

Statistical Model Based Reinforcement Learning

Statistisches Modellbasiertes Reinforcement Learning

Master thesis by Fabio d'Aquino Hilt

Date of submission: July 13, 2023

1. Review: Joe Watson
 2. Review: João Carvalho
 3. Review: Jan Peters
- Darmstadt



TECHNISCHE
UNIVERSITÄT
DARMSTADT



Erklärung zur Abschlussarbeit gemäß § 22 Abs. 7 APB TU Darmstadt

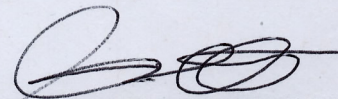
Hiermit erkläre ich, Fabio d'Aquino Hilt, dass ich die vorliegende Arbeit gemäß § 22 Abs. 7 APB der TU Darmstadt selbstständig, ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt habe. Ich habe mit Ausnahme der zitierten Literatur und anderer in der Arbeit genannter Quellen keine fremden Hilfsmittel benutzt. Die von mir bei der Anfertigung dieser wissenschaftlichen Arbeit wörtlich oder inhaltlich benutzte Literatur und alle anderen Quellen habe ich im Text deutlich gekennzeichnet und gesondert aufgeführt. Dies gilt auch für Quellen oder Hilfsmittel aus dem Internet.

Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Mir ist bekannt, dass im Falle 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 einer Thesis des Fachbereichs Architektur entspricht die eingereichte elektronische Fassung dem vorgestellten Modell und den vorgelegten Plänen.

Darmstadt, 13. Juli 2023



F. d'Aquino Hilt

Abstract

Robotic control problems are particularly challenging to solve with reinforcement learning (RL) because of the time-consuming and costly nature of sampling real-world environments. Model-based reinforcement learning (MBRL) aims to tackle this problem by learning a separate dynamics model. While Gaussian processes (GP) provide expressive models with low model bias, they face scalability issues with large datasets. This thesis uses spectral-normalized neural Gaussian processes (SNGP) to address this problem to obtain scalable GP approximations and demonstrate its expressive capabilities in a MBRL setting. Furthermore, we explore the integration of the SNGP model for policy and dynamics models and input inference for control (I2C) for stochastic trajectory optimization. This thesis contributes to the exploration of novel approaches in MBRL, highlighting the potential of SNGP models and I2C optimizations for efficient and effective control in real-world environments.¹

¹Code available at: <https://github.com/neuralskeptic/smbml>

Contents

1. Introduction	2
2. Foundations	4
2.1. Reinforcement Learning	4
2.1.1. Model-free reinforcement learning	4
2.1.2. Model-based reinforcement learning	6
2.2. Dataset aggregation for imitation learning	8
2.3. Spectral-normalized neural Gaussian process	9
2.3.1. Distance preserving hidden layers	10
2.3.2. Distance aware output layers	10
2.4. Input inference for control	12
2.4.1. Cost function as likelihood	13
2.4.2. Approximate inference	13
2.4.3. Bayesian smoothing	13
2.4.4. Temperature update	14
2.4.5. Policy as conditional action	14
3. Gaussian process model	15
3.1. Bayesian multivariate linear regression	15
3.2. Variational posterior	16
3.3. Posterior predictive and model evidence	17
3.4. Evidence lower bound loss: fitting to deterministic targets	18
3.5. Moment-matching loss: fitting to target distributions	18
4. Experiment: Gaussian process model de-risking	20
4.1. Setup	20
4.1.1. Environment: Furuta pendulum swing-up with custom cost	20
4.1.2. Setup modified SNGP model	21
4.1.3. SAC agent training	21

4.1.4. SNGP dynamics model training: from expert policy rollouts	21
4.1.5. SNGP policy training: behaviour cloning from SAC with DAgger . .	22
4.1.6. Closed-loop evaluations	23
4.2. Evaluation	24
4.2.1. SNGP dynamics model training: from expert policy rollouts	24
4.2.2. SNGP policy: behaviour cloning from SAC (DAgger)	28
4.2.3. Closed-loop evaluations	30
4.3. Summary	33
5. Experiment: input inference for control in the model-based reinforcement learning loop	34
5.1. Setup	35
5.1.1. Environment: simple pendulum swing-up	35
5.1.2. Setup models and approximate inference	35
5.1.3. Setup I2C optimization	36
5.1.4. SNGP dynamics model training: from policy rollouts with ELBO loss	38
5.1.5. I2C: find locally optimal controllers from dynamics	38
5.1.6. SNGP policy training: from optimal I2C trajectories	39
5.2. Evaluation	41
5.2.1. SNGP dynamics model learning	41
5.2.2. I2C: find locally optimal controllers	46
5.2.3. SNGP policy learning from local controllers	53
5.3. Summary	60
6. Conclusion	62
A. Matrix normal distribution	66
B. Evidence lower bound loss derivation	68
B.1. KL of posterior from prior	69
B.2. Log likelihood	70
B.3. Expected log likelihood	71
B.4. Evidence lower bound	73
C. Code	74
C.1. Quanser Qube environment reward function modifications	74

List of Abbreviations

Notation	Description
Dagger	dataset aggregation
ELBO	evidence lower bound
GP	Gaussian process
GPS	guided policy search
i.i.d.	independently and identically distributed
I2C	input inference for control
iLQR	iterative linear quadratic regulator
KL	Kullback-Leibler
MBRL	model-based reinforcement learning
MDP	Markov decision process
PILCO	probabilistic inference for learning control
RBF	radial basis functions
RFF	random Fourier features
RL	reinforcement learning
RMSE	root mean squared error
SAC	soft actor critic
SNGP	spectral-normalized neural Gaussian process
TVLGC	time-varying linear Gaussian controller
w.r.t.	with respect to

1. Introduction

Reinforcement learning (RL) solves sequential control problems via trial and error through interactions with the environment. This works very well for simulated environments like video games, which permit a huge number of interactions. However, real world environments such as robotic control problems are time-consuming and expensive to sample and are vulnerable to control errors, which can lead to destruction of equipment or bodily injury. Model-based reinforcement learning (MBRL) promises to facilitate RL in such applications by learning a separate dynamics model and policy. This reduces the required number of interactions and allows for testing of the policy before running it in the real world.

There are many different approaches to MBRL, some of which build white or grey box dynamics models. These kinds of dynamics model can be used in trajectory optimization to get optimal action sequences to be distilled into more complex policy models. Guided policy search (GPS) [1, 2, 3, 4, 5] for instance learns a linearized local dynamics model and finds the approximately optimal action sequence using iterative linear quadratic regulator (iLQR) [6, 7]. This also means GPS has to regularize the policy updates in order to prevent excessive exploitation of the inaccuracies of the local linear dynamics model. Probabilistic inference for learning control (PILCO) [8], on the other hand, trains a global Gaussian process dynamics model where the local policy is trained with backpropagation of gradients w.r.t. evaluations of the policy in the dynamics model using approximate inference. This yields excellent global dynamics models, but Gaussian processes do not scale to large datasets and backpropagation through time is brittle.

This work aims to combine the excellent properties of Gaussian processes (GP) with trajectory optimization to learn both a global dynamics model and a global policy. To obtain a scalable GP approximation we modify spectral-normalized neural Gaussian processes (SNGP) [9], and for stochastic trajectory optimization we use input inference for control (I2C) [10] with approximate non-linear inference, which improves on iLQR.

This thesis is structured as follows: Chapter 2 contains the core background concepts which are used and extended in this thesis. In Chapter 3 we introduce the modified SNGP model, which is then analyzed in Chapter 4. The analysis proceeds with Chapter 5, where a MBRL loop is set up containing a SNGP dynamics model, a SNGP policy and the I2C optimization algorithm, and the individual steps in the loop are scrutinized. The thesis concludes with a summary and directions for future work in Chapter 6.

2. Foundations

2.1. Reinforcement Learning

Reinforcement Learning (RL) is a machine learning approach to solve optimal control problems with the assumption that the system dynamics are time-discrete. This time-discretized dynamics is called Markov decision process (MDP) and is formalized by a set of states S , actions A , a transition dynamics probability function $f : S \times A \times S \rightarrow [0, 1]$, and a reward (or cost) function $R : S \times A \rightarrow \mathbb{R}$. However, the reward is not really a part of the system dynamics, since it is usually chosen by the engineer to describe the objective, and careful reward shaping helps the optimization to navigate the unknown transition dynamics and find optimal controllers. Nevertheless, it is usually treated as unknown to the optimization and part of the system dynamics. The goal is to find a policy $\pi : S \rightarrow A$ that manages to control the system in a way that the trajectory $s_0, a_0, s_1, a_1, \dots$ resulting from the alternative evaluation of the policy $a_t = \pi(s_t)$ and the transition function $s_{t+1} = f(s_t, a_t)$ starting from an initial state $s_0 \in S$ has a maximal cumulative reward. When the problem is stated for a fixed time frame it is called "finite-horizon", and "infinite-horizon" problems have an additional discount factor $0 < \gamma < 1$ which is multiplied to the future rewards to keep the cumulative reward finite.

2.1.1. Model-free reinforcement learning

Model-free RL tries to learn a (stochastic) policy from observations of rewards in the environment, without any assumptions about the structure of the transition function. Since the policy has to also learn the combined complexity of the transition dynamics and the reward, model-free policies are usually more complex and thus more flexible than their model-based counterparts, which in turn means that more observations are needed to learn a good policy. The fact that it is not possible to execute the dynamics

model offline to generate new samples additionally increases the need for environment interactions. Storing the sampled observations in a replay buffer to be used again for training, and possibly heuristically generating new samples from the buffer, slightly increases the sample-efficiency.

2.1.1.1. Soft actor critic

Soft actor critic (SAC) [11] is a model-free RL algorithm, which uses entropy regularization to optimize a stochastic policy. Entropy regularization means that the reward function is augmented with the entropy of the policy at the current state

$$\hat{R}(s, a, s') := R(s, a, s') + \alpha H[\pi(\cdot|s)] \quad (2.1)$$

To cope with complex state-dependent reward structures, it learns the reward function with an ensemble of two Q-function networks, which are optimized by minimizing the mean squared error from the current state observed reward and the next state entropy augmented reward

$$L(\phi_j, \mathcal{D}) = \mathbb{E}_{(s,a,s',r) \sim \mathcal{D}} [(Q_{\phi_j}(s, a) - \hat{r})^2] \quad (2.2)$$

$$\hat{r} = r + \mathbb{E}_{a'_\pi \sim \pi_\varphi(\cdot|s')} [Q_{\phi_j}(s', a'_\pi) - \alpha H[\pi(a|s')]] \quad (2.3)$$

where \hat{r} is considered constant (the Q-function inside is not optimized). The policy is built from a mean and a covariance neural network and is learned by minimizing the expected backwards Kullback-Leibler (KL) divergence (information projection [12]) from the policy to the exponential minimum of the two Q-functions

$$\min_{\varphi} D_{\text{KL}} (\alpha \pi_{\varphi}(a|s) \parallel \exp\{\min_{j=1,2} Q_{\phi_j}(s, a)\}) \quad (2.4)$$

which is equivalent to maximizing the entropy augmented Q-function reward over policy predictions. The temperature parameter α scales the weight of the entropy and good temperature values have to be experimentally determined for each problem individually.

2.1.2. Model-based reinforcement learning

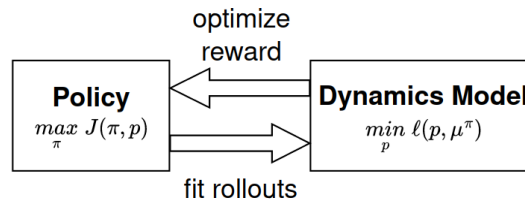


Figure 2.1.: Alternating optimization of policy and dynamics model typically used in model-based reinforcement learning

In contrast to model-free RL, where there is no explicit assumption about the distribution of the transitions observations, model-based RL (MBRL) explicitly models the transition dynamics to be able to generate new dynamics predictions during the training of the policy without needing new environment interactions. This naturally leads to a dual optimization problem of both the policy and the dynamics model, where in turns the one is held fixed while the other is optimized. Even though this dual optimization is generally more complex than optimizing a single model, the models involved in MBRL are usually simpler than the policy in model-free RL. Leveraging the dynamics model for policy optimization reduces the number of samples required, which are usually only needed to update the dynamics model.

2.1.2.1. Probabilistic inference for learning control

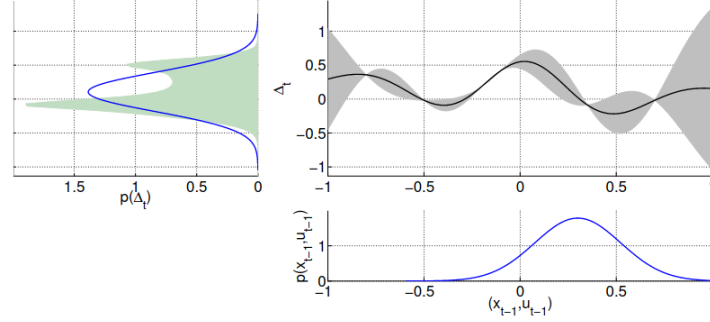


Figure 2.2.: “GP prediction at an uncertain input. The input distribution $p(x_{t-1}, u_{t-1})$ is assumed Gaussian (lower right panel). When propagating it through the GP model (upper right panel), we obtain the shaded distribution $p(\Delta_t)$, upper left panel. We approximate $p(\Delta_t)$ by a Gaussian with the exact mean and variance (upper left panel)” [8]

Probabilistic inference for learning control (PILCO) [8] is a MBRL algorithm which learns a global dynamics model from samples collected in the environment and a local policy model by optimizing the reward given the dynamics model. The global dynamics are modeled by a non-parametric Gaussian Process (GP) [13], which is learned by maximizing the marginal likelihood over all the collected transitions, which are sampled by executing the policy in the environment. The local policy is a radial basis function (RBF) network and is found by optimizing the reward received from rolling out the policy in the dynamics model and using back-propagation through time to change the policy parameters. This means that every policy is effectively a locally optimal controller given the current dynamics model. The GP dynamics model captures its own epistemic uncertainty very well, but does not scale to large amounts of transitions or more complex problems, because the marginal likelihood has to be computed over the entire data set. The local RBF policy optimization uses regularized model updates to constrain policy updates, but back-propagation through time is brittle, since gradients can easily explode or disappear if the optimization has to recurse through many time steps.

2.1.2.2. Guided policy search

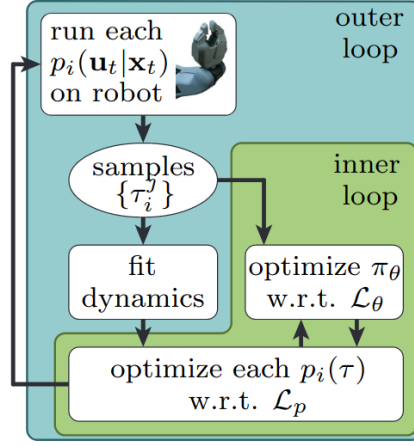


Figure 2.3.: A schematic overview of Guided Policy Search [4]

Guided policy search (GPS) [1, 2, 3, 4, 5] is a MBRL method which learns a local dynamics model and a global policy. The dynamics model is a Gaussian mixture model, i.e. time-varying linear dynamics, and is learned around the latest trajectory (policy roll-out) from the collected observations. Since only the dynamics close to the latest roll-out are modeled, the dynamics model is only locally valid. The policy is modeled by a μ and a Σ network and is found by minimizing the Kullback-Leibler (KL) divergence between the policy prediction and local controllers, where the controllers come from finding an optimal action trajectory for the linear dynamics model using (linearized) dynamic programming. The linear dynamics model allows fast and easy optimization, but linearizing the true non-linear dynamics makes an even locally quite crude model. The policy on the other hand accumulates knowledge and the maximum entropy policy update creates good regularization [14].

2.2. Dataset aggregation for imitation learning

Imitation learning is generally hard for most RL problems because of their sequential structure. For instance, the samples observed at different time steps are not i.i.d. (temporal

causality) and small prediction errors can accumulate over time giving rise to unstable or chaotic behaviour (no recovery from errors), all of which result in bad test-time performance even when the training fit is good. Dataset aggregation (DAgger) [15] uses "no-regret online learning" to tackle this problem to make imitation learning useful for RL. The idea is very simple and intuitive: instead of training on roll-outs of the expert policy, which tends to do little exploration, use the (incompletely trained) imitating policy for exploration. Then run the expert policy on the collected states to find out how to recover from those states, and aggregate the explored states and expert actions to the training data. Repeatedly training the imitating policy on the growing data set made up of states resulting from its own action errors and the expert policy recovery actions works very well and creates robust policies, since it trains the policy on expert actions over the area of state-space it is likely to encounter at test-time.

2.3. Spectral-normalized neural Gaussian process

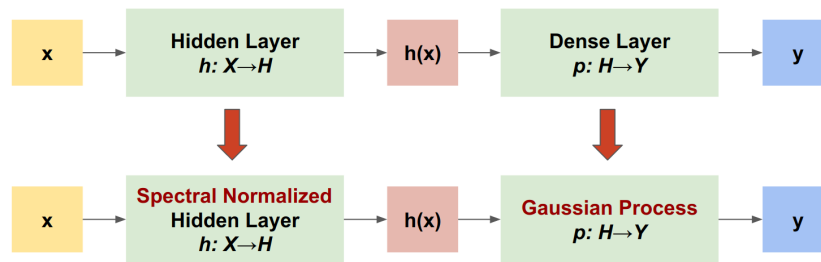


Figure 2.4.: Improvements of SNGP over the standard deep learning framework¹

Spectral-normalized neural Gaussian processes (SNGP) [9] are Gaussian processes (GP) created with the aim of being more scalable than GPs and with better input distance awareness than neural networks. Input distance awareness means having bounds on the distance of outputs $\|h(x) - h(x')\|$ given the input distances $\|x - x'\|$ and is an important guarantee for good epistemic uncertainty of GP models. As shown in Figure 2.4, SNGPs can also be thought of as GPs with spectral-normalized neural network kernel functions.

¹image source: <https://www.tensorflow.org/tutorials/understanding/sngp>

2.3.1. Distance preserving hidden layers

To make the hidden layers of a neural network distance preserving, SNGPs bound the Lipschitz constants of all nonlinear mappings (linear layers with activations) to be less than 1 [16]. This means the hidden layer has to satisfy the bi-Lipschitz condition

$$L_1 \cdot \|x - x'\| \leq \|h(x) - h(x')\| \leq L_2 \cdot \|x - x'\| \quad (2.5)$$

for positive bounded $0 < L_1 < 1 < L_2$, which encourages the hidden layers to be an approximate isometric (distance preserving) mapping. This bound is enforced by adding spectral normalization to the weight matrices W of each hidden layer: when the largest singular value of W becomes too big $\lambda > c$, the weight matrix is normalized $W \leftarrow c \cdot W / \lambda$, where the hyperparameter c can be tuned to limit the normalization to increase the expressive power of the network. Obviously it would be very computationally expensive to compute the singular value decomposition (SVD) for every weight matrix after every weight update, therefore, in practice, SNGPs approximate SVD with power iteration [17, 18]. Power iteration updates the left-approximated and right-approximated singular vectors (of the largest singular value) in an alternating algorithm which requires few iterations for a good approximation

$$v = \frac{W^\top u}{\|W^\top u\|_2} \quad (2.6)$$

$$u = \frac{Wv}{\|Wv\|_2} \quad (2.7)$$

where the resulting spectral norm is computed as $u^\top Wv$. Since the weight matrices usually do not change quickly during mini-batch training, it is often sufficient to do only one power iteration per mini-batch to keep the approximated spectral norm close to the true value.

