Control of Dynamical Systems via Deep Kernel Learning

Martin Anthony Bernard Tan

Master of Science Thesis

Delft Center for Systems and Control

Control of Dynamical Systems via Deep Kernel Learning

Master of Science Thesis

For the degree of Master of Science in Systems and Control at Delft University of Technology

Martin Anthony Bernard Tan

August 14, 2023

Faculty of Mechanical, Maritime and Materials Engineering (3mE) · Delft University of Technology

Delft University of Technology
Department of
Delft Center for Systems and Control (dcsc)

The undersigned hereby certify that they have read and recommend to the Faculty of Mechanical, Maritime and Materials Engineering (3mE) for acceptance a thesis entitled

Control of Dynamical Systems via Deep Kernel Learning

by

Martin Anthony Bernard Tan

in partial fulfillment of the requirements for the degree of

Master of Science Systems and Control

Dated: August 14, 2023

Supervisor(s): prof.dr. L.Laurenti

Reader(s): prof.dr. M.Kok

prof.dr. M.A. Sharifi Kolarijani

Abstract

In the field of Systems and Control, optimal control problem-solving for complex systems is a core task. The development of accurate mathematical models to represent these systems' dynamics is often difficult. This complexity comes from potential uncertainties, complex non-linearities, or unknown factors that might affect the system. Because of these challenges, there is a need for methods that can understand the dynamics using available data and control strategies that can work with such models without relying too much on expert knowledge or task-specific insights. These methods are essential for creating efficient and reliable solutions in a wide variety of applications within the discipline.

The need for models that do not require expert knowledge has spurred the interest in applying machine learning methods to control problems. Probabilistic Inference for Learning COntrol (PILCO) is a model-based Reinforcement Learning (RL) algorithm known for its probabilistic approach to model-based RL. By employing Gaussian Process (GP) dynamics models, PILCO integrates uncertainties into its learning process, allowing it to derive control policies from limited data. PILCO's use of the Squared Exponential (SE) kernel in its GP can restrict the learning capacity. Especially in higher-dimensional spaces, due to the SE kernel's inherent smoothness assumption that might not capture complex or non-smooth dynamics effectively. The algorithm's reliance on moment matching for approximating posterior distributions introduces another weakness, which can lead to inaccuracies in non-Gaussian or multi-modal contexts. These shortcomings may limit PILCO's efficiency and scalability in more complex, higher-dimensional tasks or in situations where the underlying dynamics are not well-captured by the chosen kernel and approximation methods.

This thesis introduces Deep Kernel PILCO (DKL PILCO), a novel framework that uses Deep Kernel Learning (DKL) for learning the dynamics, and the Unscented Transform (UT) to propagate the uncertainty. The effectiveness of this approach is demonstrated across various tasks, highlighting the potential of DKL and UT to enhance the scalability and efficiency of model-based RL methods such as PILCO, making it a promising candidate for real-world control applications.

Table of Contents

Acknowledgements

I would like to express my sincere gratitude to my supervisor prof.dr. L.Laurenti, whose guidance, support, and expertise have been invaluable throughout this research. Furthermore, I want to thank the committee for taking the time to evaluate my thesis.

My heartfelt appreciation goes to my family, friends, parents, brother, and girlfriend, Iris, who have provided constant support, encouragement, and love. Your belief in me has been my strength in all my pursuits.

Also, I would like to express my gratitude, to my colleagues at Oliver Wyman Actuarial, for the insightful experiences that helped during my thesis, fun distractions, support, and inspiration.

Lastly, I would like to thank my friends in the "iO?" group, our own thesis support group, for the needed motivations, distractions, and inspiration during the last leg of my thesis.

Delft, University of Technology
August 14, 2023

Martin Anthony Bernard Tan

Chapter 1

Introduction

Consider an autonomous vehicle assigned the task of executing a parking maneuver. The objective is to learn a sequence of actions to achieve this task successfully. Optimally, this sequence would be the most efficient, whether this efficiency is defined by fuel consumption, completion time, or another relevant metric. Such a problem is also called an optimal control problem within systems and control.

In the framework of optimal control, we would like to have a formal description (in the form of a mathematical model) of how the system (e.g., an autonomous car) responds (or evolves) from a current state to a future state, usually such a description is given in the form of an ordinary differential equation [57], such as:

$$\begin{aligned} \dot{\mathbf{x}}(t) &= \mathbf{h}(\mathbf{x}(t), \boldsymbol{u}(x,t)) \quad (t > 0) \\ \mathbf{x}(0) &= \mathbf{x_0} \end{aligned} \tag{1-1}$$

where $\mathbf{x_0} \in \mathbb{R}^n$ denotes the initial state of the dynamical system, $\mathbf{h} : \mathbb{R}^n \times A \to \mathbb{R}^n$ dictates the evolution of the dynamical system and $\boldsymbol{u} : [0, \infty) \to A$ is the control signal of the system. The optimal control solution will be a function $\boldsymbol{u}(x,t)$, which could depend on the system's time and state.

Challenges in Control

In many of the standard control algorithms, we assume that the model of the system (its dynamics, the environment, and the costs) is fully known, implicitly assuming that expert knowledge is available. However, in reality, we can have some uncertainty in the observations, noise, or the model itself can influence the system's evolution [20]. Standard stochastic control techniques can partly address the uncertainty, e.g., linear-quadratic-Gaussian control [53] or dynamic programming [9]. However, both methods still require a model of the system that is known and often depends on task-specific prior knowledge. Unfortunately, in practice, due to the ever-increasing complexity of physical systems (e.g., self-driving cars), it is increasingly difficult and expensive to have an accurate model of the system [50].

Recent Advances in Using ML for Control

Recent advances in Machine Learning (ML) have addressed several challenges in control theory, especially in the control of systems with uncertainty [20] [50] [54] [106]. Based on input and output data, ML models can automatically learn the relationship between the input and output and make predictions, even when the relationship is highly non-linear and complex, without relying on expert knowledge [33]. Within the field of ML, Reinforcement Learning (RL) is specifically concerned with learning control tasks in unknown environments through interaction [94]. RL can be subdivided into model-free and model-based algorithms. Model-free algorithms, such as Deep Q Networks (DQN) or Twin Delayed Deep Deterministic Policy Gradients (TD3), learn optimal policies by directly interacting with the environment without constructing an explicit model of the environment's dynamics [95]. A prominent example of this approach is AlphaGo, which demonstrated its capabilities by defeating the world champion in Go [91].

Limitations in Existing RL Methods

Board games and video games provide ideal environments where exhaustive exploration and navigation can be carried out with minimal expenditure, constrained only by the availability of computational resources. Nevertheless, this leads us to a fundamental limitation inherent in current state-of-the-art reinforcement learning (RL) algorithms: the necessity for extensive interactions with the environment to successfully achieve their predetermined goals [51] [20] [94]. For instance, many RL methods dealing with low-dimensional state spaces and relatively simple dynamics demand thousands of trials for learning[23]. This data inefficiency makes RL impossible to use in applications where data generation is costly or involves devices susceptible to wear and tear.

However, model-based RL offers an attractive framework for synthesizing controllers due to its potential for data efficiency. In the simulation, an RL agent can use learned dynamics models to search for effective controllers, which can minimize the need for costly trials on real robots [71]. Nonetheless, model-based RL algorithms can suffer from model bias, where the learned model might significantly deviate from the actual system, but the agent assumes the model to be entirely correct [107]. Bayesian models can be helpful in this situation. Instead of requiring an accurate model, the agent can maintain a distribution over candidate models that are compatible with its experience, reducing the risk of model bias [19].

PILCO: A Response to RL Challenges and Its Own Limitations

Probabilistic Inference for Learning COntrol (PILCO) employs Gaussian Process (GP) models for approximating one-step dynamics and utilizes Radial Basis Function (RBF) networks as policies [20]. GPs are a class of probabilistic models that allow for a non-parametric, Bayesian approach to regression and classification problems, providing both predictions and a measure of uncertainty [83]. PILCO demonstrates remarkable performance with limited data in both simulated tasks and real-world robotic applications. Nonetheless, PILCO's use of the Squared Exponential (SE) kernel in its GP can restrict the learning capacity, and it might not capture complex dynamics effectively. Furthermore, PILCO's use of moment matching to propagate uncertainty can be computationally expensive and lead to approximation errors when not using an SE kernel [31]. In addition, its computational complexity scales $O\left(Dn^3\right)$ for model training and $O\left(D^3n^2\right)$ for its long-term predictions, where n is the dataset size and D is the dimensionality of the state space [20]. As a result, PILCO's applicability is only limited to scenarios with small datasets and low-dimensional state spaces [31].

Recent developments in sparse GPs have addressed these issues [96], one of them being Deep Kernel Learning (DKL). DKL combines a Neural Network (NN) with a GP and uses structure exploiting algebra with inducing inputs for a scalable framework [104]. DKL can model complex and non-linear relationships in the data. The NN part can capture complicated patterns, while the GP provides a principled way of handling uncertainty. DKL allows for end-to-end training using gradient-based optimization methods, taking advantage of both the expressive power of deep learning and the probabilistic modeling of GPs.

Research Question

Given the complex challenges in control systems and the limitations of existing approaches, there is a pressing need to explore innovative techniques to control unknown systems. Recent advancements in DKL offer promising avenues, yet its application within the context of model-based RL remains largely unexplored. This gap motivates the central research question of this thesis:

How can Deep Kernel Learning be used to control unknown systems?

The significance of this research question lies in its potential to pioneer new pathways in RL, offering a sophisticated tool for controlling unknown systems with uncertainty.

Main Contributions

In this thesis, we introduce Deep Kernel PILCO (DKL PILCO), a novel framework based on PILCO that uses DKL for the dynamics modeling and the Unscented Transform (UT) for the propagation of uncertainty.

The core alteration in DKL PILCO is the replacement of the traditional SE kernel with a deep kernel. This deep kernel leverages the expressive power of NNs, offering a more flexible and adaptive representation of the underlying data. By handling complex relationships and non-linear patterns more naturally, the deep kernel improves the accuracy and robustness of the model.

Furthermore, DKL PILCO employs the UT instead of moment matching to propagate the uncertainty. UT provides a deterministic method to estimate the mean and covariance of a nonlinear transformation of a random variable. This change offers several advantages: increased accuracy and robustness. UT can handle nonlinear transformations with greater accuracy, reducing the approximation errors that can occur in moment matching. Moreover, by utilizing a deterministic sampling technique, UT can provide more stable predictions, minimizing sensitivity to noise in the data.

In contrast to PILCO, which relies on hardcoded gradients, DKL PILCO employs Automatic Differentiation (auto-diff). This approach enhances the framework's flexibility by enabling it to compute derivatives programmatically. Auto-diff simplifies the implementation and potentially improves the robustness and efficiency of the gradient computations, reducing the risk of human error encountered in hardcoded gradients.

DKL PILCO enhances the predictive capabilities, already seen in PILCO, and introduces a higher degree of flexibility and robustness. This framework represents a thoughtful integration of modern techniques with established principles, aiming to push the boundaries of predictive modeling.

We show that DKL PILCO is able to learn a successful policy for the mountain car problem using, on average, almost 50% fewer interactions than PILCO, demonstrating improved data efficiency. Furthermore, the policy it learns is also more robust against action, observation, and initial state disturbances than PILCO. In addition, the computational time of its multi-step ahead predictions is, on average, about an order of magnitude faster than PILCO, regardless of the prediction horizon length or dimensionality. Furthermore, its predictive power is further reinforced, achieving an Mean Squared Error (MSE) at least 10 times lower than PILCO on both the mountain car and swimmer problem and achieving a more precise and accurate multi-step ahead prediction for both low and high dimensions.

To summarize, this thesis makes the following main contributions:

- We introduce a novel framework based on PILCO that uses DKL and the UT.
- We implement the use of auto-diff instead of hardcoded gradients, resulting in a simplified implementation.
- DKL PILCO's performance and robustness are evaluated on multiple tasks, and the empirical results show that it surpasses PILCO in aspects such as data efficiency, policy robustness, numerical stability, computational complexity, and predictive accuracy.

Outline

This thesis commences with Chapter 2, laying out the groundwork by delving into the foundational concepts used, such as GPs, NNs, and PILCO. Chapter 3 introduces the DKL PILCO framework and outlines the modifications made to the original PILCO framework. Chapter 4 presents and analyzes the results of applying DKL PILCO to the mountain car and swimmer problems, comparing these outcomes with those derived from PILCO. The thesis culminates with Chapter 5, which summarizes the findings and outlines potential areas for future research.

Chapter 2

Preliminaries

This chapter, "Preliminaries", serves as a foundation for our framework Deep Kernel Learning (DKL), primarily focusing on three fundamental concepts: Gaussian Process (GP)s, Neural Network (NN)s, and Probabilistic Inference for Learning COntrol (PILCO).

We begin by introducing NNs, a cornerstone of contemporary Machine Learning (ML) techniques inspired by the structure and function of the human brain. These models, known for their capacity to learn from data and capture complex non-linear relationships, have widespread applications across various fields, ranging from image recognition to language processing [85] [91].

Next, we turn our attention to GPs, a type of statistical model that provides a robust framework for representing uncertainty [83]. These processes are crucial for making predictions in scenarios where data might be limited or noisy and will play a pivotal role in understanding the mechanisms behind PILCO.

Finally, we delve into PILCO, a model-based Reinforcement Learning (RL) algorithm. With its roots in GPs, PILCO efficiently manages uncertainties related to system modeling and control, contributing to the design of effective and efficient control strategies [20].

This chapter will serve to establish a strong foundational understanding of these three components, preparing the reader for the subsequent in-depth exploration of their integration and application.

2-1 Neural Networks

Nowadays, the ML landscape offers exciting capabilities such as object detection [85], speech recognition [18], and learning complex games (such as beating the champion of Go [91]). At the very core of these techniques usually lies NNs. NNs draw inspiration from biology by having the same rough model of how the human brain is structured.

One of the simplest forms of NN architectures, the Perceptron, was invented by Frank Rosenblatt in 1957 [87]. Perceptrons consist of a Threshold Logic Unit (TLU). The TLU takes several inputs (one of which is a bias), each with an associated weight, and yields a single output. It calculates the output by computing a weighted sum of its inputs ($z = w_1x_1 + w_2x_2 + \cdots + w_nx_n = \mathbf{xw}$) and then applying a (non-linear) activation function on the sum ($h_{\mathrm{w}}(\mathbf{x}) = \phi(z)$). Unfortunately, one of the limitations of perceptrons is that they can only classify linearly separable sets (resulting in difficulty with XOR classification problems) [33]. However, it was found that one could eliminate some of these limitations by stacking multiple perceptrons. The result is called a Multi-Layer Perceptron (MLP), and it can deal with linearly inseparable sets [33].

An MLP, also known as a Feedforward Neural Network (FNN), consists of an input layer, multiple hidden layers composed of multiple TLUs, and one final layer (the output layer). Furthermore, every layer is fully connected to the next layer. As can be seen in Figure 2-1, the information flow is in a single direction only (from the input to the output), hence the name FNN.

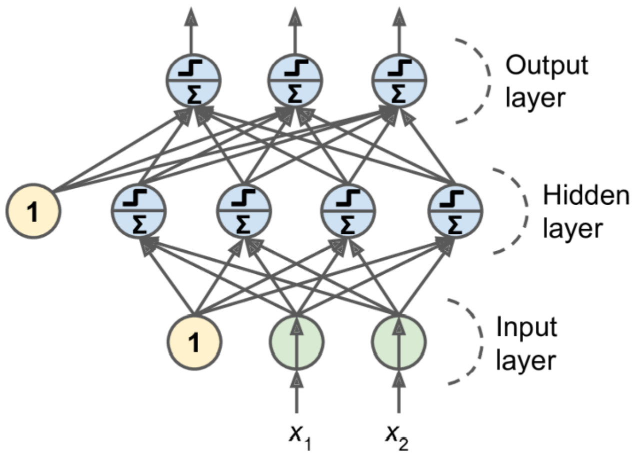


Figure 2-1: Schematic of an MLP [33]

Suppose an MLP with k hidden layers, a d dimensional input layer, and an o dimensional output layer. Then a matrix W_1 can be constructed, containing the weights between the input layer and the first hidden layer of size $d \times p_1$. For any rth hidden layer to the next $(r+1)$th hidden layer, we can put the weights between the layers in a $p_r \times p_{r+1}$ denoted by W_r. The final weight matrix, W_{k+1}, is between the final hidden layer and the output layer and has size $p_k \times o$. If the network is an FNN, then the following recursive equations can be used to calculate the outputs from the input vector $\bar{x}$ [1]:

$$\begin{aligned}
&\bar{h}_1 = \phi\left(W_1^T \bar{x}\right) && && \text{[Input to Hidden Layer]} \\
&\bar{h}_{p+1} = \phi\left(W_{p+1}^T \bar{h}_p\right) && \forall p \in \{1 \dots k-1\} && \text{[Hidden to Hidden Layer]} \\
&\bar{o} = \phi\left(W_{k+1}^T \bar{h}_k\right) && && \text{[Hidden to Output Layer]}
\end{aligned} \tag{2-1}$$

2-1-1 Activation Functions

Activation functions are an essential part of NNs. An activation function dictates the magnitude of the activation of a neuron for a given input. In other words, it decides whether the neuron's input to the whole network is essential or not [33]. One of the key reasons to use an activation function is that it adds non-linearity to the neural network. Through this non-linearity, the network can approximate any function arbitrarily well. This property is called the universal approximation theorem and was initially proposed by Cybenko [17], and is stated as follows:

Theorem 1. *Let σ be any continuous sigmoidal function. Then finite sums of the form:*

$$G(x) = \sum_{j=1}^{N} \alpha_j \sigma\left(y_j^{\mathrm{T}} x + \theta_j\right) \tag{2-2}$$

are dense in $C\left(I_n\right)$. In other words, given any $f \in C\left(I_n\right)$, there is a sum, $G(x)$, of the above form, for which:

$$|G(x) - f(x)| < \varepsilon \quad \textit{for all} \quad x \in I_n \tag{2-3}$$

where $y_j \in \mathbb{R}^n$, $x \in \mathbb{R}^n$, $\alpha_j, \theta \in \mathbb{R}$, I_n is the n-dimensional unit cube and $C\left(I_n\right)$ is the space of continuous functions on I_n. This means that any continuous function can be approximated arbitrarily well by an MLP, with at least one hidden layer and a finite number of neurons. This property, or variations of it, has also been proven for other activation functions [45].

Non-linear activation functions allow the stacking of multiple layers since the output is now a non-linear combination of the fed input [33]. Furthermore, its derivative is now related to its input and can be used in backpropagation. Many non-linear activation functions exist, e.g., Rectified Linear Unit (ReLU), sigmoid, and hyperbolic tangent.

The ReLU activation function is a widely used function commonly used for regression tasks [33]. These functions behave almost similarly to linear functions, except for the region where the function is rectified and outputs zero across half its domain. The following equation describes it:

$$\phi(x) = \max(0, x) \tag{2-4}$$

The function is differentiable except at $x = 0$. Furthermore, contrary to, e.g., sigmoid functions, the function is not bounded from above and hence can handle large positive activation of the neurons. As can be seen in Figure 2-2, for the region, the function is differentiable, its derivative has a constant value, and its second derivative is 0. This can be favorable in gradient descent, where its direction can be advantageous and cause faster convergence [36]. Since the ReLU function will not be activated for inputs smaller than 0, not all the neurons in a network will be activated at the same time [28]. This makes ReLUs more computationally

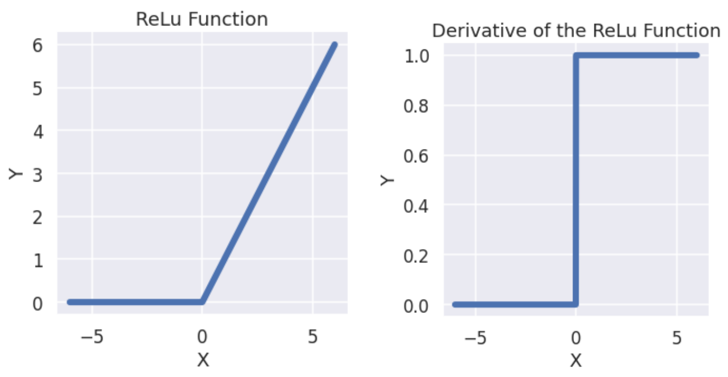


Figure 2-2: The ReLU function and its derivative

efficient than the sigmoid and hyperbolic tangent functions.
However, vanilla ReLU functions do have a significant drawback. Namely, the dying ReLU problem, where the neurons are fed a negative input, and their gradients will become immediately zero. This means the neuron will not be updating its weights and biases and can result in dead neurons, which will never learn [36]. Several modifications to the ReLU function exist to alleviate the problem of the dying ReLU: leaky ReLU, parametric ReLU, ELU, and SELU [33] [61].

2-1-2 Training a Neural Network

The goal of training an NN is to identify the network weights; given the training input, the predicted value of the network matches the desired output (for that particular training example) [33]. This can be done for a whole dataset containing multiple training examples. We can then define a single error function (or metric, also known as the loss function denoted by $\mathcal{L}$) for the whole dataset. Assume a dataset of $\mathcal{D} = \{(\mathrm{x}_i, y_i) \mid i = 1, \ldots, n\}$, then the Mean Squared Error (MSE) is defined as:

$$\mathcal{L} = \frac{1}{n} \sum_{i=1}^{n} (y_i - \hat{y}_i)^2 \tag{2-5}$$

where y_i indicates the i-th training example, $\hat{y}_i$ the i-th prediction of the network and n the number of training examples. The goal is to find the weights that minimize the error function's value. The set of values which minimizes the objective function is often denoted as $\boldsymbol{x}^* = \arg\min f(\boldsymbol{x})$.

