 specifc 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 Open AI Gym (Figure 5.1) accepts a discrete action value for either applying force to the left (-1) or to the right (1). 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. Therefore this policy has only four dimensions.

5.3 Acrobot

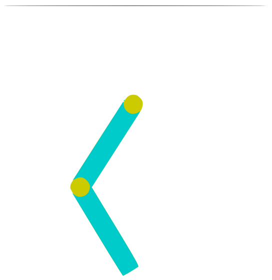

Figure 5.2.: Visualization of the Acrobot rendered by the Open AI Gym

The acrobatic robot has 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 goal is to gain enough momentum and swing the end of the second pendulum above a certain mark. 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. The discrete action can has 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.

5.4 Mountain car

In 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 hills are formed through a sine curve, therefore it is sufficient that the car's position is one dimensional only. The vertical position of the car is easily derived from the horizontal position with the sine function. The horizontal position and the car's velocity are cubically expanded (Table 5.1) to get nine state values. Also a bias parameter is added as a tenth state value.

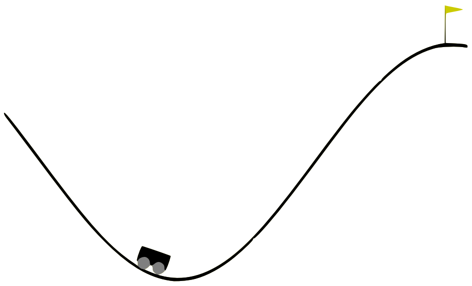

Figure 5.3.: Visualization of the mountain car rendered by the Open AI Gym

Environment	Description	Action	State feature	Policy dimension
Cart Pole	p: cart position θ: pole angle	continuous: [-1,1]	$(p, \dot{p}, \theta, \dot{\theta})$	4
		discrete: (-1,1)	$(p, \dot{p}, \theta, \dot{\theta}, 1)$	10
Acrobot	θ_1: angle of pendulum 1 θ_2: angle of 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
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
		discrete: (-1,0,1)	$(p, u, p^2, u^2, pu, p^2u, pu^2, p^3, u^3, 1)$	30

Table 5.1.: Environment parameters

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] 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.

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

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

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

[6] 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.

[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 implementation.”

A Some Appendix

Use letters instead of numbers for the chapters.