

Probabilistic Decision Making VU, WS25

Assignment 4

Gaussian Processes, Planning in a Grid World

Thomas Wedenig
thomas.wedenig@tugraz.at

Teaching Assistant: Leon Tiefenböck
Points to achieve: 20 Points
Deadline: 26.01.2026 23:59 (strict, no late submissions allowed)
Hand-in procedure: You can work in groups of **at most two people**.
Exactly one team member uploads 8 files to TeachCenter:
The report (.pdf), and all 5 Python files.
The first page of the report must be the **cover letter**.
Do not rename the Python files. Do not upload the data or figures.
Do not upload a folder. Do not zip the files.
We do not accept submissions via means other than TeachCenter.
Plagiarism: If detected, we grade *all involved parties* with
“Ungültig aufgrund von Täuschung”

General Remarks

Your submission will be graded based on:

- Correctness (Is your code doing what it should be doing? Is your derivation correct?)
- The depth of your interpretations (Usually, only a couple of lines are needed.)
- The quality of your plots (Is everything clearly readable/interpretable? Are axes labeled? ...)

Remarks:

- All results (i.e., plots, results of computations) should be included in your PDF report.
- Report all intermediate steps in pen & paper exercises—only presenting the final solution is insufficient.
- Your submission must run with Python 3.11.13 and the package versions listed in **requirements.txt**.
 - Check TeachCenter for instructions to setup a conda environment.
- Do not use any external packages except for the ones listed in **requirements.txt**.
- **Do not modify the function signatures** of the provided functions.
 - i.e., do not edit the function names and inputs
- Do not use Large Language Models (LLMs) to generate any part of your solution.
 - Evident LLM usage will be treated as plagiarism.

Failure to adhere to these rules may result in point deductions.

Task 1 — Gaussian Process Regression [14 Points]

In this task, we will consider a simple regression problem that we will tackle in a Bayesian way by using a Gaussian Process (GP). You will be given three datasets of the form

$$\mathcal{D} := \{(x^{(i)}, y^{(i)})\}_{i=1}^N, \quad x^{(i)} \in \mathbb{R}, \quad y^{(i)} \in \mathbb{R}$$

where we assume that

$$y^{(i)} = f(x^{(i)}) + \varepsilon, \quad \varepsilon \sim \mathcal{N}(0, \sigma_n^2)$$

for some unknown function $f: \mathbb{R} \rightarrow \mathbb{R}$. Also, the observation noise σ_n^2 is not known to us.

We will place a GP *prior* over functions f , which we (informally) denote as $p(f \mid \boldsymbol{\lambda}_k) = \mathcal{GP}(\mu(\cdot), k_{\boldsymbol{\lambda}_k}(\cdot, \cdot))$, where the mean function is chosen to be $\mu(x) \equiv 0$, $k_{\boldsymbol{\lambda}_k}: \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$ denotes the *kernel* (i.e., the *covariance function*), and $\boldsymbol{\lambda}_k$ denotes the hyperparameters of the kernel (e.g., length-scale, amplitude, period etc.) in log-space. We will collect all GP hyperparameters (in log-space) in $\boldsymbol{\lambda} := (\boldsymbol{\lambda}_k, \log \sigma_n)$, which is called **GPParams** in code.

Moreover, let $\mathbf{x} = (x^{(1)}, \dots, x^{(N)}) \in \mathbb{R}^N$ and $\mathbf{y} = (y^{(1)}, \dots, y^{(N)}) \in \mathbb{R}^N$ denote the training inputs and targets, and let $\mathbf{x}_* = (x_*^{(1)}, \dots, x_*^{(M)}) \in \mathbb{R}^M$, $\mathbf{f}_* = (f_*(x_*^{(1)}), \dots, f_*(x_*^{(M)})) \in \mathbb{R}^M$ denote (arbitrary) test inputs and their function values.

Task 1.1 [1.5 points] In `gp_kernels.py`, implement the following kernels using `jax`. While we provide the point-wise definitions below, note that your kernel implementations must return a *matrix* $k(\mathbf{x}, \mathbf{x}') \in \mathbb{R}^{N \times M}$ when being called with two *vectors* $\mathbf{x} \in \mathbb{R}^N, \mathbf{x}' \in \mathbb{R}^M$.