The log-likelihood can also be used as a loss function. Using the same dataset as defined previously, then the likelihood of the dataset is given by:

$$\mathcal{L} = \prod_{i=1}^{n} \frac{1}{\sqrt{2\pi}\sigma} \exp\left(\frac{-(y_i - \hat{y}_i)^2}{2\sigma^2} \right) \tag{2-6}$$

Taking the logarithm of the likelihood results in the following:

$$\log \mathcal{L} = \sum_{i=1}^{n} \frac{-(y_i - \hat{y}_i)^2}{2\sigma^2} - n \log\left(\sqrt{2\pi}\sigma\right) \tag{2-7}$$

Notice that the second term in Equation 2-7 is a constant. So then, maximizing the log-likelihood must be done through minimizing $(y_i - \hat{y}_i)^2$, which is the same squared error term as in the MSE. Hence, minimizing the MSE is equivalent to maximizing the log-likelihood of the data [10].

2-1-3 Gradient Descent

One of the most commonly used algorithms to perform this minimization is gradient descent [28][35]. Gradient descent is a numerical method in which an initial solution is guessed and gradually refined by iteratively tweaking the parameters. Gradient descent determines the local gradient of the error function with respect to the parameter vector $\boldsymbol{\theta}$, which contains the network weights and biases [33].

The gradient of a function tells us how much the function value changes given a small change in its input variable and is commonly denoted (for the univariate case) as $f', f'(x), \frac{df}{dx}$. For the multivariate case, with a vector of θ parameters, we take the derivative with respect to one of the parameters θ_i while keeping the rest of the parameters constant; we call this the partial derivative. We can then define a gradient vector of the function, which contains all partial derivatives of the function with respect to each parameter, denoted as:

$$\nabla_{\boldsymbol{\theta}} \, \mathrm{f}(\boldsymbol{\theta}) = \begin{pmatrix} \frac{\partial}{\partial \theta_0} \mathrm{f}(\boldsymbol{\theta}) \\ \frac{\partial}{\partial \theta_1} \mathrm{f}(\boldsymbol{\theta}) \\ \vdots \\ \frac{\partial}{\partial \theta_n} \mathrm{f}(\boldsymbol{\theta}) \end{pmatrix} \tag{2-8}$$

The gradient vector points in the direction of the most major change, and its magnitude indicates the size of this change.

Gradient descent then goes into the error function's descending direction (the negative gradient). We can do this iteratively, making small adjustments to the parameter vector $\boldsymbol{\theta}$ until the gradient is 0 and the error function converges to a (local) minimum.

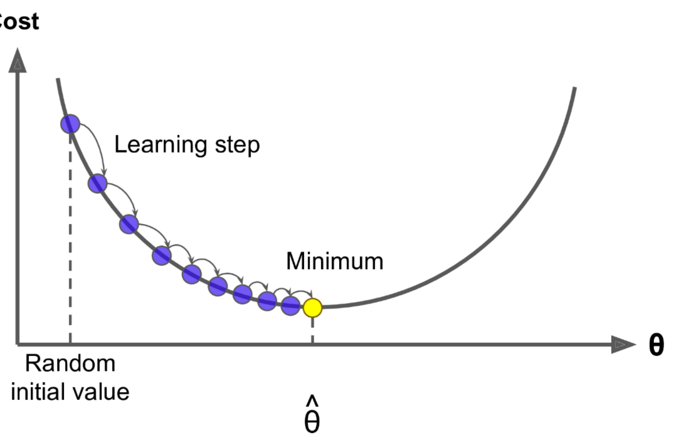


Figure 2-3: An overview of gradient descent [33]

Notice how the steps towards the minimum are not constant in Figure 2-3. Each new update of the parameter vector is determined by [28]:

$$\boldsymbol{\theta}_{n+1} = \boldsymbol{\theta}_n - \eta \nabla f(\boldsymbol{\theta}_n) \tag{2-9}$$

Where η is our learning rate. Both the learning rate and the derivative determine the step size. We can see in Figure 2-3 that the closer we get to the minimum, the smaller the step size due to the derivative approaching 0. Care should be taken with regard to selecting a learning rate. Picking a learning rate too large can result in an overshoot and the solution failing to converge. However, picking a small learning rate can result in long running times.

There are three commonly used types of gradient descent, and their crucial difference is in the amount of data they use:

Batch Gradient Descent In batch gradient descent, we calculate the gradient vector in Equation 2-8 over the full training set $\boldsymbol{X}$. The algorithm does this for each iteration step and only updates the model after evaluating the whole training set. As a result, the error gradient and convergence can be quite stable [33]. However, this can also result in slow convergence rates on vast datasets [33].

Stochastic Gradient Descent Stochastic Gradient Descent (SGD) sits at the other end of the spectrum. It picks a training example at random and computes the gradient vector, based only on that single example, at every step. As a result, the algorithm can be much faster (depending on the problem) compared to batch gradient descent and can scale well to vast datasets [33]. One of the downsides is that due to its stochastic nature, the error gradient and convergence can be pretty unstable [33]. Even close to the minimum, there can still be some erratic behavior where the value of the error function oscillates.

Mini-Batch Gradient Descent Mini-batch gradient descent sits somewhere in the middle between batch and SGD. The algorithm computes the gradient on a small random subset of the data at each step, called mini-batches. As a result, the algorithm's convergence can be less erratic compared to SGD while still being faster than batch gradient descent [33].

A comparison in convergence, displayed in parameter space, between the three methods is shown in Figure 2-4.

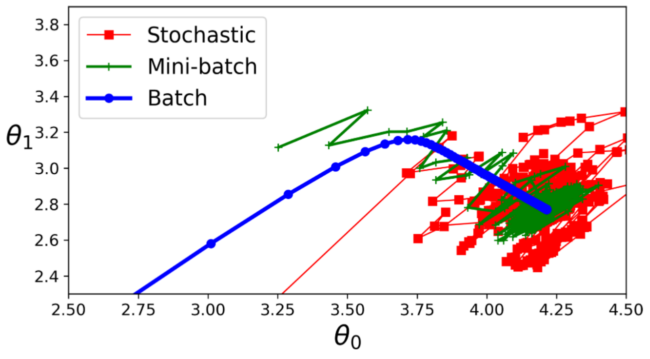


Figure 2-4: Gradient descent paths in a parameter space [33]

2-1-4 Backpropagation

The previously described algorithms can be straightforwardly applied to single-layer neural networks. However, in the case of MLPs, our error function becomes a composition function of the weights in the previous layers and is much more challenging to compute. This is where the backpropagation algorithm comes into play. It is a method to compute the gradient needed for gradient descent [28]. The backpropagation algorithm consists of two phases:

1. The forward pass, a training example, is fed into the NN in this pass. We can then compare the prediction of the network against the ground truth of the training example and compute our loss function accordingly.
2. The backward pass, our goal here is to compute the partial derivatives of the loss function with respect to all the weights in the layers. These derivatives are then used to update the weights in the network to make the prediction closer to the ground truth.

Chain Rule

In order to better understand backpropagation, we have to give a formal definition of the chain rule. Since the chain rule is heavily used in backpropagation, the more general case of the chain rule in which a dependent variable z is a differentiable function of multiple variables and each of which is, in turn, a differentiable function of multiple independent variables, can be stated as follows:

Theorem 2. ***The Chain Rule (General Version)*** *[93] Suppose that f is a differentiable function of the k functions $g_1, g_2, \ldots, g_k$ and each g_j is a function of the m variables $w_1, w_2, \ldots, w_m$. Then f is a function of $w_1, w_2, \ldots, w_m$ and*

$$\frac{\partial f\left(g_1(w), \ldots g_k(w)\right)}{\partial w} = \sum_{i=1}^{k} \frac{\partial f\left(g_1(w), \ldots g_k(w)\right)}{\partial g_i(w)} \cdot \frac{\partial g_i(w)}{\partial w} \tag{2-10}$$

for each $i = 1, 2, \ldots, m$

The Hadamard product

The notation used with backpropagation can become very index-heavy due to the many matrix-vector multiplications and vector additions. In order to make our lives easier and the notation less index-heavy, we can use the Hadamard product (also known as the Schur product). According to Horn et al. [44], the Hadamard product is defined as follows:

Definition 1. *Let two matrices A and B have the same dimensions m x n, the Hadamard product $A \odot B$ is the elementwise product of the two matrices. The resulting matrix is a matrix of the same dimensions as the operands, with its elements defined by:*

$$(A \odot B)_{ij} = (A)_{ij}(B)_{ij} \tag{2-11}$$

For matrices of different dimensions, the Hadamard product is undefined.

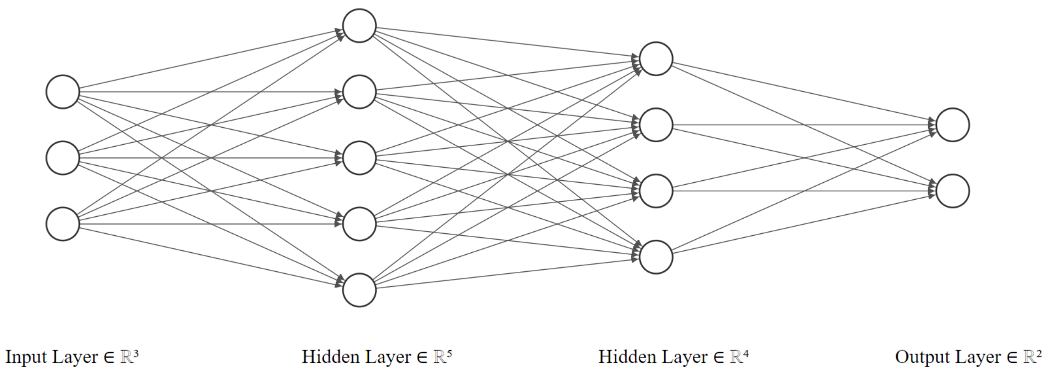


Figure 2-5: Our example neural network

Let us take an NN as depicted in Figure 2-5. The network consists of: an input layer $x_0 \in \mathbb{R}^3$, two hidden layers $x_1 \in \mathbb{R}^5$ and $x_2 \in \mathbb{R}^4$ respectively, and an output layer $x_3 \in \mathbb{R}^2$. We can then use w^l_{ij} to denote the weight between the j^{th} neuron in the $(l-1)^{\text{th}}$ layer and the i^{th} neuron in the l^{th} layer. An equivalent notation for the biases and activations can be used as well. We denote the bias and activation of the i^{th} neuron in the l^{th} by b^l_i and a^l_i, respectively.
With this in mind, we can then relate the activation of a^l_j to the activations in the previous layer by:

$$a^l_i = \phi\left(\sum_j w^l_{ij} a^{l-1}_j + b^l_i\right) \tag{2-12}$$

Where Φ denotes an arbitrary activation function, and the sum is taken over the j neurons in the previous layer. We can also collect all the weights into a weight matrix for each layer. This weight matrix W^l is defined by the entries w^l_{ij}, the entry in the ith row and jth column. Additionally, for each layer l we can also define a bias and activation vector $\mathbf{b^l}$ and $\mathbf{a^l}$, with its entries defined by b^l_j and a^l_j respectively. With this in mind, we can write the activations in the lth layer in vector form:

$$\mathbf{a^l} = \Phi\left(W^l \mathbf{a^{l-1}} + \mathbf{b^l}\right) \tag{2-13}$$

Finally, we can denote the weighted input for the activation function in the lth layer by:

$$\mathbf{z^l} \equiv W^l \mathbf{a^{l-1}} + \mathbf{b^l} \tag{2-14}$$

In backpropagation, our ultimate goal is to compute the partial derivatives of $\partial\mathcal{L}/\partial w^l_{ij}$ and $\partial\mathcal{L}/\partial b^l_i$, in order to reduce our loss function. Let us first define the error in the ith neuron in the lth layer, ε^l_i. The next step is to relate this error the partial derivatives $\partial\mathcal{L}/\partial w^l_{ij}$ and $\partial\mathcal{L}/\partial b^l_i$.
Now, suppose we perturb the weighted input to the ith neuron in the lth layer by Δz^l_i, as a result the neuron outputs $\phi\left(z^l_i + \Delta z^l_i\right)$. This perturbation causes the loss to change according to $\frac{\partial\mathcal{L}}{\partial z^l_i}\Delta z^l_i$. Imagine this perturbation Δz^l_i always acts opposite to the sign of $\frac{\partial\mathcal{L}}{\partial z^l_i}$, hence we can think $\frac{\partial\mathcal{L}}{\partial z^l_i}$ as a measure of the error in our neuron. More formally, let us define the error of the ith neuron in the lth layer by:

$$\varepsilon^l_i \equiv \frac{\partial\mathcal{L}}{\partial z^l_i} \tag{2-15}$$

The vector of errors corresponding to the lth layer can be denoted by ε^l.

Assumptions on the loss function

Before we move on, we need to have two assumptions on the loss function for the backpropagation algorithm to work. The first assumption is that we can write our loss function as an average over loss functions for each training example, that is:

$$\mathcal{L} = \frac{1}{n}\sum_x \mathcal{L}_x \tag{2-16}$$

where $\mathcal{L}$ is the average loss function and $\mathcal{L}x$ is the loss function for an individual training example. We need this assumption since the backpropagation algorithm gives us the partial derivatives: $\partial\mathcal{L}_x/\partial w$ and $\partial\mathcal{L}_x/\partial b$. We can then obtain $\partial\mathcal{L}/\partial w$ and $\partial\mathcal{L}/\partial b$ by taking the average over the training examples.

Secondly, we require that the loss function can be written as a function of the output activation layer.

The four equations for backpropagation

We will first start with the error for the output layer denoted by:

$$\varepsilon_i^L = \frac{\partial\mathcal{L}}{\partial a_i^L}\frac{\partial a_i^L}{\partial z_i^L} \tag{2-17}$$

This expression is found by applying the chain rule to Equation 2-15. The first term in Equation 2-17 is the partial derivative of the loss function with respect to the i^{th} output activation function. The second term is the partial derivative of the i^{th} output activation to its weighted input z_i^L. Finally, we can rewrite this equation in a matrix-based form:

$$\varepsilon^L = \nabla_{a^L}\mathcal{L} \odot \nabla_{z^L} a \tag{2-18}$$

where $\nabla_{a^l} C$ is the vector with its components equal to the partial derivatives $\frac{\partial\mathcal{L}}{\partial a_i^L}$ and $\nabla_{z^l} a$ is the vector with its components equal to the partial derivatives $\frac{\partial a_i^L}{\partial z_i^L}$.

The next step is to define the error ε^l in terms of the error of the next layer ε^{l+1}

$$\varepsilon^l = \left(\left(W^{l+1}\right)^T \varepsilon^{l+1}\right) \odot \nabla_{z^l} a \tag{2-19}$$

Looking at Equation 2-19, we can see that in order to compute the error in the previous layer, we first multiply the error with the weight matrix W^{l+1}. In doing so, we obtain the error of the output of the $l^t h$ layer. If we take the Hadamard product between this error and $\nabla_{z^l} a$, we can think of this as moving the error through the activation function in layer l. We obtain the error ε^l in terms of the weighted input to the l^{th} layer. With these two equations (Equation 2-17 and Equation 2-19), we can now compute ε^l for any layer. We first start with Equation 2-17 to compute ε^L. Then, we apply Equation 2-19 to compute ε^{L-1}, recursively applying Equation 2-19 we can work our way through the whole network.

Next up, the partial derivative of the loss function with respect to any bias in the network is as follows:

$$\frac{\partial \mathcal{L}}{\partial b^l} = \varepsilon^l \tag{2-20}$$

where b^l is the bias in the lth layer.

Finally, we need the partial derivative of the loss function with respect to any weight in the network.

$$\frac{\partial \mathcal{L}}{\partial w_{ij}^l} = a_j^{l-1} \varepsilon_i^l \tag{2-21}$$

This equation contains quantities that we already know how to compute.

The Algorithm

Having treated the four equations: Equation 2-17, Equation 2-19, Equation 2-20 and Equation 2-21, we can now summarize these steps in Algorithm 1. For this example, we will assume we have a dataset of $X = x_1, x_2, ..., x_m$ training examples. Furthermore, we will only show one step of the gradient descent learning step. An outer loop is also needed to step through multiple epochs of the training process.

Algorithm 1 Backpropagation for a single gradient descent learning step

```
Require: an input set of training examples $X = x_1, x_2, ..., x_m$
 1: for $x = x_1, x_2, ..., x_m$ do
 2:     Set the input activation layer $a^{x,0}$
 3:     for $l = 1, 2, 3, ...L$ do                                   ▷ Feedforward
 4:         Compute $z^{x,1} \equiv W^l \mathbf{a}^{x,1-1} + \mathbf{b}^1$
 5:         Compute $\mathbf{a}^{x,1} = \Phi\left(W^l \mathbf{a}^{x,1-1} + \mathbf{b}^1\right)$
 6:     end for
 7:     Compute the error of the output layer $\varepsilon^{x,L} = \nabla_{a^{x,L}} \mathcal{L} \odot \nabla_{z^{x,L}a}$
 8:     for $l = L-1, L-2, ..., 1$ do                                ▷ Backpropagation
 9:         Compute $\varepsilon^{x,l} = \left(\left(W^{l+1}\right)^T \varepsilon^{x,l+1}\right) \odot \nabla_{z^{x,l}a}$
10:     end for
11:     Compute the gradient of the loss function given by: $\frac{\partial \mathcal{L}}{\partial w_{ij}^l} = a_j^{l-1}\varepsilon_i^l$ and $\frac{\partial \mathcal{L}}{\partial b_i^l} = \varepsilon_i^l$
12: end for
13: for $l = L, L-1, ..., 1$ do                                       ▷ Gradient descent step
14:     Update $W^l \to W^l - \frac{\eta}{m}\sum_x \varepsilon^{x,l}\left(a^{x,l-1}\right)^T$
15:     Update $b^l \to b^l - \frac{\eta}{m}\sum_x \varepsilon^{x,l}$
16: end for
```

2-1-5 RBF Networks

Radial Basis Function (RBF) networks constitute a class of NNs typically employed in various ML tasks such as pattern recognition, function approximation, and regression analysis [3]

RBF networks are FNNs and share similarities with MLP networks. However, they differ in terms of their activation functions. These networks make use of RBF as activation functions. The RBF, which is typically a Gaussian function, ensures the network's response decreases (or increases) monotonically and smoothly as the input moves away from a central point.

The architecture of an RBF network consists of three layers. The first layer is the input layer that takes in the raw data, followed by a hidden layer with a non-linear RBF activation function, and finally, a linear output layer that combines the outputs of the hidden layer. The number of nodes in the input and output layers corresponds to the dimensionality of the input data and output data, respectively.

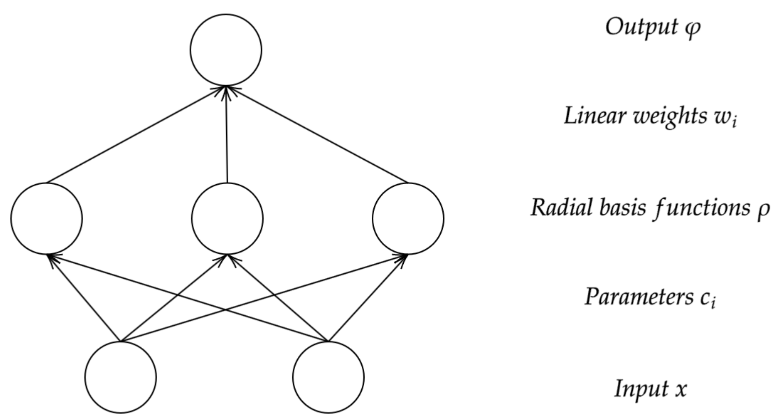


Figure 2-6: Schematic of a RBF network - A given input vector $\mathbf{x}$ feeds into multiple radial basis functions, each having distinct parameters. The network's final output emerges from the linear combination of outputs across these RBFs.