2.3.2. Distance aware output layers

Making the output layers distance aware is mainly a problem of choosing a scalable approximation of the exact GP posterior and marginal likelihood. SNGPs first approximate the infinite-dimensional GP with finite dimensional Bayesian linear model using random Fourier features (RFF) [19] to create expressive and scalable feature functions while keeping the neural network output layer size manageable. Secondly, SNGPs use Laplace approximation to get tractable posteriors.

2.3.2.1. Random Fourier features

For exact GP inference, the marginal likelihood has to be computed, which requires the entire dataset and scales cubically $\mathcal{O}(N^3)$ in the number of data points N . If the infinite-dimensional kernel function is approximated using a finite feature space of M dimensions, this can be thought of as a "reverse kernel trick", which is advantageous when the feature space dimension is less than the number of data points $M < N$. Random Fourier features hinge on Bochners theorem (see [19] or [20]), which states that every positive definite shift-invariant (stationary) kernel is proportional to the Fourier transform of a kernel-specific probability distribution

$$k(x - y) \propto \frac{1}{2\pi} \int p(\omega) e^{j\omega^\top(x-y)} d\omega \quad (2.8)$$

For instance, the Gaussian (radial basis function) kernel $e^{-\frac{\|x-y\|^2}{2\sigma^2}}$ is equal to the Fourier transform of the standard normal distribution $\mathcal{N}(0, 1)$. Since both the kernel and the probability distribution are real, we can further rewrite

$$k(x - y) \propto \mathbb{E}_\omega[\cos(\omega^\top(x - y))] = \mathbb{E}_\omega[z_\omega(x)^\top z_\omega(y)] \quad (2.9)$$

using the Fourier feature functions

$$z_\omega(x) := \left[\cos(\omega^\top x), \sin(\omega^\top x) \right]^\top \quad (2.10)$$

We can therefore get unbiased estimates of the kernel by using monte-carlo integration

$$k(x - z) \propto \mathbb{E}_\omega[z_\omega(x)^\top z_\omega(y)] \approx \frac{1}{D} \sum_{i=1}^D z_{\omega_i}(x)^\top z_{\omega_i}(y) = z(x)^\top z(y) \quad (2.11)$$

where $\omega_1, \dots, \omega_D$ are i.i.d. samples drawn from $p(\omega)$ and the D-dimensional RFF function $z(x)$ becomes

$$z(x) := \sqrt{\frac{1}{D}} \left[\cos(\omega_1^\top x), \sin(\omega_1^\top x), \dots, \cos(\omega_D^\top x), \sin(\omega_D^\top x) \right]^\top \quad (2.12)$$

Conveniently, the estimation converges uniformly and exponentially fast in D. However, RFFs can suffer from variance starvation [21], because with an increasing number of data points, the finite degrees of freedom can get pinned more and more, resulting in overconfident predictions. Variance starvation can be countered by increasing the number of RFFs at the expense of higher computational cost.

2.3.2.2. Laplace approximation of the posterior

To approximate the generally non-Gaussian weight posterior with a Gaussian distribution, SNGPs use Laplace’s approximation [22]. The approximated mean is the posterior mode, which is the maximum a posteriori solution, and is updated together with the hidden layer weights on every minibatch using stochastic gradient descent and squared loss. The approximated precision is the observed Fischer information matrix, which is the Hessian of the posterior log-likelihood at the posterior mode, and is only computed on the final epoch using the learned mean. Since Bayesian linear regression with RFF is a finite-rank model, the Bernstein von Mises theorem states that Laplace’s approximation is asymptotically exact, i.e. it converges to the true posterior with many samples.

2.4. Input inference for control

Input inference for control (I2C) [23, 24, 10] is a stochastic optimal control (SOC) method which can be used for trajectory optimization. For Gaussian uncertainties it formulates optimal control as input inference using approximate Bayesian smoothing [25] where the found trajectories are joint Gaussian distributions over the state and action for each time step in a finite-horizon MDP (see Section 2.1). It requires a state-action cost function $C : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$ and a stochastic state transition dynamics $f : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \rightarrow [0, 1]$, both of which can be arbitrarily complex (e.g. non-linear), but I2C has to be able to randomly evaluate them. Since propagating Gaussian priors through non-linear dynamics and cost generally results in non-Gaussian posteriors, I2C uses approximate inference [26] to approximate the posteriors with Gaussians, such that they can be used as priors for the next time step. I2C starts from an initial action sequence and first state and recursively predicts the next state distribution with the dynamics function applied to the current state and action from the sequence. To find optimal joint state-action trajectories with minimal cumulative cost, I2C nudges the joint state-action distribution of each time step towards lower cost. Smoothing this nudged state-action trajectory results in an action sequence with lower cumulative cost. The strength of the nudge is controlled by a temperature parameter α , which in turn is optimized after every smoothing pass similarly to expectation maximization [27]. The alternating of smoothing and optimizing the temperature parameter until convergence results in a locally optimal posterior state-action trajectory which can be synthesized into a Gaussian feedback controller by computing the Gaussian conditional action distribution (in closed form).

2.4.1. Cost function as likelihood

To use the cost function in an inference setting, I2C applies *belief in optimality* [28] to create a likelihood of optimality

$$p(\text{optimal}|x_t, u_t) \propto \exp(-\alpha C(x_t, u_t)) \quad (2.13)$$

with an inverse temperature parameter α which controls the variance of the likelihood. This likelihood has its peak at the minimal cost and for a quadratic cost function the likelihood is Gaussian.

2.4.2. Approximate inference

Since analytic posterior distributions are often non-Gaussian and the integrals of smoothing are not tractable, posteriors are usually locally approximated by Gaussian distributions. In I2C three approximation methods are presented: The first method, local linearization only uses the posterior mean and its derivatives, which makes it brittle but easy to compute. The second method is the spherical Gaussian (cubature) quadrature rule, which samples the function at $2d + 1$ so called *sigma points*, spaced in each of the d dimensions along the principal axes of the covariance matrix, and weighs each sample such that the sum approximates the integral. This method is similar to the unscented transform popularized by the unscented Kalman filter [29]. The third method is Gauss-Hermite quadrature which builds a mesh of samples and hence does not scale well to large dimensions, but is able to more accurately capture local non-linear behaviour.

2.4.3. Bayesian smoothing

The central I2C algorithm is based on the Rauch Trung Striebel (RTS) smoother [25, 30]. It has a filtering (forward) pass and a smoothing (backward) pass resulting in temporal chains of probability distributions, which are created by recursively propagating the initial state distribution through the action generating function (current optimal action sequence) and the observation function (dynamics) for every time step. But differently from the RTS smoother it also propagates the state-action distribution through the cost likelihood function, which nudges the trajectory towards lower cost. This nudge is artificially added by the optimizer and is not a part of the dynamics, and therefore warps the filtering and smoothing trajectories, but allows the optimization to be framed as input estimation to

figure out the unknown actions in each time step. The goal of the optimization is to find a trajectory where the nudge disappears, which means that it has reached minimal cost.

2.4.4. Temperature update

To achieve good and fast convergence, a good temperature update schedule has to be chosen, but how to find an optimal schedule is still a matter of active research. Common choices are a linear or a simulated annealing schedule, as well as Polyak step sizes, which are exact for linear models.

2.4.5. Policy as conditional action

The optimization results in a cost-optimal joint state-action posterior distribution trajectory. Conditioning the joint distribution of every time step on the state results in a conditional action trajectory, which is a time-varying linear Gaussian controller (TVLGC), similarly to the optimal linear quadratic Gaussian controller.

3. Gaussian process model

The Gaussian process (GP) model used in our experiments to learn a dynamics model and a policy is coded from scratch based on the SNGP (see Section 2.3) but contains some important modifications. Firstly, we are doing multiple regression without learning separate models for each output dimension, which entails extending Bayesian linear regression to the multivariate setting. Secondly, we use a variational posterior instead of Laplace’s approximation to model the GP posterior, and we have different losses to optimize the posterior and train the model for deterministic and stochastic training data.

3.1. Bayesian multivariate linear regression

Let a multivariate linear regression model [31, 32] be defined by the mapping from a length (d_x) input vector \mathbf{x} to a length (d_y) output vector \mathbf{y} with a $(d_x \times d_y)$ weight matrix \mathbf{W} and zero-mean i.i.d. multivariate normal distributed length (d_y) error vectors \mathbf{e}

$$\mathbf{y}^\top = \mathbf{x}^\top \mathbf{W} + \mathbf{e}^\top \quad (3.1)$$

$$\mathbf{e} \sim \mathcal{N}(\mathbf{0}, \Sigma_\epsilon) . \quad (3.2)$$

The **likelihood** of a set of input-output pairs is normally distributed

$$\mathbf{y}^\top \sim p(\mathbf{y}^\top | \mathbf{x}^\top, \mathbf{W}, \Sigma_\epsilon) = \mathcal{N}(\mathbf{y}^\top | \mathbf{x}^\top \mathbf{W}, \Sigma_\epsilon) \quad (3.3)$$

and on a set of i.i.d. data points $\mathcal{D} = \{(\mathbf{x}_0, \mathbf{y}_0), \dots, (\mathbf{x}_n, \mathbf{y}_n)\}$ we can compute the **joint likelihood**

$$p(\mathbf{Y} | \mathbf{X}, \mathbf{W}, \Sigma_\epsilon) = \prod_i^n p(\mathbf{y}_i^\top | \mathbf{x}_i^\top, \mathbf{W}, \Sigma_\epsilon) = \mathcal{M}\mathcal{N}(\mathbf{Y} | \mathbf{X}\mathbf{W}, \mathbf{I}_n, \Sigma_\epsilon) \quad (3.4)$$

where we used the $(n \times d_x)$ matrix \mathbf{X} of vertically stacked inputs, the $(n \times d_y)$ matrix \mathbf{Y} of vertically stacked outputs, and the definition of the matrix normal distribution (see Appendix A).

The **conjugate weight prior**

$$\mathbf{W}_0 \sim p(\mathbf{W}) = \mathcal{MN}(\mathbf{W} | \mathbf{M}_0, \Sigma_{0,in}, \Sigma_{0,out}) \quad (3.5)$$

is a matrix normal distribution as well and has a $(d_x \times d_y)$ mean matrix \mathbf{M}_0 , a positive definite $(d_x \times d_x)$ input covariance $\Sigma_{0,in}$ and a positive definite $(d_y \times d_y)$ output covariance $\Sigma_{0,out}$. To keep the prior non-informative with minimal regularization we choose constant

$$\mathbf{M}_0 = \mathbf{0} \quad (3.6)$$

$$\Sigma_{0,in} = \sigma_{0,in}^2 \cdot \mathbf{I}_{d_x} = (1 + 1 \times 10^{-3}) \cdot \mathbf{I}_{d_x} \quad (3.7)$$

and learnable

$$\Sigma_{0,out} = \Sigma_{q,out} = \Sigma_{\epsilon,out} \stackrel{init}{=} 1 \times 10^{-2} \cdot \mathbf{I}_{d_y} \quad (3.8)$$

where the output covariance $\Sigma_{0,out}$ is diagonal and parametrized in square root form to maintain non-negativity, and is shared¹ with the variational posterior and the likelihood (error).

3.2. Variational posterior

The **variational weight posterior**

$$\mathbf{W}_q \sim q(\mathbf{W}) = \mathcal{MN}(\mathbf{W} | \mathbf{M}_q, \Sigma_{q,in}, \Sigma_{q,out}) \quad (3.9)$$

approximates the exact (analytical) posterior $p(\mathbf{W} | \mathcal{D})$. Its $(d_x \times d_y)$ mean matrix \mathbf{M}_q , positive definite $(d_x \times d_x)$ input covariance $\Sigma_{q,in}$, and positive definite $(d_y \times d_y)$ output covariance $\Sigma_{q,out}$ are learned from these initial values

$$\mathbf{M}_q \stackrel{init}{=} \mathbf{0} \quad (3.10)$$

$$\Sigma_{q,in} \stackrel{init}{=} \mathbf{I}_{d_x} \quad (3.11)$$

$$\Sigma_{q,out} = \Sigma_{\epsilon,out} = \Sigma_{0,out} \stackrel{init}{=} 1 \times 10^{-2} \cdot \mathbf{I}_{d_y} \quad (3.12)$$

¹The reason for sharing the output covariance is explained in Section 3.3

where the output covariance $\Sigma_{q,out}$ is diagonal and parametrized in square root form to maintain non-negativity, and is shared² with the prior and the likelihood (error). To maintain positive definiteness the covariances are parametrized in Cholesky factorized form with a softplus constraint on the diagonal.

Finding the optimal variational posterior is equivalent to finding the information projection (I-projection) [12] of the true posterior onto the set of parametrized variational posteriors. Since in our case both are Gaussians, the fit will be perfect.

3.3. Posterior predictive and model evidence

The **posterior predictive** results from the model likelihood given the variational posterior on the m new data points $\mathbf{Y}^*, \mathbf{X}^*$

$$p(\mathbf{Y}^*|\mathbf{X}^*) = \int p(\mathbf{Y}^*|\mathbf{X}^*, \mathbf{W})q(\mathbf{W})d\mathbf{W} \quad (3.13)$$

$$= \int \mathcal{MN}(\mathbf{Y}^*|\mathbf{X}^*\mathbf{W}, \mathbf{I}_n, \Sigma_\epsilon)\mathcal{MN}(\mathbf{W}|\mathbf{M}_q, \Sigma_{q,in}, \Sigma_{q,out})d\mathbf{W} \quad (3.14)$$

$$\propto \exp \left\{ -\frac{1}{2} \text{vec}(\mathbf{Y}^* - \underbrace{\mathbf{X}^*\mathbf{M}_q}_{\mathbf{M}_{y^*}})^\top \right. \quad (3.15)$$

$$\cdot \underbrace{\left[(\Sigma_\epsilon \otimes \mathbf{I}_m) + (\Sigma_{q,out} \otimes \mathbf{X}^*\Sigma_{q,in}\mathbf{X}^{*\top}) \right]}_{\Sigma_{y^*,\text{vec}}^{-1}} \quad (3.16)$$

$$\cdot \text{vec}(\mathbf{Y}^* - \underbrace{\mathbf{X}^*\mathbf{M}_q}_{\mathbf{M}_{y^*}}) \} \quad (3.17)$$

The resulting vectorized covariance $\Sigma_{y^*,\text{vec}}^{-1}$ is however irreducible for arbitrary covariances Σ_ϵ , $\Sigma_{q,in}$ and $\Sigma_{q,out}$. In order to be able to parametrize the posterior in terms of the input and output covariances we have to constrain either $\Sigma_{q,out} = \Sigma_\epsilon$ or $\Sigma_{q,in} = \mathbf{I}_m$. We choose to set $\Sigma_{q,out} = \Sigma_\epsilon$ and let $\Sigma_{q,in}$ be free, which results in the posterior predictive

$$p(\mathbf{Y}^*|\mathbf{X}^*, \mathcal{D}) = \mathcal{MN}(\mathbf{Y}^*|\mathbf{M}_y, \Sigma_\epsilon, \Sigma_y) \quad (3.18)$$

$$= \mathcal{MN}(\mathbf{Y}^*|\mathbf{X}^*\mathbf{M}_q, \Sigma_\epsilon, \mathbf{I}_m + \mathbf{X}^*\Sigma_{q,in}\mathbf{X}^{*\top}) . \quad (3.19)$$

²The reason for sharing the output covariance is explained in Section 3.3

The **model evidence** (also called marginal likelihood) is the posterior predictive evaluated on the entire (training) dataset, i.e. the same equations but without the asterisks.

3.4. Evidence lower bound loss: fitting to deterministic targets

To optimize our model given a set of deterministic data points, we would like to maximize the model evidence on the entire data set, preferably in closed form. Using a variational posterior, the model evidence can be conveniently separated into the evidence lower bound (ELBO) and backward Kullback-Leibler (KL) divergence.

$$\log p(\mathcal{D}) = \mathcal{L}_{ELBO}(\mathcal{D}) + D_{KL}[q(\mathbf{W})||p(\mathbf{W}|\mathcal{D})] \quad (3.20)$$

showcasing the relation between the optimization of the feature neural network parameters and the variational posterior parameters. The former increases the model evidence and thus the ELBO and the posterior backward KL divergence. The latter does not change the evidence, but by lowering the posterior backward KL divergence, increases the ELBO, thus moving it closer to the evidence. In our experiments we have found the variational posterior optimization (I-Projection) to converge so much faster than the feature network optimization that we have decided to maximize the ELBO w.r.t. both sets of parameters simultaneously on every minibatch.

Finding the true posterior and the model evidence is computationally expensive, but we can easily compute the prior, likelihood and variational posterior. Therefore, we choose to maximize the following form of the ELBO as our (negative) loss function

$$\mathcal{L}_{ELBO}(\mathbf{X}, \mathbf{Y}) = \mathbb{E}_{\mathbf{W} \sim q(\mathbf{W})} [\log p(\mathbf{Y}|\mathbf{X}, \mathbf{W})] - D_{KL}[q(\mathbf{W})||p(\mathbf{W})] \quad (3.21)$$

$$\ell_{ossELBO} := -\mathcal{L}_{ELBO}(\mathbf{X}, \mathbf{Y}) \quad (3.22)$$

which we can compute in closed form (see Appendix B and [33]). The loss can also be computed in batches, since the log likelihood of the entire dataset is the sum of the mini-batch log likelihoods.

3.5. Moment-matching loss: fitting to target distributions

To learn from stochastic targets we can do moment-matching of the stochastic model predictions and stochastic targets. This is called the moment projection (M-projection) [12]

and means minimizing the (forward) KL divergence of the target distributions from the posterior predictive distributions of the model on the training data. For multivariate Gaussian targets (and Gaussian model predictions) only the first two moments need to be matched and the forward KL divergence loss can be computed in closed form

$$\ell_{oss_{mm}} := D_{\text{KL}}[\mathcal{N}(\mu_{\mathbf{Y}^*}, \Sigma_{\mathbf{Y}^*}) || p_{\text{model}}(\mathbf{Y}^* | \mathbf{X}^*, \mathcal{D})] \quad (3.23)$$

which can also be computed in batches. However, the M-Projection notoriously overestimates the support of the observations, which means that the model learned from non-Gaussian observations (e.g. multi-modal) might be a bad approximation of the true data distribution. How bad this approximation is in practice depends on the context and use of the model.