- In `rbf_kernel`, implement the *radial basis function* (RBF) (*squared exponential*) kernel

$$k(x, x') := \sigma^2 \exp\left(-\frac{\|x - x'\|^2}{2\ell^2}\right)$$

where σ is the amplitude and ℓ is a length-scale parameter.

- In `matern32_kernel`, implement the *Matérn* kernel with $\nu = 3/2$:

$$k(x, x') := \sigma^2 \left(1 + \frac{\sqrt{3}\|x - x'\|}{\ell}\right) \exp\left(-\frac{\sqrt{3}\|x - x'\|}{\ell}\right)$$

where σ is the amplitude and ℓ is a length-scale parameter.

- In `periodic_kernel`, implement the *periodic* (*sinusoidal*) kernel

$$k(x, x') := \sigma^2 \exp\left(-2 \frac{\sin^2(\pi\|x - x'\|/p)}{\ell^2}\right)$$

where σ is the amplitude, ℓ is a length-scale parameter, and p is the period.

Task 1.2 [0.5 points] In `gp_regression.py`, implement `gp_prior_sample`, which draws `num_functions` times from $p(\mathbf{f}_* \mid \mathbf{x}_*, \boldsymbol{\lambda}_k)$. Note that for numerical stability, we add a small value (`jitter`) to the diagonal of the covariance (kernel) matrix. Use `jax.random.multivariate_normal` to sample from a multivariate Gaussian.

Task 1.3 [1 point] In the `gp_regression_ml2` function of `gp_regression.py`, pick reasonable initial hyperparameters $\boldsymbol{\lambda}_{\text{init}} := (\log \sigma, \log \ell, \log p, \log \sigma_n)$ (that will be shared across kernels)¹. For each dataset and for each kernel, call `gp_prior_sample` to draw 5 functions from the prior, which will then be passed to the plotting function: This should populate the subplots in the first row of the figures that are saved to `figures_task1/dataset_name_ml2.pdf`. For each kernel, comment on (1) the shape of the samples for your choice of initial hyperparameters, and (2) the influence of each hyperparameter on the shape of the samples.

¹You can choose the same initial values for all datasets, or you can choose different values for each dataset.

Task 1.4 [3 points] Implement `gp_posterior`, which computes the finite-dimensional posterior

$$p(\mathbf{f}_* | \mathbf{x}_*, \mathcal{D}, \boldsymbol{\lambda}) = \mathcal{N}(\mathbf{f}_*; \mu_{\text{post}}(\mathbf{x}_*), \Sigma_{\text{post}}(\mathbf{x}_*)) \quad (1)$$

where

$$\mu_{\text{post}}(\mathbf{x}_*) = K_{*t} (K_{tt} + \sigma_n^2 I)^{-1} \mathbf{y}, \quad \Sigma_{\text{post}}(\mathbf{x}_*) = K_{**} - K_{*t} (K_{tt} + \sigma_n^2 I)^{-1} K_{*t}^\top$$

with $K_{tt} = k_{\lambda_k}(\mathbf{x}, \mathbf{x})$, $K_{*t} = k_{\lambda_k}(\mathbf{x}_*, \mathbf{x})$, and $K_{**} = k_{\lambda_k}(\mathbf{x}_*, \mathbf{x}_*)$.

Note: For numerical stability, we again add a small value (`jitter`) to the diagonal of K_{tt} , and we never want to use direct matrix inverses (`jnp.linalg.inv`): Instead, we will use `jnp.linalg.solve`², which is numerically more stable³.

In `gp_regression_ml2`, call `gp_posterior` to compute $\mu_{\text{post}}, \Sigma_{\text{post}}$ (called `mean_init`, `cov_init` in the code) with the initial hyperparameters $\boldsymbol{\lambda}_{\text{init}}$ that you have chosen previously. $\mu_{\text{post}}, \Sigma_{\text{post}}$ are then passed to the plotting function, which should populate the subplots in the second row of the figures that are saved to `figures_task1/dataset_name_ml2.pdf`.