Let the input vector of the network be given as $\mathbf{x} \in \mathbb{R}^n$. Then, the output of the network is the scalar function $\varphi : \mathbb{R}^n \to \mathbb{R}$ defined by the following equation:

$$\varphi(\mathbf{x}) = \sum_{i=1}^{N} w_i \rho\left(\|\mathbf{x} - \mathbf{c}_i\|\right) \tag{2-22}$$

where N is the number of basis functions used in the hidden layer, $\boldsymbol{c_i}$ is the center location vector for each basis function i, and a_i is the weight of the ith basis function in the linear output layer. Functions that rely solely on the distance from a central vector exhibit radial symmetry around that vector, which is why they are referred to as RBFs. The Euclidian distance is typically taken as the norm and the Gaussian distribution as the RBF [10].

$$\rho\left(\|\mathbf{x} - \mathbf{c}_i\|\right) = \exp\left[-l_i^2 \|\mathbf{x} - \mathbf{c}_i\|^2\right] \tag{2-23}$$

where l_i are the same length-scales as previously discussed. A significant aspect of RBF networks is their localized nature of learning, which results in the network's ability to adapt to changes in the local environment while leaving other regions unaffected. This property enables RBF networks to provide robust performance even in scenarios with non-uniform data distribution. In other words, changing the parameters of one basis function will only have a small effect on input values that are far away from the center of that particular basis function since:

$$\lim_{\|x\| \to \infty} \rho\left(\|\mathbf{x} - \mathbf{c}_i\|\right) = 0 \tag{2-24}$$

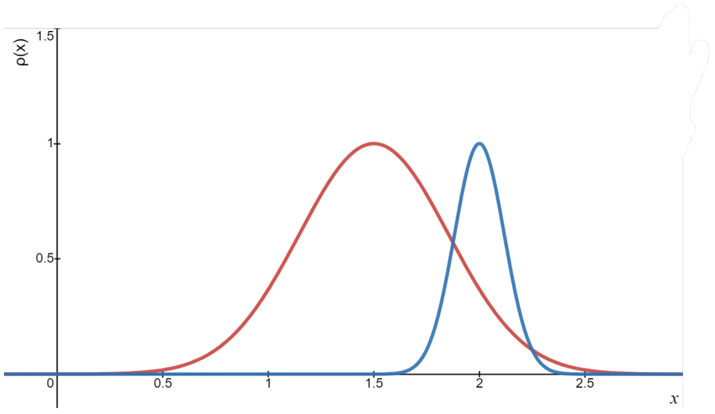


Figure 2-7: Example of two Gaussian basis functions with $\boldsymbol{c} = \begin{bmatrix} 1.5 & 2 \end{bmatrix}$ and $\boldsymbol{\Lambda} = \begin{bmatrix} 2 & 6 \end{bmatrix}$

RBF networks are particularly well-suited for problems where the mapping from input to output space is complex and non-linear. Given specific conditions regarding the shape of the activation function, RBF networks are universal function approximators on a compact subset of $\mathbb{R}^n$ [76]. This implies that given a sufficient number of basis functions, an RBF network can approximate any continuous function within a closed and bounded set with an arbitrary degree of accuracy.

However, certain drawbacks are associated with RBF networks. Deciding the number of basis functions and the length scales of the RBF is non-trivial and can significantly impact the network performance [66], e.g., using too many basis functions can increase the risk of overfitting. Proper tuning can be challenging and time-consuming.

RBF networks are impacted by the curse of dimensionality, meaning that as the dimensionality of the input space grows, the number of basis functions required for good input space coverage increases exponentially [80]. However, this can also increase the computational demand since the distance between each data point and every basis function has to be calculated. Furthermore, the distance metric used in RBF becomes less meaningful as the dimensionality of the space increases [108].

Some extensions have been proposed to address the drawbacks of RBF networks, e.g., an adaptive kernel incorporating the cosine distance measure alongside the Euclidian distance measure [55].

2-2 Gaussian Processes

To understand GPs better, we first have to define stochastic processes. We can view stochastic processes as collections of random variables [53]. Or, more formally, according to [53]:

Definition 2. *Let T be an index set, such as time, and Ω be a sample space, such as $\mathbb{R}^D$. A stochastic process $\{x(t), t \in T\}$ is a function of two arguments $\{x(t, \omega) : t \in T, \omega \in \Omega\}$. For fixed $t \in T, x(t, \cdot)$ is thus a random variable, and for fixed $\omega, x(\cdot, \omega)$ is a function of time which is called a realization of the process, a trajectory or a path.*

GPs are stochastic processes that describe probability distributions over a set of candidate functions [83]. These probability distributions dictate the properties of the candidate functions. Since we use probability distributions, we can use Bayes' rule to update our belief and uncertainty on the candidate functions after observing some training data [64].

Definition 3. *A GP is a collection of random variables, any finite number of which have a joint Gaussian distribution [83].*

A GP is fully defined by its mean and covariance function, according to:

$$\begin{aligned} m_h(\mathbf{x}) &= \mathbb{E}[h(\mathbf{x})] \\ k_h(\mathbf{x}, \mathbf{x}') &= \mathbb{E}\left[(h(\mathbf{x}) - m_h(\mathbf{x}))\left(h(\mathbf{x}') - m_h(\mathbf{x}')\right)\right] \end{aligned} \tag{2-25}$$

with the GP written as:

$$h(\mathbf{x}) \sim \mathcal{GP}(m_h(\mathbf{x}), k_h(\mathbf{x}, \mathbf{x}')) \tag{2-26}$$

Usually, the mean $m_h(\mathbf{x})$ is assumed to be zero everywhere since we can add a constant term or uncertainty about the mean to the kernel [25], and after training, this might not be zero anymore (depending if the training data has been centered). However, the kernel choice of a GP is crucial since the kernel determines what kind of structure can be captured and how the GP generalizes. There is a wide range of options regarding the different types of kernels, to name a few: Squared Exponential (SE) kernel, linear kernel, and Matern kernels [83][104][73].

Useful features of Gaussian Processes GPs enjoy some useful properties which make them an attractive choice for modeling purposes:

- **Closed-form inference**. The predictive posterior distribution of a GP can be computed analytically in closed-form for regression tasks [83]. More specifically, the predictive distribution is a multivariate Gaussian distribution. Meaning we can perform inference analytically.
- **Confidence intervals**. Since the prediction of GPs is Gaussian, we can compute confidence intervals for our predictions instead of just getting the point estimates [83].
- **Model selection.** We can compute the marginal likelihood of the data for a given model [25]. As a result, we can use Bayesian model comparison for model selection.

The downsides of Gaussian Processes Unfortunately, GPs also come with some limitations which can make them difficult to use:

- **Costly inference**. Inference with GPs involves the inversion of its covariance matrix, which scales in $\mathcal{O}\left(N^3\right)$ time, making them scale poorly to large datasets [104]. However, several methods exist to alleviate this problem by relying on sparse methods [81], [63], [96], and [42].
- **Kernel choice.** On the one hand, a wide range in kernel selection can be regarded as a helpful property [25]. However, on the other hand, it also comes with the risk of misspecifying the kernel, which can make the convergence slow [25].

2-2-1 Kernel

A GP's kernel (or covariance function) models the covariance between each pair of points in the input space. A kernel is called valid if its matrix $\mathbf{K}$ is Positive Semi-Definite (PSD). $\mathbf{K}$ (with $\mathbf{K}$ a real $n \times n$ matrix) is PSD if it satisfies $Q(\mathbf{v}) = \mathbf{v}^\top K\mathbf{v} \geq 0 \ \forall \mathbf{v} \in \mathbb{R}^n$ [83]. This requirement is actually the only restriction for selecting a valid kernel function [83].

We can further make a distinction between stationary and non-stationary kernels. Stationary kernels are invariant to translations in the input space [83]:

$$\mathbf{K}(\mathbf{x_a}, \mathbf{x_b}) = \mathbf{K}(\mathbf{x_a} - \mathbf{x_b}, 0) \tag{2-27}$$

Non-stationary kernels do not satisfy this property.

There exists a wide range of kernels [83], and then it boils down to selecting a kernel that correctly captures the behavior of the latent function $\mathbf{h}$. Choosing a proper kernel is crucial since misspecification can lead to poor predictions [52]. Luckily, a few popular kernels give good results when applied practically.

The most popular kernel is the SE (also known as the RBF) and is defined as follows [10]:

$$k_{\mathrm{SE}}\left(\mathbf{x}_p, \mathbf{x}_q\right) := \alpha^2 \exp\left(-\frac{1}{2}\left(\mathbf{x}_p - \mathbf{x}_q\right)^\top \Lambda^{-1}\left(\mathbf{x}_p - \mathbf{x}_q\right)\right) + \delta_{pq}\sigma_\varepsilon^2, \quad \mathbf{x}_p, \mathbf{x}_q \in \mathbb{R}^D \tag{2-28}$$

where α^2 is the signal variance of h (describing the magnitude of deviations from the mean values, which can be seen in Figure 2-8), $\Lambda^{-1} = \operatorname{diag}\left(\left[\ell_1^2, \ldots, \ell_D^2\right]\right)$ is a diagonal matrix consisting of the squared lengthscales. Finally, we have a term that takes into account the independent measurement noise through $\delta_{pq}\sigma_E^2$. δ_{pq} is called the Kronecker delta and has the following property [21]:

$$\delta_{pq} = \begin{cases} 0 & \text{if } p \neq q \\ 1 & \text{if } p = q \end{cases} \tag{2-29}$$

The aforementioned parameters are called the hyperparameters of the function h and are jointly collected in the vector $\boldsymbol{\theta} = \left[\begin{array}{ccccc} \alpha & \sigma_\varepsilon & l_1 & \ldots & l_D \end{array}\right]$.

The SE kernel belongs to the class of stationary kernels since the value only depends on the distance between $\mathbf{x}_p - \mathbf{x}_q$, and hence is translation invariant [83]. When the distance between the two points is small, the value of the kernel between the points will be high, and as the distance increases, the kernel value decreases exponentially [10]. A high kernel value indicates a high correlation between the two points in consideration, and information about $h(\boldsymbol{x}_p)$ has a significant impact on the prediction of $h(\boldsymbol{x}_q)$. This brings us back to the length scales; they capture how fast the kernel value (and hence the correlation) decays with the distance. Smaller values will allow for faster decay and also more oscillatory and varying function behavior [83] (see Figure 2-8). Also, if the length-scales $(\ell_1^2, \ldots, \ell_D^2)$ are allowed to have different values, the kernel is said to have the Automatic Relevance Determination (ARD) property [21].

One of the most valuable properties of the SE kernel is its universal approximation property [69]. This property states that there is always a function $\mathbf{h}^*$ that is in the space of all functions that the GP can generate, which can be as close as desired to the latent function $\mathbf{h}$. The full proof of this property for the SE kernel can be found in [69].

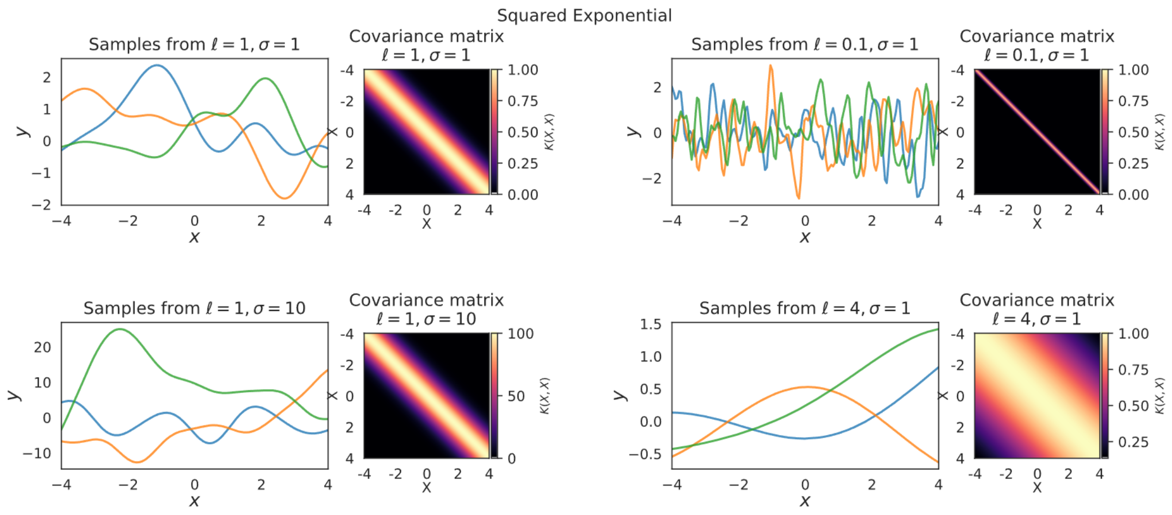


Figure 2-8: We have constructed four different SE kernels. The kernels have smaller and larger length scales on the top and bottom right, respectively. Notice how the amount of fluctuations in the sampled functions is inversely related to the length scale, and the spread of the covariance matrix is proportional to the length scale. The bottom left kernel has a larger α and note how the sampled function values can differ more from the mean, while the covariance matrix has remained the same.

Other examples of commonly used kernels are the Matern and sinusoidal kernels. The Matern kernel has a parameter that can control its differentiability and thus allows for modeling functions that are not infinitely smooth [83]. The sinusoidal kernel can be used for modeling cyclical patterns [83].

2-2-2 Inference with Gaussian Processes

This subsection will explore how GPs can be used to make predictions based on observed data and provide an insight into how GPs provide a measure of uncertainty for their predictions. We will split our data set $\mathcal{D}$ in a training set for which we know the output values y_i and a test set for which the output values are unknown.

Deterministic Inputs with Noisy Observations

Suppose we have a dataset $\mathcal{D} := \left\{\mathbf{X} := [\mathbf{x}_1, \ldots, \mathbf{x}_n]^\top,\ \mathbf{y} := [y_1, \ldots, y_n]^\top\right\}$, with $y_i = h(\mathbf{x}_i) + \varepsilon_i$, where $h : \mathbb{R}^D \to \mathbb{R}$ is the latent function that we want to learn and $\varepsilon_i \sim \mathcal{N}(0, \sigma_\varepsilon^2)$ Independent and Identically Distributed (iid) Gaussian noise. Furthermore, define the test set as $\mathbf{X}_* := [\mathbf{x}_{*1}, \ldots, \mathbf{x}_{*m}]^\top$; and $\mathbf{x}, \mathbf{x}_* \in \mathbb{R}^D$.

Then the function values for both the test and training inputs are jointly Gaussian, using the definition of the GP [19]:

$$p(\mathbf{h}, \mathbf{h}_* \mid \mathbf{X}, \mathbf{X}_*) = \mathcal{N}\left(\begin{bmatrix} m_h(\mathbf{X}) \\ m_h(\mathbf{X}_*) \end{bmatrix}, \begin{bmatrix} \mathbf{K} & k_h(\mathbf{X}, \mathbf{X}_*) \\ k_h(\mathbf{X}_*, \mathbf{X}) & \mathbf{K}_* \end{bmatrix}\right) \tag{2-30}$$

where $\mathbf{h} := [h(\mathbf{x}_1), \ldots, h(\mathbf{x}_n)]^\top$ and $\mathbf{h}_* := [h(\mathbf{x}_{*_1}), \ldots, h(\mathbf{x}_{*_m})]^\top$. Furthermore, the kernel matrix $\mathbf{K}$ is the full covariance matrix of all function values $\boldsymbol{h}$, and similarly for $\mathbf{K}_*$ for all functions values $\boldsymbol{h}_*$. And finally, $k_h(\mathbf{X}, \mathbf{X}_*) = k_h(\mathbf{X}_*, \mathbf{X})^T$ is the kernel matrix for $[k_h(\mathbf{x}_1, \mathbf{x}_{*1}), k_h(\mathbf{x}_2, \mathbf{x}_{*1}), \ldots, k_h(\mathbf{x}_n, \mathbf{x}_{*m})]$.

Univariate Predictions

Assume we have a scalar training target $y_i \in \mathbb{R}$ and a deterministic test input $\mathbf{x}_* \in \mathbb{R}^D$. We can then condition on the observed function values and dataset. The result is a posterior Gaussian distribution. The predictive equations for our regression problem are the following Gaussian: $p(h(\mathbf{x}_*) \mid \mathcal{D}, \mathbf{x}_*)$ and its mean and variance are given by [19]:

$$\begin{aligned} \mu_* := m_h(\mathbf{x}_*) &:= \mathbb{E}_h[h(\mathbf{x}_*) \mid \mathbf{X}, \mathbf{y}] = k_h(\mathbf{x}_*, \mathbf{X})\left(\mathbf{K} + \sigma_\varepsilon^2 \mathbf{I}\right)^{-1} \mathbf{y} \\ &= k_h(\mathbf{x}_*, \mathbf{X})\boldsymbol{\beta} = \sum_{i=1}^{n} \beta_i k_h(\mathbf{x}_i, \mathbf{x}_*) \end{aligned} \tag{2-31}$$

$$\sigma_*^2 := \sigma_h^2(\mathbf{x}_*) := \operatorname{var}_h[h(\mathbf{x}_*) \mid \mathbf{X}, \mathbf{y}] = k_h(\mathbf{x}_*, \mathbf{x}_*) - k_h(\mathbf{x}_*, \mathbf{X})\left(\mathbf{K} + \sigma_\varepsilon^2 \mathbf{I}\right)^{-1} k_h(\mathbf{X}, \mathbf{x}_*) \tag{2-32}$$

with $\boldsymbol{\beta} := (\mathbf{K} + \sigma_\varepsilon^2 \mathbf{I})^{-1} \mathbf{y}$. Note that we have a mean of our prediction and the variance, which tells us how confident our model is about the prediction. Furthermore, $k_h(\mathbf{x}_*, \mathbf{X})(\mathbf{K} + \sigma_\varepsilon^2 \mathbf{I})^{-1} k_h(\mathbf{X}, \mathbf{x}_*)$ (which encodes how much information can be transferred from the training set to the test set) is positive definite, hence the posterior variance $\sigma_h^2(\mathbf{x}_*)$ cannot be larger than the prior variance $k_h(\mathbf{x}_*, \mathbf{x}_*)$ (which represents the prior model uncertainty plus measurement noise) [19].

Multivariate Predictions

For the case $\mathbf{y} \in \mathbb{R}^E$ being a multivariate training target, we assume that the function values $h_1(\mathbf{x}), \ldots, h_E(\mathbf{x})$ are conditionally independent. Thus, given an input $\mathbf{x}$, $h_i(\mathbf{x})$ is independent of $h_j(\mathbf{x})$ for $i \neq j$. We still use the same training inputs $\mathbf{X} = [\mathbf{x}_1, \ldots, \mathbf{x}_n], \mathbf{x}_i \in \mathbb{R}^D$, but different training targets $\mathbf{y}_a = [y_1^a, \ldots, y_n^a]^\top$, $a = 1, \ldots, E$. The function values are still fully jointly Gaussian, within the same dimension a. As a result, the distribution for a single target dimension a is given by Equation 2-31 and Equation 2-32. Then the predictive distribution of $h(\mathbf{x}_*)$ is Gaussian and described by [19]:

$$\begin{aligned} \boldsymbol{\mu}_* &= \left[\begin{array}{ccc} m_{h_1}(\mathbf{x}_*) & \dots & m_{h_E}(\mathbf{x}_*) \end{array}\right]^\top \\ \boldsymbol{\Sigma}_* &= \operatorname{diag}\left(\left[\begin{array}{ccc} \sigma_{h_1}^2 & \dots & \sigma_{h_E}^2 \end{array}\right]\right) \end{aligned} \tag{2-33}$$

Uncertain Inputs

In this section, we will treat the case of uncertain inputs, resulting in the test input having a probability distribution, according to $\mathbf{x}_* \sim \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$. Now if we map the Gaussian input through a nonlinear function, the predictive distribution:

$$p(h(\mathbf{x}_*) \mid \boldsymbol{\mu}, \boldsymbol{\Sigma}) = \int p(h(\mathbf{x}_*) \mid \mathbf{x}_*)\, p(\mathbf{x}_* \mid \boldsymbol{\mu}_{\mathbf{x}_*}, \boldsymbol{\Sigma}_{\mathbf{x}_*})\, \mathrm{d}\mathbf{x}_* \tag{2-34}$$

will generally not be Gaussian. As a result, the predictive distribution cannot be computed analytically. However, we can use moment matching to approximate the distribution $p(h(\mathbf{x}_*) \mid \boldsymbol{\mu}, \boldsymbol{\Sigma})$ by a Gaussian with the same mean and variance [34].

Univariate Case

Let $\mathbf{x}_* \sim \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma}) \in \mathbb{R}^D$ be our Gaussian distributed test input and $y_* \in \mathbb{R}$ our target value. We already have the mean (Equation 2-31) and variance (Equation 2-32) of the predictive distribution $p(h(\mathbf{x}_*) \mid \mathbf{x}_*)$. As a next step, we have to determine the mean and variance of the distribution Equation 2-34. We can compute the mean μ_* using the law of total expectations [88] [34] [21]:

$$\mu_* = \iint h(\mathbf{x}_*)\, p(h, \mathbf{x}_*)\, \mathrm{d}(h, \mathbf{x}_*) = \mathbb{E}_{\mathbf{x}_*, h}[h(\mathbf{x}_*) \mid \boldsymbol{\mu}, \boldsymbol{\Sigma}] = \mathbb{E}_{\mathbf{x}_*}[\mathbb{E}_h[h(\mathbf{x}_*) \mid \mathbf{x}_*] \mid \boldsymbol{\mu}, \boldsymbol{\Sigma}] \tag{2-35}$$

$$= \mathbb{E}_{\mathbf{x}_*}[m_h(\mathbf{x}_*) \mid \boldsymbol{\mu}, \boldsymbol{\Sigma}] = \int m_h(\mathbf{x}_*)\, \mathcal{N}(\mathbf{x}_* \mid \boldsymbol{\mu}, \boldsymbol{\Sigma})\, \mathrm{d}\mathbf{x}_* \tag{2-36}$$

We already had an expression for the predictive mean μ_* through the Equation 2-31 and recall that $\boldsymbol{\beta} := \left(\mathbf{K} + \sigma_\varepsilon^2 \mathbf{I}\right)^{-1} \mathbf{y}$, then [34] [21]:

$$\mu_* = \boldsymbol{\beta}^\top \mathbf{q} \tag{2-37}$$

where $\mathbf{q} = [q_1, \dots, q_n]^\top \in \mathbb{R}^n$ with

$$q_i := \int k_h(\mathbf{x}_i, \mathbf{x}_*)\, \mathcal{N}(\mathbf{x}_* \mid \boldsymbol{\mu}, \boldsymbol{\Sigma})\, \mathrm{d}\mathbf{x}_* \tag{2-38}$$

q_i can be interpreted as the expected covariance between the function values $h(\mathbf{x}_i)$ and $h(\mathbf{x}_*)$ [21]. The result of this integral is dependent on the choice of the kernel.

