RL for Statisticians: an Incomplete Introduction to Model-Based Reinforcement Learning

Beatrice Cantoni

Spring 2022

Contents

1 Introduction

This report was written as final project for the class SDS383D in Spring 2022. Its aim is to review and understand one of the most well known approaches in model-based Reinforcement Learning (RL) field, named PILCO by its inventors [1]. The report is designed to be useful for people with graduate-level statistical knowledge but without deep knowledge of RL literature and techniques. A public library, available online, was also used for simulations. Some difficulties encountered in dealing with the necessary packages and environments will be mentioned. A drawback encountered was the unavailability of certain packages for Windows, as well as some recent updates of important packages such as Tensorflow.

2 RL; Model-based versus Model-free

Reinforcement learning [5] is a class of learning problems in which an agent (or controller) interacts with a stochastic dynamic and incompletely known environment with the goal of finding an action-selection strategy or policy to optimize some measure of its long-term performance. The interaction is often modeled conventionally as a *Markov decision process* (MDP). In particular, RL proposes that good policies are inferred by observing rewards (dependent on the actions taken) and adjusting the associated policy that maps states to actions and thence to future rewards. This approach is loosely seen as lying between supervised and unsupervised learning. One big classification within RL models is the distinction between value function and policy search models. Firstly, value function methods approximate a function that maps states (or state-action pairs) to expected returns. Secondly, policy search approaches use a parametrised policy and, by sampling trajectories from the system, estimate the gradient of a reward (or cost) function with respect to the policy parameters and propose a new value.

Model-based vs Model-free Traditionally, RL algorithms have been categorized as being either model-based or model-free. In the former category, the agent uses the collected data to first build a model of the domain's dynamics and then uses this model to optimize its policy. In the latter case, the agent directly learns an optimal (or good) action-selection strategy from the collected data. These is some evidence that model-free methods are usually more flexible, whereas model-based methods offer more data efficiency. Generally speaking, model-based methods are more promising to efficiently extract valuable information from available data than model-free methods. On the other hand, model based methods are not widely used in learning from scratch since they suffer from model bias, i.e., they inherently assume that the learned dynamics model sufficiently resembles the real environment. Model bias is especially an issue when only a few samples and no informative prior knowledge about the task to be learned is available.

3 PILCO

Deisenroth and Rasmussen [1] proposed PILCO, a **model-based, policy search** method based on a probabilistic approach for learning the underlying dynamics of the model. **Its main advantage is the reduced model bias**, which is one of the key problems of model-based reinforcement learning. The key for such an improvement lies in the probabilistic approach itself, since it allows for explicitly incorporating model uncertainty into long-term planning. When such an intrinsic uncertainty is not in place, approximations of $p(\mathbf{x}_t)$ and the associated long term predictions risk being essentially arbitrary. This is even more harmful when few data are available. By contrast, a probabilistic function approximator, obtained using a Gaussian Process, as we will see, places a posterior distribution over the transition function, thereby expressing the level of uncertainty about the model.

3.1 Notation and setting

The key elements are:

- A dynamic system

  $$\mathbf{x}_t = f(\mathbf{x}_{t-1}, \mathbf{u}_{t-1}),$$

  where $\mathbf{x} \in \mathbb{R}^D$ is a continuous-valued state, $\mathbf{u}$ are the controls, and f encodes the unknown transition dynamic. For $\mathbf{x}_0$, we are assuming

  $$\mathbf{x}_0 \sim N(\mu_0, \Sigma_0)$$

- A *policy/controller* π

  $$\mathbf{x} :\rightarrow \pi(\mathbf{x}) = \mathbf{u}$$

  is a function parametrised by θ.

- The expected return of following a certain π for T steps is

  $$J^{\pi}(\theta) = \sum_{t=0}^{T} E_{\mathbf{x}_t}[c(\mathbf{x}_t)],$$

  where $c(\mathbf{x})$ is the cost of being in state $\mathbf{x}$ at time t, which encodes some information about a target state $\mathbf{x}_{target}$.

The goal is to find a deterministic *policy/controller* π that minimises the *expected return.*

Examples For a better understanding of this last element of the PILCO strategy, I will give two examples, coming from the applications mentioned by the authors in the paper.

Robot Unycycle One of the implementations described by the authors is the one of a robotic unicycle system. The robot consists of a wheel, a frame, and a flywheel mounted perpendicularly to the frame. The goal of the experiment was to ride the unicycle or, in other words, to train it to learn how to from falling. In this case, the input vector is 2 dimensional: a first torque applied directly on the wheel mimics a human rider using pedals and a second torque applied laterally to the flywheel. The last element needed to specify the setting is the selected controller. In this case, the authors opted for a linear controller $\pi(\mathbf{x}, \theta) = \mathbf{A}\mathbf{x} + \mathbf{b}$; hence in this example $\theta = \{\mathbf{A}, \mathbf{b}\}$.