Note that we compute the posterior over *function values* at test points \mathbf{x}_* , i.e., $p(\mathbf{f}_* | \mathbf{x}_*, \mathcal{D}, \boldsymbol{\lambda})$. How would the computation change if we instead wanted to compute the *posterior predictive* $p(\mathbf{y}_* | \mathbf{x}_*, \mathcal{D}, \boldsymbol{\lambda})$? How would this change the plot in the second row?

Task 1.5 [0.5 points] Use big- O notation to quantify the computational complexity of performing posterior inference in a GP (Eq. 1) as a function of N , the number of training data points. Briefly comment on the computational complexity.

Task 1.6 [2 points] Implement `gp_log_marginal_likelihood`, which computes the log of the *marginal likelihood*

$$p(\mathbf{y} | \mathbf{x}, \boldsymbol{\lambda}) = \int p(\mathbf{y} | f, \mathbf{x}, \boldsymbol{\lambda}) p(f | \boldsymbol{\lambda}) df.$$

In a GP, this can be computed analytically as

$$\log p(\mathbf{y} | \mathbf{x}, \boldsymbol{\lambda}) = -\frac{1}{2} (\mathbf{y}^\top \Sigma^{-1} \mathbf{y} + \log |\Sigma| + N \log 2\pi). \quad (2)$$

where $\Sigma := K_{tt} + \sigma_n^2 I$. Hint: Again use `jnp.linalg.inv` to avoid explicit matrix inverses, and use `jnp.linalg.slogdet` to compute the determinant directly in log-space.

Provide an informal, intuitive explanation of this quantity. What does it tell us about the model?

Task 1.7 [2 points] We can use the *marginal likelihood* to fit the hyperparameters $\boldsymbol{\lambda}$ of the GP, i.e., by finding

$$\boldsymbol{\lambda}_{\text{ML2}} = \underset{\boldsymbol{\lambda}}{\operatorname{argmax}} p(\mathbf{y} | \mathbf{x}, \boldsymbol{\lambda}). \quad (3)$$

This is often called *Type-II Maximum Likelihood Estimation*.

- The function `fit_hyperparams_ml2` is already implemented for you and uses the Adam optimizer to find a local maximum of the log marginal likelihood (LML)⁴. Pass $\boldsymbol{\lambda}_{\text{init}}$ as the starting point for optimization, pick the learning rate `lr`, and run a reasonable number of optimization steps to obtain $\hat{\boldsymbol{\lambda}}_{\text{ML2}}$ and an array of loss values (the respective negative LMLs).
- When maximizing the LML in Eq. 2, what does the term $-\frac{1}{2} \mathbf{y}^\top \Sigma^{-1} \mathbf{y}$ measure and what does the term $-\frac{1}{2} \log |\Sigma|$ encourage?

²https://docs.jax.dev/en/latest/_autosummary/jax.numpy.linalg.solve.html

³In most implementations, one typically uses a Cholesky decomposition since it is faster and more stable, but we avoid this here for simplicity.

⁴Thanks to `jax`, this is easy to implement as $\nabla_{\boldsymbol{\lambda}} \log p(\mathbf{y} | \mathbf{x}, \boldsymbol{\lambda})$ is available to us via autodiff.

- Compute the mean and covariance of the new posterior $p(\mathbf{f}_* | \mathbf{x}_*, \mathcal{D}, \hat{\boldsymbol{\lambda}}_{\text{ML2}})$, which are passed to the plotting function (`mean_optimized`, `cov_optimized`) (third row).
- Also compute $\log p(\mathbf{y} | \mathbf{x}, \boldsymbol{\lambda}_{\text{init}})$ (called `lml_init`) and $\log p(\mathbf{y} | \mathbf{x}, \hat{\boldsymbol{\lambda}}_{\text{ML2}})$ (called `lml_optimized`), which are then passed to the plotting function, together with the loss array obtained earlier (fourth row in the plot).
- Include all plots in your report. For each dataset, comment on how the choice of kernel influences (1) the posterior before type-II maximum likelihood, and (2) the posterior after type-II maximum likelihood. Are there posteriors (before or after type-II maximum likelihood) that do not model the data well, while being very confident? If so, why?
- What happens to the posterior in regions where there are no training data points? How does this relate to the kernel choice?
- For each dataset, report which kernel and hyperparameters achieve the highest marginal likelihood. Comment on these results.