4. Experiment: Gaussian process model de-risking

In this chapter we describe the separate building and training of the modified SNGP (from Section 3) as a policy and a dynamics model in a controlled context with arbitrary amounts of data to verify that it is expressive enough to be used in model-based reinforcement learning (MBRL).

4.1. Setup

In this experiment we first describe the environment and modified SNGP implementation. After that, we acquire an expert policy, a SAC model trained on the environment. To train the dynamics model, we run the SAC policy to collect environment transitions and train the SNGP dynamics model to minimize the ELBO loss on the collected transitions. Finally, we use DAgger to imitate the SAC policy with the modified SNGP policy using the ELBO loss to fit to the collected data.

4.1.1. Environment: Furuta pendulum swing-up with custom cost

The task to be solved is encoded in a python gym environment¹, which simulates the Furuta pendulum used by Quanser Inc. in their Cube robot. The reward function is modified for training purposes (see Appendix C.1). This robot has two rotary joints. The first joint is actuated and allows rotation of a shaft about the vertical axis through the center of the (fixed) base of the robot. The shaft is L-shaped with two parts at a right

¹code at <https://git.ias.informatik.tu-darmstadt.de/quanser/clients.git> with parameters `id=qube-500-v0`, `gamma=0.99`, `horizon=3000`

angle to each other, such that the first part stays in the rotation axis of the first joint and the other moves in the horizontal plane above the robot base. The second joint sits at the end of the second part of the shaft and allows rotation of the end-effector about the axis through the second part of the shaft. In the Furuta pendulum configuration, the end-effector is a shaft at right angle to the second joint axis.

4.1.2. Setup modified SNGP model

The modified SNGP model is specified as a linear Bayesian model with a custom feature function, where the feature network architecture and the number of RFFs are configurable from the main script or the command line. The model is set up to transparently apply static whitening of inputs and outputs, which has to be initialized on a representative (or the full) data set beforehand. For the spectral normalization we used the pytorch spectral-norm module, which performs a power iteration step per call. This means, every call returns a slightly different prediction, the sequence of which does converge. To stop the power iteration and get deterministic predictions for evaluation, the module has to be set to eval mode. The residual network is coded from scratch and skips activations, but not the linear layers.

4.1.3. SAC agent training

For the expert SAC agent we used the implementation of SAC in mushroom-rl². The model contains an actor network and two critic networks, which are both two-layer dense neural networks with 64 hidden nodes and are both trained with a learning rate of $lr = 3 \times 10^{-4}$. We trained the model for 240 epochs of 2000 environment steps each, where both the actor and the critics are fit after each step on one random batch from the replay buffer. The maximal size of the replay buffer is 500 000 steps and it is pre-filled before training with 3000 steps. The soft update coefficient is set to $\tau = 0.005$ and the learning rate for the entropy coefficient is set to $lr_{\alpha} = 5 \times 10^{-5}$.

4.1.4. SNGP dynamics model training: from expert policy rollouts

The dynamics model learns the unknown state transition dynamics function, which maps from a state and an action to the next state. We encode two sensible assumptions in the

²code at <https://github.com/MushroomRL/mushroom-rl.git>

model. The first assumption is that physical systems with masses can not instantaneously change their state, which means the dynamics model can learn a mapping from states and actions to state *deltas*, and the next state is the sum of the current state and the predicted delta. This makes the mapping essentially a residual model, which makes rollout trajectories less jerky and the training more stable. The second assumption is that the degree or radiant scale is does not represent physical rotary joint positions naturally, which instead are periodic, continuous, and move in a circle around the joint axis. Mapping all the angles to a two-dimensional sin-cos-space $\alpha \mapsto [\sin(\alpha), \cos(\alpha)]$ encodes these assumptions elegantly and provides a bijective mapping to the physical joint positions. The joint velocities are not mapped to the sin-cos space, since these assumptions do not match them. The delta predictions are also not mapped to the sin-cos space, because small angle deltas correctly map to small physical joint position deltas.

The modified SNGP dynamics model is trained for 500 epochs on 100 roll-outs, where each step is with a 50% probability either a *stochastic* prediction by the trained SAC agent or white Gaussian noise. The differences between consecutive states are used to compute the delta-state targets. The feature network of the SNGP dynamics is a two-layer network with 128 hidden nodes and 256 RFFs. The ELBO loss with a learning rate of $lr = 1 \times 10^{-3}$ is used for training.

4.1.5. SNGP policy training: behaviour cloning from SAC with DAgger

The policy learns a mapping from states to actions and is trained to be able to robustly imitate the behaviour of the expert SAC agent. Just like the dynamics model in Section 4.1.4, the policy angular (not velocity) inputs are mapped to the sin-cos space to encode sensible assumptions and provide a bijective mapping to physical joint positions. Since directly fitting policy predictions does not result in robust controllers, the training procedure uses DAgger (see Section 2.2) with a training buffer (replay buffer) with a maximum size of 10 000 steps. The modified SNGP policy is trained for 10 000 epochs on this replay buffer. The buffer is initially filled with 10 roll-outs of *deterministic* predictions by the trained SAC agent, and once every 10 epochs the SNGP policy is rolled out in the environment and the *deterministic* SAC predictions for the observed states are aggregated to the buffer. The feature network of the SNGP policy is a two-layer network with 128 hidden nodes and 256 RFFs. The ELBO loss with a learning rate of $lr = 1 \times 10^{-4}$ is used for training.

4.1.6. Closed-loop evaluations

To validate the learned policy and dynamics models and compare them against the SAC and true environment baselines, we have set up a separate script, which can load stored models given its directory path. A policy type (SNGP or SAC) and a dynamics model type (SNGP or gym environment) have to be selected, are used to run and plot 10 roll-outs. We use this script to test the performance of the SNGP policy against the true environment, the SAC policy against the SNGP dynamics and the SNGP policy and SNGP dynamics jointly.

4.2. Evaluation

4.2.1. SNGP dynamics model training: from expert policy rollouts

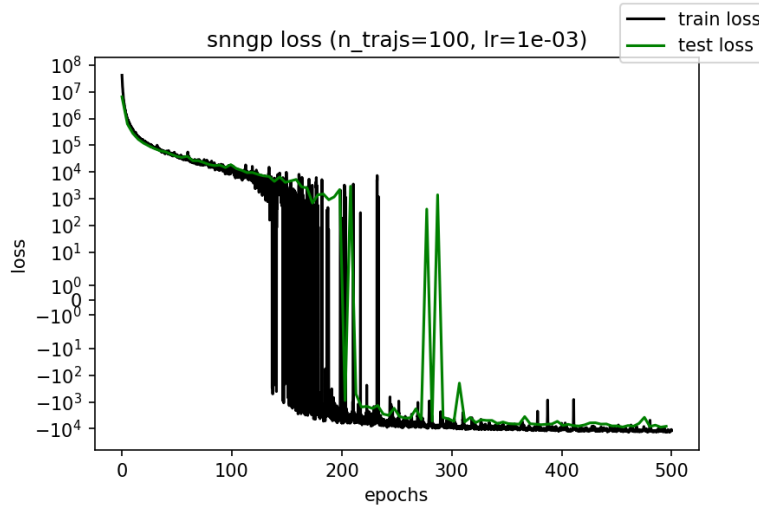


Figure 4.1.: ELBO loss of training dynamics model. To properly visualize the log probabilities moving in a very large range, the y-axis has a symmetric logarithm (symlog) scale, which maps values like this:

$$x \mapsto \begin{cases} -\log(-x) & \text{if } x < -1, \\ x & \text{if } -1 < x < 1, \\ \log(x) & \text{if } x > 1. \end{cases}$$

The SNGP dynamics seems to learn quite well with the ELBO loss on the test data set going down and staying low in Figure 4.1. It seems that most of the loss is minimized after 300 epochs.

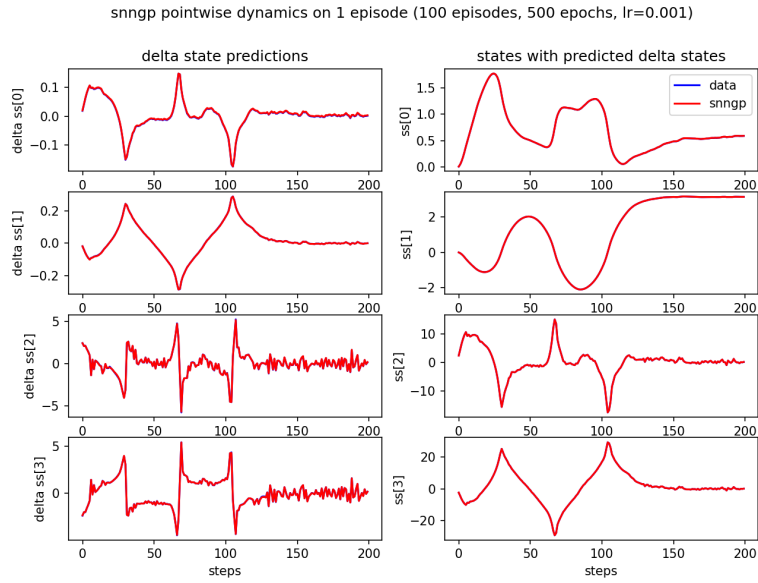


Figure 4.2.: Pointwise dynamics prediction fit: the left side shows the predicted deltas, the right side integrates the deltas starting from the initial state

The point-wise fit on the training data in Figure 4.2 set is visually excellent, with the red SNGP predictions completely covering the blue training data line, which matches the very low loss seen in the loss plot.

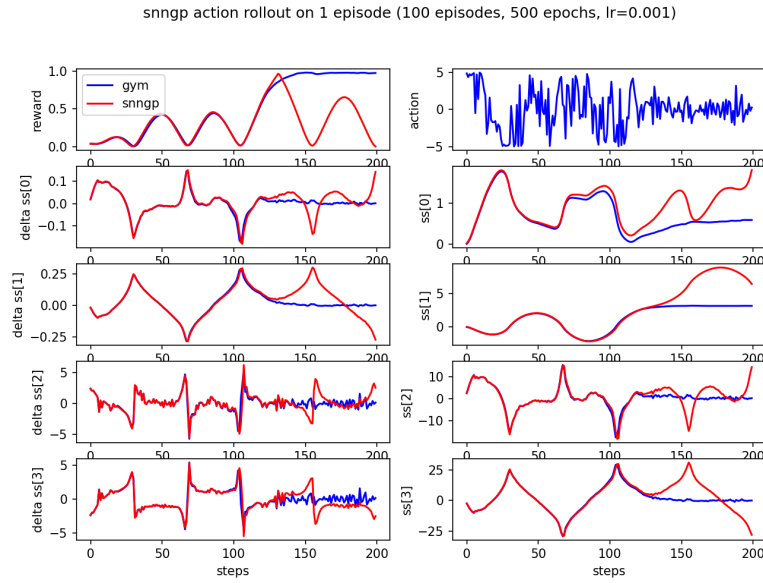


Figure 4.3.: Dynamics roll-out using expert actions replay

The roll-outs of an expert (SAC) action trajectory in Figure 4.3 is less perfect and errors seem to accumulate until the trajectory finally diverges. However, the trajectories are still very similar until more than half-way through the episode, which seems sufficiently good, given that the policy is static and a real policy would probably steer the system back to an optimal trajectory.



4.2.2. SNGP policy: behaviour cloning from SAC (Dagger)

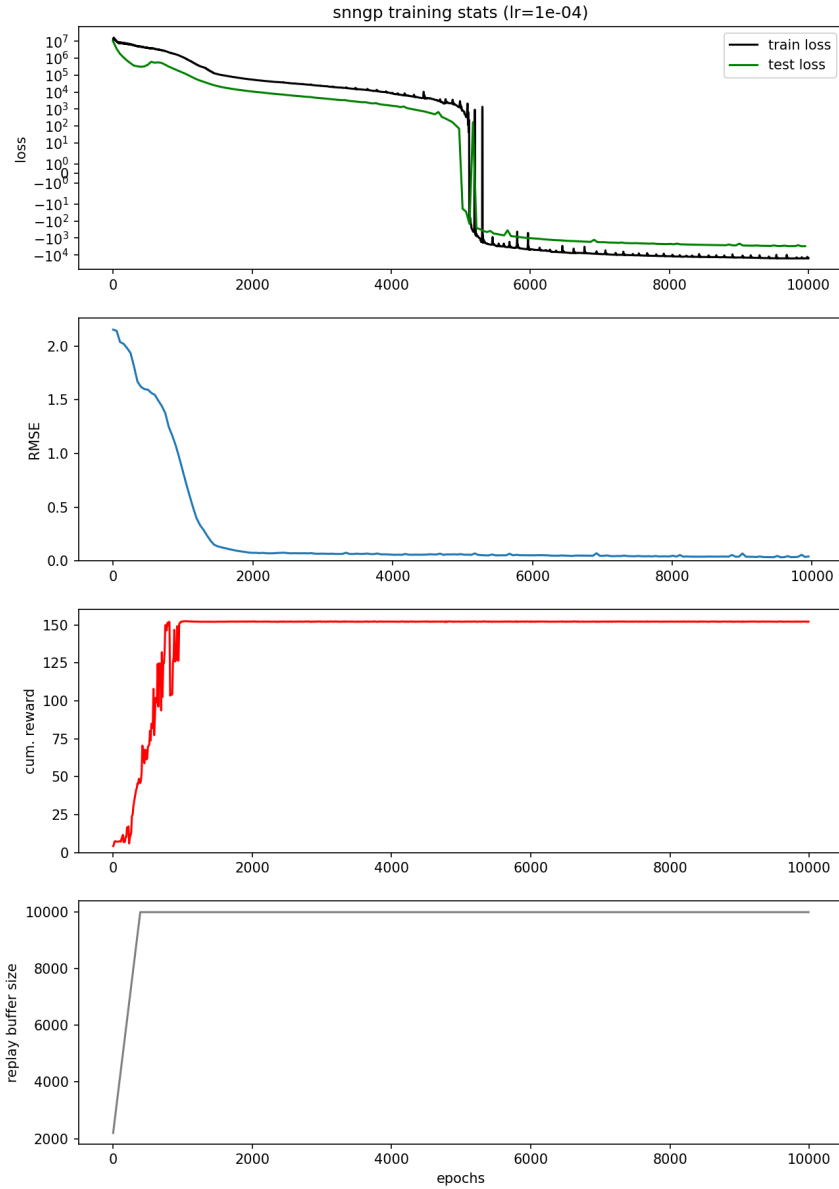


Figure 4.4.: Training statistics for SNGP policy imitating SAC agent (loss plot uses a symlog scaled y-axis like in Figure 4.1)

In Figure 4.4 we can see that the SNGP policy seems to converge to a good policy with low loss and root mean squared error (RMSE) and converged cumulative reward. However, in contrast to the dynamics model training, we had to use many more epochs and a much higher epoch number to get a good model. We assume this has to do with DAgger continuously overwriting data in the size-limited buffer, thus requiring slower learning and more epochs to stay stable throughout the training.

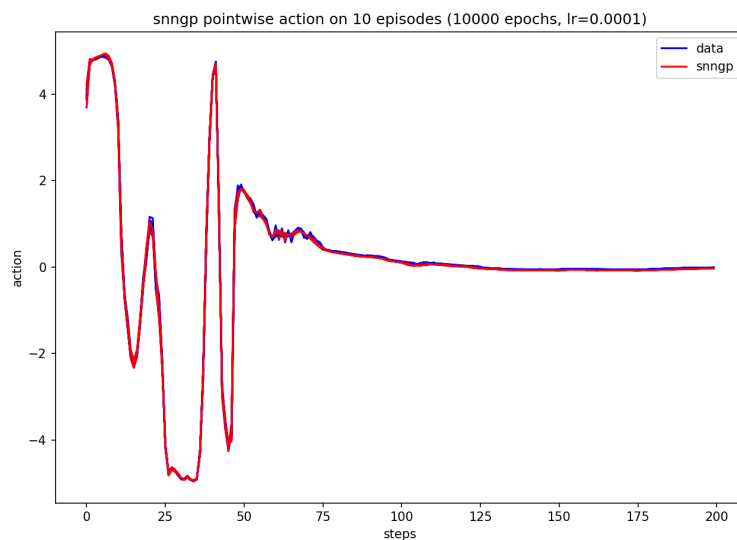


Figure 4.5.: Pointwise prediction fit of SNGP policy and SAC roll-out data

The point-wise fit on the training data in Figure 4.5 seems sufficiently good, though not perfect, since the blue line of the training labels is often not exactly covered by the red SNGP predictions.

4.2.3. Closed-loop evaluations

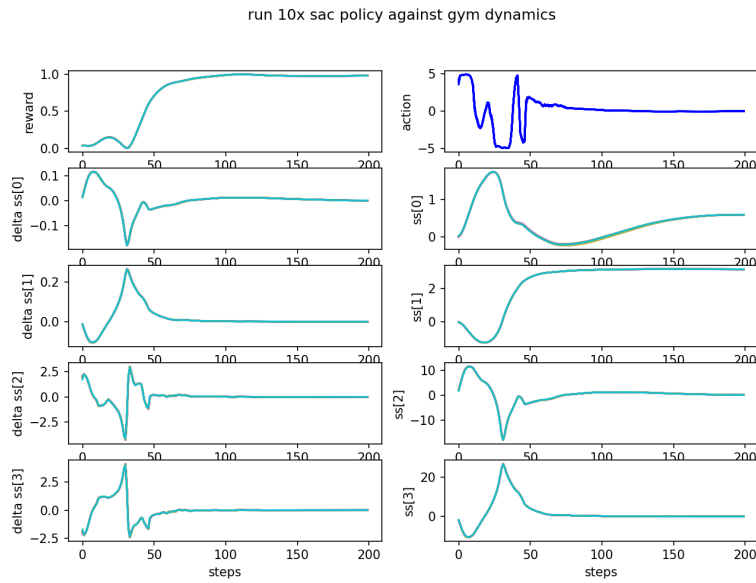


Figure 4.6.: 10 roll-outs of the SAC policy and the true (gym) dynamics

In Figure 4.6 we can see that the SAC policy manages to solve the environment consistently over 10 different environment seeds. This serves as a baseline to the SNGP policy and dynamics models.