Cart-Pole Swing-up Again from the original paper, PILCO was also applied to learning to control

a real cartpole system. The system consisted of a cart with mass 0.7 kg running on a track and a freely swinging pendulum with mass 0.325 kg attached to the cart. In this case, the state of the system is the position of the cart, the velocity of the cart, the angle of the pendulum, and the angular velocity. A horizontal force $\mathbf{u}$ could be applied to the cart. The objective was to learn a controller capable to swing the pendulum up and to balance it in the inverted position in the middle of the track. As the authors claim, a linear controller cannot be used in this setting. Rather, the learned state feedback controller was a nonlinear Radial Basis Function (RBF) network:

$$\pi(\mathbf{x}, \theta) = \sum_{i=1}^{n} w_i \phi_i(\mathbf{x})$$
$$\phi_i(\mathbf{x}) = \exp\left(-\frac{1}{2}\left(\mathbf{x} - \mu_i\right)^\top \mathbf{\Lambda}^{-1}\left(\mathbf{x} - \mu_i\right)\right)$$

Cart-double pendulum swing-up Similarly, the authors applied the PILCO learning strategy to the physics of the Cart-double Pendulum Swing-up. The cart-double pendulum system consists of a cart running on a track and a freely swinging two-link pendulum attached to it. The state of the system is the position, the velocity of the cart, the angles and the angular velocities of both attached pendulums. In this case, $\mathbf{u}$ corresponds to horizontal forces applied to the cart. Similarly to the Cart-pole Swing-up problem, a linear controller is not capable of solving this problem. A standard control approach to solving the cart-double pendulum task is to design two separate controllers, one for the swing up and one linear controller for the balancing task.

3.2 Modeling the underlying dynamics using a GP

As said, PILCO's underlying dynamics are implemented using a GP. In particular:

- the touples
$$\tilde{\mathbf{x}}_{t-1} := [\mathbf{x}_{t-1}, \mathbf{u}_{t-1}] \in \mathbb{R}^{D+F}$$
are the training inputs. For future reference and for a better understanding of the following sections, recall that
$$\mathbf{x}_t = f(\mathbf{x}_{t-1}, \mathbf{u}_{t-1}). \tag{1}$$
- the targets or output are the differences $\Delta_t = \mathbf{x}_t - \mathbf{x}_{t-1}$.

Since we are using a GP, we have:

$$\Delta_t = \mathbf{x}_t - \mathbf{x}_{t-1} + \epsilon \in \mathbb{R}^D \tag{2}$$

with $\epsilon \sim N(0, \Sigma_\epsilon)$. The authors also assume $\Sigma_\epsilon = diag([\sigma_{\epsilon_1}, ..., \sigma_{\epsilon_2}])$.
Going through the same derivations that we saw in class, one obtains that:

$$m_f\left(\tilde{\mathbf{x}}_*\right) = \mathbb{E}_f\left[\Delta_*\right] = \mathbf{k}_*^\top \left(\mathbf{K} + \sigma_\varepsilon^2 \mathbf{I}\right)^{-1} \mathbf{y} = \mathbf{k}_*^\top \beta, \tag{3}$$
$$\sigma_f^2\left(\Delta_*\right) = \mathrm{var}_f\left[\Delta_*\right] = k_{**} - \mathbf{k}_*^\top \left(\mathbf{K} + \sigma_\varepsilon^2 \mathbf{I}\right)^{-1} \mathbf{k}_*, \tag{4}$$

respectively, where $\mathbf{k}_* := k\left(\tilde{\mathbf{X}}, \tilde{\mathbf{x}}_*\right), k_{**} := k\left(\tilde{\mathbf{x}}_*, \tilde{\mathbf{x}}_*\right)$, $\beta := \left(\mathbf{K} + \sigma_\varepsilon^2 \mathbf{I}\right)^{-1} \mathbf{y}$, and $\mathbf{K}$ being the Gram matrix with entries $K_{ij} = k\left(\tilde{\mathbf{x}}_i, \tilde{\mathbf{x}}_j\right)$.