Task 1.8 [3.5 points] Up until this point, we have only been *partially* Bayesian since we have not considered the hyperparameters of the GP $\boldsymbol{\lambda}$ as random variables (which we would integrate out in a Bayesian approach), but instead came up with a single point estimate $\hat{\boldsymbol{\lambda}}_{\text{ML2}}$. In this task, we will switch to being fully Bayesian by (1) placing a prior $p(\boldsymbol{\lambda})$ over the hyperparameters, and (2) (approximately) integrating them out to estimate

$$p(\mathbf{f}_* | \mathbf{x}_*, \mathcal{D}) = \int p(\mathbf{f}_* | \mathbf{x}_*, \mathcal{D}, \boldsymbol{\lambda}) p(\boldsymbol{\lambda} | \mathcal{D}) d\boldsymbol{\lambda}. \quad (4)$$

- In `numpyro_lambda_model`, pick a reasonable prior $p(\boldsymbol{\lambda})$ over the hyperparameters using `numpyro.sample`. Justify your choices in your report.
- To get intuition how functions from the prior look like, we wish to sample from the hyperparameter-marginalized prior $p(\mathbf{f}_* | \mathbf{x}_*) = \int p(\mathbf{f}_* | \mathbf{x}_*, \boldsymbol{\lambda}) p(\boldsymbol{\lambda}) d\boldsymbol{\lambda}$. Briefly describe how you can do this and implement it in `gp_regression_fully_bayesian`. Draw 5 functions and pass them to the plotting function. Hint: Use `sample_prior_lambda` to sample from the prior $p(\boldsymbol{\lambda})$ you have previously defined. If you have issues with numerics, try a slightly larger value for `jitter`.
- Finally, we wish to sample many times from the hyperparameter-marginalized *posterior* $p(\mathbf{f}_* | \mathbf{x}_*, \mathcal{D})$ as defined in Eq. 4. To this end, we will use MCMC (NUTS) to sample from the hyperparameter posterior $p(\boldsymbol{\lambda} | \mathcal{D})$ by noticing that

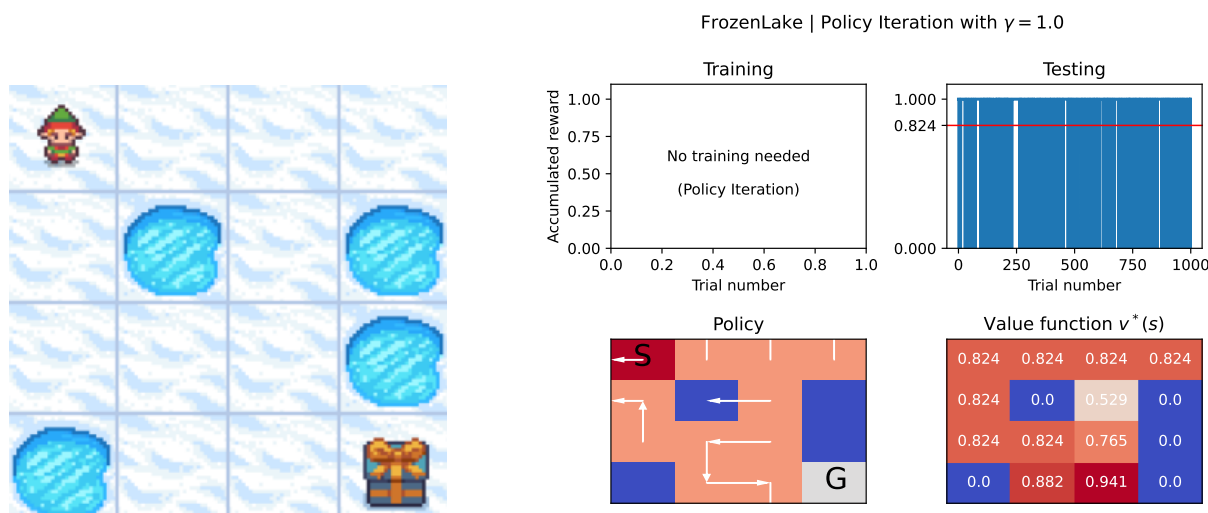
$$p(\boldsymbol{\lambda} | \mathcal{D}) = p(\boldsymbol{\lambda} | \mathbf{x}, \mathbf{y}) \propto p(\mathbf{y} | \mathbf{x}, \boldsymbol{\lambda}) p(\boldsymbol{\lambda}),$$