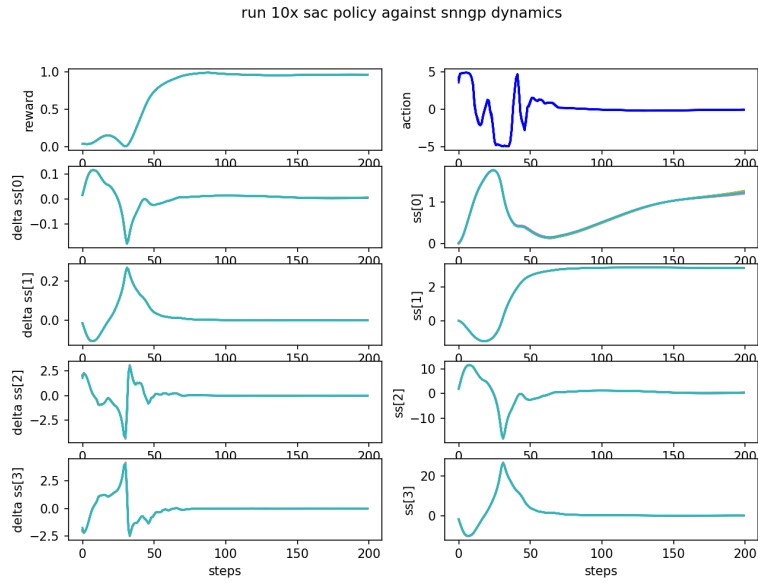


Figure 4.7.: 10 roll-outs of the SAC policy and the trained SNGP dynamics

In Figure 4.7 the SNGP dynamics seems to capture the true dynamics around the SAC optimal trajectories so well, that SAC manages to stabilize the system in a trajectory and final state, which are very like the baseline of SAC with the true dynamics.

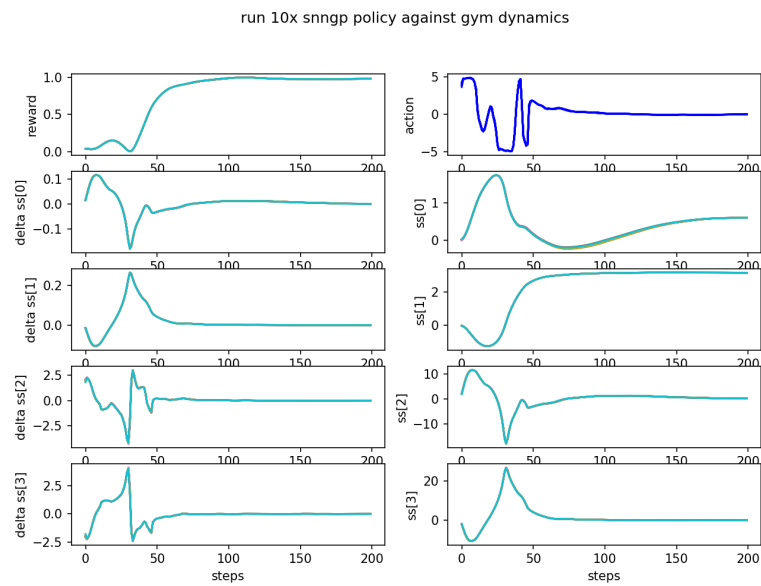


Figure 4.8.: 10 roll-outs of the trained SNGP policy and the true (gym) dynamics

In Figure 4.8 the SNGP policy seems to fare very well against the true dynamics as well. The SNGP policy even seems to produce smoother action trajectories (see action, delta ss[2], and delta ss[3] subplots).

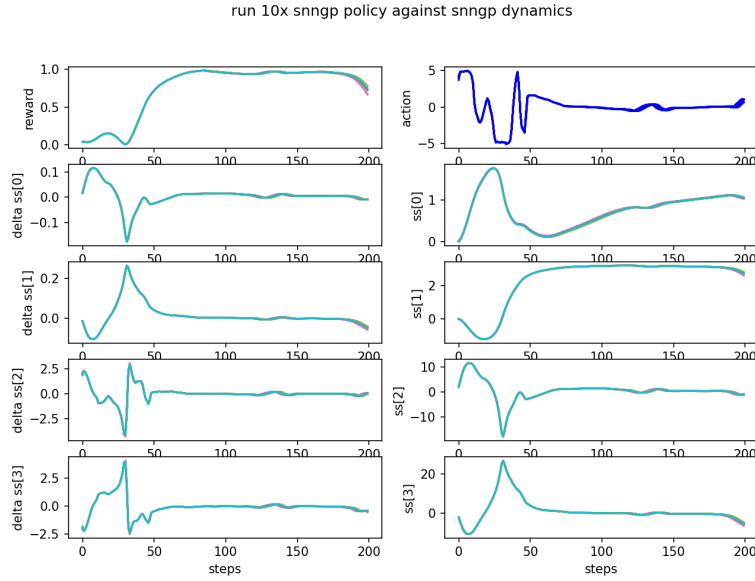


Figure 4.9.: 10 roll-outs of the trained SNGP policy and the trained SNGP dynamics

Finally we can see in Figure 4.9 that the SNGP policy and the SNGP dynamics seem to work together reasonably. The only noticeable difference from the optimal SAC trajectory is the bumpier stabilization between time steps 100 and 150 and the final part of the roll-out, where the trajectories of different seeds diverge slightly.

4.3. Summary

From these experiments we see that the modified SNGP is expressive enough to learn the true environment dynamics or imitate an expert policy. Training an SNGP dynamics model with the ELBO loss on expert policy roll-outs yields such an accurate representation of the true dynamics, that the closed-loop roll-outs match the nominal trajectory perfectly and the open-loop roll-outs are almost perfect for over 100 time steps. The SNGP policy trained using DAGger and the ELBO loss manages to imitate a SAC expert policy perfectly and even produces smoother action trajectories than SAC.

5. Experiment: input inference for control in the model-based reinforcement learning loop

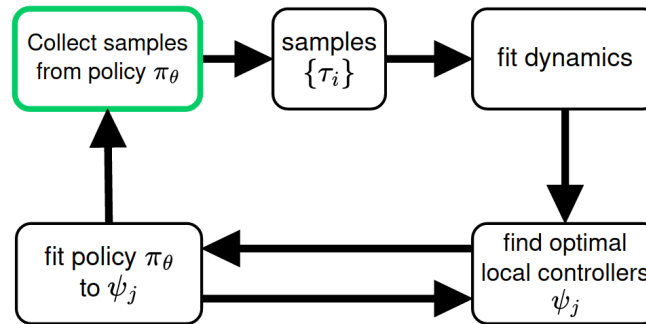


Figure 5.1.: Model-based reinforcement learning loop with global dynamics model, global policy and optimization to get local optimal controllers

In Chapter 4 we have confirmed the expressiveness of the modified SNGP model for our purposes. Now, we connect them with the I2C optimization and create the model-based RL (MBRL) loop seen in Figure 5.1. The loop starts off with an untrained exploration policy to collect samples. The collected transitions are stored in a buffer and are used to train the dynamics model. Using this dynamics model, I2C is tasked with finding cost-optimal state-action trajectories, which can be encoded into time-varying linear Gaussian controllers (TVLGC). Finally, the policy is trained to imitate the locally optimal state-action trajectories or the equivalent controllers. Rolling out the updated policy in the real system to collect new samples closes the loop. The backwards arrow from the policy to the I2C optimization indicates that the currently learned policy can be used to warm-start the optimization with the latest policy instead of starting from random actions. Note that

the only true environment interactions are from the rollouts used to train the dynamics model, as indicated by the green outline in Figure 5.1. The optimization of I2C and the policy are only based on the learned dynamics model entirely in simulation.

5.1. Setup

In this experiment we will first define the environment and describe the models, approximate inference, and I2C optimization, then set up experiments to cover the individual steps of dynamics training, I2C optimization, and policy training. We are not evaluating the full closed loop yet because there are still issues with exploration to learn a representative dynamics model and the I2C optimization when using said dynamics model, which lead to a cascading divergence of the close loop optimization.

5.1.1. Environment: simple pendulum swing-up

Instead of the Furuta pendulum used in Chapter 4, here we learn a simpler, single-joint pendulum swing-up task. The pendulum has one under-actuated joint and a point mass at the end of the mass-less pendulum rod. The pendulum dynamics are integrated with the forward Euler method, which contain damping, which is proportional to the angular velocity, and the control signal, which acts on the angular acceleration. The cost contains the squared angular velocity, the squared control signal and the squared distance of one minus the cosine of the joint angle, which encodes the task of swinging up from the bottom to the top and stabilizing the pendulum.

5.1.2. Setup models and approximate inference

For the MBRL experiments, all models are specified using a modular model wrapper which facilitates experiments and ablation studies. In order to be wrapped, models can be the modified SNGP, a simple neural network or even a constant cost function, as long as they are callable functions. The model wrapper encapsulates the callable model and an approximate inference method to provide a unified interface for stochastic and deterministic predictions as well as propagating input distributions through the model using approximate inference to get output distributions. Furthermore, the model wrapper allows transparent use of torch evaluation or training mode as well as moving the model

between torch devices. Model wrappers are used to encapsulate the dynamics and policy models, but also the cost function and environment. Additionally, the model decorator can be used to easily create and apply input and output transformations to model wrappers, making the model specification in the main script more versatile and verbose. Predefined wrappers include input transformers (clamping or angle to sin-cos mapping), output transformers (predicting delta targets or adding dithering) or transformers for both input and output (moving the inputs to the cpu, calling the model, and moving the outputs back to the torch device).

The type of model to use for the dynamics model and the policy, including the feature network architecture and the number of RFFs for GPs, are individually configurable from the main script or the command line.

Every callable model is assigned an appropriate approximate inference method to propagate Gaussian inputs through the model. The dynamics and policy callables use spherical Gaussian quadrature ($\alpha = 1, \beta = 0, \kappa = 0$), which also incorporate the predicted uncertainty of stochastic models. The approximate inference for the cost callable uses importance sampling, to move the state-action distribution towards lower cost by using the cost likelihood from Section 2.4.1. Since we have found the cost approximation to have a greater impact on the optimization, we are using the more precise Gauss-hermite quadrature to get a more accurate local cost approximation.

5.1.3. Setup I2C optimization

Our implementation of I2C iterates forwards and backwards over sequences of consecutive state and state-action distributions, which matches the filtering and smoothing perspective [25] and not the equivalent forward and backward messages framework often used for probabilistic Gaussian models or hidden Markov models [30].

The initial policy can either be Gaussian noise or a prior policy to warm-start the optimization. From the second iteration on, the policy is the optimal TVLGC of the previous iteration. Every optimization iteration starts with a forward filtering pass, where the initial state distribution is rolled out through the policy, cost likelihood and dynamics model using approximate inference, leading to a joint state-action trajectory, which is slightly warped towards lower cost according to the temperature parameter. The smoothing pass then iterates backwards starting from the final state of the filtering pass, and adjusts the trajectory from the filtering pass. After every forward and backward pass, the TVLGC

gains are computed from the joint posterior trajectory, and the temperature is updated for the next iteration.

5.1.3.1. Open-loop and closed-loop controller

The TVLGC introduced in Section 2.4.5 is a closed loop controller

$$a \sim \pi_t(s) = \mathcal{N}(k + Ks, \Sigma_{aa_t}) \quad (5.1)$$

but in the implementation the controller can also be switched to open loop mode which internally uses a zero feedback gain $K = 0$. This is important for optimization, because excessive exploitation of the feedback gain notoriously gives rise to numerical instabilities, making it hard to optimize closed loop controllers without regularization. Both the open loop and closed loop controllers are learned from a Gaussian time-varying joint posterior distribution:

$$p(s, a; t) = \mathcal{N} \left(\begin{pmatrix} \mu_{s_t} \\ \mu_{a_t} \end{pmatrix}, \begin{pmatrix} \Sigma_{ss_t} & \Sigma_{sa_t} \\ \Sigma_{as_t} & \Sigma_{aa_t} \end{pmatrix} \right) \quad (5.2)$$

The closed loop controller is essentially the conditional distribution of the posterior action, as a function of the state

$$p(a|s; t) = \mathcal{N}(\mu_{a_t} + \Sigma_{sa_t} \Sigma_{ss_t}^{-1}(s - \mu_{s_t}), \Sigma_{aa_t} - \Sigma_{as_t} \Sigma_{ss_t}^{-1} \Sigma_{sa_t}) \quad (5.3)$$

but the conditional variance is simplified to prevent numerical instabilities due to the involved covariance matrix inversion resulting in the following closed loop gains

$$K^{cl} = \Sigma_{sa_t} \Sigma_{ss_t}^{-1} \quad (5.4)$$

$$k^{cl} = \mu_{a_t} - K^{cl} \mu_{s_t} \quad (5.5)$$

For the open loop controller the feedback gain is zero, which means the open loop controller is the marginal action posterior, where the variance is now exact:

$$K^{ol} = \mathbf{0} \quad (5.6)$$

$$k^{ol} = \mu_{a_t} \quad (5.7)$$

For every joint time-varying posterior distribution that is supplied, both open- and closed-loop controllers are created and the appropriate controller can be chosen at inference time via a boolean parameter. Both the posteriors used to create the controllers and the states for which the controllers predict actions take an arbitrary number of batch dimensions and scale the controller gains and predictions accordingly.

5.1.4. SNGP dynamics model training: from policy rollouts with ELBO loss

Just like in Section 4.1.4 we learn the global unknown state transition dynamics function, and just like before we use two sensible assumptions: the former is that physical systems with masses can not instantaneously change their state, thus we learn a mapping from states and actions to state *deltas* in the model; the latter is that rotatory joint positions (but not velocities) are better represented by mapping the angles to a two-dimensional orthogonal space $\alpha \mapsto [\sin(\alpha), \cos(\alpha)]$. Both of these assumptions are implemented by decorators to the model wrapper.

For the dynamics model we run two different experiments, since it sits at the start of the MBRL loop: first, we train it on environment rollouts of an untrained exploratory policy, and then we train it on environment rollouts of a trained policy¹, since these different policies produce very different trajectories. To increase exploration we add dithering $\sigma = 5 \times 10^{-2}$ to both policies during the roll-out. Both dynamics models are trained on 50 roll-outs, with 12 additional roll-outs used for evaluation. The differences between consecutive states are used to compute the delta-state targets. The feature network of the SNGP dynamics is a five-layer network with 128 hidden nodes and 256 RFFs. The ELBO loss with a learning rate of $lr = 5 \times 10^{-4}$ is used for training. Instead of training for a fixed number of epochs, we are using a threshold, such that the training is halted when both the training and evaluation loss are below $\ell_{\text{th}} = 1 \times 10^{-3}$.

5.1.5. I2C: find locally optimal controllers from dynamics

To distill a good global policy we need to find multiple local optimal controllers, because each local controller is optimal only for a specific trajectory starting from a specific initial state. Therefore, we run the I2C optimization for several different initial states before every policy distillation. Unfortunately, I2C can not be directly parallelized, due to the chained recursions of the forward and backward passes of filtering and smoothing, where each computation depends on previous states. Hence we vectorize the computation of I2C with pytorch, such that the optimization can run on the CPU in parallel, starting from a vector of initial states, which results in a significant speed-up to almost constant run-time w.r.t. the number of initial states.

¹we use the policy trained with DAGger and the ELBO loss (description in Section 5.1.6.2, evaluation in Section 5.2.3.2)

To find cost-optimal action trajectories, we execute I2C starting from a set of 10 initial states randomly sampled from the environment². These initial states are assigned a low variance of $\sigma_0 = 1 \times 10^{-6}$, which tells the optimization to trust these initial states and be bold in the optimization. The step-size is computed using the Polyak temperature strategy. To make sure all the individual optimizations from all initial states have converged, we do not optimize for a fixed number of iterations. Instead, we use a convergence criterion, such that the optimization is halted when the cumulative cost of all optimizations stabilizes. Specifically, the change of the cost over the last 5 iterations has to be lower than the threshold³ $\Delta_{c_{th}} = 1$. We evaluate the effect of a learned SNGP dynamics model on the optimal trajectories by running the optimization first using the true dynamics, then using the SNGP dynamics model learned from roll-outs of an untrained SNGP policy, and finally using the SNGP dynamics model learned from roll-outs of a (Dagger) trained SNGP policy.

5.1.6. SNGP policy training: from optimal I2C trajectories

The goal for policy training is to distill the knowledge of several optimal I2C trajectories into one global policy. Due to the Markovian structure of most RL problems, policies are usually harder to train directly with supervised learning than dynamics models. Therefore, we have designed two different approaches for policy distillation: the first approach uses the moment matching loss to directly fit the optimal trajectories, the second uses DAgger and the ELBO loss to match the policy to an importance weighed mixture of the trajectories.

5.1.6.1. Moment matching loss

We could directly fit the optimal conditional action distributions found by I2C, but since every optimization yields only one action for each time-step, we decided to augment our dataset by adding the sigma points of the Gaussian action distributions to the dataset as well, effectively quintupling our samples. The feature network of the SNGP policy is a two-layer network with 128 hidden nodes and 256 RFFs. The moment matching loss with a learning rate of $lr = 1 \times 10^{-4}$ is used for training. Instead of training for a fixed

²The currently used environment allows direct access to the initial state distribution, but the system can be easily extended to sample from an empirical distribution of collected initial states, when the initial state distribution is unknown

³This value was found to be reasonable for the used environment given the typical cumulative costs of 150 to 800.

number of epochs, we are using a threshold, such that the training is halted when both the training and evaluation loss are below $\ell_{\text{th}} = 1 \times 10^{-3}$.

5.1.6.2. DAgger using importance weighed mixture controller and ELBO loss

Alternatively to an M-Projection we can also use DAgger to distill a policy from a set of locally optimal controllers. This should provide a higher robustness and allow the model to learn to recover better from a disturbance. To combine the information contained in the conditional action distributions

$$a_{t,\text{expert}}^i(s) \sim \mathcal{N}(\mu_{t,a|s}^i, \Sigma_{t,a|s}^i) = p_t^i(a|s) = \frac{p_t^i(a, s)}{p_t^i(s)} \quad (5.8)$$

$$p_t^i(s) = \int p_t^i(a, s) da \quad (5.9)$$

where $p_t^i(s, a)$ is the i -th joint state-action posterior distribution for the t -th timestep returned by I2C. We compute the weighed sum of their moments for each time step

$$\mu_{t,a|s} = \frac{\sum_i \mu_{t,a|s}^i p_t^i(s)}{\sum_i p_t^i(s)} \quad (5.10)$$

$$\Sigma_{t,a|s} = \frac{\sum_i \Sigma_{t,a|s}^i p_t^i(s)}{\sum_i p_t^i(s)} \quad (5.11)$$

$$a_{t,\text{expert}} \sim \mathcal{N}(\mu_{t,a|s}, \Sigma_{t,a|s}) \quad (5.12)$$

where each weight is the likelihood of seeing the current state s assuming that the states are distributed according to the controller state distribution $p_t^i(s)$. The weighing procedure is similar to a kernel density estimation [34, 35] with Gaussian (radial basis function) kernels. This results in a mixture policy which pools the local policies' information and is therefore locally good in the union of the sections of state-space where the single policies are good. However, this weighing does not qualitatively distinguish between the different local policies and might thus follow the trajectory of a worse local policy if it is closer to the current state than a better one.