Mean function and Covariance function The prior mean function is chosen to be $m \equiv 0$. We have seen in class that the selection of the covariance function represents a crucial point when

modeling a process using a GP. In this case, the authors use the squared exponential kernel (SE), defined as:

$$k(\tilde{\mathbf{x}}, \tilde{\mathbf{x}}') = \alpha^2 \exp(-\frac{1}{2}(\tilde{\mathbf{x}} - \tilde{\mathbf{x}}')' \Lambda^{-1} (\tilde{\mathbf{x}} - \tilde{\mathbf{x}}')),$$

where we can think of α^2 as the variance of the latent function f. We have $\Lambda := diag([l_1^2, ..., l_D^2])$, so that the kernel function is parameterised by the hyperparameters $l_i, \alpha^2, \Sigma_\epsilon$. With the SE function, l_i have the role of a length-scale. Those hyperparameters are learned by *evidence maximization*, which, using a more statistical vocabulary, corresponds to a maximum likelihood strategy. As we commented in class, the GP with this covariance function is very smooth. This is due to the fact that this covariance function is infinitely differentiable. More details about it can be found in [4]. While such strong smoothness assumptions are unrealistic for modelling many physical processes, the squared exponential is probably the most widely-used kernel within the kernel machines field and turns out to not be harmful for models such PILCO.

3.3 From the GP to the expected return

Recall that our goal is to find the optimal π, which minimizes the *expected return* that we defined in previous sections. One can easily see how, in order to evaluate it, we need to first obtain an expression for $p(\mathbf{x}_t)$. This section is dedicated to explaining how $p(\mathbf{x}_t)$ is obtained in the model, starting from the GP structure we introduced in the previous section.

First note that, using (1) and (2), we can say that using a GP to model Δ_t implies:

$$p\left(\mathbf{x}_t \mid \mathbf{x}_{t-1}, \mathbf{u}_{t-1}\right) = \mathcal{N}\left(\mathbf{x}_t \mid \mu_t, \mathbf{\Sigma}_t\right) \tag{5}$$

$$\mu_t = \mathbf{x}_{t-1} + \mathbb{E}_f\left[\Delta_t\right], \tag{6}$$

$$\mathbf{\Sigma}_t = \mathrm{var}_f\left[\Delta_t\right] \tag{7}$$

Now it is time for a further (trivial) observation: in order to obtain $p(\mathbf{x}_t)$, we need to get rid of $\mathbf{u}_t$ and f in expression (5), (6), (7). This goal is achieved by constructing an approximation for both the joint distribution $p(\tilde{x}_{t-1}) = p(\mathbf{x}_{t-1}, \mathbf{u}_{t-1})$ and the predictive $p(\Delta_t)$. Both are approximated using Gaussian distributions with carefully selected means and covariances. For further reference and notation, we can note that those approximations allow us to write

$$p(\mathbf{u}_t) \approx N(\mu_u, \Sigma_u), \quad p(\Delta_t) \approx N(\mu_\Delta, \Sigma_\delta)$$

The details are not trivial, and a more comprehensive explanation is given in the appendix. At this point, using properties of Gaussians (hence combining the GP model with the approximations for the joint with $\mathbf{u}_t$ as well as the Gaussian approximation for $p(\Delta)$), one can write

$$p(\mathbf{x}_t) \approx N(\mu_t, \Sigma_t)$$

with

$$\mu_t = \mu_{t-1} + \mu_\Delta \tag{8}$$

$$\mathbf{\Sigma}_t = \mathbf{\Sigma}_{t-1} + \mathbf{\Sigma}_\Delta + \mathrm{cov}\left[\mathbf{x}_{t-1}, \Delta_t\right] + \mathrm{cov}\left[\Delta_t, \mathbf{x}_{t-1}\right] \tag{9}$$

$$\mathrm{cov}\left[\mathbf{x}_{t-1}, \Delta_t\right] = \mathrm{cov}\left[\mathbf{x}_{t-1}, \mathbf{u}_{t-1}\right] \mathbf{\Sigma}_u^{-1} \mathrm{cov}\left[\mathbf{u}_{t-1}, \Delta_t\right] \tag{10}$$

Details about how to derive $cov[\mathbf{x}_t, \Delta_t]$ can be found in Deisenroth 2010 [2].

Now that we obtained an approximation for $p(\mathbf{x}_t)$, we can evaluate the expected return J^π by computing the expected values

$$\mathbb{E}_{\mathbf{x}_t}[c(\mathbf{x}_t)] = \int c(\mathbf{x}_t)p(\mathbf{x}_t)d\mathbf{x}_t$$

In particular, the authors assumed c to be modeled as a squared exponential subtracted from unity, i.e.

$$c(\mathbf{x}) = 1 - exp(-||\mathbf{x} - \mathbf{x}_{target}||^2/\sigma_c^2) \in [0, 1]$$

where σ_c^2 is a terms that controls the width of c. This choice for c implies that the expectations can be found analytically.