We can also similarly determine the variance through the law of total variance [88]:

$$\sigma_*^2 = \operatorname{var}_{\mathbf{x}_*, h}[h(\mathbf{x}_*) \mid \boldsymbol{\mu}, \boldsymbol{\Sigma}] = \mathbb{E}_{\mathbf{x}_*}[\operatorname{var}_h[h(\mathbf{x}_*) \mid \mathbf{x}_*] \mid \boldsymbol{\mu}, \boldsymbol{\Sigma}] + \operatorname{var}_{\mathbf{x}_*}[\mathbb{E}_h[h(\mathbf{x}_*) \mid \mathbf{x}_*] \mid \boldsymbol{\mu}, \boldsymbol{\Sigma}] \tag{2-39}$$

We can rewrite this expression by using the following: $m_h(\mathbf{x}_*) = \mathbb{E}_h[h(\mathbf{x}_*) \mid \mathbf{x}_*]$, $\sigma_h^2(\mathbf{x}_*) = \mathrm{var}_h[h(\mathbf{x}_*) \mid \mathbf{x}_*]$ and $\mathrm{Var}(X) = \mathrm{E}[X^2] - \mathrm{E}[X]^2$:

$$= \mathbb{E}_{\mathbf{x}_*}\left[\sigma_h^2(\mathbf{x}_*) \mid \boldsymbol{\mu}, \boldsymbol{\Sigma}\right] + \left(\mathbb{E}_{\mathbf{x}_*}\left[m_h(\mathbf{x}_*)^2 \mid \boldsymbol{\mu}, \boldsymbol{\Sigma}\right] - \mathbb{E}_{\mathbf{x}_*}[m_h(\mathbf{x}_*) \mid \boldsymbol{\mu}, \boldsymbol{\Sigma}]^2\right) \tag{2-40}$$

Now, we can go a step further by substituting Equation 2-31 for $m_h(\mathbf{x}_*)$, Equation 2-32 for $\sigma_h^2(\mathbf{x}_*)$ and Equation 2-37 for $\mathbb{E}_{\mathbf{x}_*}[m_h(\mathbf{x}_*) \mid \boldsymbol{\mu}, \boldsymbol{\Sigma}]$:

$$\begin{aligned}\sigma_*^2 = &\int k_h(\mathbf{x}_*, \mathbf{x}_*) - k_h(\mathbf{x}_*, \mathbf{X})\left(\mathbf{K} + \sigma_\varepsilon^2\mathbf{I}\right)^{-1} k_h(\mathbf{X}, \mathbf{x}_*)\, p(\mathbf{x}_*)\, \mathrm{d}\mathbf{x}_* \\ &+ \int \underbrace{k_h(\mathbf{x}_*, \mathbf{X})}_{1\times n} \boldsymbol{\beta}\boldsymbol{\beta}^\top \underbrace{k_h(\mathbf{X}, \mathbf{x}_*)}_{n\times 1} p(\mathbf{x}_*)\, \mathrm{d}\mathbf{x}_* - \left(\boldsymbol{\beta}^\top \mathbf{q}\right)^2\end{aligned} \tag{2-41}$$

Again, the evaluation of this integral is dependent on the kernel.

Multivariate Case

Let us consider the multivariate case. Our target value is now $\mathbf{y}_* \in \mathbb{R}^E$. As a result we have a predictive mean vector μ_* of Equation 2-34 which contains all E independently predicted means [21]:

$$\boldsymbol{\mu}_* \mid \boldsymbol{\mu}, \boldsymbol{\Sigma} = \begin{bmatrix} \boldsymbol{\beta}_1^\top \mathbf{q}_1 & \dots & \boldsymbol{\beta}_E^\top \mathbf{q}_E \end{bmatrix}^\top \tag{2-42}$$

Contrary to the deterministic case, where the covariance matrix is a diagonal matrix (Equation 2-33), our covariance matrix is no longer diagonal [21]:

$$\boldsymbol{\Sigma}_* \mid \boldsymbol{\mu}, \boldsymbol{\Sigma} = \begin{bmatrix} \mathrm{var}_{h,\mathbf{x}_*}[h_1^* \mid \boldsymbol{\mu}, \boldsymbol{\Sigma}] & \dots & \mathrm{cov}_{h,\mathbf{x}_*}[h_1^*, h_E^* \mid \boldsymbol{\mu}, \boldsymbol{\Sigma}] \\ \vdots & \ddots & \vdots \\ \mathrm{cov}_{h,\mathbf{x}_*}[h_E^*, h_1^* \mid \boldsymbol{\mu}, \boldsymbol{\Sigma}] & \dots & \mathrm{var}_{h,\mathbf{x}_*}[h_E^* \mid \boldsymbol{\mu}, \boldsymbol{\Sigma}] \end{bmatrix} \tag{2-43}$$

This can be attributed to the fact that the target dimensions covary in this case. For abbrevation $h_a(\mathbf{x}_*)$ is denoted by $h_{a,}^* a \in \{1, \dots, E\}$. The diagonal of the covariance matrix is filled by the variances as defined in Equation 2-41. The cross-covariances are defined by:

$$\mathrm{cov}_{h,\mathbf{x}_*}[h_a^*, h_b^* \mid \boldsymbol{\mu}, \boldsymbol{\Sigma}] = \mathbb{E}_{h,\mathbf{x}_*}[h_a^* h_b^* \mid \boldsymbol{\mu}, \boldsymbol{\Sigma}] - \boldsymbol{\mu}_a^* \boldsymbol{\mu}_b^* \tag{2-44}$$

Now, we can assume h_a and h_b are conditionally independent given $\mathbf{x}_*$, resulting in [21]:

$$\mathbb{E}_{h,\mathbf{x}_*}[h_a^* h_b^* \mid \boldsymbol{\mu}, \boldsymbol{\Sigma}] = \mathbb{E}_{\mathbf{x}_*}[\mathbb{E}_{h_a}[h_a^* \mid \mathbf{x}_*]\, \mathbb{E}_{h_b}[h_b^* \mid \mathbf{x}_*] \mid \boldsymbol{\mu}, \boldsymbol{\Sigma}] = \int m_h^a(\mathbf{x}_*)\, m_h^b(\mathbf{x}_*)\, p(\mathbf{x}_*)\, \mathrm{d}\mathbf{x}_* \tag{2-45}$$

Recall from Equation 2-31 our expression for the mean function, which leads to [21]:

$$\begin{aligned}\mathbb{E}_{h,\mathbf{x}_*}[h_a^* h_b^* \mid \boldsymbol{\mu}, \boldsymbol{\Sigma}] &= \int m_h^a(\mathbf{x}_*)\, m_h^b(\mathbf{x}_*)\, p(\mathbf{x}_*)\, \mathrm{d}\mathbf{x}_* \\ &= \int \underbrace{k_h^a(\mathbf{x}_*, \mathbf{X})\, \boldsymbol{\beta}_a}_{\in\mathbb{R}} \underbrace{k_h^b(\mathbf{x}_*, \mathbf{X})\, \boldsymbol{\beta}_b}_{\in\mathbb{R}} p(\mathbf{x}_*)\, \mathrm{d}\mathbf{x}_* \\ &= \boldsymbol{\beta}_a^\top \underbrace{\int k_h^a(\mathbf{x}_*, \mathbf{X})^\top k_h^b(\mathbf{x}_*, \mathbf{X})\, p(\mathbf{x}_*)\, \mathrm{d}\mathbf{x}_*}_{=:\mathbf{Q}} \boldsymbol{\beta}_b\end{aligned} \tag{2-46}$$

2-2-3 Learning the Hyper-Parameters

In the previous section, we have treated the hyper-parameters $\boldsymbol{\theta}$ and the kernel as known. However, we do not know these values and function a priori. Let us define the following marginal likelihood [83]:

$$p\left(\boldsymbol{\theta} \mid \mathbf{y}, X, \mathcal{M}_i\right) = \frac{p\left(\mathbf{y} \mid X, \boldsymbol{\theta}, \mathcal{M}_i\right) p\left(\boldsymbol{\theta} \mid \mathcal{M}_i\right)}{p\left(\mathbf{y} \mid X, \mathcal{M}_i\right)} \tag{2-47}$$

where $p\left(\mathbf{y} \mid X, \boldsymbol{\theta}, \mathcal{M}_i\right)$ is the likelihood term describing how likely it is that a given set of hyper-parameters describes the data. $p\left(\boldsymbol{\theta} \mid \mathcal{M}_i\right)$ is the prior for the hyper-parameters (also called the hyper-prior [83]), which describes our prior beliefs about hyper-parameters given a kernel. The normalization constant is obtained by marginalizing out the hyper-parameters:

$$p\left(\mathbf{y} \mid X, \mathcal{M}_i\right) = \int p\left(\mathbf{y} \mid X, \boldsymbol{\theta}, \mathcal{M}_i\right) p\left(\boldsymbol{\theta} \mid \mathcal{M}_i\right) d\boldsymbol{\theta} \tag{2-48}$$

Given the observed data and chosen kernel, we have obtained a posterior distribution that describes a probability distribution over the hyper-parameters.

2-2-4 Maximum Likelihood

Our goal is to find the maximum value of $p\left(\boldsymbol{\theta} \mid X, \boldsymbol{y}, \mathcal{M}_i\right)$, which should correspond to the optimal set of hyper-parameters $\hat{\boldsymbol{\theta}}$ [83]. We can simplify Equation 2-47, noting that the optimal $\hat{\boldsymbol{\theta}}$ will not be affected by the normalization constant Equation 2-48. Dropping this term means we want to find the maximum of Equation 2-47 up to the normalization constant:

$$p\left(\boldsymbol{\theta} \mid X, \boldsymbol{y}, \mathcal{M}_i\right) \propto p\left(\boldsymbol{y} \mid X, \boldsymbol{\theta}, \mathcal{M}_i\right) p\left(\boldsymbol{\theta} \mid \mathcal{M}_i\right) \tag{2-49}$$

If we assume that we have no prior information on the values of $\hat{\boldsymbol{\theta}}$, then Equation 2-49 becomes proportional to only the marginal likelihood $p\left(\boldsymbol{y} \mid X, \boldsymbol{\theta}, \mathcal{M}_i\right)$, which we can optimize to find $\hat{\boldsymbol{\theta}}$. For convenience, we will take the log marginal likelihood [83]:

$$\mathcal{L}(\boldsymbol{\theta}) = \log p(\boldsymbol{y} \mid X, \boldsymbol{\theta}) = -\frac{1}{2}\boldsymbol{y}^T K_y^{-1}\boldsymbol{y} - \frac{1}{2}\log|K_y| - \frac{n}{2}\log 2\pi \tag{2-50}$$

where $K_y = K_f + \sigma_n^2 I$ is the covariance matrix assuming noisy observations, and we assume the kernel has already been chosen (or will be chosen), hence the dependency on $\mathcal{M}_i$ has been dropped. We can identify three terms in the log marginal likelihood in Equation 2-50. Namely [83]: the data-fit term $\frac{1}{2}\boldsymbol{y}^T K_y^{-1}\boldsymbol{y}$ which involves the observed targets, the complexity penalty term $\frac{1}{2}\log|K_y|$ which involves only the kernel and a constant term $\frac{n}{2}\log 2\pi$. This equation actually gives a trade-off between a model that can accurately predict the training data (model fit) and a simple model (to avoid overfitting) [83]

The optimal set of hyper-parameter can be found by solving the following optimization problem:

$$\hat{\boldsymbol{\theta}} = \operatorname{argmax} \mathcal{L}(\boldsymbol{\theta}) \tag{2-51}$$

We can use gradient descent to solve for the hyper-parameters by making use of the partial derivatives of the marginal likelihood with respect to the hyper-parameters [83]:

$$\begin{aligned}\frac{\partial}{\partial \theta_j} \log p(\mathbf{y} \mid X, \boldsymbol{\theta}) &= \frac{1}{2}\mathbf{y}^\top K^{-1}\frac{\partial K}{\partial \theta_j}K^{-1}\mathbf{y} - \frac{1}{2}\operatorname{tr}\left(K^{-1}\frac{\partial K}{\partial \theta_j}\right) \\ &= \frac{1}{2}\operatorname{tr}\left(\left(\boldsymbol{\alpha}\boldsymbol{\alpha}^\top - K^{-1}\right)\frac{\partial K}{\partial \theta_j}\right) \text{ where } \boldsymbol{\alpha} = K^{-1}\mathbf{y}\end{aligned} \tag{2-52}$$

where $\operatorname{tr}(\cdot)$ denotes the trace of a matrix.

2-2-5 Sparse Gaussian Processes

One of the key limitations of GPs is their scalability with larger data. The bottleneck lies in the inversion of the $(\mathbf{K} + \sigma_\varepsilon^2 \mathbf{I})$ which requires $\mathcal{O}(n^3)$ computations per gradient step and storage $O(n^2)$ [96]. The scalability issue is further exacerbated in the case of multivariate training targets with uncertain inputs, resulting in $\mathcal{O}(E^2 n^2 D)$ (with D the dimensionality of the training inputs and E the dimensionality of the training targets) [19].
This is where sparse GPs comes into the picture. The main idea behind sparse GPs is to improve the scalability of the GP while still retaining its predictive quality. Several methods have been developed, e.g., subset-of-data (SoD), which trains the GP on a smaller subset of the data [40], or sparse approximations, which approximate either the prior [81][92], posterior [96][8] or exploit specific structures of the kernel [105].

2-2-6 Deep Kernel Learning

GPs are a popular choice for modeling due to their explainable nature and reliable uncertainty estimates [104]. However, GPs have trouble with high-dimensional and structured data [75]. On the other hand, NNs can learn generally sophisticated representations, which can aid predictions. Unfortunately, NNs suffer from unreliable uncertainty estimates [104]. Bayesian Neural Networks (BNN) address these issues. However, inference remains costly due to the complex posteriors and the tremendous amount of parameters [104]. DKL combines the two frameworks in order to obtain the "best of both worlds." The main idea is to use an NN to map the high-dimensional input data to a low-dimensional intermediate feature space, which the GP then uses. The general structure of DKL is shown in Figure 2-9.

Starting from a given kernel $k : \mathbb{R}^D \times \mathbb{R}^D \mapsto \mathbb{R}$ and an NN $\Phi : \mathbb{R}^M \mapsto \mathbb{R}^D$, the deep kernel is composition between the two:

$$k(\mathbf{x}_i, \mathbf{x}_j \mid \boldsymbol{\theta}) \to k(\Phi(\mathbf{x}_i, \mathbf{w}), \Phi(\mathbf{x}_j, \mathbf{w}) \mid \boldsymbol{\theta}, \mathbf{w}) \tag{2-53}$$

where $\Phi(\mathbf{x}, \mathbf{w})$ is the NN parameterized by weights $\mathbf{w}$.

In this framework the deep kernel hyper-parameters $\boldsymbol{\gamma} = \{\mathbf{w}, \boldsymbol{\theta}\}$, which contain the weights and biases of the NN $\mathbf{w}$, and the parameters of the base kernel $\boldsymbol{\theta}$, are jointly learnt by maximizing the log marginal likelihood $\mathcal{L}$ [104]:

$$\log p(\mathbf{y} \mid \boldsymbol{\gamma}, X) \propto -\left[\mathbf{y}^\top \left(K_\gamma + \sigma^2 I\right)^{-1} \mathbf{y} + \log\left|K_\gamma + \sigma^2 I\right|\right] \tag{2-54}$$

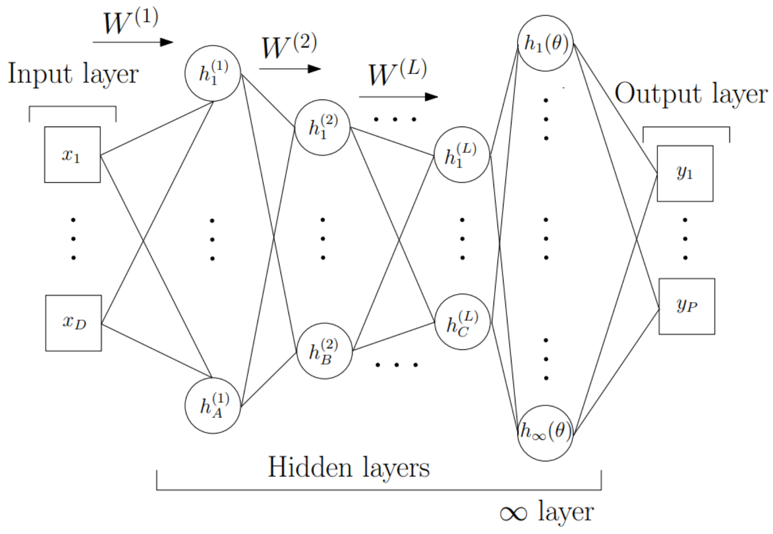


Figure 2-9: *Note: this figure is produced by Wilson in 2016 [104].* A GP featuring DKL transforms D-dimensional input values x by employing L hidden layers, followed by a hidden layer consisting of an infinite amount of basis functions. These functions utilize the base kernel hyper-parameters, denoted as θ. A GP integrated with a deep kernel generates a probabilistic mapping consisting of an infinite series of adaptive basis functions parameterized by $\gamma = (w, \theta)$. The entirety of the parameters γ is determined through the GP's marginal likelihood.

where K_γ depends on both $\boldsymbol{\theta}$ and $\mathbf{w}$. We can then use the chain rule to compute the gradients with respect to the base kernel hyper-parameters $\boldsymbol{\theta}$ as follows [104]:

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{\theta}} = \frac{\partial \mathcal{L}}{\partial K_\gamma} \frac{\partial K_\gamma}{\partial \boldsymbol{\theta}} \tag{2-55}$$

which follows the same route as a standard GP. The gradients with regards to the parameter $\mathbf{w}$ are computed by applying the chain rule as well [104]:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{w}} = \frac{\partial \mathcal{L}}{\partial K_\gamma} \frac{\partial K_\gamma}{\partial \Phi(\mathbf{x}, \mathbf{w})} \frac{\partial \Phi(\mathbf{x}, \mathbf{w})}{\partial \mathbf{w}} \tag{2-56}$$

As a next step, Wilson et al. [104] proposed to absorb the noise covariance $\sigma^2 I$ into the covariance matrix and regard it as part of the hyper-parameters $\boldsymbol{\theta}$. As a result, the partial derivative derivative of $\mathcal{L}$ with respect to the deep kernel matrix $\boldsymbol{K}_\gamma$ is determined by:

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{K}_\gamma} = \frac{1}{2} \left(\boldsymbol{K}_\gamma^{-1} \mathbf{y}\mathbf{y}^\top \boldsymbol{K}_\gamma^{-1} - \boldsymbol{K}_\gamma^{-1} \right) \tag{2-57}$$

$\frac{\partial K_\gamma}{\partial \boldsymbol{\theta}}$ is the partial derivative of the DKL matrix with respect to the base kernel hyper-parameters $\boldsymbol{\theta}$, conditioned on the NN output. Then, $\frac{\partial K_\gamma}{\Phi(\mathbf{x},\mathbf{w})}$ is the partial derivative of the deep kernel matrix with respect to the output of the NN, conditioned on $\boldsymbol{\theta}$. Finally, $\frac{\partial \Phi(\mathbf{x},\mathbf{w})}{\partial \mathbf{w}}$ is the gradient of the NN, which can be obtained through backpropagation.

Furthermore, it is also possible to make the whole framework scalable to larger datasets. A sparse GP is used for scalability, and the deep kernel K_γ is replaced by the Kernel Interpolation for Scalable Structured Gaussian Processes (KISS-GP) (or Structured Kernel Interpolation (SKI)) [103]:

$$K_\gamma \approx M K_{U,U}^{\text{deep}} M^\top := K_{\text{KISS}} \tag{2-58}$$

where $K_{U,U}$ is the covariance matrix created from the deep kernel, over m inducing points $U = [\mathbf{u}_i]_{i=1} \dots m$ and M is a sparse matrix of interpolation weights. The training itself scales as $\mathcal{O}(n + h(m))$ (with $h(m)$ approximately linear in m, where m are the inducing points) [104]. As a result, this framework allows $m \approx n$ to improve the accuracy in its approximation and has a prediction cost of $\mathcal{O}(1)$ per test point [104].

Shortcomings of Deep Kernel Learning

However, DKL does come with its shortcomings. Ober et al. (2021) [75] showed that there were cases where DKL could lead to worse overfitting than a standard NN trained through maximum likelihood. They argued that maximizing the marginal likelihood term encourages the complexity penalty to be minimized. Then, the complexity penalty promotes strong correlations among various points. An overparameterized covariance function could potentially yield abnormal results, as it enables all points to be correlated in the prior rather than exclusively in the points where the correlations should manifest [75]. The improved performance was attributed to the implicit regularization induced by stochastic mini batching [75]. They recommended a fully Bayesian approach to the hyper-parameters to mitigate the risk of overfitting [75].

Moreover, the structured kernel interpolation framework DKL uses, KISS-GP, limits the dimensionality of the input to $d \leq 5$ due to computational complexity [103] [104]. DKL addresses this issue by projecting the input feature space to a lower dimensional space. However, this might result in feature collapse and an underestimation of the uncertainty [99]. Van Amersfoort et al. (2021) proposed to use a bi-Lipschitz constraint, which would approximately preserve the distances between data points. Hence, implementing the sparse GP approximation could also require the implementation of the constraint proposed by Van Amersfoort et al. (2021). Another mitigation to this issue could be the implementation of Structured Kernel Interpolation of Products (SKIP), which does not suffer from the same curse of dimensionality as KISS-GP [32] and decomposes the kernel matrix K into a product of kernels.