The feature network of the SNGP policy is a two-layer network with 128 hidden nodes and 256 RFFs. To implement DAgger we collect and aggregate a new roll-out every 10 epochs where the weighed mixture policy is used as the expert policy. The collected aggregated data-set is fit using the ELBO loss with a learning rate of $lr = 1 \times 10^{-4}$. Instead of training for a fixed number of epochs we are using a threshold, such that the training is halted when both the training and evaluation loss are below $\ell_{\text{th}} = 1 \times 10^{-3}$.

5.2. Evaluation

5.2.1. SNGP dynamics model learning

5.2.1.1. From untrained SNGP policy roll-outs with dithering

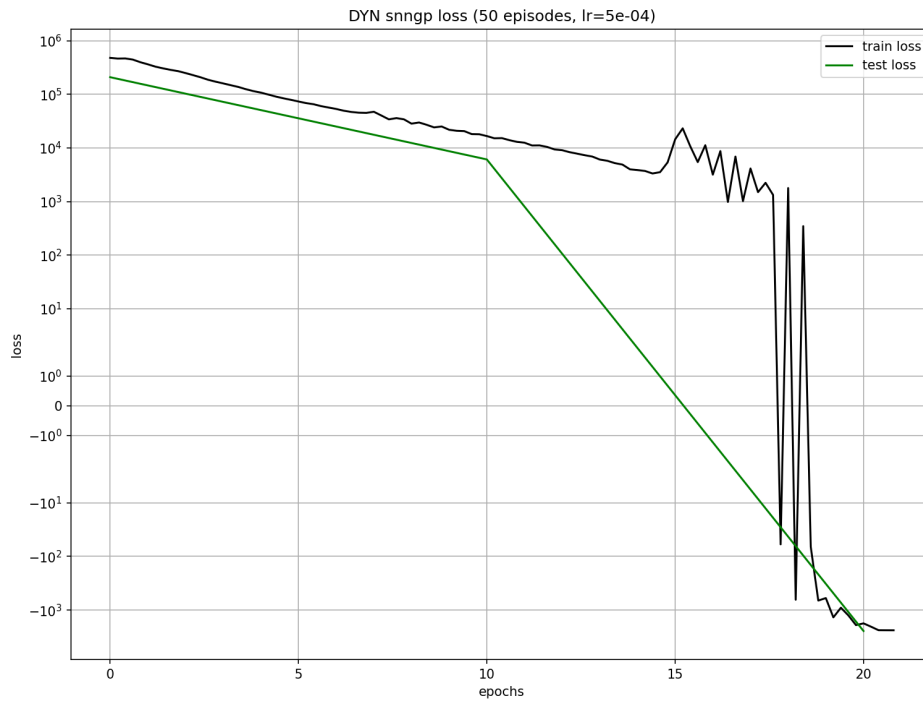


Figure 5.2.: ELBO loss on training and evaluation roll-outs. The training stopped after about 21 epochs because both the test and evaluation loss were below the threshold $\ell_{\text{th}} = 1 \times 10^{-3}$ (loss plot uses a symlog scaled y-axis explained in Figure 4.1).

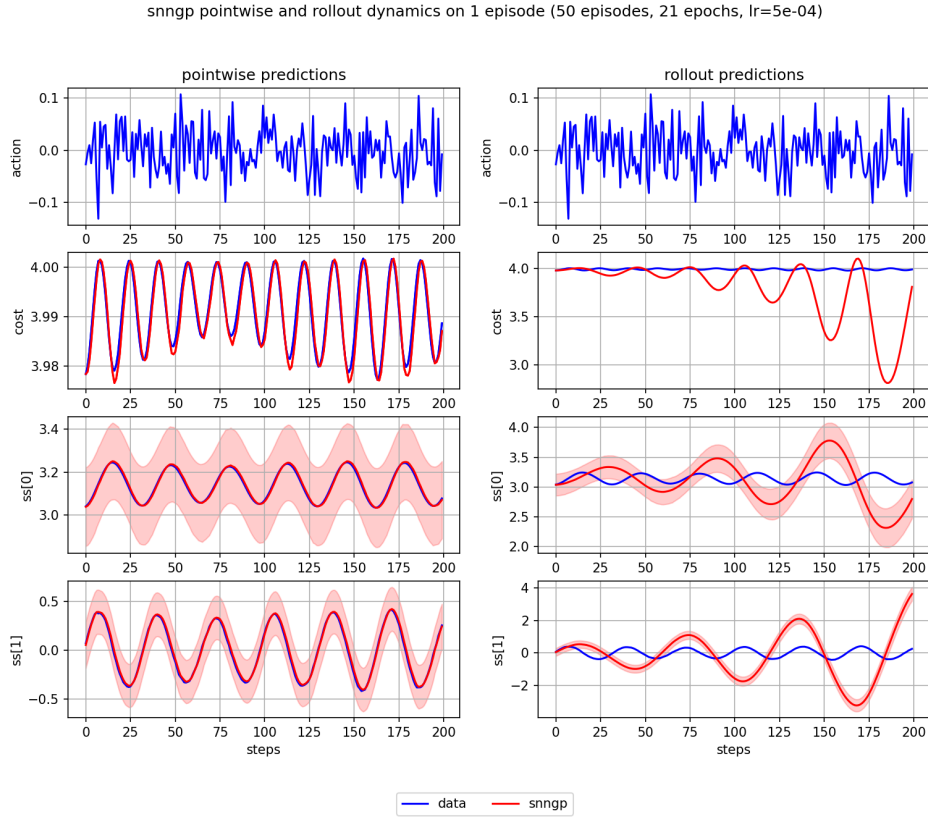


Figure 5.3.: Dynamics prediction fit on one evaluation episode: the left side shows the point-wise prediction fit, the right side shows a roll-out of the dynamics model from the starting state and the (fixed) action sequence from the point-wise data

In Figure 5.2 we can see the SNGP dynamics training and evaluation loss curves. The training of the model converges much faster than during the de-risking experiments in Chapter 4 even though the learning rate is lower. This might be because the training data used in Chapter 4 come from expert policy roll-outs which entails more exploration than a random noise policy, and the dynamics model therefore has to learn the dynamics of a larger area of the phase space. In Figure 5.3 we can see the point-wise and roll-out fit of the dynamics model. The point-wise fit is not perfect, but it seems reasonable, whereas the roll-out fit has the wrong frequency and seems to gain energy. We assume that this is

due to the high noise in the actions, which probably forces the model to learn an averaged mapping (since the model is not trained to over-fit) and the errors accumulate when the model is applied recursively.

5.2.1.2. From trained SNGP policy roll-outs with dithering

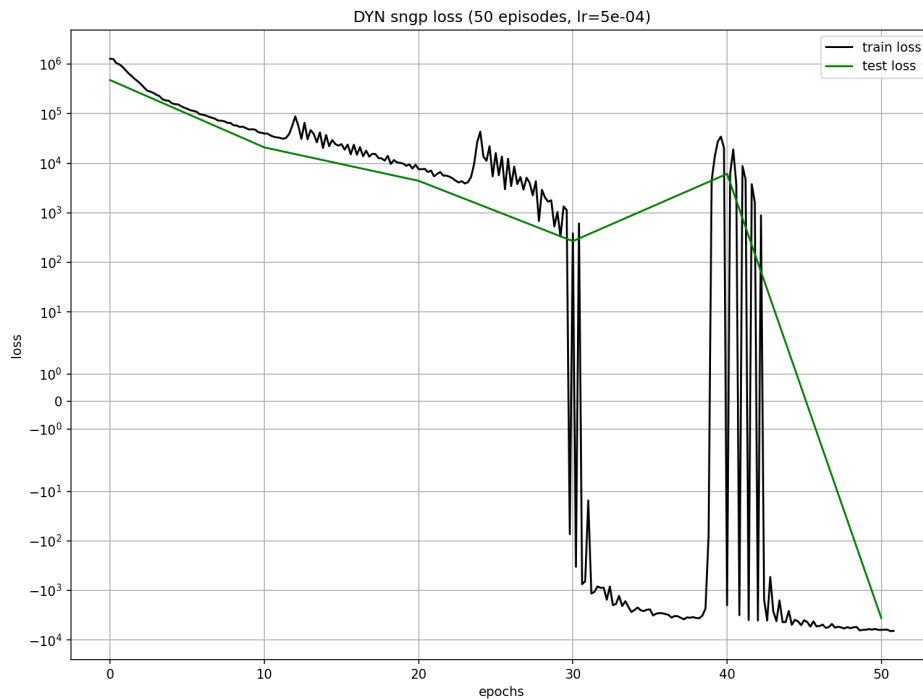


Figure 5.4.: ELBO loss on training and evaluation roll-outs. The training stopped after 51 epochs because both the test and evaluation loss were below the threshold $\ell_{\text{th}} = 1 \times 10^{-3}$ (loss plot uses a symlog scaled y-axis explained in Figure 4.1).

sngp pointwise and rollout dynamics on 1 episode (50 episodes, 51 epochs, lr=5e-04)

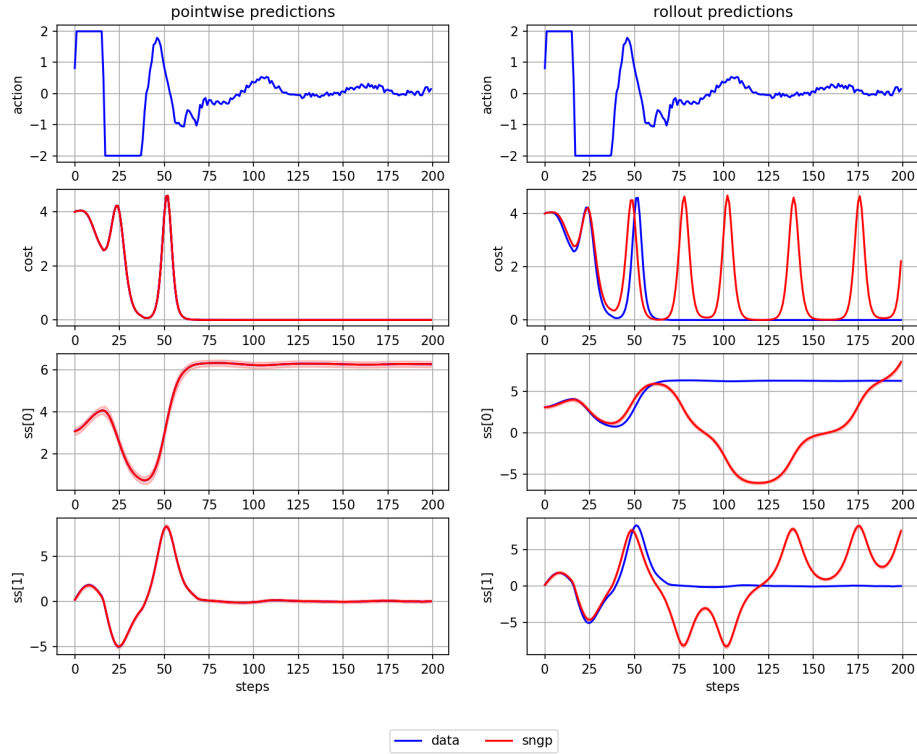


Figure 5.5.: Dynamics prediction fit on one evaluation episode: the left side shows the point-wise prediction fit, the right side shows a roll-out of the dynamics model from the starting state and the (fixed) action sequence from the point-wise data

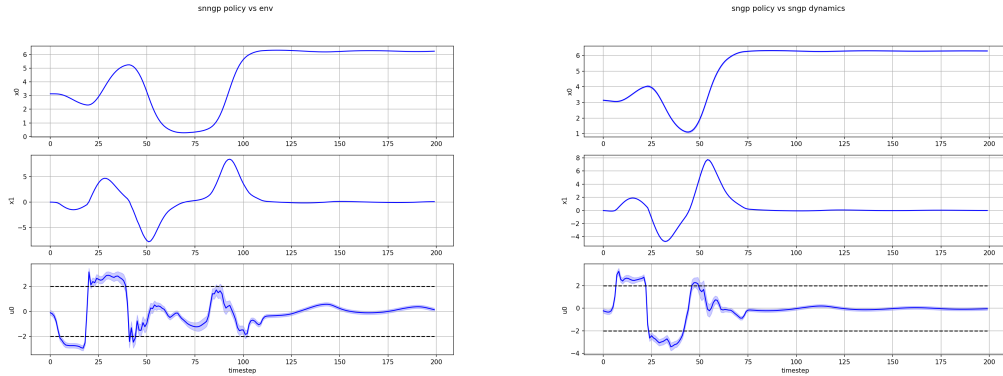


Figure 5.6.: Roll-out of the trained policy in the true environment (left) and the learned SNGP dynamics model (right) from the same initial state.

In Figure 5.4 we can see the SNGP dynamics training and evaluation loss curves. The training of the model still converges faster than during the de-risking experiments in Chapter 4, which we assume partly comes from the greater number of policy roll-outs used for this experiment and from using a deeper feature network, but slower than when training from untrained policy roll-outs (as in Section 5.2.1.1) because the roll-outs cover a wider area of the state-space. In Figure 5.5 we can see the point-wise and roll-out fit of the dynamics model. The point-wise fit looks perfect, whereas the roll-out is good for the first 60 time steps and then diverges rapidly. However, the divergence happens at an unstable equilibrium point where the pendulum is upright, the cost is zero, the state roughly $6 \approx 2\pi$ (lower right center plot), and the angular velocity zero (bottom right plot). Unstable equilibrium points of nonlinear systems are very sensitive, which in our case means that a slightly wrong action applied to the upright pendulum can decide whether the pendulum stays upright, swings right or left. In Figure 5.6 we can see that these accumulated errors are not critical in practice, because a good policy (as opposed to a static action replay) can compensate slight deviations and still manage to swing up the pendulum.

5.2.2. I2C: find locally optimal controllers

5.2.2.1. Using true dynamics

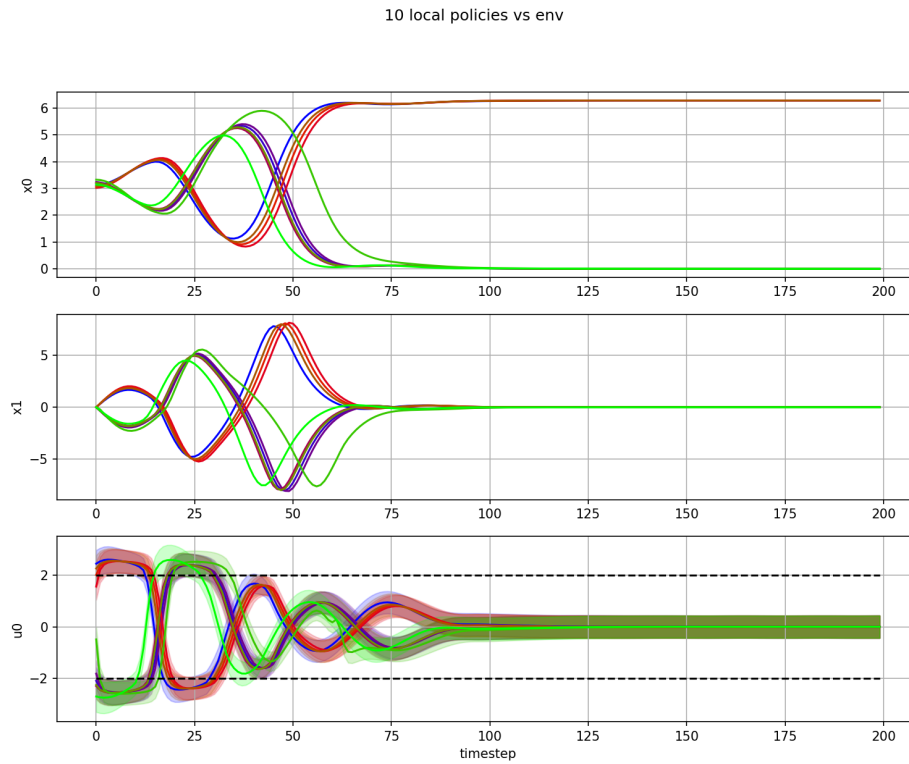


Figure 5.7.: Roll-outs of all 10 local optimal controllers in the environment (true dynamics).

10 i2c metrics (temp.strategy: PolyakStepSize)

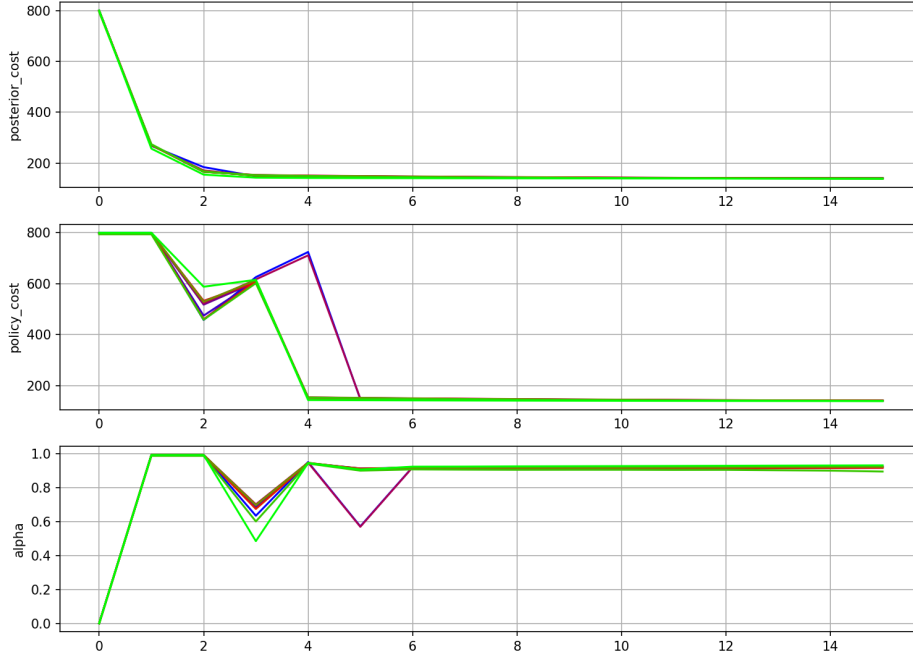


Figure 5.8.: Metrics of the I2C optimization over optimization iterations: the posterior cost is the cumulative cost of the smoothed trajectory including the temperature-controlled nudge towards lower cost, the policy cost is like the posterior cost, but without the nudge towards lower cost and α is the temperature. The optimization stopped after 15 steps because both the posterior and the policy costs were below the threshold $\Delta c_{th} = 1$ for 5 iterations.

In Figure 5.7 we can see the performance of the 10 different local controllers in the environment (true dynamics). Considering that the initial states are randomly sampled from the environment and are not identical to the initial states chosen for the optimization, the controllers seem reasonably good and robust. From the optimization metrics in Figure 5.8 we can deduce that the optimization has converged properly, since the posterior and the policy costs are similar, which means that every controller takes (locally) cost-

optimal actions in every time step⁴.