3.4 From the expected return to the optimal policy

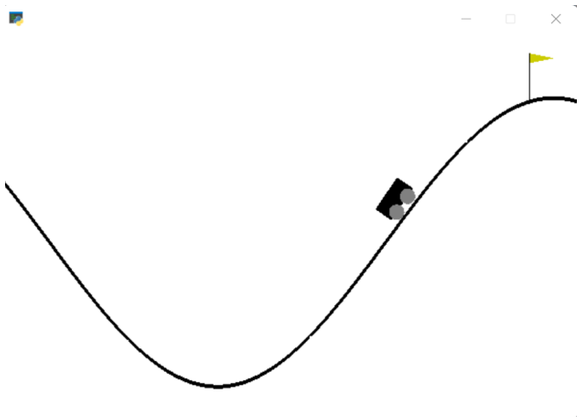
We are now left with the last bite of the PILCO strategy. Recall that examples of policy functions (or controllers) were given earlier in this model section. As said, the controller π can be thought as a function parameterised by θ. Hence, finding the optimal policy boils down to finding the optimal θ. This is done by analytically computing the gradients of the expected return J^π with respect to the policy parameter θ. Note that the parameter space will be different for different applications, since the controllers will have different functional forms. In particular, the derivative $\mathrm{d}J^\pi/\mathrm{d}\theta$ is obtained by repeated application of the chain rule. Details depend on the specific parametrization of the policy π.

4 Mountain car implementation

Implementing PILCO from scratch would have been incredibly satisfactory, but also an incredibly unfeasible task for the time available in preparing this project. Still, I was interested in having the chance to look at the mechanics of the model in more operational terms. Hence, I decided to run some experiments based on the code posted online by Nikitas Rontsis and Kyriakos Polymenakos on their GitHub repositories. Tensorflow is used to optimize the parameters in π (that is, in the explanation of the model I just gave, to compute the gradients), while the package GPflow is used to train the GP. The code provides the uses with different PILCO tools: a PILCO basic skeleton which combines the training of the GP and the optimization of the parameters, as well as different specifications of controllers. The user can combine the different pieces and construct a new experiment in the environments offered by OpenAI Gym. Within the OpenAI Gym environments, the most advanced and complex (in terms of both graphics and model structure) are the ones offered in the subgroup Mujoco. Installing those toolkits can be tedious sometimes, but the satisfaction of seeing them in action is worth the effort. Here, a quite simple environment is presented as an example. More elaborate environments can be found in the Mujoco library, presenting 3D rendering and complex objects structures. Unfortunately, due to the last changes in policies and in the packages to download, I had some issues with the more complex Mujoco environments, that I plan to fix soon in order to explore more advanced simulations. More explanation on further directions as well as on software problems encountered are given in the Final Comments section and in the code section in the Appendix.

The mountain car environment

In this environment, there is a car starting in between two hills. The goal is for the car to reach the top of the hill on the right. The car does not have enough engine power to reach the top of the hill by just accelerating forward. To win, the car must build momentum by swinging itself backwards and forwards until it has enough speed to reach the flag.



The setting is quite simple. In this case, $\mathbf{x}_t$ corresponds to the car position and velocity so that, in our notation, $D = 2$. The car only has 3 possible actions ($\mathbf{u}$) at every state: it can either accelerate forwards, accelerate backwards, or do nothing. Again, the goal is to learn what actions to take at each state to help it reach the flag. Hence, in our notation, the flag will represent $\mathbf{x}_{target}$. Following the PILCO strategy I just presented, the car will at every state look at all of its possible actions (accelerate forward, backward, or do not accelerate). It can then calculate the expected value of the future states and select the best action.
As we have seen, a key selection is the choice of the functional form of the policy function π. In this case, similarly to what the authors suggest to use for the Cart-pole swing-up environment, I used a RBF controller (see above in section 3.1). The code is written in Python, and I found its structure very pedagogical. Few comments on it follow:

- Recall that the car needs to learn the strategy on its own. More specifically, we can quote PILCO's inventors: *"Increasing data efficiency requires either having informative prior knowledge or extracting more information from available data. In this paper, we do not assume that any expert knowledge is available (e.g., in terms of demonstrations or differential equations for the dynamics). Instead, we elicit a general policy-search framework for data-efficient learning from scratch"*. In the code, this is achieved by running an initial random rollout to generate a dataset.

- In the actual environment, after the first initial rollout, the variables (hence also the environment reference measures) are normalised for computational stability.

- The packages GP flow and Tensorflow play a key role in training the GP process and optimizating the parameters faster and more efficiently from a computational point of view.