2-3 PILCO

PILCO is a model-based RL algorithm that uses GPs to model the dynamics of the environment [20]. State trajectories are predicted using Gaussian distributions, which allows the framework to incorporate uncertainty [20]. This section outlines the PILCO framework, which will serve as a significant foundation for our framework.

The PILCO framework considers dynamical systems described by [20]:

$$\boldsymbol{x}_{t+1} = f(\boldsymbol{x}_t, \boldsymbol{u}_t) + \boldsymbol{w}, \quad \boldsymbol{w} \sim \mathcal{N}(\mathbf{0}, \boldsymbol{\Sigma}_w) \tag{2-59}$$

where $\boldsymbol{x} \in \mathbb{R}^D$ are continuous-valued states, $\boldsymbol{u} \in \mathbb{R}^F$ are the controls, $\boldsymbol{w}$ is additive Gaussian noise, and f are the unkown system dynamics. The goal is to find a policy $\pi : \boldsymbol{x} \mapsto \pi(\boldsymbol{x}, \boldsymbol{\theta}) = \boldsymbol{u}$, which minimizes the expected long-term cost:

$$J^\pi(\boldsymbol{\theta}) = \sum_{t=0}^{T} \mathbb{E}_{\boldsymbol{x}_t}\left[c\left(\boldsymbol{x}_t\right)\right], \quad \boldsymbol{x}_0 \sim \mathcal{N}\left(\boldsymbol{\mu}_0, \boldsymbol{\Sigma}_0\right) \tag{2-60}$$

where the policy π is a function parameterized by $\boldsymbol{\theta}$. The cost of being in state $\mathbf{x}$ at time t is denoted by $c\left(\boldsymbol{x}_t\right)$, and T is the final time step [20]. The solution is a policy π^* and is found by going through the following four high-level steps in Algorithm 2:

Algorithm 2 PILCO

```
 1: init: Sample controller parameters $\boldsymbol{\theta} \sim \mathcal{N}(\mathbf{0}, \boldsymbol{I})$. Apply random control signals and
    record data.
 2: repeat
 3:     Learn probabilistic GP dynamics models using all data
 4:     repeat
 5:         Approximate inference for policy evaluation, get: $J^\pi(\boldsymbol{\theta})$.
 6:         Gradient-based policy improvement, get: $\mathrm{d}J^\pi(\boldsymbol{\theta})/\mathrm{d}\boldsymbol{\theta}$
 7:         Update parameters $\boldsymbol{\theta}$
 8:     until convergence; return $\theta^*$
 9:     Set $\pi^* \leftarrow \pi\left(\boldsymbol{\theta}^*\right)$
10:     Apply $\pi^*$ to the system and record data
11: until task learned
```

Generally, the policy π is represented as a function approximator, which can be either local (e.g., an RBF network) or global (e.g., an NN).

2-3-1 Learning the Model

PILCO uses a GP to model the system's dynamics. The training inputs are the tuples $(\boldsymbol{x}_t, \boldsymbol{u}_t) \in \mathbb{R}^{D+F}$ and the state differences $\Delta_t = \boldsymbol{x}_{t+1} - \boldsymbol{x}_t \in \mathbb{R}^D$ are used as training targets. This means that the GP will model the state's change rather than the state's absolute value. Note, that in this case the inputs are: $\tilde{\boldsymbol{x}} := \left[\boldsymbol{x}^\top \boldsymbol{u}^\top\right]^\top$. The hyper-parameters of the GP are learned through maximum likelihood by using the training inputs $\tilde{\boldsymbol{X}} = [\tilde{\boldsymbol{x}}_1, \ldots, \tilde{\boldsymbol{x}}_n]$ and corresponding targets $\boldsymbol{y} = [\Delta_1, \ldots, \Delta_n]^\top$. The resulting GP model will predict one-step ahead, and the predicted state $\boldsymbol{x}_{t+1}$ is Gaussian as well, with [20]:

$$p\left(\boldsymbol{x}_{t+1} \mid \boldsymbol{x}_t, \boldsymbol{u}_t\right) = \mathcal{N}\left(\boldsymbol{x}_{t+1} \mid \boldsymbol{\mu}_{t+1}, \boldsymbol{\Sigma}_{t+1}\right) \tag{2-61}$$

$$\boldsymbol{\mu}_{t+1} = \boldsymbol{x}_t + \mathbb{E}_f\left[\boldsymbol{\Delta}_t\right], \quad \boldsymbol{\Sigma}_{t+1} = \operatorname{var}_f\left[\boldsymbol{\Delta}_t\right] \tag{2-62}$$

PILCO requires long-term predictions $p\left(\boldsymbol{x}_1\right), \ldots, p\left(\boldsymbol{x}_T\right)$ under a given policy. At each time step t, PILCO computes the predictive state distribution $p\left(\boldsymbol{x}_{t+1}\right)$ by approximating it as a Gaussian. Uncertain inputs propagated through the GP for multi-step ahead predictions generally result in a non-Gaussian distribution that cannot be solved analytically [20]. PILCO uses an approximation method to solve this issue, called moment matching. The solutions are discussed in the next subsection and also in [20] [34].

2-3-2 Policy Evaluation

The policy has to be evaluated using simulations of the system through the dynamical model to enable optimization. The simulation requires cascading multiple one-step-ahead predictions together to obtain the predictive state distributions $p(\mathbf{x}_1|\pi), \ldots, p(\mathbf{x}_T|\pi)$ [20]. From here on out, the conditioning on the policy π will be left out for notational convenience.

Suppose that trajectories begin from a specified initial state $\boldsymbol{x}_0 \sim \mathcal{N}(\boldsymbol{\mu}_{x_0}, \Sigma_{x_0})$. To predict $\boldsymbol{x}_{t+1}$ from $p(\boldsymbol{x}_t)$, $p(\boldsymbol{\Delta}_t)$ needs to be determined, which requires $p(\tilde{\boldsymbol{x}}_t) = p(\boldsymbol{x}_t, \boldsymbol{u}_t)$ [20]:

$$p(\boldsymbol{\Delta}_t) = \iint p(f(\tilde{\boldsymbol{x}}_t) \mid \tilde{\boldsymbol{x}}_t) p(\tilde{\boldsymbol{x}}_t) \, \mathrm{d}f \; \mathrm{d}\tilde{\boldsymbol{x}}_t \tag{2-63}$$

where the resulting predictive distribution $p(\boldsymbol{\Delta}_t)$ will generally not be a Gaussian, but it can be approximated by a Gaussian through moment matching. Now, the predictive distribution of the successor state can be approximated again with moment matching as $p(\boldsymbol{x}_{t+1}) = \mathcal{N}(\boldsymbol{x}_{t+1} \mid \boldsymbol{\mu}_{t+1}, \boldsymbol{\Sigma}_{t+1})$ and with:

$$\begin{aligned} \boldsymbol{\mu}_{t+1} &= \boldsymbol{\mu}_t + \boldsymbol{\mu}_\Delta \\ \boldsymbol{\Sigma}_{t+1} &= \boldsymbol{\Sigma}_t + \boldsymbol{\Sigma}_{\boldsymbol{\Delta}} + \operatorname{cov}[\boldsymbol{x}_t, \boldsymbol{\Delta}_t] + \operatorname{cov}[\boldsymbol{\Delta}_t, \boldsymbol{x}_t] \end{aligned} \tag{2-64}$$

where the mean $\boldsymbol{\mu}_{\boldsymbol{\Delta}}$ and variance $\boldsymbol{\Sigma}_{\boldsymbol{\Delta}}$ are assumed known from $p(\boldsymbol{\Delta}_t)$.

In the end, the long-term cost $J^\pi(\boldsymbol{\theta})$ needs to be determined, but that requires the expected cost associated with each state distribution:

$$\mathbb{E}_{\boldsymbol{x}_t}[c(\boldsymbol{x}_t)] = \int c(\boldsymbol{x}_t) \mathcal{N}(\boldsymbol{x}_t \mid \boldsymbol{\mu}_t, \boldsymbol{\Sigma}_t) \, \mathrm{d}\boldsymbol{x}_t \tag{2-65}$$

where $t = 1, \ldots, T$, and the cost function c.

Moment Matching

PILCO uses moment matching to propagate the uncertainty during multi-step ahead predictions. The following equations outline the moment matching used in PILCO and originate from Deisenroth et al. (2015) [20]. Note that the following terms have to be determined: $\boldsymbol{\mu}_\Delta$, $\boldsymbol{\Sigma}_\Delta$, $\operatorname{cov}[\boldsymbol{x}_t, \boldsymbol{\Delta}_t]$ and $\operatorname{cov}[\boldsymbol{\Delta}_t, \boldsymbol{x}_t]$. Suppose we have the target dimensions $a = 1, .., D$, then the predictive mean can be determined by using the tower rule [20]:

$$\begin{aligned} \mu_\Delta^a &= \mathbb{E}_{\tilde{\boldsymbol{x}}_t}[\mathbb{E}_{f_a}[f_a(\tilde{\boldsymbol{x}}_t) \mid \tilde{\boldsymbol{x}}_t]] = \mathbb{E}_{\tilde{\boldsymbol{x}}_t}[m_{f_a}(\tilde{\boldsymbol{x}}_t)] \\ &= \int m_{f_a}(\tilde{\boldsymbol{x}}_t) \mathcal{N}\left(\tilde{\boldsymbol{x}}_t \mid \tilde{\boldsymbol{\mu}}_t, \tilde{\boldsymbol{\Sigma}}_t\right) \mathrm{d}\tilde{\boldsymbol{x}}_t = \boldsymbol{\beta}_a^\top \boldsymbol{q}_a \\ \boldsymbol{\beta}_a &= \left(\boldsymbol{K}_a + \sigma_{w_a}^2\right)^{-1} \boldsymbol{y}_a, \end{aligned} \tag{2-66}$$

where $\boldsymbol{q}_a = [q_{a_1}, \ldots, q_{a_n}]^\top \in \mathbb{R}^n$ and its entries are given by:

$$q_{a_i} = \sigma_{f_a}^2 \left|\tilde{\boldsymbol{\Sigma}}_t \boldsymbol{\Lambda}_a^{-1} + \boldsymbol{I}\right|^{-\frac{1}{2}} \exp\left(-\frac{1}{2} \boldsymbol{\nu}_i^\top \left(\tilde{\boldsymbol{\Sigma}}_t + \boldsymbol{\Lambda}_a\right)^{-1} \boldsymbol{\nu}_i\right) \tag{2-67}$$

where $\boldsymbol{\nu}_i := (\tilde{\boldsymbol{x}}_i - \tilde{\boldsymbol{\mu}}_t)$ is the difference between the training input $\tilde{\boldsymbol{x}}_i$ and the mean of $p(\boldsymbol{x}_t, \boldsymbol{u}_t)$.

The law of total covariance is then used to determine the predictive covariance matrix $\boldsymbol{\Sigma}_{\boldsymbol{\Delta}} \in \mathbb{R}^{D\times D}$ for the target dimensions $a, b = 1, ..., D$ [20]:

$$\begin{aligned} \sigma_{aa}^2 &= \mathbb{E}_{\tilde{\boldsymbol{x}}_t}\left[\operatorname{var}_f\left[\Delta_a \mid \tilde{\boldsymbol{x}}_t\right]\right] + \mathbb{E}_{f,\tilde{\boldsymbol{x}}_t}\left[\Delta_a^2\right] - \left(\boldsymbol{\mu}_{\boldsymbol{\Delta}}^a\right)^2, \\ \sigma_{ab}^2 &= \mathbb{E}_{f,\tilde{\boldsymbol{x}}_t}\left[\Delta_a\Delta_b\right] - \boldsymbol{\mu}_{\Delta}^a\boldsymbol{\mu}_{\boldsymbol{\Delta}}^b, \quad a \neq b, \end{aligned} \tag{2-68}$$

Note that PILCO assumes that the GP models are conditionally independent from each other, hence:

$$\mathbb{E}_{\tilde{\boldsymbol{x}}_t}\left[\operatorname{cov}_f\left[\Delta_a, \Delta_b \mid \tilde{\boldsymbol{x}}_t\right]\right] = 0 \tag{2-69}$$

The next step again uses the tower rule, assuming $p\left(\tilde{\boldsymbol{x}}_t\right) = \mathcal{N}\left(\tilde{\boldsymbol{x}}_t \mid \tilde{\boldsymbol{\mu}}_t, \tilde{\boldsymbol{\Sigma}}_t\right)$ and again using the conditional independence of the GPs, the following expression is obtained [20]:

$$\mathbb{E}_{f,\tilde{\boldsymbol{x}}_t}\left[\Delta_a\Delta_b\right] = \mathbb{E}_{\tilde{\boldsymbol{x}}_t}\left[\mathbb{E}_f\left[\Delta_a \mid \tilde{\boldsymbol{x}}_t\right]\mathbb{E}_f\left[\Delta_b \mid \tilde{\boldsymbol{x}}_t\right]\right] \tag{2-70}$$

Then through the definition of the mean function of the GP [20]:

$$\begin{aligned} \mathbb{E}_{f,\tilde{\boldsymbol{x}}_t}\left[\Delta_a\Delta_b\right] &= \boldsymbol{\beta}_a^\top \boldsymbol{Q}\boldsymbol{\beta}_b, \\ \boldsymbol{Q} &:= \int k_a\left(\tilde{\boldsymbol{x}}_t, \tilde{\boldsymbol{X}}\right)^\top k_b\left(\tilde{\boldsymbol{x}}_t, \tilde{\boldsymbol{X}}\right) p\left(\tilde{\boldsymbol{x}}_t\right)\mathrm{d}\tilde{\boldsymbol{x}}_t \end{aligned} \tag{2-71}$$

Furthermore, due to the Gaussian assumptions and approximations made, the entries of Q_{ij} of $\boldsymbol{Q} \in \mathbb{R}^{n\times n}$ can be determined with [20]:

$$Q_{ij} = |\boldsymbol{R}|^{-\frac{1}{2}} k_a\left(\tilde{\boldsymbol{x}}_i, \tilde{\boldsymbol{\mu}}_t\right) k_b\left(\tilde{\boldsymbol{x}}_j, \tilde{\boldsymbol{\mu}}_t\right) \exp\left(\frac{1}{2}\boldsymbol{z}_{ij}^\top \boldsymbol{T}^{-1}\boldsymbol{z}_{ij}\right) \tag{2-72}$$

with

$$\begin{aligned} \boldsymbol{R} &:= \tilde{\boldsymbol{\Sigma}}_t\left(\boldsymbol{\Lambda}_a^{-1} + \boldsymbol{\Lambda}_b^{-1}\right) + \boldsymbol{I}, \quad \boldsymbol{T} := \boldsymbol{\Lambda}_a^{-1} + \boldsymbol{\Lambda}_b^{-1} + \tilde{\boldsymbol{\Sigma}}_t^{-1}, \\ \boldsymbol{z}_{ij} &:= \boldsymbol{\Lambda}_a^{-1}\boldsymbol{\nu}_i + \boldsymbol{\Lambda}_b^{-1}\boldsymbol{\nu}_j, \end{aligned} \tag{2-73}$$

Finally, the diagonal terms of $\boldsymbol{\Sigma}_\Delta$ can also be fully determine using the following in addition [20]:

$$\mathbb{E}_{\tilde{\boldsymbol{x}}_t}\left[\operatorname{var}_f\left[\Delta_a \mid \tilde{\boldsymbol{x}}_t\right]\right] = \sigma_{f_a}^2 - \operatorname{tr}\left(\left(\boldsymbol{K}_a + \sigma_{w_a}^2\boldsymbol{I}\right)^{-1}\boldsymbol{Q}\right) + \sigma_{w_a}^2 \tag{2-74}$$

where $\sigma_{w_a}^2$ is the noise variance of the ath target dimension [20]. Hence, the predictive covariance matrix is now fully determined [20].

The final terms that have to be determined are the cross-covariances $\operatorname{cov}\left[\boldsymbol{x}_t, \boldsymbol{\Delta}_t\right]$, which is done through calculating the cross-covariance between an uncertain state-action pair and the corresponding predicted state evolution [20]:

$$\operatorname{cov}\left[\tilde{\boldsymbol{x}}_t, \boldsymbol{\Delta}_t\right] = \mathbb{E}_{\tilde{\boldsymbol{x}}_t, f}\left[\tilde{\boldsymbol{x}}_t\boldsymbol{\Delta}_t^\top\right] - \tilde{\boldsymbol{\mu}}_t\boldsymbol{\mu}_{\boldsymbol{\Delta}}^\top \tag{2-75}$$

Through the tower rule for each target dimension $a = 1, .., D$ [20]:

$$\begin{aligned} \mathbb{E}_{\tilde{\boldsymbol{x}}_t, f}\left[\tilde{\boldsymbol{x}}_t\Delta_t^a\right] = \mathbb{E}_{\tilde{\boldsymbol{x}}_t}\left[\tilde{\boldsymbol{x}}_t\mathbb{E}_f\left[\Delta_t^a \mid \tilde{\boldsymbol{x}}_t\right]\right] &= \int \tilde{\boldsymbol{x}}_t m_f^a\left(\tilde{\boldsymbol{x}}_t\right) p\left(\tilde{\boldsymbol{x}}_t\right)\mathrm{d}\tilde{\boldsymbol{x}}_t \\ &= \int \tilde{\boldsymbol{x}}_t\left(\sum_{i=1}^n \beta_{a_i} k_f^a\left(\tilde{\boldsymbol{x}}_t, \tilde{\boldsymbol{x}}_i\right)\right) p\left(\tilde{\boldsymbol{x}}_t\right)\mathrm{d}\tilde{\boldsymbol{x}}_t, \end{aligned} \tag{2-76}$$

which can be rewritten as:

$$\begin{aligned}
&\mathbb{E}_{\tilde{\boldsymbol{x}}_t,f}\left[\tilde{\boldsymbol{x}}_t\Delta_t^a\right] \\
&=\sum_{i=1}^{n}\beta_{a_i}\int\tilde{\boldsymbol{x}}_t\underbrace{c_1\mathcal{N}\left(\tilde{\boldsymbol{x}}_t \mid \tilde{\boldsymbol{x}}_i,\boldsymbol{\Lambda}_a\right)}_{=k_f^a(\tilde{\boldsymbol{x}}_t,\tilde{\boldsymbol{x}}_i)}\underbrace{\mathcal{N}\left(\tilde{\boldsymbol{x}}_t \mid \tilde{\boldsymbol{\mu}}_t,\tilde{\boldsymbol{\Sigma}}_t\right)}_{p(\tilde{\boldsymbol{x}}_t)}\mathrm{d}\tilde{\boldsymbol{x}}_t
\end{aligned} \tag{2-77}$$

where $k_f^a\left(\tilde{\boldsymbol{x}}_t,\tilde{\boldsymbol{x}}_i\right)=c_1\mathcal{N}\left(\tilde{\boldsymbol{x}}_t \mid \tilde{\boldsymbol{x}}_i,\boldsymbol{\Lambda}_a\right)$ can be defined as an unnormalized Gaussian probability distribution in $\tilde{\boldsymbol{x}}_t$, with $c_1:=\sigma_{f_a}^2(2\pi)^{\frac{D+F}{2}}\left|\boldsymbol{\Lambda}_a\right|^{\frac{1}{2}}$ Equation 2-77 contains the product of two Gaussians which yields a Gaussian [20]:

$$c_2^{-1}\mathcal{N}\left(\tilde{\boldsymbol{x}}_t \mid \boldsymbol{\psi}_i,\boldsymbol{\Psi}\right) \tag{2-78}$$

$$\begin{aligned}
c_2^{-1}=&(2\pi)^{-\frac{D+F}{2}}\left|\boldsymbol{\Lambda}_a+\tilde{\boldsymbol{\Sigma}}_t\right|^{-\frac{1}{2}} \\
&\times\exp\left(-\frac{1}{2}\left(\tilde{\boldsymbol{x}}_i-\tilde{\boldsymbol{\mu}}_t\right)^{\top}\left(\boldsymbol{\Lambda}_a+\tilde{\boldsymbol{\Sigma}}_t\right)^{-1}\left(\tilde{\boldsymbol{x}}_i-\tilde{\boldsymbol{\mu}}_t\right)\right),\\
\boldsymbol{\Psi}=&\left(\boldsymbol{\Lambda}_a^{-1}+\tilde{\boldsymbol{\Sigma}}_t^{-1}\right)^{-1},\quad \boldsymbol{\psi}_i=\boldsymbol{\Psi}\left(\boldsymbol{\Lambda}_a^{-1}\tilde{\boldsymbol{x}}_i+\tilde{\boldsymbol{\Sigma}}_t^{-1}\tilde{\boldsymbol{\mu}}_t\right).
\end{aligned} \tag{2-79}$$

The integral in Equation 2-77 can then be simplified to [20]:

$$\begin{aligned}
\mathbb{E}_{\tilde{\boldsymbol{x}}_t,f}\left[\tilde{\boldsymbol{x}}_t\Delta_t^a\right]&=\sum_{i=1}^{n}c_1c_2^{-1}\beta_{a_i}\boldsymbol{\psi}_i,\quad a=1,\ldots,D,\\
\operatorname{cov}_{\tilde{\boldsymbol{x}}_t,f}\left[\tilde{\boldsymbol{x}}_t,\Delta_t^a\right]&=\sum_{i=1}^{n}c_1c_2^{-1}\beta_{a_i}\boldsymbol{\psi}_i-\tilde{\boldsymbol{\mu}}_t\mu_{\boldsymbol{\Delta}}^a,
\end{aligned} \tag{2-80}$$

where $c_1c_2^{-1}=q_{a_i}$ and $\boldsymbol{\psi}_i=\boldsymbol{\Sigma}_t\left(\boldsymbol{\Sigma}_t+\boldsymbol{\Lambda}_a\right)^{-1}\tilde{\boldsymbol{x}}_i+\boldsymbol{\Lambda}\left(\boldsymbol{\Sigma}_t+\boldsymbol{\Lambda}_a\right)^{-1}\tilde{\boldsymbol{\mu}}_t$. The cross-covariance can then be further simplified to [20]:

$$\operatorname{cov}_{\tilde{\boldsymbol{x}}_t,f}\left[\tilde{\boldsymbol{x}}_t,\Delta_t^a\right]=\sum_{i=1}^{n}\beta_{a_i}q_{a_i}\tilde{\boldsymbol{\Sigma}}_t\left(\tilde{\boldsymbol{\Sigma}}_t+\boldsymbol{\Lambda}_a\right)^{-1}\left(\tilde{\boldsymbol{x}}_i-\tilde{\boldsymbol{\mu}}_t\right) \tag{2-81}$$

where $\operatorname{cov}\left[\boldsymbol{x}_t,\Delta_t\right]\in\mathbb{R}^{(D+F)\times D}$. Then, $\operatorname{cov}\left[\boldsymbol{x}_t,\Delta_t\right]$ is the $D\times D$ submatrix of $\operatorname{cov}\left[\boldsymbol{x}_t,\Delta_t\right]$ [20], which concludes the moment matching.