5.2.2.2. Using SNGP dynamics from untrained SNGP policy roll-outs

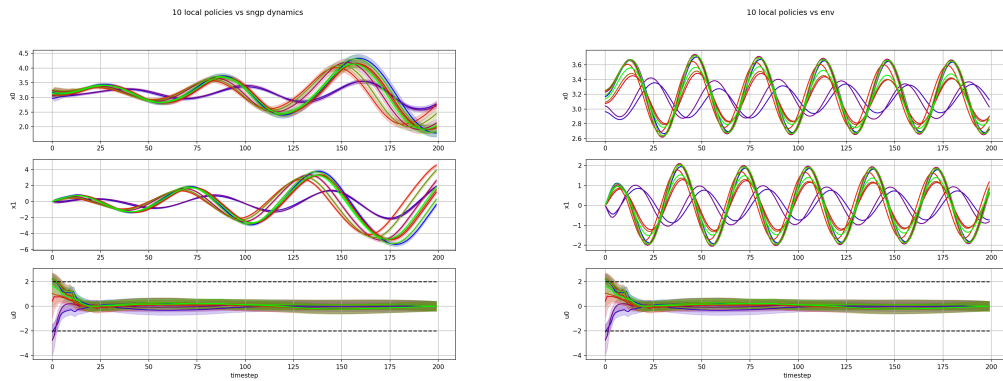


Figure 5.9.: Roll-outs of all 10 local optimal controllers in the learned SNGP dynamics (left) and in the environment (right).

In Figure 5.9 we can see the performance of the 10 different local controllers in the SNGP dynamics and the environment (true dynamics). Both sets of roll-outs are obviously non-optimal, which means that the optimization has not converged to any optimal controllers. This is also apparent in the action sequences in the bottom of the plots, which, except for the first two dozen steps, are mostly zero. The differences between the roll-outs in the SNGP dynamics and the environment seem to indicate that the SNGP dynamics model might not model the environment adequately enough in the area of the state-space encountered in the optimization. This mirrors our observations about the training of this dynamics model in Section 5.2.1.1 and can be seen particularly in the apparent gain of kinetic energy in the SNGP dynamics (the amplitudes of the oscillations increase), while the controller actions are almost zero and should not add energy to the system.

⁴Strictly speaking, the locally optimal nature of the solutions means that the controllers take optimal actions not for the *true cost* but for the *local quadrature approximated cost*. Therefore, the optimization might not actually have converged to a local cost-optimal trajectory yet, but the minimum of the approximated cost would be close to the current trajectory. This would result in weak nudging and small trajectory updates, slowing the optimization until it would appear to converge.



10 i2c metrics (temp.strategy: PolyakStepSize)

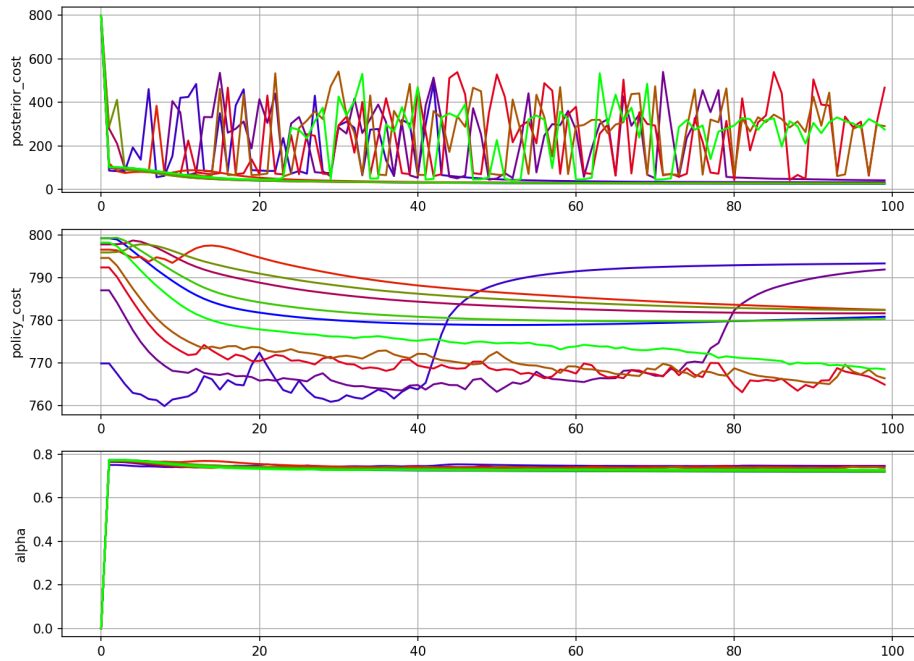


Figure 5.10.: Metrics⁵ of the I2C optimization over optimization iterations. The optimization stopped after 100 steps, which is the iteration limit of the optimization, because the convergence threshold is not reached.

The optimization metrics in Figure 5.10 prove that the controllers have not converged and document an erratic optimization. The posterior cost often comes close to its minimum but keeps jumping up and down. Most policy costs seem to converge to a sub-optimal policy with very high cost, but some controllers have jagged and some even rising costs. Our hypothesis to explain this behaviour is that the learned SNGP dynamics are so variable locally that the quadrature approximation becomes an oversimplification which misleads the optimization. All in all, the resulting controllers do not seem good enough to warrant trying to distill them into a global policy.

⁵see Figure 5.8 for explanation

5.2.2.3. Using SNGP dynamics from trained SNGP policy roll-outs

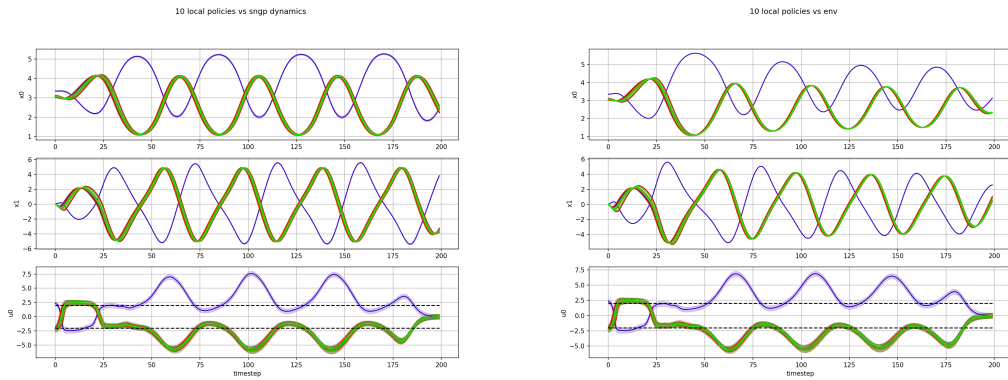


Figure 5.11.: Roll-outs of all 10 local optimal controllers in the learned SNGP dynamics (left) and in the environment (right).

10 i2c metrics (temp.strategy: PolyakStepSize)

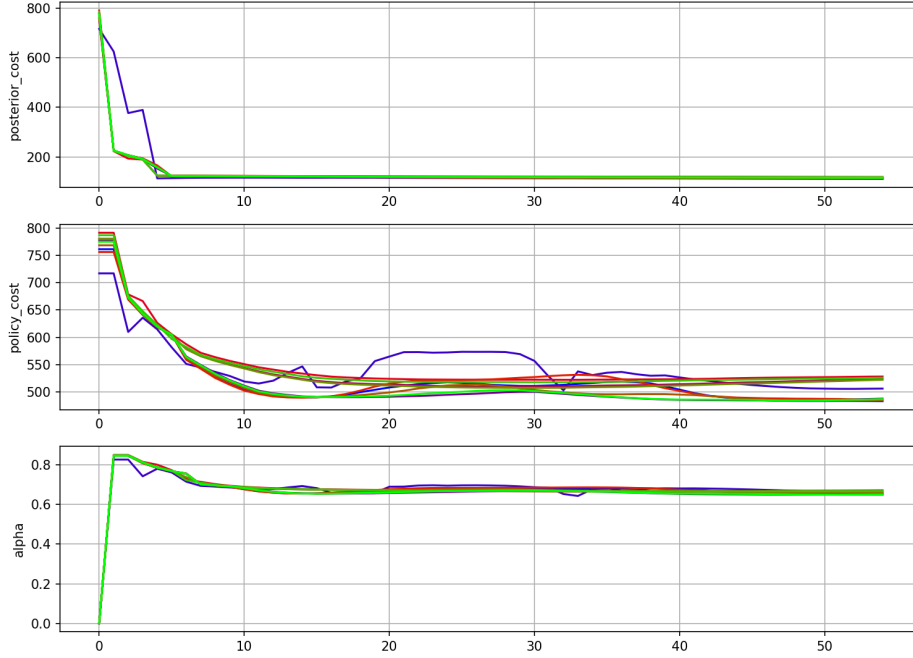


Figure 5.12.: Metrics⁶ of the I2C optimization over iterations. The optimization stopped after roughly 50 steps because both the posterior and the policy costs were below the threshold $\Delta c_{\text{th}} = 1$ for 5 iterations.

In Figure 5.11 we can see the performance of the 10 different local controllers in the SNGP dynamics and the environment (true dynamics). Both sets of roll-outs are obviously non-optimal, which means that the optimization has not found any optimal controllers. The action sequences are mostly outside of the allowed action interval, which means they are almost always unable to apply effective control to the system. Together with the optimization metrics in Figure 5.12 it becomes apparent that the optimization plateaus without reaching a minimum: once the optimization pushes the action sequences outside the valid action interval, the effect of a change in an action vanishes because their effect

⁶see Figure 5.8 for explanation

is clipped by the action limiter. Therefore, the cost stagnates and the optimization starves. However, it is still unclear to us why the action trajectories were pushed outside of their limits in the first place. Just like the controllers in Section 5.2.2.2, these controllers do not seem good enough to warrant trying to distill them into a global policy.

On a different note, the roll-outs in this SNGP dynamics (learned from a trained SNGP policy) are much more similar to the environment roll-outs than in Section 5.2.2.2, where the SNGP dynamics is learned from an untrained policy. Together with the much less jagged posterior and policy costs, this seems to show that the SNGP dynamics used here is a more accurate model of the environment than the one in Section 5.2.2.2.

5.2.3. SNGP policy learning from local controllers

5.2.3.1. Using moment matching loss

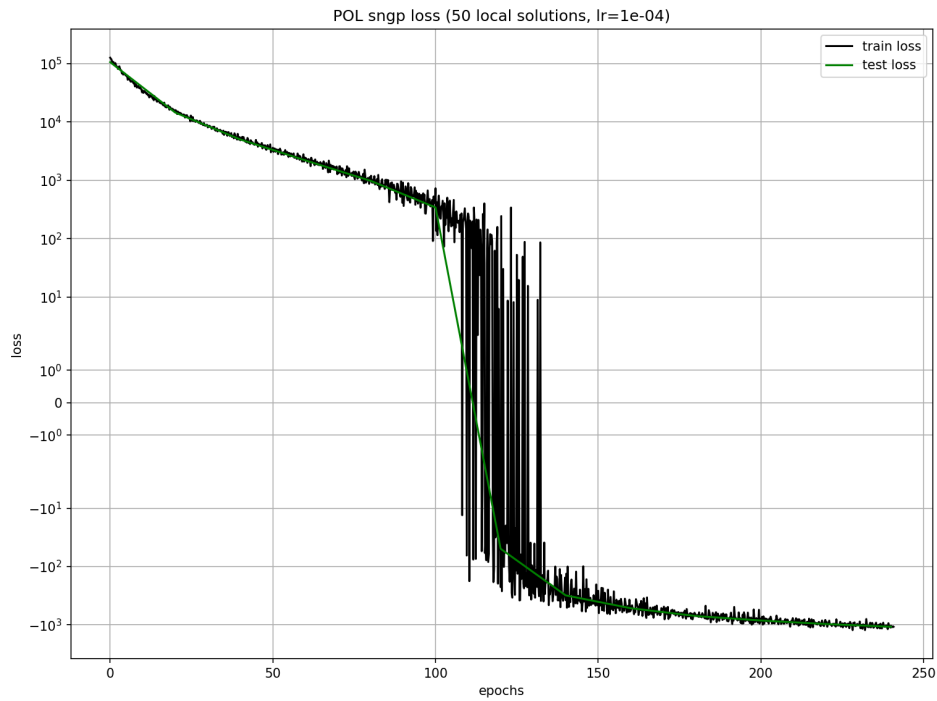


Figure 5.13.: Moment matching loss on training and evaluation data. The training stopped after 240 epochs because both the test and evaluation loss were below the threshold $\ell_{\text{th}} = 1 \times 10^{-3}$ (loss plot uses a symlog scaled y-axis explained in Figure 4.1).

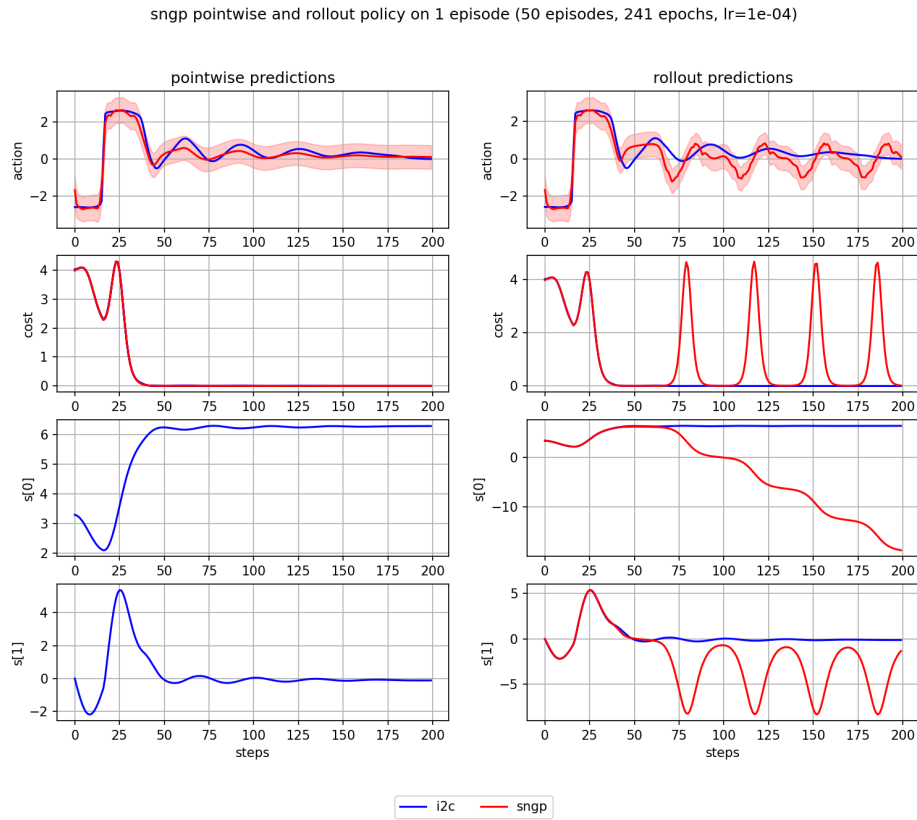


Figure 5.14.: Policy prediction fit on one evaluation episode: the left side shows the point-wise prediction fit, the right side shows a roll-out of the policy from the starting state from the point-wise data

In Figure 5.13 we can see the SNGP policy training and evaluation loss curves. The training of the model converges much faster than the policy during the de-risking experiments in Chapter 4, which matches our expectation that DAgger training takes longer than supervised learning. In Figure 5.14 we can see the point-wise and roll-out fit of the policy. Generally, the fit seems quite good, but the roll-outs diverge after reaching the target state and the pendulum starts looping without managing to stop. This could indicate a bad robustness of the controller.

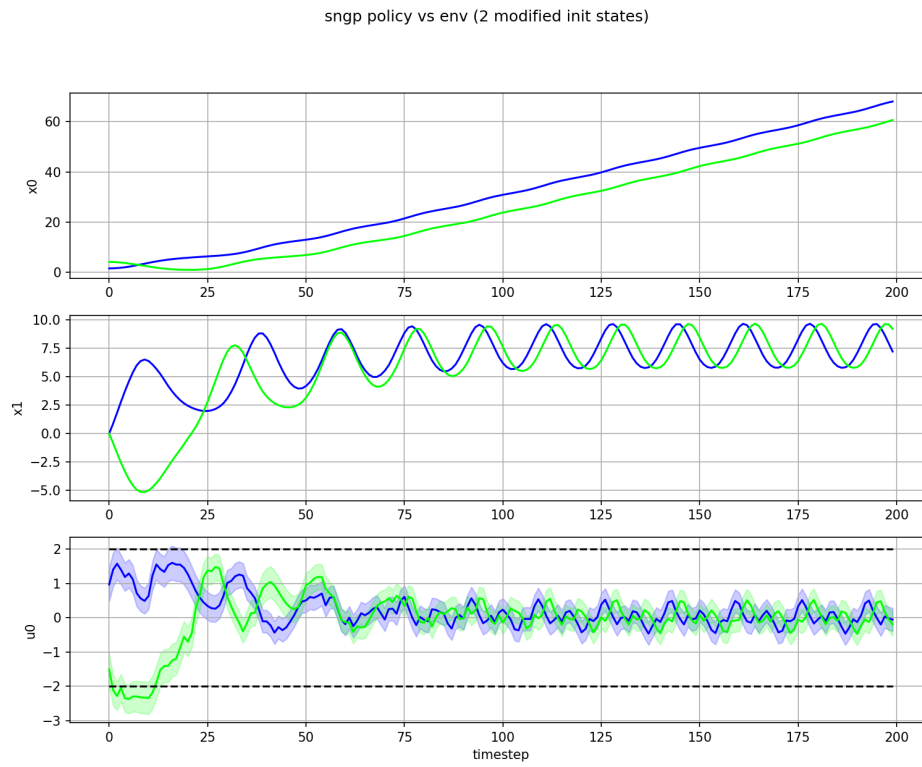


Figure 5.15.: Learned policy (M-projection) roll-out starting from 2 modified initial states far from the initial states used in the I2C optimization to find the controllers distilled into the policy

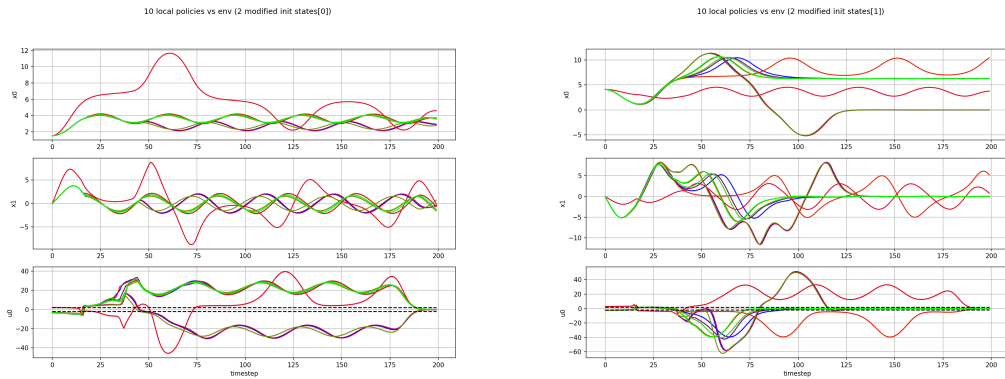


Figure 5.16.: Local controller roll-out starting from 2 modified initial states far from the starting states used to find them: the left side show roll-outs from the first modified initial state, the right side from the second.