- Notably, the code is modular. This makes it possible to try out the results when choosing different policy functions.

- Importantly, and more on this will be said in the "Further comments on code" section in the Appendix, some recent modifications to OpenAI Gym and Tensorflow create some further

incompatibles when running this code. This problems are in addition to the already tricky usage of Mujoco on Windows machines.

5 Further applications and role in the literature

The examples given in the paper were presented earlier in section 4.1. Importantly, given its flexible structure, PILCO can be employed in both more standard benchmark problems and also high-dimensional control problems. Many other applications can be found in the literature. Indeed, after the first seminal paper about PILCO was published, PILCO rapidly became one of the most used strategies for model-baed reinforcement learning. Some videos of those implementations can be found at the corresponding YouTube channel: https://www.youtube.com/user/PilcoLearner.
An example of usage of PILCO in the literature can be given by the work of Rosenthal et al. [3], who made use of the PILCO framework for designing an algorithm for *safe* policy search. Their work was motivated by the fact that in many real application domains a primary concern is that of safety, particularly the avoidance of specific states or sets of states which are considered dangerous for the system or more generally undesirable (for example, the avoidance of obstacles). Using the probabilistic setting provided by the GP, one can define these constraints *a priori* and then estimate the risk of violating the constraints before any candidate policy is implemented in the actual system. This example, which is one of the many new models that have been built on the PILCO framework, shows once again the potential that statistics, which brings tools to quantify uncertainty, can bring to the field.

6 Final comments

The main purpose of this project was to take a first step in understanding one possible way in which statistical tools can be useful for Reinforcing Learning strategies. Also, the goal was to look at the role played by some of the tools we had the chance to look closer at in class. In this case, this tool was the Gaussian Process used to model the underlying evolution of the system.

My feeling is that this project exposed me to learn many new things, some of which are, I believe, really important if I aim to conduct research in the field of RL. In the first place, I had the chance to look closer at the OpenAI Gym, with its really cool environments but also its pitfalls in terms of installing and permit issues. I plan to be able to fix some limitations on my computer and to be able to explore more Mujoco environments in future projects. I also learned that when dealing with Reinforcement Learning research, a good knowledge of tools such as TensorFlow is a prerequisite for making the computation feasible. While I cannot claim to have coded PILCO from scratch, I did some experiments based on the code provided by Nikitas Rontsis and Kyriakos Polymenakos on their websites.

Importantly, this project introduced me to the great potential that statistics can have as a tool for robotics and machine learning techniques as a general category: model uncertainty. As seen with PILCO, this can reduce model bias and provide more efficient procedures. I believe that at least two directions can be taken making use of what learned from this project. On one side, one can think about further applications of the PILCO strategy and about how to export its building blocks to other settings. Moreover, one can ask himself the following question: can we make improvements

to the PILCO setting from a statistical point of view? Can we make the characteristics of the GP (its covariance function, for example) more specific for different environments? On the other side, which is maybe the more open and intellectually challenging one, we can think about other ways in which probabilistic and statistical tools can operate in bringing uncertainty evaluation in the RL field. For example, is it possible to evaluate the uncertainty of well established model-free tools, such as Q-learning? How random are the results obtained with heuristic well know techniques? Can agents be more prone to adapt to new environments by using probabilistic tools to improve their flexibility?

References

[1] Marc Deisenroth and Carl E Rasmussen. Pilco: A model-based and data-efficient approach to policy search. In *Proceedings of the 28th International Conference on machine learning (ICML-11)*, pages 465–472. Citeseer, 2011.

[2] Marc Peter Deisenroth. *Efficient reinforcement learning using Gaussian processes*, volume 9. KIT Scientific Publishing, 2010.

[3] Kyriakos Polymenakos, Alessandro Abate, and Stephen Roberts. Safe policy search with gaussian process models. *arXiv preprint arXiv:1712.05556*, 2017.

[4] Michael L Stein. *Interpolation of spatial data: some theory for kriging*. Springer Science & Business Media, 1999.