2-3-3 Cost Function

The PILCO framework uses a saturating immediate cost function, which depends solely on a geometric distance d between the current state and the target state [19]:

$$c(\mathbf{x})=1-\exp\left(-\frac{1}{2a^2}d\left(\mathbf{x},\mathbf{x}_{\text{target}}\right)^2\right) \tag{2-82}$$

The expected immediate cost can be calculated according to the following [19]:

$$\mathbb{E}_{\mathbf{x}}[c(\mathbf{x})]=1-\int c(\mathbf{x})p(\mathbf{x})\mathrm{d}\mathbf{x}=1-\int\exp\left(-\frac{1}{2}\left(\mathbf{x}-\mathbf{x}_{\text{target}}\right)^{\top}\mathbf{T}^{-1}\left(\mathbf{x}-\mathbf{x}_{\text{target}}\right)\right)p(\mathbf{x})\mathrm{d}\mathbf{x} \tag{2-83}$$

where $\mathbf{T}^{-1}$ is the precision matrix of $p(\boldsymbol{x})$. If the input vector $\boldsymbol{x}$ has the same representation as the target, then $\boldsymbol{T}^{-1}$ is a diagonal matrix with entries either unity or zero. Thus for $\mathbf{x} \sim \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$ this results in the following expected immediate cost [19]:

$$\mathbb{E}_{\mathbf{x}}[c(\mathbf{x})] = 1 - \left|\mathbf{I} + \boldsymbol{\Sigma}\mathbf{T}^{-1}\right|^{-1/2} \exp\left(-\frac{1}{2}\left(\boldsymbol{\mu} - \mathbf{x}_{\text{target}}\right)^{\top} \tilde{\mathbf{S}}_1 \left(\boldsymbol{\mu} - \mathbf{x}_{\text{target}}\right)\right) \tag{2-84}$$

where $\tilde{\mathbf{S}}_1 := \mathbf{T}^{-1}\left(\mathbf{I} + \boldsymbol{\Sigma}\mathbf{T}^{-1}\right)^{-1}$.

2-3-4 Policy Optimization

Recall that the original goal is to find a policy which minimizes $J^{\pi}(\boldsymbol{\theta})$. The gradient of the long-term cost with respect to the policy parameters $\mathrm{d}J^{\pi}(\boldsymbol{\theta})/\mathrm{d}\boldsymbol{\theta}$ can be used to achieve this goal. Now, in order to determine these gradients, the following assumptions are required [20]:

- The expected cost $\mathbb{E}_{\boldsymbol{x}_t}\left[c\left(\boldsymbol{x}_t\right)\right]$ is differentiable with respect to the first two moments of the state distribution.
- The first two moments of the control distribution $\boldsymbol{\mu}_u$ and $\boldsymbol{\Sigma}_u$ can be computed analytically and are differentiable with respect to the policy parameters $\boldsymbol{\theta}$.

The gradient of the long-term cost $\mathrm{d}J^{\pi}/\mathrm{d}\boldsymbol{\theta}$ can be calculated with [20]:

$$\begin{aligned}\frac{\mathrm{d}J^{\pi}(\boldsymbol{\theta})}{\mathrm{d}\boldsymbol{\theta}} &= \sum_{t=1}^{T} \frac{\mathrm{d}\mathbb{E}_{\boldsymbol{x}_t}\left[c\left(\boldsymbol{x}_t\right)\right]}{\mathrm{d}\boldsymbol{\theta}} \\ \frac{\mathrm{d}\mathbb{E}_{\boldsymbol{x}_t}\left[c\left(\boldsymbol{x}_t\right)\right]}{\mathrm{d}\boldsymbol{\theta}} &= \frac{\mathrm{d}\mathbb{E}_{\boldsymbol{x}_t}\left[c\left(\boldsymbol{x}_t\right)\right]}{\mathrm{d}p\left(\boldsymbol{x}_t\right)} \frac{\mathrm{d}p\left(\boldsymbol{x}_t\right)}{\mathrm{d}\boldsymbol{\theta}} := \frac{\partial\mathbb{E}_{\boldsymbol{x}_t}\left[c\left(\boldsymbol{x}_t\right)\right]}{\partial\boldsymbol{\mu}_t} \frac{\mathrm{d}\boldsymbol{\mu}_t}{\mathrm{d}\boldsymbol{\theta}} + \frac{\partial\mathbb{E}_{\boldsymbol{x}_t}\left[c\left(\boldsymbol{x}_t\right)\right]}{\partial\boldsymbol{\Sigma}_t} \frac{\mathrm{d}\boldsymbol{\Sigma}_t}{\mathrm{d}\boldsymbol{\theta}}\end{aligned} \tag{2-85}$$

where the expression can be solved with repeated applications of the chain rule, resulting in partial derivatives of both the state and controller actions with respect to the policy parameters [20]:

$$\begin{aligned}\frac{\mathrm{d}p\left(\boldsymbol{x}_t\right)}{\mathrm{d}\boldsymbol{\theta}} &= \frac{\partial p\left(\boldsymbol{x}_t\right)}{\partial p\left(\boldsymbol{x}_{t-1}\right)} \frac{\mathrm{d}p\left(\boldsymbol{x}_{t-1}\right)}{\mathrm{d}\boldsymbol{\theta}} + \frac{\partial p\left(\boldsymbol{x}_t\right)}{\partial\boldsymbol{\theta}}, \\ \frac{\partial p\left(\boldsymbol{x}_t\right)}{\partial p\left(\boldsymbol{x}_{t-1}\right)} &= \left\{\frac{\partial\boldsymbol{\mu}_t}{\partial p\left(\boldsymbol{x}_{t-1}\right)}, \frac{\partial\boldsymbol{\Sigma}_t}{\partial p\left(\boldsymbol{x}_{t-1}\right)}\right\}.\end{aligned} \tag{2-86}$$

As an example, the term $\mathrm{d}\boldsymbol{\mu}_t/\mathrm{d}\boldsymbol{\theta}$ is shown, a more detailed derivation is shown in [19].

$$\begin{aligned}\frac{\mathrm{d}\boldsymbol{\mu}_t}{\mathrm{d}\boldsymbol{\theta}} &= \frac{\partial\boldsymbol{\mu}_t}{\partial\boldsymbol{\mu}_{t-1}} \frac{\mathrm{d}\boldsymbol{\mu}_{t-1}}{\mathrm{d}\boldsymbol{\theta}} + \frac{\partial\boldsymbol{\mu}_t}{\partial\boldsymbol{\Sigma}_{t-1}} \frac{\mathrm{d}\boldsymbol{\Sigma}_{t-1}}{\mathrm{d}\boldsymbol{\theta}} + \frac{\partial\boldsymbol{\mu}_t}{\partial\boldsymbol{\theta}} \\ \frac{\partial\boldsymbol{\mu}_t}{\partial\boldsymbol{\theta}} &= \frac{\partial\boldsymbol{\mu}_{\Delta}}{\partial p\left(\boldsymbol{u}_{t-1}\right)} \frac{\partial p\left(\boldsymbol{u}_{t-1}\right)}{\partial\boldsymbol{\theta}} = \frac{\partial\boldsymbol{\mu}_{\Delta}}{\partial\boldsymbol{\mu}_u} \frac{\partial\boldsymbol{\mu}_u}{\partial\boldsymbol{\theta}} + \frac{\partial\boldsymbol{\mu}_{\Delta}}{\partial\boldsymbol{\Sigma}_u} \frac{\partial\boldsymbol{\Sigma}_u}{\partial\boldsymbol{\theta}}\end{aligned} \tag{2-87}$$

Since PILCO uses moment matching and an RBF network for the parameterization of the policy, it is possible to obtain analytic expressions for the policy gradients, and the optimizer Broyden–Fletcher–Goldfarb–Shanno (BFGS) (a second-order gradient-based optimization method) is used to solve for the optimized policy parameters $\boldsymbol{\theta}^*$ [20].

2-3-5 PILCO's Shortcomings

Despite a list of successful applications of PILCO in a variety of settings [20] [31] [68], the framework does have its limitations.

- **Kernel function:** PILCO uses the SE kernel [20]. However, the SE kernel assumes that the state transition functions are infinitely differentiable [83], which can be restrictive and lead to unrealistic smoothness assumptions [68]. Other kernels, e.g., Matérn, and a deep kernel, also exist and might perform better.
- **Moment matching approximations:** the framework requires approximate inference since the integrals used in calculating the posterior distribution are generally analytically intractable. In PILCO, the approximate inference is made through moment matching. However, moment matching can be computationally inflexible. For example, it cannot be used with NNs [77] unless linearization is used, which introduces approximation errors [34]. In addition, the performance of moment matching may be sensitive to the selection of hyper-parameters in the underlying model, such as kernel parameters in the GP [68].
- **Measurement noise:** PILCO treats measurement (or observation) noise as system noise, and as a result, it assumes there is direct access to the complete state of the system [20]. In reality, observation noise can result in the state being only partially observable. Fitting a GP over noisy measurement autoregressively can lead to a poor dynamical model [68]. Recent work has addressed this issue [67], where a partially observed Markov decision process (POMDP) is considered through a filtering process during policy evaluation.
- **Hard-coded gradients:** PILCO uses hard-coded gradients for the policy improvement. While this approach can lead to a more efficient optimization [20], it decreases flexibility. For each new problem-setting, the gradients would have to be derived by hand again, which also increases the risk of human-made error.
- **Temporal correlation:** successive state transitions are considered to be independent from each other [20]. This can result in an underestimation of the state uncertainty, eventually leading to diminished performance [68].

Chapter 3

Deep Kernel PILCO

In the final section of the previous chapter, we examined some of the most prominent shortcomings of Probabilistic Inference for Learning COntrol (PILCO). Among the main limitations of PILCO that we intend to address is the restricted capacity of the utilized kernel function and the constraints imposed by the moment matching approximation. Specifically, employing the Squared Exponential (SE) kernel can lead to unrealistic smoothness assumptions on the underlying function, which may be too restrictive and result in approximation errors [68]. Furthermore, it was shown that PILCO struggles in higher-dimensional spaces, which might be attributed to the SE kernel [31]. Additionally, moment matching for uncertainty propagation provides exact first- and second-order moments only with linear or Gaussian covariance functions [34]. In other cases, this method approximates the first- and second-order moments [34]. These approximations can lead to significant errors if the transformation is highly nonlinear [31].

This chapter presents our framework, Deep Kernel PILCO (DKL PILCO), specifically created for data-efficient deep Reinforcement Learning (RL). In contrast to PILCO's use of the SE kernel, our framework integrates Deep Kernel Learning (DKL), a Neural Network (NN) in conjunction with a Gaussian Process (GP), making it more compatible with high-dimensional observations while maintaining the ability to manage output uncertainties. Furthermore, instead of moment matching, DKL PILCO uses the Unscented Transform (UT) to propagate uncertainty. Like PILCO, our policy search algorithm employs a cyclical procedure that includes adjusting a dynamics model based on observed transition data, assessing the policy utilizing anticipated future states and costs derived from the dynamics model, and then improving the policy.

3-1 Deep Kernel PILCO

We will look at stochastic discrete-time systems described by the following stochastic difference equation:

$$\boldsymbol{x}_{t+1} = f\left(\boldsymbol{x}_t, \boldsymbol{u}_t\right) + \boldsymbol{w}, \quad \boldsymbol{w} \sim \mathcal{N}\left(\mathbf{0}, \boldsymbol{\Sigma}_w\right) \tag{3-1}$$

where $t \in \mathbb{N}$, continuous-valued states $\mathbf{x} \in \mathbb{R}^D$, control actions $\mathbf{u} \in \mathbb{R}^F$, $\boldsymbol{w}$ is i.i.d. Gaussian system noise, and $f : \mathbb{R}^D \times \mathbb{R}^F \to \mathbb{R}^D$ is the unknown function representing the transition dynamics of the system. We assume that all states are fully observable. Our objective is to find a policy (or controller):

$$\pi : \boldsymbol{x} \mapsto \pi(\boldsymbol{x}, \boldsymbol{\theta}) = \boldsymbol{u} \tag{3-2}$$

where $\boldsymbol{\theta}$ are the parameters of the policy which minimizes the expected long-term cost:

$$J^\pi(\boldsymbol{\theta}) = \sum_{t=0}^{T} \mathbb{E}_{\boldsymbol{x}_t}\left[c\left(\boldsymbol{x}_t\right)\right], \quad \boldsymbol{x}_0 \sim \mathcal{N}\left(\boldsymbol{\mu}_0, \boldsymbol{\Sigma}_0\right) \tag{3-3}$$

following a policy π for a finite horizon of T time-steps, with $c\left(\boldsymbol{x}_t\right)$ the associated cost of being in a state $\boldsymbol{x}$ at time-step t. The policy π is a function that is completely parameterized by $\boldsymbol{\theta}$ and maps the states to actions. We aim to find an optimal (or good) policy π^* that results in a minimal value of $J^{\pi^*}(\theta)$, using the initial state distribution $p\left(\mathbf{x}_0\right)$, all while interacting minimally with the system.

3-2 Learning the Model

DKL PILCO uses DKL to model the system's dynamics. The training inputs are the tuples $\left(\boldsymbol{x}_t, \boldsymbol{u}_t\right) \in \mathbb{R}^{D+F}$ and the state differences $\Delta_t = \boldsymbol{x}_{t+1} - \boldsymbol{x}_t \in \mathbb{R}^D$ are used as training targets. This means that the DKL will model the state's change rather than the state's absolute value. Note, that in this case the inputs are: $\tilde{\boldsymbol{x}} := \left[\boldsymbol{x}^\top \boldsymbol{u}^\top\right]^\top$.

Our application of DKL will differ slightly since the Structured Kernel Interpolation (SKI) approach is not used. However, since SKI does not scale well with dimensionality [103], we will not be forced to project the feature space to a lower dimension.

Suppose we have the training inputs, $\tilde{\boldsymbol{X}} = \left[\tilde{\boldsymbol{x}}_1, \ldots, \tilde{\boldsymbol{x}}_n\right]$ and the training targets $\boldsymbol{y} = \left[\Delta_1, \ldots, \Delta_n\right]^\top$. Then, we can define a prior mean function of $m \equiv 0$ and the SE kernel $K : \mathbb{R}^{D+F} \times \mathbb{R}^{D+F} \mapsto \mathbb{R}$:

$$k\left(\tilde{\boldsymbol{x}}_p, \tilde{\boldsymbol{x}}_q\right) = \sigma_f^2 \exp\left(-\frac{1}{2}\left(\tilde{\boldsymbol{x}}_p - \tilde{\boldsymbol{x}}_q\right)^\top \boldsymbol{\Lambda}^{-1}\left(\tilde{\boldsymbol{x}}_p - \tilde{\boldsymbol{x}}_q\right)\right) + \delta_{pq}\sigma_w^2 \tag{3-4}$$

with $\boldsymbol{\Lambda} := \operatorname{diag}\left(\left[\ell_1^2, \ldots, \ell_{D+F}^2\right]\right)$ contain the length-scales, σ_f^2 is the variance of the latent function f, σ_w^2 is the noise variance, and we collect all the SE kernel hyper-parameters in the vector $\boldsymbol{\Omega}$.

Furthermore, we can also define an NN $\Phi : \mathbb{R}^{D+F} \mapsto \mathbb{R}^{D+F}$. Subsequently, the deep kernel is a composition between the NN and the SE kernel:

$$k\left(\tilde{\boldsymbol{x}}_i, \tilde{\boldsymbol{x}}_j \mid \boldsymbol{\Omega}\right) \to k\left(\Phi\left(\tilde{\boldsymbol{x}}_i, \mathbf{w}\right), \Phi\left(\tilde{\boldsymbol{x}}_j, \mathbf{w}\right) \mid \boldsymbol{\Omega}, \mathbf{w}\right) \tag{3-5}$$

where $\Phi(\tilde{\boldsymbol{x}}, \mathbf{w})$ is the NN parameterized by weights $\mathbf{w}$.

The hyper-parameters of the DKL, $\boldsymbol{\Omega}$ and $\mathbf{w}$, are learned jointly through maximum likelihood by using the training inputs $\tilde{\boldsymbol{X}} = [\tilde{\boldsymbol{x}}_1, \dots, \tilde{\boldsymbol{x}}_n]$ and corresponding targets $\boldsymbol{y} = [\Delta_1, \dots, \Delta_n]^\top$. The resulting DKL model will predict one-step ahead, and the predicted state $\boldsymbol{x}_{t+1}$ is Gaussian as well, with [20]:

$$p(\boldsymbol{x}_{t+1} \mid \boldsymbol{x}_t, \boldsymbol{u}_t) = \mathcal{N}(\boldsymbol{x}_{t+1} \mid \boldsymbol{\mu}_{t+1}, \boldsymbol{\Sigma}_{t+1}) \tag{3-6}$$

$$\boldsymbol{\mu}_{t+1} = \boldsymbol{x}_t + \mathbb{E}_f[\boldsymbol{\Delta}_t], \quad \boldsymbol{\Sigma}_{t+1} = \operatorname{var}_f[\boldsymbol{\Delta}_t] \tag{3-7}$$

where the mean and the variance of the DKL predictions are, $\mathbb{E}_f[\boldsymbol{\Delta}_t]$ and $\operatorname{var}_f[\boldsymbol{\Delta}_t]$ [20]:

$$\mathbb{E}_f[\boldsymbol{\Delta}_t] = m_f(\tilde{\boldsymbol{x}}_t) = \boldsymbol{k}_*^\top \left(\boldsymbol{K} + \sigma_w^2 \boldsymbol{I}\right)^{-1} \boldsymbol{y} = \boldsymbol{k}_*^\top \boldsymbol{\beta} \tag{3-8}$$

$$\operatorname{var}_f[\boldsymbol{\Delta}_t] = k_{**} - \boldsymbol{k}_*^\top \left(\boldsymbol{K} + \sigma_w^2 \boldsymbol{I}\right)^{-1} \boldsymbol{k}_* \tag{3-9}$$

where $\boldsymbol{k}_* := k\left(\Phi(\tilde{\boldsymbol{X}}), \Phi(\tilde{\boldsymbol{x}}_t)\right)$, $k_{**} := k(\Phi(\tilde{\boldsymbol{x}}_t), \Phi(\tilde{\boldsymbol{x}}_t))$, and $\boldsymbol{\beta} := (\boldsymbol{K} + \sigma_w^2 \boldsymbol{I})^{-1} \boldsymbol{y}$ where $\boldsymbol{K}$ is the kernel matrix with entries $K_{ij} = k(\Phi(\tilde{\boldsymbol{x}}_i), \Phi(\tilde{\boldsymbol{x}}_j))$.

3-2-1 Batch Normalization

Furthermore, we have opted to use batch normalization layers. Batch normalization is a technique used in deep learning that can improve the performance and stability of neural networks by normalizing each layer's outputs in a mini-batch [48]. In a standard neural network, each layer's input distribution can change during training as the parameters of the previous layers change, which is further amplified as the network becomes deeper [48]. This phenomenon, known as internal covariate shift [90], can slow down training, as each layer must adapt to the new distribution at every step. To be more precise, the term 'internal covariate shift' is used to describe the alteration in the distribution of network activations due to modifications in the network parameters throughout the training process [48]. Batch normalization aims to mitigate this problem by normalizing the activations of each layer (for each mini-batch), effectively stabilizing the distribution of inputs to subsequent layers [48]. Furthermore, it also scales and shifts the normalized activations to retain the layer's representative power [48]. Doing so allows each layer to learn from a more stable distribution of inputs, which can accelerate the training process [5]. Now consider a mini-batch $\mathcal{B}$ of size m. Then the Batch Normalizing Transform is done as follows:

- **Mini-batch mean**:
$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m} \sum_{i=1}^{m} x_i \tag{3-10}$$
- **Mini-batch variance**:
$$\sigma_{\mathcal{B}}^2 = \frac{1}{m} \sum_{i=1}^{m} (x_i - \mu_{\mathcal{B}})^2 \tag{3-11}$$
- **Normalization step**:
$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \tag{3-12}$$

- **Scale and shift**:

$$y_i \leftarrow \gamma \widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \tag{3-13}$$

Batch normalization has several benefits. It allows the use of higher learning rates, making the network more resilient to the initialization of parameters, reduces the need for other regularization methods such as dropout, and often results in both improved performance and faster training times [48].