In figures 5.15 and 5.16 we can see the performance of the learned SNGP policy and the local controllers. The SNGP policy does not manage to swing the pendulum up from this state, whereas some (but not all) controllers succeed at stabilizing the system. This supports our hypothesis that the M-projection SNGP policy is not more robust than the controllers and does not generalize well to outside of its training data.

5.2.3.2. Using DAgger with importance weighed mixture policy and ELBO loss

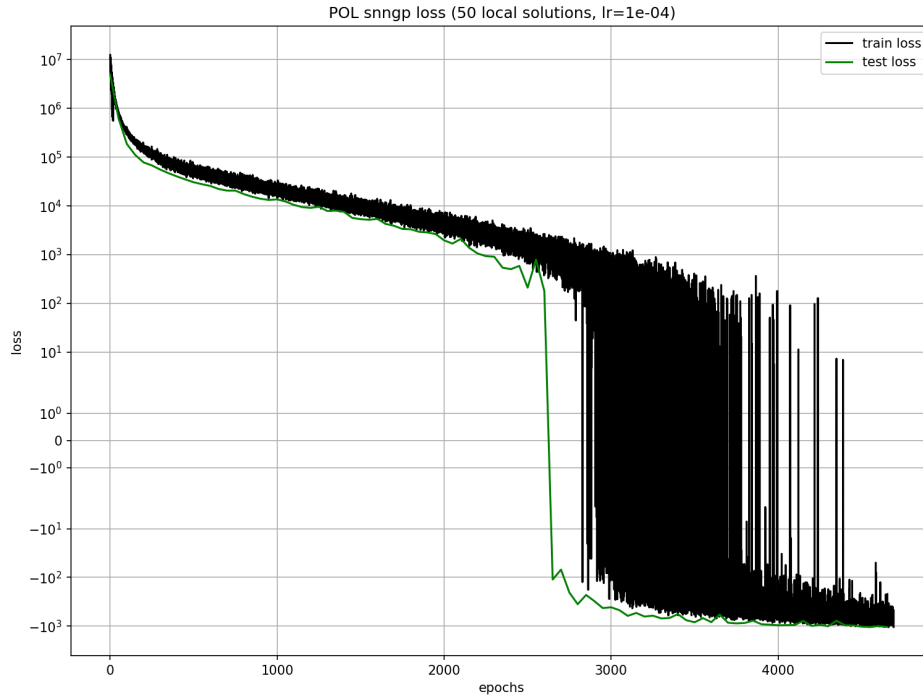


Figure 5.17.: ELBO loss on aggregating (Dagger) training and evaluation data. The training stopped after about 4500 epochs because both the test and evaluation loss were below the threshold $\ell_{\text{th}} = 1 \times 10^{-3}$ (loss plot uses a symlog scaled y-axis explained in Figure 4.1).

snnpg pointwise and rollout policy on 1 episode (50 episodes, 4692 epochs, lr=1e-04)

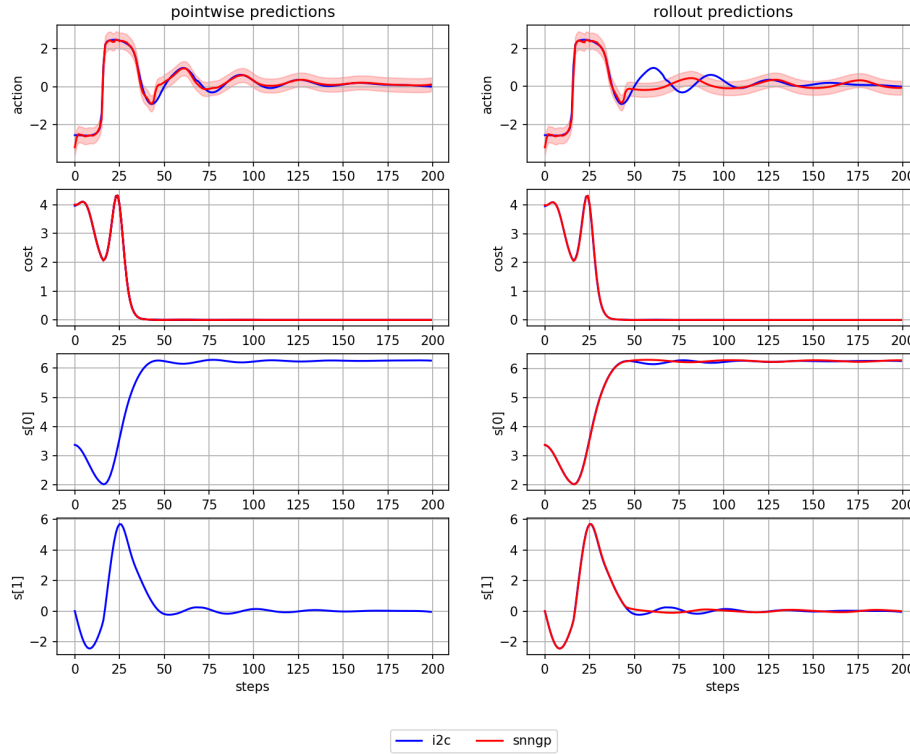


Figure 5.18.: Policy prediction fit on one evaluation episode: the left side shows the point-wise prediction fit, the right side shows a roll-out of the policy from the starting state from the point-wise data

In Figure 5.17 we can see the SNGP policy training and evaluation loss curves. The training of the model seems to converge roughly at the same speed as the policy during the de-risking experiments in Chapter 4 for the same model size and learning rate. In Figure 5.18 we can see the point-wise and roll-out fit of the policy. Generally, the fit seems very good. Especially the deviation of the roll-out predictions on the right from the blue nominal trajectory leading to an even smoother equilibrium without destabilizing the pendulum, seem to indicate a good robustness.

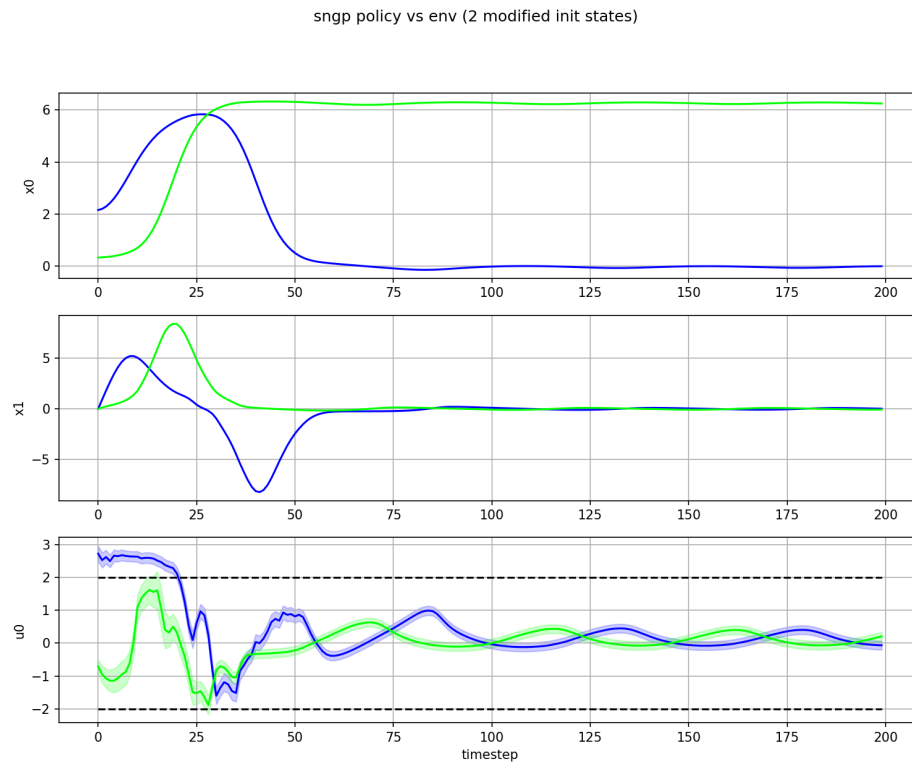


Figure 5.19.: Learned policy (Dagger) roll-out starting from 2 modified initial states far from the initial states used in the I2C optimization to find the controllers distilled into the policy

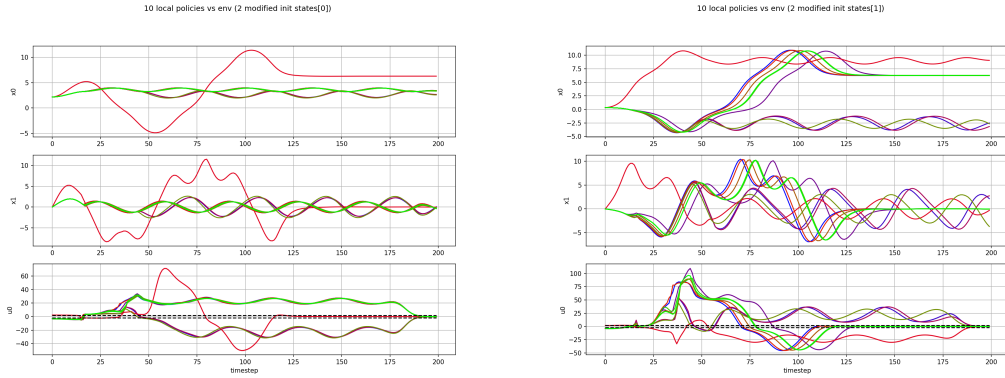



Figure 5.20.: Local controller roll-out starting from 2 modified initial states far from the starting states used to find them: the left side show roll-outs from the first modified initial state, the right side from the second.

In figures 5.19 and 5.20 we can see the performance of the learned SNGP policy and the local controllers for modified initial states. This confirms our observation that the SNGP policy is not only robust, but even more robust than the optimal controllers.

5.3. Summary

In these experiments we have seen that the SNGP dynamics model improves with the state-space coverage of its data. The initial SNGP dynamics trained on random action roll-outs learns a good pointwise fit, but the open-loop trajectory diverges quickly, because of the low exploration and high frequency noise of the random actions. The SNGP dynamics trained on good trajectories has a much better open-loop fit, and results in a more plausible dynamics model. From the experiments with the I2C optimization we have learned that the convergence criterion works as intended, but the optimization using a learned SNGP dynamics model requires further tuning to converge. The first SNGP dynamics model, which was trained from random actions, is too noisy to be optimized, and the optimization of the second SNGP dynamics model, which was trained on good trajectories, pushes the action sequences outside the action constraint, which clips the actions and stalls the optimization. Furthermore, the policy learning experiments have shown that using DAGger and the ELBO loss is a promising approach to distill the local optimal controllers found by



I2C into a SNGP policy, and results in a smoother and more robust policy, which sensibly combines the knowledge of the different optimal controllers. On the other hand, training the SNGP policy with the moment matching loss converged to a sub-optimal and brittle policy, with bad pointwise fit and even worse open-loop fit.

6. Conclusion

The intention of this thesis was to set up a model-based reinforcement learning (MBRL) loop with a global SNGP dynamics model and a global SNGP policy connected by I2C optimizations to find cost-optimal action trajectories. In Chapter 4 we have seen that the modified SNGP model is expressive enough to imitate expert policies and the true environment dynamics well enough to be used in a MBRL setting. In Chapter 5 we have set up and analyzed the individual transitions in the MBRL loop. We have seen that the SNGP dynamics model improves with the state-space coverage of its data, which is why a SNGP dynamics trained on random action roll-outs does not learn a good representation of the true dynamics, whereas a SNGP dynamics trained on good trajectories seems to learn a good model. The I2C optimization has been shown to work when using the true dynamics, but when using any learned SNGP dynamics model the training does not converge to promising, let alone optimal trajectories. To train the SNGP policy, we have found the DAgger approach to result in smoother and more robust policies than the moment matching loss, albeit for a significantly longer training time.

In future, to allow closing the MBRL loop and analyzing it, it is probably necessary to improve the exploration when collecting roll-outs to train the SNGP dynamics, especially in the initial iteration when there is no good policy to guide the environment into interesting areas of the state-space. Furthermore, the I2C optimization has to be tuned or regularized in a way to allow finding optimal trajectories from an SNGP dynamics model. This could include trying different temperature strategies or convergence criteria. To successfully distill the optimized controllers into the SNGP policy we propose using the DAgger approach. It might be necessary to filter or weigh the controllers based on their policy cost to prevent the distillation of sub-optimal controllers into the policy, while considering that sub-optimal behaviour might also improve exploration.

Bibliography

- [1] S. Levine and V. Koltun, “Variational policy search via trajectory optimization,” in *Advances in Neural Information Processing Systems* (C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Weinberger, eds.), vol. 26, Curran Associates, Inc., 2013.
- [2] S. Levine and V. Koltun, “Guided policy search,” in *Proceedings of the 30th International Conference on Machine Learning* (S. Dasgupta and D. McAllester, eds.), vol. 28 of *Proceedings of Machine Learning Research*, (Atlanta, Georgia, USA), pp. 1–9, PMLR, 17–19 Jun 2013.
- [3] S. Levine and P. Abbeel, “Learning neural network policies with guided policy search under unknown dynamics,” in *Advances in Neural Information Processing Systems* (Z. Ghahramani, M. Welling, C. Cortes, N. Lawrence, and K. Weinberger, eds.), vol. 27, Curran Associates, Inc., 2014.
- [4] S. Levine, C. Finn, T. Darrell, and P. Abbeel, “End-to-end training of deep visuomotor policies,” Apr. 2015.
- [5] S. Levine, “Reinforcement learning and control as probabilistic inference: Tutorial and review,” May 2018.
- [6] W. Li and E. Todorov, “Iterative linear quadratic regulator design for nonlinear biological movement systems,” in *International Conference on Informatics in Control, Automation and Robotics*, SciTePress - Science and Technology Publications, 2004.
- [7] Y. Tassa, T. Erez, and E. Todorov, “Synthesis and stabilization of complex behaviors through online trajectory optimization,” in *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, IEEE, oct 2012.

-
-
- [8] M. P. Deisenroth and C. E. Rasmussen, “Pilco: A model-based and data-efficient approach to policy search,” in *Proceedings of the 28th International Conference on International Conference on Machine Learning, ICML’11*, (Madison, WI, USA), p. 465–472, Omnipress, 2011.
 - [9] J. Z. Liu, Z. Lin, S. Padhy, D. Tran, T. Bedrax-Weiss, and B. Lakshminarayanan, “Simple and principled uncertainty estimation with deterministic deep learning via distance awareness,” June 2020.
 - [10] J. Watson and J. Peters, “Advancing trajectory optimization with approximate inference: Exploration, covariance control and adaptive risk,” Mar. 2021.
 - [11] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine, “Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor,” Jan. 2018.
 - [12] F. Nielsen, “WHAT IS...an information projection?,” *Notices of the American Mathematical Society*, vol. 65, p. 1, mar 2018.
 - [13] C. E. Rasmussen and C. K. I. Williams, *Gaussian Processes for Machine Learning*. The MIT Press, 2005.
 - [14] B. Eysenbach and S. Levine, “Maximum entropy rl (provably) solves some robust rl problems,” Mar. 2021.
 - [15] S. Ross, G. J. Gordon, and J. A. Bagnell, “A reduction of imitation learning and structured prediction to no-regret online learning,” Nov. 2010.
 - [16] T. Miyato, T. Kataoka, M. Koyama, and Y. Yoshida, “Spectral normalization for generative adversarial networks,” Feb. 2018.
 - [17] R. V. Mises and H. Pollaczek-Geiringer, “Praktische verfahren der gleichungsaufloesung .,” *ZAMM - Zeitschrift für Angewandte Mathematik und Mechanik*, vol. 9, no. 2, pp. 152–164, 1929.
 - [18] G. H. Golub and H. A. van der Vorst, “Eigenvalue computation in the 20th century,” *Journal of Computational and Applied Mathematics*, vol. 123, pp. 35–65, nov 2000.
 - [19] S. Markou, “Random fourier features.” website.
 - [20] A. Rahimi and B. Recht, “Random features for large-scale kernel machines,” p. 1177–1184, 2007.

-
-
- [21] J. T. Wilson, V. Borovitskiy, A. Terenin, P. Mostowsky, and M. P. Deisenroth, “Efficiently sampling functions from gaussian process posteriors,” *International Conference on Machine Learning*, 2020, Feb. 2020.
- [22] R. Kass, L. Tierney, and J. Kadane, “Laplace’s method in bayesian analysis,” 1991.
- [23] J. Watson, H. Abdulsamad, and J. Peters, “Stochastic optimal control as approximate input inference,” Oct. 2019.
- [24] J. Watson, H. Abdulsamad, R. Findeisen, and J. Peters, “Efficient stochastic optimal control through approximate bayesian input inference,” May 2021.
- [25] S. Särkkä, *Bayesian Filtering and Smoothing*. Cambridge University Press, sep 2013.
- [26] S. Särkkä, J. Hartikainen, L. Svensson, and F. Sandblom, “On the relation between gaussian process quadratures and sigma-point methods,” Apr. 2015.
- [27] A. P. Dempster, N. M. Laird, and D. B. Rubin, “Maximum likelihood from incomplete data via the em algorithm,” *Journal of the Royal Statistical Society. Series B (Methodological)*, vol. 39, no. 1, pp. 1–38, 1977.
- [28] P. Dayan and G. E. Hinton, “Using expectation-maximization for reinforcement learning,” *Neural Computation*, vol. 9, pp. 271–278, feb 1997.
- [29] S. Julier and J. Uhlmann, “Unscented filtering and nonlinear estimation,” *Proceedings of the IEEE*, vol. 92, pp. 401–422, mar 2004.
- [30] T. P. Minka, “From hidden markov models to linear dynamical systems,” tech. rep., 1999.
- [31] T. P. Minka, “Bayesian linear regression,” techreport, 1998 (revised 2010).
- [32] J. Soch, T. B. O. S. Proofs, T. J. Faulkenberry, K. Petrykowski, and C. Allefeld, “Statproofbook/statproofbook.github.io: Statproofbook 2020,” 2020.
- [33] J. Watson, “Bayesian linear regression technical note,” tech. rep., 2022.
- [34] M. Rosenblatt, “Remarks on some nonparametric estimates of a density function,” *The Annals of Mathematical Statistics*, vol. 27, pp. 832–837, sep 1956.
- [35] E. Parzen, “On estimation of a probability density function and mode,” *The Annals of Mathematical Statistics*, vol. 33, pp. 1065–1076, sep 1962.