[5] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

Appendix

Approximating the input joint distribution

Note that, at the stage given by equations (3), (4), (5), a joint distribution $p\left(\mathbf{x}_{t-1}, \mathbf{u}_{t-1}\right)$ is required for predicting $\mathbf{x}_t$ from $\mathbf{x}_{t-1}$. Also, note that those one-step-predictions are crucial for evaluating the expected return J, since the state distributions $p(\mathbf{x}_1, ..., \mathbf{x}_T)$ are obtained by cascading one step predictions. Finally, recall $\mathbf{u}_{t-1} = \pi\left(\mathbf{x}_{t-1}, \theta\right)$; hence, the distribution associated computations will depend on the parametrization of the policy π. The authors compute the desired joint distribution as follows. First, they compute the mean μ_u and the covariance $\mathbf{\Sigma}_u$ of the predictive control distribution $p\left(\mathbf{u}_{t-1}\right)$ by integrating out the state. Subsequently, the cross-covariance $\operatorname{cov}\left[\mathbf{x}_{t-1}, \mathbf{u}_{t-1}\right]$ is computed. Finally, they approximate the joint state-control distribution $p\left(\tilde{\mathbf{x}}_{t-1}\right) = p\left(\mathbf{x}_{t-1}, \mathbf{u}_{t-1}\right)$ with a Gaussian with the correct mean and covariance.

Approximating the output predictive distribution, $p(\Lambda_t)$

We now simply report the reasoning followed in the paper for completeness of this report.

MEAN OF THE PREDICTION Recall that D is the dimension of $\mathbf{x}_t$. Then, following the law of iterated expectations and for target dimensions $a = 1, \ldots, D$, one has

$$\mu_{\Delta}^{a} = \mathbb{E}_{\tilde{\mathbf{x}}_{t-1}}\left[\mathbb{E}_f\left[f\left(\tilde{\mathbf{x}}_{t-1}\right) \mid \tilde{\mathbf{x}}_{t-1}\right]\right] = \mathbb{E}_{\tilde{\mathbf{x}}_{t-1}}\left[m_f\left(\tilde{\mathbf{x}}_{t-1}\right)\right] \tag{11}$$

$$= \int m_f\left(\tilde{\mathbf{x}}_{t-1}\right) \mathcal{N}\left(\tilde{\mathbf{x}}_{t-1} \mid \tilde{\mu}_{t-1}, \tilde{\mathbf{\Sigma}}_{t-1}\right) \mathrm{d}\tilde{\mathbf{x}}_{t-1} \tag{12}$$

$$= \beta_a^{\top} \mathbf{q}_a \tag{13}$$

with $\beta_a = \left(\mathbf{K}_a + \sigma_{\varepsilon_a}^2\right)^{-1} \mathbf{y}_a$ and $\mathbf{q}_a = \left[q_{a_1}, \ldots, q_{a_n}\right]^{\top}$. With m_f given in equation(3), the entries of $\mathbf{q}_a \in \mathbb{R}^n$ are

$$q_{a_i} = \int k_a\left(\tilde{\mathbf{x}}_i, \tilde{\mathbf{x}}_{t-1}\right) \mathcal{N}\left(\tilde{\mathbf{x}}_{t-1} \mid \tilde{\mu}_{t-1}, \tilde{\mathbf{\Sigma}}_{t-1}\right) \mathrm{d}\tilde{\mathbf{x}}_{t-1}$$

$$= \frac{\alpha_a^2}{\sqrt{\left|\tilde{\mathbf{\Sigma}}_{t-1} \mathbf{\Lambda}_a^{-1} + \mathbf{I}\right|}} \exp\left(-\frac{1}{2} \nu_i^{\top}\left(\tilde{\mathbf{\Sigma}}_{t-1} + \mathbf{\Lambda}_a\right)^{-1} \nu_i\right),$$

$$\nu_i := \left(\tilde{\mathbf{x}}_i - \tilde{\mu}_{t-1}\right).$$

Where ν_i is the difference between the training input $\tilde{\mathbf{x}}_i$ and the mean of the "test" input distribution $p\left(\mathbf{x}_{t-1}, \mathbf{u}_{t-1}\right)$.

COVARIANCE MATRIX OF THE PREDICTION To compute the predictive covariance matrix $\mathbf{\Sigma}_{\Delta} \in \mathrm{R}^{D \times D}$ the authors distinguish between diagonal elements and off-diagonal elements. Using the law of iterated variances, one obtains for target dimensions $a, b = 1, \ldots, D$

$$\sigma_{aa}^2 = \mathbb{E}_{\tilde{\mathbf{x}}_{t-1}}\left[\operatorname{var}_f\left[\Delta_a \mid \tilde{\mathbf{x}}_{t-1}\right]\right] + \mathbb{E}_{f, \tilde{\mathbf{x}}_{t-1}}\left[\Delta_a^2\right] - \left(\mu_{\Delta}^{a}\right)^2 \tag{14}$$

$$\sigma_{ab}^2 = \mathbb{E}_{f, \tilde{\mathbf{x}}_{t-1}}\left[\Delta_a \Delta_b\right] - \mu_{\Delta}^{a} \mu_{\Delta}^{b}, \quad a \neq b \tag{15}$$

respectively, where μ_{Δ}^{a} is known from equation (13). The off-diagonal terms do not contain the additional term $\mathbb{E}_{\tilde{\mathbf{x}}_{t-1}}\left[\operatorname{cov}_f\left[\Delta_a, \Delta_b \mid \tilde{\mathbf{x}}_{t-1}\right]\right]$ because of the conditional independence assumption of the GP models: different target dimensions do not covary for given $\tilde{\mathbf{x}}_{t-1}$.