which is just the marginal likelihood times the prior, which is already implemented in `numpyro_lambda_model` when `use_likelihood=True`. Use a single chain and a reasonable number of warmup steps to draw 100 MCMC samples from $p(\boldsymbol{\lambda} | \mathcal{D})$ in `gp_regression_fully_bayesian`.

- Which of the hyperparameter posteriors $p(\boldsymbol{\lambda} | \mathcal{D})$ is hardest to sample from? Why?
- For each MCMC sample $\boldsymbol{\lambda}$, draw a sample from $p(\mathbf{f}_* | \mathbf{x}_*, \mathcal{D}, \boldsymbol{\lambda})$, which are then passed to the plotting function (together with the average function sample).
- The plots will be saved as `figures_task1/dataset_name_fully_bayesian.pdf`. Include all plots in your report. For each dataset, comment on the quality of the posterior samples. Are there posteriors that do not model the data well, while being very confident?
- Compare the uncertainties of the type-II maximum likelihood and fully Bayesian approaches. Explain your findings.

Task 2 — Planning in a Grid World [6 Points]

A grid world is a typical environment with finite action and state spaces. We will solve the FrozenLake⁵ environment from the `gymnasium` package (a fork of OpenAI's Gym package). The agent controls the movement of a character in a grid world. Some tiles of the grid are walkable, and others lead to the agent falling into the water. Additionally, the movement direction of the agent is uncertain and only partially depends on the chosen direction (the lake is slippery). The agent receives a reward of 1 if it moves the goal state and 0 else. The episode ends as soon as the agent falls into the water or reaches the goal state. Further information can be found in the Gymnasium Documentation.



(a) FrozenLake environment. The starting state is on the top left while the goal state is on the bottom right.

(b) Value function and policy that were acquired using *Policy Iteration* ($\gamma = 1$). Testing the policy using 1000 episodes yields an average sum of rewards of 0.824.

Task 2.1 [5 points] Assume we know the dynamics of the underlying MDP (i.e., the state transition probabilities and the expected rewards). Fill in the TODOs in `policy_iteration.py`: You are supposed to implement the *Policy Iteration* algorithm, which consists of repeatedly *evaluating a policy* (using *Iterative Policy Evaluation*) and improving this policy by acting greedily w.r.t. the Q function of the old policy. When finished with the TODOs, execute the file: This will run policy iteration until convergence for both the undiscounted case ($\gamma = 1$) and a discounted case ($\gamma = 0.95$). In both scenarios, the code framework will produce a plot which includes (1) the policy you have found, (2) the corresponding value function, and (3) the reward that was accumulated in 1000 simulated episodes (and its average). Include both plots ($\gamma = 1$ and $\gamma = 0.95$) in your report. Explain any differences between the two policies, the two value functions, and between the average test success rate (i.e., the fraction of simulated episodes in which the agent reached the goal state). Briefly discuss why these differences appear.

Task 2.2 [0.5 points] Assume that s_{start} is the starting state. In the undiscounted case ($\gamma = 1$), what's the relationship between $v^*(s_{\text{start}})$ and the average reward you encounter when acting according to π^* for N episodes (i.e., `np.mean(policy_iteration.test_policy(num_episodes=N))`)? What happens as $N \rightarrow \infty$? Write down the definition of the v function in this particular case ($\gamma = 1$, binary reward per episode).

Task 2.3 [0.5 points] In general: Is the optimal policy π^* unique? Is the optimal value function v^* unique? Explain your reasoning.

⁵https://gymnasium.farama.org/v1.0.0/environments/toy_text/frozen_lake/

Task 3 – Peer Review [0 Points]

Each group member must write a single paragraph outlining their opinion on the work distribution within the group. Did every group member contribute equally? Did you split up tasks in a fair manner, or jointly worked through the exercises? Do you think that some members of your group deserve a different grade from others?