However, there are also certain considerations when using batch normalization. One is that it is not suited for online learning since it adjusts the test data based on statistics that are heavily influenced by the current task, which can inadvertently lead to catastrophic forgetting, although variations exist which mitigate this issue [79]. Furthermore, the performance is sensitive to the batch size (with small batch sizes resulting in poor performance) and variance in the underlying dataset (large variances causing poor performance) [62].

3-2-2 Implemented NN Architecture

We have used the NN architecture, depicted in Table 3-1, in all experiments.

Parameter	**Value**
Number of hidden layers	3
Number of hidden units per layer	$[32, 64, 16]$
Non-linearity	ReLU and batch-norm

Table 3-1: Architecture of the NN

3-3 Policy Evaluation

The policy has to be evaluated using simulations of the system through the dynamical model to enable optimization. The simulation requires cascading multiple one-step-ahead predictions together to obtain the predictive state distributions $p(\mathbf{x}_1|\pi), \ldots, p(\mathbf{x}_T|\pi)$ [20]. From here on out, the conditioning on the policy π will be left out for notational convenience.

Suppose that trajectories begin from a specified initial state $\boldsymbol{x}_0 \sim \mathcal{N}(\boldsymbol{\mu}_{x_0}, \Sigma_{x_0})$. To predict $\boldsymbol{x}_{t+1}$ from $p(\boldsymbol{x}_t)$, $p(\boldsymbol{\Delta}_t)$ needs to be determined, which requires $p(\tilde{\boldsymbol{x}}_t) = p(\boldsymbol{x}_t, \boldsymbol{u}_t)$ [20]:

$$p(\boldsymbol{\Delta}_t) = \iint p(f(\tilde{\boldsymbol{x}}_t) \mid \tilde{\boldsymbol{x}}_t) p(\tilde{\boldsymbol{x}}_t) \,\mathrm{d}f \;\mathrm{d}\tilde{\boldsymbol{x}}_t \tag{3-14}$$

where the resulting predictive distribution $p(\boldsymbol{\Delta}_t)$ and successor state $p(\boldsymbol{x}_{t+1})$ will generally not be a Gaussian.

In the end, the long-term cost $J^\pi(\boldsymbol{\theta})$ needs to be determined, but that requires the expected cost associated with each state distribution to be evaluated:

$$\mathbb{E}_{\mathbf{x}_t}[c(\mathbf{x}_t)] = \int c(\mathbf{x}_t) p(\mathbf{x}_t) \,\mathrm{d}\mathbf{x}_t \tag{3-15}$$

where $t = 1, \ldots, T$, and the cost function c.

In the original PILCO framework, moment matching was employed to propagate uncertainty. The use of the SE kernel allowed the problem to be analytically solved [19]. However, introducing DKL rendered the previously derived equations for moment matching with an SE kernel, in some cases, unstable. Hence, we have chosen to use the UT, which uses a set of deterministic points. Replacing moment matching with UT opens the avenue for using other functions to parameterize the policy since these methods do not rely on analytical or closed-form solutions [102].

3-3-1 Unscented Transform

The UT, a critical component of the Unscented Kalman Filter (UKF), employs a set of weighted deterministic samples, called sigma points, that share the same mean and covariance as the original distribution and may also have comparable higher-order moments [102]. Their purpose is to approximate the statistics of random variables undergoing a nonlinear transformation allowing for more accurate state and covariance estimation in nonlinear systems [102]. The UT has three parameters, α, β, κ, that can be tuned to achieve the desired performance [74]. The α and κ parameters determine the spread of the sigma points around the mean, whereas the α parameter mainly ensures numerical stability [74]. β incorporates prior knowledge about the distribution of the state (often set to 2 for Gaussian distributions; higher values can accommodate for heavy-tailed distributions) and impacts the weights of the transformed sigma points [74].

Let $\boldsymbol{x} \in \mathbb{R}^D$ be a random variable, with a mean $\boldsymbol{\mu} \in \mathbb{R}^D$, and covariance matrix $\boldsymbol{P} \in \mathbb{R}^{D \times D}$:

$$\begin{aligned} \boldsymbol{\mu} &= \mathbb{E}[\boldsymbol{x}] \\ \boldsymbol{P} &= \mathbb{E}\left[(\boldsymbol{x} - \boldsymbol{\mu})(\boldsymbol{x} - \boldsymbol{\mu})^T\right] \end{aligned} \tag{3-16}$$

Define a set $\mathcal{S}$ which contains $2D + 1$ sigma points and a set of weights $\boldsymbol{w}$, where D is the dimension of the random variable. Then, the sample mean and sample covariance of the nonlinear transformation $f(\boldsymbol{x})$ are determined as follows [102].

1. Calculate $2D + 1$ sigma points according to:

$$\begin{aligned} \mathcal{S}_{[0]} &= \boldsymbol{\mu} & \boldsymbol{w}_0^m &= \tfrac{\lambda}{D+\lambda} & \boldsymbol{w}_0^c &= \tfrac{\lambda}{D+\lambda} + (1 - \alpha^2 + \beta) \\ \mathcal{S}_{[i]} &= \boldsymbol{\mu} + (\sqrt{(D+\kappa)\boldsymbol{P}})_{[i]} & \boldsymbol{w}_i^m &= \boldsymbol{w}_i^c = \tfrac{1}{2(D+\lambda)} \\ \mathcal{S}_{[i+D]} &= \boldsymbol{\mu} - (\sqrt{(D+\kappa)\boldsymbol{P}})_{[i]} & \boldsymbol{w}_{i+D}^m &= \boldsymbol{w}_{i+D}^c = \tfrac{1}{2(D+\lambda)} \end{aligned} \tag{3-17}$$

   for $i \in \{1, \cdots, D\}$, where $(\sqrt{(D+\kappa)\boldsymbol{P}})_{[i]}$ is the ith column of $\sqrt{(D+\kappa)\boldsymbol{P}}$, $\boldsymbol{w}_i$ is the weight corresponding to the ith sigma point and where $\lambda = \alpha^2(D+\kappa) - D$. α determines the spread of the sigma points around the mean, and β is the prior knowledge of the distribution of $\boldsymbol{x}$ (with $\beta = 2$ for Gaussian distributions). κ is a secondary scaling parameter and is typically set to $\kappa = D - 3$ to minimize fourth-order errors [26].

2. Propagate the sigma points through the nonlinear function in order to obtain the transformed sigma points:

$$f\left(\mathcal{S}_{[i]}\right) \tag{3-18}$$

3. Determine the sample mean and sample covariance of the transformed sigma points:

$$\begin{aligned} \boldsymbol{m}_{\mathcal{S}} &= \sum_{i=0}^{2D} \boldsymbol{w}_i^m f\left(\mathcal{S}_{[i]}\right) \\ \boldsymbol{S}_{\mathcal{S}} &= \sum_{i=0}^{2D} \boldsymbol{w}_i^c \left(f\left(\mathcal{S}_{[i]}\right) - \boldsymbol{m}_{\mathcal{S}}\right)\left(f\left(\mathcal{S}_{[i]}\right) - \boldsymbol{m}_{\mathcal{S}}\right)^T \end{aligned} \tag{3-19}$$

Illustrative Example Unscented Transform

In this example, the effectiveness of the UT will be compared against Monte Carlo sampling. Let $\mathbf{X}$ be a random variable distributed according to $\boldsymbol{p} = \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$, where $\boldsymbol{\mu} = \begin{bmatrix} 1.0 & 0.5 \end{bmatrix}^T$ and $\boldsymbol{\Sigma} = \begin{bmatrix} 0.1 & 0 \\ 0 & 0.05 \end{bmatrix}$. The random variables will be propagated through the following non-linear transformation $f(\boldsymbol{x}) = e^{\boldsymbol{x}}$. Then, the following cases are compared against a pseudo-ground truth: MC sampling with $n = 10, 100, 1000$ samples and the UT. The pseudo-ground truth is created by drawing $n = 10^6$ samples from the distribution $\boldsymbol{p}$, propagating those samples, and estimating the sample mean and covariance from those transformed samples.

Method	**Normed Residual Mean**	**Normed Residual Covariance**
MC Sampling $n = 10$	$1.52 \cdot 10^{-1}$	$2.77 \cdot 10^{-1}$
MC Sampling $n = 100$	$1.11 \cdot 10^{-1}$	$1.00 \cdot 10^{-1}$
MC Sampling $n = 1000$	$2.13 \cdot 10^{-2}$	$1.12 \cdot 10^{-1}$
Unscented Transform	$2.19 \cdot 10^{-3}$	$4.31 \cdot 10^{-2}$

Table 3-2: Comparison between the normed residuals of the mean and covariance matrix between MC sampling and UT.

In Table 3-2, the norm of the mean vector and covariance matrix residuals are shown with respect to the pseudo-ground truth. Which is the difference between the predicted and actual values: $\|\mathbf{X}_{\text{truth}} - \mathbf{X}_{\text{prediction}}\|_2$. It can be seen that while the UT only uses 5 points (shown in Figure 3-1), it still manages to approximate the pseudo-ground truth the closest.

Shortcomings of the Unscented Transform

The UT is based on the assumption that the random variables follow a Gaussian distribution [26]. Note that moment matching uses the same assumption. Due to this assumption, the odd-powered moments in the approximation of the true mean and covariance are always zero due to their symmetry. This introduces significant approximation errors in situations where the odd-powered moments of the distribution of x are non-zero, and the transformation is highly nonlinear. Extensions to the UT have been proposed, such as the generalized UT, which addresses this issue [26].

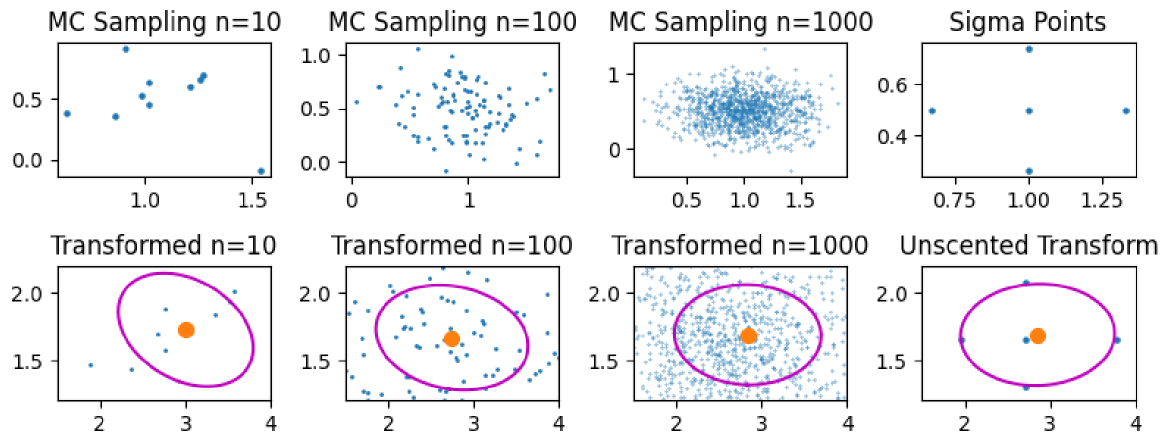


Figure 3-1: Comparison between Monte Carlo sampling and Unscented transform.

The resulting UT implementation for DKL PILCO

Recall that we have defined the training inputs as $\tilde{\boldsymbol{x}} := \left[\boldsymbol{x}^\top \boldsymbol{u}^\top\right]^\top \in \mathbb{R}^{D+F}$, and the training targets as the state differences $\Delta_t = \boldsymbol{x}_{t+1} - \boldsymbol{x}_t \in \mathbb{R}^D$. The full training dataset was defined as $\tilde{\boldsymbol{X}} = [\tilde{\boldsymbol{x}}_1, \ldots, \tilde{\boldsymbol{x}}_n]$ and the training targets $\boldsymbol{y} = [\Delta_1, \ldots, \Delta_n]^\top$. Then, as previously defined, we also have the predictive mean of a test input propagated through the DKL:

$$\mathbb{E}_f\left[\boldsymbol{\Delta}_t\right] = m_f\left(\tilde{\boldsymbol{x}}_t\right) = \boldsymbol{k}_*^\top \left(\boldsymbol{K} + \sigma_w^2 \boldsymbol{I}\right)^{-1} \boldsymbol{y} \tag{3-20}$$

Furthermore, suppose we have an initial state $\boldsymbol{m}_{init} \in \mathbb{R}^D$ and variance $\boldsymbol{S}_{init} \in \mathbb{R}^{D\times D}$. In addition, we assume we have an RBF controller that constrains the control signal between the limits specified by the control environment:

$$\pi : \boldsymbol{x} \mapsto \pi(\boldsymbol{x}, \boldsymbol{\theta}) = \boldsymbol{a}_{\max} \tanh \frac{\mathbf{1}}{\boldsymbol{a}_{\mathbf{max}}} \tilde{\boldsymbol{u}} = \boldsymbol{u} \tag{3-21}$$

where $\tilde{\boldsymbol{u}} \in \mathbb{R}^F$ specifies the unconstrained control action, $\boldsymbol{a}_{max}$ the limits for the controller and $\boldsymbol{u}$ the squashed control action. Then we can define the specific implementation of the UT for DKL PILCO as in Algorithm 3.

Note that similar to PILCO, we also make a Gaussian approximation of the successor state distribution $p\left(\boldsymbol{x}_{t+1}\right)$, as $\mathcal{N}\left(\boldsymbol{x}_{t+1} \mid \boldsymbol{m}_{x_{t+1}}, \boldsymbol{S}_{x_{t+1}}\right)$.

Constraining the control input

In many continuous control tasks, actions must lie within specific bounds. Directly optimizing over these constraints can be challenging, and often squashing functions are used to address this issue [38] [19]. PILCO used the sine function to bound the actions [20]. However, we have used the tanh to constrain the actions. First of all, the behavior of the tanh can be understood more intuitively since the function saturates at its extremes, in contrast to the sine function, which produces oscillatory behavior. Secondly, the tanh derivative is positive and smooth across its domain, in contrast to the sine function, which oscillates and can change signs.

Algorithm 3 Unscented Transform

```
1: Initialize: Set $t = 0$, $\boldsymbol{m}_{x_0} = \boldsymbol{m}_{init}$, $\boldsymbol{s}_{x_0} = \boldsymbol{S}_{init}$, UT parameters for DKL $[\alpha_1, \beta_1, \kappa_1]$ and policy $[\alpha_2, \beta_2, \kappa_2]$
2: repeat
3:     Initialize $2D+1$ sigma points, $\mathcal{S}$,using $\mathcal{N}(\boldsymbol{m}_{x_t}, \boldsymbol{s}_{x_t})$, and calculate the weights.
4:     Propagate samples through the controller.
5:     Estimate mean weighted mean $\boldsymbol{m}_{u_t}$, weighted covariance $\boldsymbol{S}_{u_t}$ and weighted input-output covariance $\boldsymbol{C}_{xu_t}$ from the propagated samples.
```

$$\begin{aligned}
\boldsymbol{m}_{u_t} &= \sum_{i=0}^{2D} \boldsymbol{w}_i^m \pi\left(\mathcal{S}_{[i]}, \boldsymbol{\theta}\right) \\
\boldsymbol{S}_{u_t} &= \sum_{i=0}^{2D} \boldsymbol{w}_i^c \left(\pi\left(\mathcal{S}_{[i]}, \boldsymbol{\theta}\right) - \boldsymbol{m}_{u_t}\right)\left(\pi\left(\mathcal{S}_{[i]}, \boldsymbol{\theta}\right) - \boldsymbol{m}_{u_t}\right)^T \\
\boldsymbol{C}_{xu_t} &= \sum_{i=0}^{2D} \boldsymbol{w}_i^c \left(\mathcal{S}_{[i]} - \boldsymbol{m}_{x_t}\right)\left(\pi\left(\mathcal{S}_{[i]}, \boldsymbol{\theta}\right) - \boldsymbol{m}_{u_t}\right)^T
\end{aligned} \tag{3-22}$$

```
6:     Determine full mean $\tilde{\boldsymbol{m}}_t \in \mathbb{R}^{D+F}$ and covariance $\tilde{\boldsymbol{S}}_t \in \mathbb{R}^{(D+F)\times(D+F)}$.
```

$$\tilde{\boldsymbol{m}}_t = [\boldsymbol{m}_{x_t}, \boldsymbol{m}_{u_t}] \tag{3-23}$$

$$\tilde{\boldsymbol{S}}_t = \begin{bmatrix} \boldsymbol{s}_{x_t} & \boldsymbol{s}_{x_t}\boldsymbol{C}_{xu_t} \\ \boldsymbol{C}_{xu_t}^T \boldsymbol{s}_{x_t}^T & \boldsymbol{S}_{u_t} \end{bmatrix} \tag{3-24}$$

```
7:     Initialize $2(D+F)+1$ sigma points using $\mathcal{N}\left(\tilde{\boldsymbol{m}}_t, \tilde{\boldsymbol{S}}_t\right)$, and calculate the weights.
8:     Propagate samples through the dynamics model.
9:     Estimate weighted mean $\boldsymbol{m}_{\Delta_{t+1}}$, weighted covariance $\boldsymbol{s}_{\Delta_{t+1}}$ and weighted input-output covariance $\boldsymbol{c}_{\Delta u_{t+1}}$ from the propagated samples.
```

$$\begin{aligned}
\boldsymbol{m}_{\Delta_{t+1}} &= \sum_{i=0}^{2D} \boldsymbol{w}_i^m m_f\left(\mathcal{S}_{[i]}\right) \\
\boldsymbol{s}_{\Delta_{t+1}} &= \sum_{i=0}^{2D} \boldsymbol{w}_i^c \left(m_f\left(\mathcal{S}_{[i]}\right) - \boldsymbol{m}_{\Delta_{t+1}}\right)\left(m_f\left(\mathcal{S}_{[i]}\right) - \boldsymbol{m}_{\Delta_{t+1}}\right)^T \\
\boldsymbol{C}_{\Delta u_{t+1}} &= \sum_{i=0}^{2D} \boldsymbol{w}_i^c \left(\mathcal{S}_{[i]} - \tilde{\boldsymbol{m}}_t\right)\left(m_f\left(\mathcal{S}_{[i]}\right) - \boldsymbol{m}_{\Delta_{t+1}}\right)^T
\end{aligned} \tag{3-25}$$

```
10:    Calculate $\boldsymbol{m}_{x_{t+1}}$ and $\boldsymbol{s}_{x_{t+1}}$, where:
```

$$\begin{aligned}
\boldsymbol{m}_{x_{t+1}} &= \boldsymbol{m}_{x_t} + \boldsymbol{m}_{\Delta_{t+1}} \\
\boldsymbol{s}_{x_{t+1}} &= \boldsymbol{s}_{x_t} + \boldsymbol{s}_{\Delta_{t+1}} + \begin{bmatrix} \boldsymbol{s}_{x_t} & \boldsymbol{s}_{x_t}\boldsymbol{C}_{xu_t} \end{bmatrix} \boldsymbol{C}_{\Delta u_{t+1}} + \boldsymbol{C}_{\Delta u_{t+1}}^T \begin{bmatrix} \boldsymbol{s}_{x_t} & \boldsymbol{s}_{x_t}\boldsymbol{C}_{xu_t} \end{bmatrix}^T
\end{aligned} \tag{3-26}$$

```
11:    $t \leftarrow t+1$
12: until Planning horizon $T$ is reached.
```

Cost Function

The DKL PILCO framework will use a decaying reward function, which depends solely on a geometric distance d between the current state and the target state [19]:

$$c(\mathbf{x}) = \exp\left(-\frac{1}{2a^2} d\left(\mathbf{x}, \mathbf{x}_{\text{target}}\right)^2\right) \tag{3-27}$$

where $\frac{1}{2a^2}$ is a decay rate for the specific dimension of the target goal, high values for a can make the reward decay faster and result in a sparser reward signal. However, setting a too high can impede learning since it will make exploration harder. On the other hand, setting a too low can result in a very wide reward signal and, consequently, uninformative gradients, which can also impede the learning process.

The expected immediate cost can be calculated according to the following [20]:

$$\mathbb{E}_{\mathbf{x}}[c(\mathbf{x})] = \int c(\mathbf{x}) p(\mathbf{x}) \mathrm{d}\mathbf{x} = \int \exp\left(-\frac{1}{2}\left(\mathbf{x} - \mathbf{x}_{\text{target}}\right)^\top \mathbf{T}^{-1}\left(\mathbf{x} - \mathbf{x}_{\text{target}}\right)\right) p(\mathbf{x}) \mathrm{d}\mathbf{x} \tag{3-28}$$

where $\mathbf{T}^{-1}$ is the precision matrix of $p(\boldsymbol{x})$. If the input vector $\boldsymbol{x}$ has the same representation as the target, then $\boldsymbol{T}^{-1}$ is a diagonal matrix with entries either unity or zero, scaled by the decay factors $\frac{1}{2a^2}$. Recall that we made a Gaussian approximation for the successor state distribution, thus for $\mathbf{x} \sim \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$ this results in the following expected immediate cost [20]:

$$\mathbb{E}_{\mathbf{x}}[c(\mathbf{x})] = \left|\mathbf{I} + \boldsymbol{\Sigma}\mathbf{T}^{-1}\right|^{-1/2} \exp\left(-\frac{1}{2}\left(\boldsymbol{\mu} - \mathbf{x}_{\text{target}}\right)^\top \tilde{\mathbf{S}}_1\left(\boldsymbol{\mu} - \mathbf{x}_{\text{target}}\right)\right) \tag{3-29}$$

where $\tilde{\mathbf{S}}_1 := \mathbf{T}^{-1}\left(\mathbf{I} + \boldsymbol{\Sigma}\mathbf{T}^{-1}\right)^{-1}$.