First, the terms that are common to both the diagonal and off-diagonal entries are computed.

With the Gaussian approximation $\mathcal{N}\left(\tilde{\mathbf{x}}_{t-1} \mid \tilde{\mu}_{t-1}, \tilde{\mathbf{\Sigma}}_{t-1}\right)$ of $p\left(\tilde{\mathbf{x}}_{t-1}\right)$ and the law of iterated expectations and using (3)

$$\begin{aligned}\mathbb{E}_{f,\tilde{\mathbf{x}}_{t-1}}\left[\Delta_a \Delta_b\right] &= \mathbb{E}_{\tilde{\mathbf{x}}_{t-1}}\left[\mathbb{E}_f\left[\Delta_a \mid \tilde{\mathbf{x}}_{t-1}\right] \mathbb{E}_f\left[\Delta_b \mid \tilde{\mathbf{x}}_{t-1}\right]\right] \\ &= \int m_f^a\left(\tilde{\mathbf{x}}_{t-1}\right) m_f^b\left(\tilde{\mathbf{x}}_{t-1}\right) p\left(\tilde{\mathbf{x}}_{t-1}\right) \mathrm{d}\tilde{\mathbf{x}}_{t-1}\end{aligned}$$

due to the conditional independence of Δ_a and Δ_b given $\tilde{\mathbf{x}}_{t-1}$. Using again the definition of the mean function m_f in (3), one obtains

$$\begin{aligned}&\mathbb{E}_{f,\tilde{\mathbf{x}}_{t-1}}\left[\Delta_a \Delta_b\right] = \beta_a^{\top} \mathbf{Q} \beta_b \\ &\mathbf{Q} := \int k_a\left(\tilde{\mathbf{X}}, \tilde{\mathbf{x}}_{t-1}\right) k_b\left(\tilde{\mathbf{X}}, \tilde{\mathbf{x}}_{t-1}\right)^{\top} p\left(\tilde{\mathbf{x}}_{t-1}\right) \mathrm{d}\tilde{\mathbf{x}}_{t-1}\end{aligned}$$

Now, one can make use of standard results from Gaussian multiplications and integration, and obtain the entries Q_{ij} of $\mathbf{Q} \in \mathbb{R}^{n \times n}$ $Q_{ij} = \frac{k_a(\tilde{\mathbf{x}}_i, \tilde{\mu}_{t-1}) k_b(\tilde{\mathbf{x}}_j, \tilde{\mu}_{t-1})}{\sqrt{|\mathbf{R}|}} \exp\left(\frac{1}{2} \mathbf{z}_{ij}^{\top} \mathbf{R}^{-1} \tilde{\mathbf{\Sigma}}_{t-1} \mathbf{z}_{ij}\right)$, where we defined $\mathbf{R} := \tilde{\mathbf{\Sigma}}_{t-1}\left(\mathbf{\Lambda}_a^{-1} + \mathbf{\Lambda}_b^{-1}\right) + \mathbf{I}$ and $\mathbf{z}_{ij} := \mathbf{\Lambda}_a^{-1} \nu_i + \mathbf{\Lambda}_b^{-1} \nu_j$. Hence, the off-diagonal entries of $\mathbf{\Sigma}_{\Delta}$ are fully determined by the other results we just obtained.

Finally, from equation (14), we see that the diagonal entries of $\mathbf{\Sigma}_{\Delta}$ contain an additional term

$$\mathbb{E}_{\tilde{\mathbf{x}}_{t-1}}\left[\mathrm{var}_f\left[\Delta_a \mid \tilde{\mathbf{x}}_{t-1}\right]\right] = \alpha_a^2 - \mathrm{tr}\left(\left(\mathbf{K}_a + \sigma_{\varepsilon_a}^2 \mathbf{I}\right)^{-1} \mathbf{Q}\right)$$

Further comments on code

In order to run the code for the examples, we need to execute the following operations. First, OpenAI Gym 0.15.3 and mujoco-py 2.0.2.7. have to be installed manually. Keep in mind that the creators of Mujoco have only recently changed the usage policy. These changes make those environments more accessible, but the installation trickier in a sense. Then, the PILCO package can be installed by running:

```
git clone https://github.com/nrontsis/PILCO && cd PILCO
python setup.py develop
```

The attached code can be used to run some experiments in the MountainCar OpenAI Gym environment.