A. Matrix normal distribution

A random $(d_{in} \times d_{out})$ matrix X is matrix-normally distributed

$$X \sim \mathcal{MN}(\cdot | M, \Sigma_{in}, \Sigma_{out}) \quad (\text{A.1})$$

with real $(d_{in} \times d_{out})$ matrix M , positive definite $(d_{in} \times d_{in})$ input (or row) covariance Σ_{in} and positive definite $(d_{out} \times d_{out})$ output (or column) covariance Σ_{out} if and only if its density is¹

$$\mathcal{MN}_{d_{in}, d_{out}} = \frac{\exp \left[-\frac{1}{2} \text{tr} \left\{ \Sigma_{out}^{-1} (X - M)^\top \Sigma_{in}^{-1} (X - M) \right\} \right]}{(2\pi)^{\frac{d_{in} d_{out}}{2}} |\Sigma_{out}|^{\frac{d_{in}}{2}} |\Sigma_{in}|^{\frac{d_{out}}{2}}} . \quad (\text{A.2})$$

The matrix normal distribution is equivalent to the vectorized normal distribution such that if X is matrix normally distributed as in (A.1), then $\text{vec}(X)$ is multivariate normally distributed²:

$$\text{vec}(X) \sim \mathcal{N}(\text{vec}(M), \Sigma_{out} \otimes \Sigma_{in}) \quad (\text{A.3})$$

Note that there is a scale ambiguity between the two covariances³, such that for any real scalar $t > 0$:

$$\mathcal{MN}(\cdot | M, \Sigma_{in}, \Sigma_{out}) = \mathcal{MN}(\cdot | M, t\Sigma_{in}, \frac{1}{t}\Sigma_{out}) . \quad (\text{A.4})$$

The first and second order moments are⁴

$$\mathbb{E}[X] = M \quad (\text{A.5})$$

$$\mathbb{E}[(X - M)^\top (X - M)] = \Sigma_{in} \text{tr}\{\Sigma_{out}\} \quad (\text{A.6})$$

$$\mathbb{E}[(X - M)(X - M)^\top] = \Sigma_{out} \text{tr}\{\Sigma_{in}\} . \quad (\text{A.7})$$

¹<https://statproofbook.github.io/D/matn>

²<https://statproofbook.github.io/P/matn-mvn>

³https://en.wikipedia.org/wiki/Matrix_normal_distribution

⁴https://en.wikipedia.org/wiki/Matrix_normal_distribution#Expected_values

The Kullback-Leibler divergence between two matrix normal distributions (same dimensions)

$$X \sim \mathcal{MN}_0(\cdot | M_0, \Sigma_{0,in}, \Sigma_{0,out}) \quad (\text{A.8})$$

$$X \sim \mathcal{MN}_1(\cdot | M_1, \Sigma_{1,in}, \Sigma_{1,out}) \quad (\text{A.9})$$

is given by⁵

$$\begin{aligned} D_{\text{D}_{\text{KL}}}[\mathcal{MN}_0 \parallel \mathcal{MN}_1] &= \frac{1}{2} \left[\text{vec}(M_1 - M_0)^\top \text{vec}(\Sigma_{1,in}^{-1}(M_1 - M_0)\Sigma_{1,out}^{-1}) \right. \\ &\quad + \text{tr} \left\{ (\Sigma_{1,out}^{-1}\Sigma_{0,out}) \otimes (\Sigma_{1,in}^{-1}\Sigma_{0,in}) \right\} \\ &\quad \left. - d_{in} \log \frac{|\Sigma_{0,out}|}{|\Sigma_{1,out}|} - d_{out} \log \frac{|\Sigma_{0,in}|}{|\Sigma_{1,in}|} - d_{in}d_{out} \right] \quad (\text{A.10}) \end{aligned}$$

$$\begin{aligned} &= \frac{1}{2} \text{tr} \{ \Sigma_{1,in}^{-1} \Sigma_{0,in} \} \text{tr} \{ \Sigma_{1,out}^{-1} \Sigma_{0,out} \} \\ &\quad + \frac{1}{2} \text{tr} \left\{ (M_1 - M_0)^\top \Sigma_{1,in}^{-1} (M_1 - M_0) \Sigma_{1,out}^{-1} \right\} \\ &\quad - \frac{1}{2} \left(d_{in}d_{out} + d_{out} \log \frac{|\Sigma_{0,in}|}{|\Sigma_{1,in}|} + d_{in} \log \frac{|\Sigma_{0,out}|}{|\Sigma_{1,out}|} \right) . \quad (\text{A.11}) \end{aligned}$$

which is rearranged by using the following properties:

$$\text{tr}(A^\top B) = \text{vec}(A)^\top \text{vec}(B) \quad 6 \quad (\text{A.12})$$

$$\text{tr}\{A \otimes B\} = \text{tr}\{A\} \cdot \text{tr}\{B\} \quad 7 \quad (\text{A.13})$$

$$|A^{-1}| = |A|^{-1} . \quad 8 \quad (\text{A.14})$$

⁵<https://statproofbook.github.io/P/matn-kl.html>

⁶[https://en.wikipedia.org/wiki/Vectorization_\(mathematics\)](https://en.wikipedia.org/wiki/Vectorization_(mathematics))

⁷[https://en.wikipedia.org/wiki/Trace_\(linear_algebra\)](https://en.wikipedia.org/wiki/Trace_(linear_algebra))

⁸<https://en.wikipedia.org/wiki/Determinant>

B. Evidence lower bound loss derivation

The **loss function** used to learn the variational posterior hyper-parameters is the **evidence lower bound (ELBO)** [33]

$$\mathcal{L}_{ELBO}(\mathcal{D}) = \mathbb{E}_{W \sim q(W|\mathcal{D})}[\log p(D|W, \Sigma_{\epsilon, out})] - D_{\text{D}_{\text{KL}}}[W_q \parallel W_0] \quad (\text{B.1})$$

where the first term is the expected log likelihood for weights distributed according to the variational weight posterior, and the second term is the Kullback-Leibler divergence of the variational posterior from the prior.

B.1. KL of posterior from prior

Using (A.10) the **Kullback-Leibler divergence** of the **variational posterior** W_q from the **prior** W_0 is

$$\begin{aligned}
 D_{\text{D}_{\text{KL}}}[W_q \parallel W_0] &= \frac{1}{2} \text{tr}\{\Sigma_{0,in}^{-1} \Sigma_{q,in}\} \text{tr}\{\Sigma_{0,out}^{-1} \Sigma_{q,out}\} \\
 &\quad + \frac{1}{2} \text{tr}\left\{(M_0 - M_q)^\top \Sigma_{0,in}^{-1} (M_0 - M_q) \Sigma_{0,out}^{-1}\right\} \\
 &\quad - \frac{1}{2} \left(d_x d_y + d_y \log \frac{|\Sigma_{q,in}|}{|\Sigma_{0,in}|} + d_x \log \frac{|\Sigma_{q,out}|}{|\Sigma_{0,out}|} \right) \quad (\text{B.2})
 \end{aligned}$$

$$\begin{aligned}
 &= \frac{1}{2} d_y \text{tr}\{\Sigma_{0,in}^{-1} \Sigma_{q,in}\} \\
 &\quad + \frac{1}{2} \text{tr}\left\{M_q^\top \Sigma_{0,in}^{-1} M_q \Sigma_{\epsilon,out}^{-1}\right\} \\
 &\quad - \frac{1}{2} \left(d_x d_y + d_y \log \frac{|\Sigma_{q,in}|}{|\Sigma_{0,in}|} \right) \quad (\text{B.3})
 \end{aligned}$$

$$\begin{aligned}
 &= \frac{1}{2} d_y \sigma_{0,in}^{-2} \text{tr}\{\Sigma_{q,in}\} \\
 &\quad + \frac{1}{2} \sigma_{0,in}^{-2} \text{tr}\left\{M_q^\top M_q \Sigma_{\epsilon,out}^{-1}\right\} \\
 &\quad - \frac{1}{2} \left(d_x d_y + d_x d_y \log \sigma_{0,in}^{-2} + d_y \log |\Sigma_{q,in}| \right) \quad (\text{B.4})
 \end{aligned}$$

where the trace and log terms of the output covariances $\Sigma_{0,out}$ cancel. The prior hyperparameters were also inserted and we used that

$$|A \otimes B| = |A|^m \cdot |B|^n, \quad \text{where } A \in \mathbb{R}^{n \times n}, B \in \mathbb{R}^{m \times m}. \quad (\text{B.5})$$

B.2. Log likelihood

To find the expected log likelihood we first derive the **log likelihood**

$$\begin{aligned} \log p(\mathcal{D}|W, \Sigma_{\epsilon, out}) &= -\frac{nd_y}{2} \log 2\pi - \frac{n}{2} \log |\Sigma_{\epsilon, out}| - \cancel{\frac{d_y}{2} \log |I_n|} \\ &\quad - \frac{1}{2} \text{tr}\{\Sigma_{\epsilon, out}^{-1} (Y - XW)^\top I_n (Y - XW)\} \end{aligned} \quad (\text{B.6})$$

$$\begin{aligned} &= -\frac{nd_y}{2} \log 2\pi - \frac{n}{2} \log |\Sigma_{\epsilon, out}| \\ &\quad - \frac{1}{2} \text{tr}\{\Sigma_{\epsilon, out}^{-1} (Y - XW)^\top (Y - XW)\} \end{aligned} \quad (\text{B.7})$$

$$\begin{aligned} &= -\frac{nd_y}{2} \log 2\pi - \frac{n}{2} \log |\Sigma_{\epsilon, out}| \\ &\quad - \frac{1}{2} \text{tr}\{\Sigma_{\epsilon, out}^{-1} (Y^\top Y - Y^\top XW - W^\top X^\top Y + W^\top X^\top XW)\} \end{aligned} \quad (\text{B.8})$$

$$\begin{aligned} &= -\frac{nd_y}{2} \log 2\pi - \frac{n}{2} \log |\Sigma_{\epsilon, out}| \\ &\quad - \frac{1}{2} \text{tr}\{\Sigma_{\epsilon, out}^{-1} (Y^\top Y - 2Y^\top XW + W^\top X^\top XW)\} \end{aligned} \quad (\text{B.9})$$

where we used that

$$\text{tr}\{A^\top\} = \text{tr}\{A\} \quad (\text{B.10})$$

together with the symmetry of the covariance matrix $\Sigma_{\epsilon, out}$ to collect the mixed terms.

B.3. Expected log likelihood

Now the **expected log likelihood** using weights from the variational posterior follows

$$\begin{aligned} & \mathbb{E}_{W \sim q(W|\mathcal{D})} [\log p(\mathcal{D}|W, \Sigma_{\epsilon, out})] \\ &= \mathbb{E}_{W \sim q} \left[-\frac{nd_y}{2} \log 2\pi - \frac{n}{2} \log |\Sigma_{\epsilon, out}| \right. \\ & \quad \left. - \frac{1}{2} \text{tr} \{ \Sigma_{\epsilon, out}^{-1} (Y^\top Y - 2Y^\top XW + W^\top X^\top XW) \} \right] \end{aligned} \quad (\text{B.11})$$

$$\begin{aligned} &= -\frac{nd_y}{2} \log 2\pi - \frac{n}{2} \log |\Sigma_{\epsilon, out}| \\ & \quad - \frac{1}{2} \text{tr} \left\{ \Sigma_{\epsilon, out}^{-1} \cdot \mathbb{E}_{W \sim q} \left[Y^\top Y - 2Y^\top XW + W^\top X^\top XW \right] \right\} \end{aligned} \quad (\text{B.12})$$

$$\begin{aligned} &= -\frac{nd_y}{2} \log 2\pi - \frac{n}{2} \log |\Sigma_{\epsilon, out}| \\ & \quad - \frac{1}{2} \text{tr} \left\{ \Sigma_{\epsilon, out}^{-1} \left(Y^\top Y - 2 \cdot \mathbb{E}_{W \sim q} \left[Y^\top XW \right] \right) \right\} \\ & \quad - \frac{1}{2} \text{tr} \left\{ \Sigma_{\epsilon, out}^{-1} \left(\mathbb{E}_{W \sim q} \left[W^\top X^\top XW \right] \right) \right\} \end{aligned} \quad (\text{B.13})$$

$$\begin{aligned} &= -\frac{nd_y}{2} \log 2\pi - \frac{n}{2} \log |\Sigma_{\epsilon, out}| \\ & \quad - \frac{1}{2} \text{tr} \left\{ \Sigma_{\epsilon, out}^{-1} \left(Y^\top Y - 2Y^\top XM_q \right) \right\} \\ & \quad - \frac{1}{2} \text{tr} \left\{ \Sigma_{\epsilon, out}^{-1} \left(\Sigma_{q, out} \text{tr} \{ \Sigma_{q, in} X^\top X \} + M_q^\top X^\top XM_q \right) \right\} \end{aligned} \quad (\text{B.14})$$

$$\begin{aligned} &= -\frac{nd_y}{2} \log 2\pi - \frac{n}{2} \log |\Sigma_{\epsilon, out}| \\ & \quad - \frac{1}{2} \text{tr} \left\{ \Sigma_{\epsilon, out}^{-1} \left(Y^\top Y - 2Y^\top XM_q + M_q^\top X^\top XM_q \right) \right\} \\ & \quad - \frac{1}{2} \text{tr} \left\{ \Sigma_{\epsilon, out}^{-1} \Sigma_{q, out} \right\} \text{tr} \left\{ \Sigma_{q, in} X^\top X \right\} \end{aligned} \quad (\text{B.15})$$

$$\begin{aligned} &= -\frac{nd_y}{2} \log 2\pi - \frac{n}{2} \log |\Sigma_{\epsilon, out}| \\ & \quad - \frac{1}{2} \text{tr} \left\{ \Sigma_{\epsilon, out}^{-1} \left(Y^\top Y - 2Y^\top XM_q + M_q^\top X^\top XM_q \right) \right\} \\ & \quad - \frac{1}{2} d_y \text{tr} \left\{ \Sigma_{q, in} X^\top X \right\} \end{aligned} \quad (\text{B.16})$$

where we used the following properties with $X \sim \mathcal{MN}(M, \Sigma_{in}, \Sigma_{out})$:

$$\text{tr}\{A \text{tr}\{B\}\} = \text{tr}\{A\} \text{tr}\{B\} \quad ^1 \quad (\text{B.17})$$

$$\mathbb{E}[\text{tr}\{X\}] = \text{tr}\{\mathbb{E}[X]\} \quad (\text{B.18})$$

$$\mathbb{E}_X[CX] = CM \quad (\text{B.19})$$

$$\mathbb{E}_X[X^\top] = M^\top \quad (\text{B.20})$$

$$\mathbb{E}_X[X^\top BX] = \Sigma_{out} \text{tr}\{\Sigma_{in} B^\top\} + M^\top BM \quad ^2 \quad (\text{B.21})$$

¹since $\text{tr}\{B\}$ is a constant, we can pull it out of the outer trace

²https://en.wikipedia.org/wiki/Matrix_normal_distribution

B.4. Evidence lower bound

Finally the **ELBO** is:

$$\begin{aligned}
\mathcal{L}_{ELBO}(\mathcal{D}) = & -\frac{nd_y}{2} \log 2\pi - \frac{n}{2} \log |\Sigma_{\epsilon, out}| \\
& - \frac{1}{2} \text{tr} \left\{ \Sigma_{\epsilon, out}^{-1} \left(Y^\top Y - 2Y^\top X M_q + M_q^\top X^\top X M_q \right) \right\} \\
& - \frac{1}{2} \text{tr} \left\{ \Sigma_{\epsilon, out}^{-1} \Sigma_{q, out} \right\} \text{tr} \left\{ \Sigma_{q, in} X^\top X \right\} \\
& - \frac{1}{2} \text{tr} \left\{ \Sigma_{0, in}^{-1} \Sigma_{q, in} \right\} \text{tr} \left\{ \Sigma_{0, out}^{-1} \Sigma_{q, out} \right\} \\
& - \frac{1}{2} \text{tr} \left\{ (M_0 - M_q)^\top \Sigma_{0, in}^{-1} (M_0 - M_q) \Sigma_{0, out}^{-1} \right\} \\
& + \frac{1}{2} \left(d_x d_y + d_y \log \frac{|\Sigma_{q, in}|}{|\Sigma_{0, in}|} + d_x \log \frac{|\Sigma_{q, out}|}{|\Sigma_{0, out}|} \right) \tag{B.22}
\end{aligned}$$

$$\begin{aligned}
= & -\frac{nd_y}{2} \log 2\pi - \frac{n}{2} \log |\Sigma_{\epsilon, out}| \\
& - \frac{1}{2} \text{tr} \left\{ \Sigma_{\epsilon, out}^{-1} \left(Y^\top Y - 2Y^\top X M_q + M_q^\top X^\top X M_q \right) \right\} \\
& - \frac{1}{2} d_y \text{tr} \left\{ \Sigma_{q, in} X^\top X \right\} \\
& - \frac{1}{2} d_y \sigma_{0, in}^{-2} \text{tr} \left\{ \Sigma_{q, in} \right\} \\
& - \frac{1}{2} \sigma_{0, in}^{-2} \text{tr} \left\{ M_q^\top M_q \Sigma_{\epsilon, out}^{-1} \right\} \\
& + \frac{1}{2} \left(d_x d_y + d_y \log |\Sigma_{q, in}| + d_y \log \sigma_{0, in}^{-2} \right) \tag{B.23}
\end{aligned}$$

C. Code

C.1. Quanser Qube environment reward function modifications

Listing C.1: original reward function

```
def _rwd(self, x, a):
    th, al, thd, ald = x
    al_mod = al % (2 * np.pi) - np.pi
    cost = (
        al_mod**2
        + 5e-3 * ald**2
        + 1e-1 * th**2
        + 2e-2 * thd**2
        + 3e-3 * a[0] ** 2
    )
    done = not self.state_space.contains(x)
    rwd = np.exp(-cost) * self.timing.dt_ctrl
    return np.float32(rwd), done
```

Listing C.2: modified reward function

```
def _tweaked_rwd(self, x, a):
    th, al, thd, ald = x
    al_mod = al % (2 * np.pi) - np.pi
    done = not self.state_space.contains(x)

    factor = [0.96, 0.039, 0.001] # [0.9, 0.05, 0.05]
    scales = [np.pi, 2.0, 5.0]

    err_dist = th
```

```
err_rot = al_mod
err_act = a[0]

rotation_rew = (1 - np. abs(err_rot / scales[0])) ** 2
distance_rew = (1 - np. abs(err_dist / scales[1])) ** 2
action_rew = (1 - np. abs(err_act / scales[2])) ** 2

# Reward should be roughly between [0, 1]
rew = (
    factor[0] * rotation_rew
    + factor[1] * distance_rew
    + factor[2] * action_rew
)
return np.float32(np.clip(rew, 0, 1)), done
```