Unfortunately, recent changes in the Tensorflow package and the the OpenAI Gym environments create some incompatibilities that create serious issues when running code for PILCO. Fixing those problems would require quite some time. Few "wins" on my side were the following:

- I managed to download Mujoco and OpenAI Gym on Windows when the installation for Windows was not available (which was until two months ago, when I started reading the material for this project). New changes in the following weeks made this effort useless.
- I recently managed to have Mujoco running using the administrator account on my computer.

I believe that new adjustments to the packages will be made in the next months that will make fixing the issues more feasible. On my side, if I will have the chance to do research on related topics, I plan to delve more into the mechanics of those tools. As said previously, this is a pitfall of my project. On the other side, I guess this is a lesson in terms of how research doesn't always go as smooth as we would hope it to go.

```
#import relevant packages
import numpy as np
import gym
from gpflow import set_trainable
np.random.seed(0)


#import models from reference code
from pilco.models import PILCO #import main structure
from pilco.controllers import RbfController #import selected controller
from pilco.rewards import ExponentialReward #import type of reward used for the
↪  cost function
from utils import rollout, Normalised_Env

SUBS = 5
T = 25
env = gym.make('MountainCarContinuous-v0')

# Initial random rollouts to generate a dataset
X1,Y1, _, _ = rollout(env=env, pilco=None, random=True, timesteps=T, SUBS=SUBS,
↪  render=True)
for i in range(1,5):
    X1_, Y1_,_,_ = rollout(env=env, pilco=None, random=True,  timesteps=T,
    ↪  SUBS=SUBS, render=True)
    X1 = np.vstack((X1, X1_))
    Y1 = np.vstack((Y1, Y1_))
env.close()

# A new (normalised) environment is created
env = Normalised_Env('MountainCarContinuous-v0', np.mean(X1[:,:2],0),
↪  np.std(X1[:,:2], 0))

# results of previous rollouts are normalised and used as initial dataset
X = np.zeros(X1.shape)
X[:, :2] = np.divide(X1[:, :2] - np.mean(X1[:,:2],0), np.std(X1[:,:2], 0))
X[:, 2] = X1[:,-1] # control inputs are not normalised
Y = np.divide(Y1 , np.std(X1[:,:2], 0))

#various setup
#dimension set up
state_dim = Y.shape[1]
control_dim = X.shape[1] - state_dim
```

```
#prior setup for GP
m_init =  np.transpose(X[0,:-1,None])
S_init =  0.5 * np.eye(state_dim)
#controller setup
controller = RbfController(state_dim=state_dim, control_dim=control_dim,
↪  num_basis_functions=25)
#cost/reward type setup
R = ExponentialReward(state_dim=state_dim,
                      t=np.divide([0.5,0.0] - env.m, env.std),
                      W=np.diag([0.5,0.1])
                     )

pilco = PILCO((X, Y), controller=controller, horizon=T, reward=R, m_init=m_init,
↪  S_init=S_init)

best_r = 0
all_Rs = np.zeros((X.shape[0], 1))
for i in range(len(all_Rs)):
    all_Rs[i,0] = R.compute_reward(X[i,None,:-1], 0.001 * np.eye(state_dim))[0]

ep_rewards = np.zeros((len(X)//T,1))

for i in range(len(ep_rewards)):
    ep_rewards[i] = sum(all_Rs[i * T: i*T + T])

for model in pilco.mgpr.models:
    model.likelihood.variance.assign(0.05)
    set_trainable(model.likelihood.variance, False)

r_new = np.zeros((T, 1))
for rollouts in range(5):
    pilco.optimize_models()
    pilco.optimize_policy(maxiter=100, restarts=3)
    import pdb; pdb.set_trace()
    X_new, Y_new,_,_ = rollout(env=env, pilco=pilco, timesteps=T, SUBS=SUBS,
    ↪  render=True)

    for i in range(len(X_new)):
            r_new[:, 0] = R.compute_reward(X_new[i,None,:-1], 0.001 *
            ↪  np.eye(state_dim))[0]
    total_r = sum(r_new)
    _, _, r = pilco.predict(m_init, S_init, T)
```

```
77
78      print("Total ", total_r, " Predicted: ", r)
79      X = np.vstack((X, X_new)); Y = np.vstack((Y, Y_new));
80      all_Rs = np.vstack((all_Rs, r_new)); ep_rewards = np.vstack((ep_rewards,
        ↪  np.reshape(total_r,(1,1))))
81      pilco.mgpr.set_data((X, Y))
```