The total expected long-term cost J^π is simply calculated by evaluating Equation 3-29 for each $t = 1, \ldots, T$.

$$J^\pi(\boldsymbol{\theta}) = \sum_{t=0}^{T} \mathbb{E}_{\boldsymbol{x}_t}\left[c\left(\boldsymbol{x}_t\right)\right], \quad \boldsymbol{x}_0 \sim \mathcal{N}\left(\boldsymbol{\mu}_0, \boldsymbol{\Sigma}_0\right) \tag{3-30}$$

3-4 Policy Optimization

In the original PILCO framework, the derivative calculations involved in the optimization process were achieved through hard-coded gradients. While this approach has its merits, it requires considerable manual computation and coding effort. For optimization, PILCO initially employed the Limited-Memory BFGS (L-BFGS) algorithm, a quasi-Newton method known for its effectiveness in solving large-scale optimization problems. L-BFGS, while powerful, has its limitations, including the requirement for computation and storage of a Hessian matrix or an approximation thereof [12]. This necessity can significantly increase the computational overhead, especially for problems with many parameters [12].

We have chosen to use Automatic Differentiation (auto-diff) and the Adaptive Moment Estimation (Adam) optimizer. Auto-diff provides a mechanism to compute derivatives accurately and efficiently, sidestepping the manual calculation required for hard-coded gradients [37]. At its core, auto-diff applies the chain rule of calculus to compute the derivatives of functions whose analytic expressions are known [37]. It is not numerical differentiation, which approximates derivatives by finite differences, nor is it symbolic differentiation. This technique leverages symbolic rules for differentiation, offering higher accuracy than finite difference approximations [39]. Distinct from a wholly symbolic strategy, auto-diff performs numerical evaluations early in the calculations, sidestepping the need for extensive symbolic computations. Hence, auto-diff computes derivatives at specific numerical values without building symbolic expressions for the derivatives [65].

There are two main variants of auto-diff, forward mode, and reverse mode, each with its specific applications. Forward mode auto-diff computes derivatives from the 'inside out,' propagating derivative information from inputs to outputs [37]. On the other hand, reverse mode auto-diff, the method employed by backpropagation in neural networks, computes derivatives from the 'outside in,' propagating from outputs back to inputs, leveraging computational graphs to apply this process to complex functions [39].

The central advantage of auto-diff is its ability to compute exact derivatives efficiently, regardless of the complexity of the function [65]. This stands in contrast to numerical differentiation, which can suffer from numerical instability and inaccuracy, and symbolic differentiation, which can lead to long redundant expressions [65]. Furthermore, auto-diff can mitigate the risk of human errors, which is a risk in hard-coded gradients. Finally, auto-diff reduces maintenance. If the function which has to be differentiated changes, then auto-diff handles these changes automatically [37]. The hard-coded gradients would have to be recomputed and recoded.

However, there are some considerations when choosing auto-diff. Auto-diff, particularly in its reverse mode, can be memory-intensive as it needs to store intermediate values for backpropagation [47]. Hard coding might be preferable if memory is a significant concern and the function's gradients are relatively straightforward to compute. Auto-diff tools generally provide an efficient computation of gradients. For elementary functions, a hard-coded gradient could potentially be faster.

The Adam optimizer is a first-order gradient-based optimization algorithm [56]. It has become a popular choice for training deep learning models due to its efficiency and performance across a variety of tasks and architectures [7]

Adam operates by computing adaptive learning rates for different parameters. It achieves this by maintaining a moving average of the gradients (the first moment) and the square of the gradients (the second raw moment) [56]. By keeping track of these moment estimates, Adam adjusts the learning rate for each weight in the model individually, according to the magnitude of their gradients. This property makes Adam particularly well-suited to sparse data and large-scale problems [7].

Another key characteristic of Adam is its incorporation of the momentum principle. By maintaining an exponentially decaying average of past gradients, Adam introduces a form of momentum that can help mitigate the effects of noisy gradient information and accelerate convergence toward the minimum of the loss function [84].

Algorithm 4 The Adam optimization algorithm

```
1:  input: $\gamma$ (learning rate),
2:  $\beta_1$, $\beta_2$ (exponential decay rates moment estimates),
3:  $f(\theta)$ (objective function with parameters $\theta$)
4:  initialize: $\theta_0$ (parameter vector),
5:  $m_0 \leftarrow 0$ ($1^{st}$ moment vector),
6:  $v_0 \leftarrow 0$ ($2^{nd}$ moment vector),
7:  $t \leftarrow 0$ (timestep)
8:  repeat
9:      $t \leftarrow t + 1$
10:     $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$    ▷ Gradient w.r.t. objective function at $t$
11:     $m_t \leftarrow \beta_1 m_{t-1} + (1 - \beta_1) g_t$    ▷ Update biased first moment estimate
12:     $v_t \leftarrow \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$    ▷ Update biased second raw moment estimate
13:     $\widehat{m_t} \leftarrow m_t / (1 - \beta_1^t)$    ▷ Compute bias-corrected first moment estimate
14:     $\widehat{v_t} \leftarrow v_t / (1 - \beta_2^t)$    ▷ Compute bias-corrected second raw moment estimate
15:     $\theta_t \leftarrow \theta_{t-1} - \gamma \widehat{m_t} / (\sqrt{\widehat{v_t}} + \epsilon)$    ▷ Update parameters
16: until the maximum number of steps has been reached or $\theta_t$ has converged.
17: return $\theta_t$
```

Finally, Adam has the advantage of requiring less initial learning rate tuning than other optimization methods. Its default settings (as suggested by Kingma et al. [56]) often provide a good starting point: $\gamma = 1e - 3$, $\beta_1 = 0.9$, $\beta_2 = 0.999$, and $\varepsilon = 1e - 8$. Note that ε is a constant that helps to avoid division by zero when computing the update step [56].

In summary, the Adam optimizer combines elements of other optimization algorithms to provide a tool that is efficient, effective and generally well-suited to the demands of deep learning. Its adaptive learning rate and momentum make it robust to various data scales and conditions, and its ease of use further contributes to its popularity in the field.

Nevertheless, there have been cases where Adam failed to converge, e.g., online learning, large output spaces, or a critical point in a nonconvex setting [84]. Extensions to Adam have been proposed, e.g., AMSGrad, where the key difference is in maintaining the maximum of all v_t and using it to normalize the gradient's running average [84]. However, we will use the original Adam implementation since further research and practical implementations have found mixed results on the performance of AMSGrad compared to Adam [98].

3-5 Overview of DKL PILCO

Algorithm 5 shows an overview of the proposed DKL PILCO framework. Note that the kernel matrix $\mathbf{K}$ is based on the mini-batch during the model learning part. Hence, at the end of the model learning part, the training data contained within the DKLs are reset with the full data $\tilde{\mathbf{X}}$ and $\mathbf{y}$.

Algorithm 5 DKL PILCO

```
init policy $\pi(\theta)$, cost function $c(\mathbf{x})$, deep kernels $k(\Phi(\cdot,\mathbf{w}),\Phi(\cdot,\mathbf{w}))$, number of DKL optimization steps $N_{DKL}$, number of policy optimization steps $N_{opt}$, batch size $m$, learning rate for DKL models $\gamma_1$, learning rate for the policy $\gamma_2$, UT parameters for DKL [$\alpha_1$, $\beta_1$, $\kappa_1$], UT parameters for the policy [$\alpha_2$, $\beta_2$, $\kappa_2$], planning horizon $T$

Apply random control signals to the system, and record data into $\tilde{\mathbf{X}}$ and $\mathbf{y}$.

while task not learned do
    1) Model Learning:
    Divide $\tilde{\mathbf{X}}$ and $\mathbf{y}$ into mini-batches of size $m$, resulting in $B$ mini-batches with $\tilde{\mathbf{X}}_k$ and $\mathbf{y}_k$ denoting each mini-batch
    for $j = 1, ..., N_{DKL}$ do
        for $k = 1, ..., B$ do
            Calculate the predictions of the DKL models $\mathbf{y}_{\text{pred}}$
            Calculate the marginal log-likelihood $\mathcal{L}(\mathbf{y}_{\text{pred}}, \mathbf{y}_k)$
            Compute $-\nabla_{[\mathbf{\Omega},\mathbf{w}]}\mathcal{L}(\mathbf{\Omega}_k, \mathbf{w}_k)$ by performing backpropagation using Adam.
            Update parameters $\mathbf{\Omega}_{k+1} \leftarrow \mathbf{\Omega}_{k+1} - \gamma_1 \cdot \widehat{m}_t / \left(\sqrt{\widehat{v}_t} + \epsilon\right)$
            Update parameters $\mathbf{w}_{k+1} \leftarrow \mathbf{w}_{k+1} - \gamma_1 \cdot \widehat{m}_t / \left(\sqrt{\widehat{v}_t} + \epsilon\right)$
        end for
    end for reset the training data DKL $\leftarrow (\tilde{\mathbf{X}}, \mathbf{y})$
    2) Policy Optimization:
    Initialize state $\mathbf{m}_{init}$ and covariance $\mathbf{S}_{init}$.
    for $i = 1, ..., N_{opt}$ do
        Perform multi-step ahead prediction using the DKL, UT,
        and $\pi(\theta_i)$ until $T$ is reached.
        Compute $\hat{J}(\theta_i)$ using the previously computed multi-step ahead predictions.
        Compute $\nabla_\theta \hat{J}(\theta_i)$ and perform backpropagation using Adam.
        Update parameters $\theta_{i+1} \leftarrow \theta_i - \gamma_2 \cdot \widehat{m}_t / \left(\sqrt{\widehat{v}_t} + \epsilon\right)$.
    end for set $\pi^* \leftarrow \pi(\theta^*)$
    3) Policy Execution:
    Apply $\pi^*$ to the system, record data and append to $\tilde{\mathbf{X}}$ and $\mathbf{y}$
end while
return final policy $\pi^*$ and learned DKL models
```

Chapter 4

Experiments

In this chapter, we focus on a comprehensive comparison between the traditional Probabilistic Inference for Learning COntrol (PILCO) framework and our extended framework, Deep Kernel PILCO (DKL PILCO). Both frameworks have unique characteristics and aspects, and our aim is to highlight the comparative strengths and potential weaknesses of each.

DKL PILCO extends the original PILCO framework by incorporating a Neural Network (NN) for feature extraction within the Gaussian Process (GP) model and using the Unscented Transform (UT) for uncertainty propagation. These modifications have been designed with the aim of refining the prediction accuracy in complex control problems and improving the learning process.

Our comparative analysis will examine how both frameworks manage parameter variations, respond to noise and disturbances, and review overall performance metrics and the efficiency of their learning processes.

This chapter aims to provide valuable insights into the impact of our modifications on the performance of the traditional PILCO framework. By the end of this chapter, readers should have a clear understanding of the potential advantages and any potential trade-offs that our DKL PILCO framework might present.

Through a methodical comparison, we hope to contribute essential research for future applications and inspire further developments within the DKL PILCO framework across a multitude of control system scenarios.

4-1 Experimental Settings

4-1-1 Experimental Set-Up

The experiments were all run on Windows 11 and in Python v3.9.10. In addition, we used the following libraries: PyTorch v1.12.0, GPyTorch v1.8.0, GPflow v2.6.0, NumPy v.1.22.4, Gym v.0.26.2, Tensorflow v.2.11.0. In addition, the tests were all run on the same system with the following specifications: a Ryzen 9 5900HX CPU (8 cores and 16 threads, with a base clock speed of 3.3 GHz), 16GB DDR4 RAM, and an NVIDIA RTX 3060 GPU. However, the CUDA cores were not used during training or testing. Furthermore, both frameworks will use an RBF network to parameterize the policy using the same amount of parameters and parameter initialization.

4-1-2 Experiments

We tested our framework on the Mountain Car problem without any modifications. Except for the following: we train the framework on a closed-form reward function, similarly to [19]. However, we evaluate the framework on the original reward function. The presented results are averaged over ten random seeds. We tried each seed five times for a more precise policy evaluation at each stage and calculated the mean. We test the framework on three aspects:

- **Rewards obtained**: the average reward obtained indicates the found policy's quality. A higher reward is better.
- **Data efficiency**: data efficiency is defined as the number of random trials needed to converge to a successful policy. A lower number is better.
- **Predictive performance**: is measured by comparing the Mean Absolute Error (MAE) and Mean Squared Error (MSE) for the one-step ahead predictions. Furthermore, Quantile-Quantile (Q-Q) plots will be compared to validate the normality of the residuals. Finally, the numerical stability and the accuracy of the multi-step ahead predictions will be compared.

Furthermore, we also tested our framework on the swimmer problem. Unfortunately, both frameworks failed to converge on the swimmer problem. But we will still compare their predictive performance for both one- and multi-step ahead predictions.

4-1-3 Numerical Precision

One of the issues encountered when using moment matching was the numerical imprecision and the propagation of errors throughout the calculations. As a result, the covariance matrix may lose two of its essential properties: symmetry and positive definiteness. The numerical stability was improved through the use of higher precision floats.

PyTorch, an often-used Python library for deep learning applications, stores numeric values in 32-bit floats by default and usually occupies 32-bits in computer memory. A 32-bit float, also known as a single-precision floating-point format, is a way to represent real numbers in binary form that can accommodate a wide range of values. In decimal terms, this precision is equivalent to approximately 7.225 decimal digits (since $\log_{10}(2^{24}) \approx 7.225$). Thus, about 7 to 8 decimal digits (before and after the decimal point) in any number stored as a 32-bit float will be accurately represented. Any number beyond the precision limit of a 32-bit float will be rounded.

On the other hand, double-precision floats occupy 64 bits in computer memory to achieve a higher significand precision, resulting in a precision of $\log_{10}(2^{53}) \approx 15.955$ digits. High precision can be essential for both PILCO and DKL PILCO since both models use multi-step ahead predictions in their decision-making, and rounding errors will then be propagated in the calculations. For example, the rounding errors can accumulate and sometimes result in a covariance matrix that is no longer symmetric or positive definite, or ill-conditioned. When the covariance matrix becomes ill-conditioned due to rounding errors, it can cause numerical instability in subsequent computations.

However, using 64-bit floats does have its limitations. First, 64-bit floats take up twice as much memory as 32-bit floats [86]. This can be a concern when working with large data sets, and memory footprint is a limiting factor. Furthermore, graphics processing units (GPUs) are usually optimized for 32-bit operations since these are most common in video games and graphics [86]. This could limit further scalability since most machine learning models are more efficiently trained on GPUs due to more considerable parallel processing capabilities and higher memory bandwidth [86].

4-2 Mountain Car

The Mountain Car problem involves an under-powered car randomly situated (between $-0.6m$ and $-0.4m$) at the bottom of a valley shaped like a sine wave set with an initial velocity of $0.0\frac{m}{s}$, as depicted in Figure 4-1. The car's potential actions are accelerations, which can be executed in either direction. The objective is to use strategic accelerations to reach the goal state, positioned atop the right hill (at a position of $0.45m$) [97].

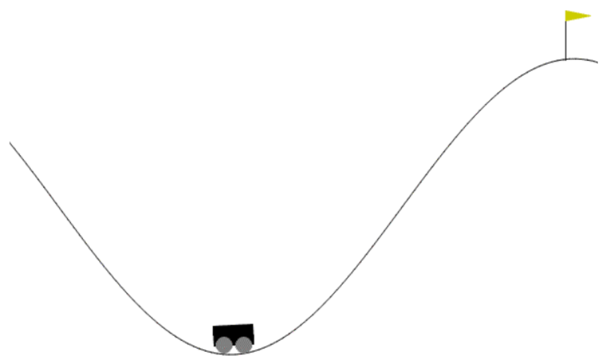


Figure 4-1: The Mountain Car problem as implemented by OpenAI gym

The state space of the mountain car is two-dimensional and specifies the car's position and velocity, depicted in Table 4-1.

State	**Min**	**Max**	**Unit**
Position of the car along the x-axis	-1.2	0.6	m
Velocity of the car	-0.07	0.07	$\frac{m}{s}$

Table 4-1: State space of the mountain car.

The action space of the mountain car is one-dimensional and specifies the force applied to the car. In this case, the control action can take continuous in the specified range in Table 4-2. Negative values will eventually move the car to the left, while positive values will move the car to the right. Note that the force is multiplied with a scaling factor of 0.0015 in the actual dynamics. Furthermore, the velocity will be set to 0 if a collision occurs at either end of the environment [97]. The governing equations for the dynamics of the Mountain Car are defined as follows:

$$\begin{aligned} s_{t+1} &= s_t + v_t \\ v_{t+1} &= v_t + U_t \cdot 0.0015 - 0.0025 \cdot \cos(3 \cdot s_t) \end{aligned} \tag{4-1}$$

where s_t is the position of the car at time t, v_t the velocity of the car and U_t is the control action at time t.

Action	**Min**	**Max**	**Unit**
Force applied to the car	-1	1	N

Table 4-2: Action space of the mountain car.

The reward signal of the environment consists of two components:

- **Running cost:** Large actions are penalized, for every timestep there is $-0.1u^2$.
- **Terminal reward:** If the mountain car reaches the goal, a positive reward of $+100$ is added to the negative reward for that time step, and the episode will terminate.

Note that we will use a different reward signal for the policy optimization part since the environment's reward signal is not differentiable. Instead, we use the goal position as the reward signal, where the following parameters are specified:

$$\mathbf{x}_t = \begin{bmatrix} 0.5 & 0 \end{bmatrix} \quad \mathbf{W}_t = \begin{bmatrix} 0.5 & 0 \end{bmatrix} \tag{4-2}$$

where $\mathbf{x}_t$ parameterizes the goal state and $\mathbf{W}_t$ the decay rates.

At first glance, this problem might seem trivial to solve. However, finding a successful policy can be challenging [60]. Two key aspects drive this complexity. Firstly, the environment exhibits adversarial dynamics - the environmental conditions, for example, gravity and slope, work against the car's motion toward the goal, limiting the likelihood of reaching the target state through random exploration. Secondly, the car's engine lacks the strength to accelerate directly from its starting position to the goal. An effective policy would initially steer the car left to gather momentum, then utilize this momentum to propel the car towards the target [60].

Figure 4-2 demonstrates the difficulty of finding a successful policy for the mountain car problem. Twin Delayed Deep Deterministic Policy Gradients (TD3) and Soft Actor-Critic (SAC) are both considered to be state-of-the-art Reinforcement Learning (RL) algorithms [30] [38], but they needed 300k and 50k time steps to find a successful policy [82]. However, both PILCO and DKL PILCO needed far fewer environmental interactions; both algorithms needed less than 1000 steps, with DKL PILCO requiring fewer interactions than PILCO (240 vs. 470-time steps, on average). Table 4-3 compares the average number of random trials needed and the spread between DKL PILCO and PILCO. The increased data efficiency of DKL PILCO could be attributed to a more accurate dynamics model and the possibility that the UT is handling the non-linearities occurring in the system better than moment matching. The combined effect might result in more accurate multi-step ahead predictions (which will be discussed later in the chapter) and, thus, fewer interactions needed to learn a successful policy.

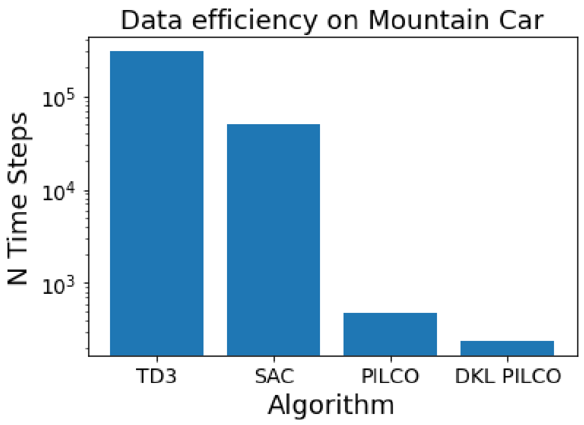


Figure 4-2: Time steps needed until a successful policy is found. The benchmark scores of TD3 and SAC were obtained from a publicly available baseline [82]. Note that the y-axis is on a log scale.

	Average random trials	**Standard deviation**
PILCO	4.7	0.95
DKL PILCO	2.4	1.07

Table 4-3: Comparison of data efficiency

In Figure 4-3, the average reward is plotted against the iteration. An iteration is defined as a policy optimization cycle. The shaded area is the standard deviation between the results of the random seeds. Note that the significant standard deviation for PILCO is caused by two random seeds that were unsuccessful in finding a good policy. On the other hand, DKL PILCO immediately found a successful policy over all ten random seeds after the first iteration. Furthermore, the policy it found was not forgotten in subsequent iterations, while the learning rate for the policy optimization part was kept constant, highlighting the local approximation benefit of Radial Basis Function (RBF) networks. Table 4-4 shows a comparison between PILCO and DKL PILCO for the average reward found after four iterations and the spread.

The observed data provide strong evidence that DKL PILCO exhibits a significant improvement in data efficiency and reward acquisition compared to its counterpart, PILCO, on the

	Average Final Reward	**Standard deviation**
PILCO	74.4	39.6
DKL PILCO	95.1	0.72

Table 4-4: Comparison of average reward after four iterations

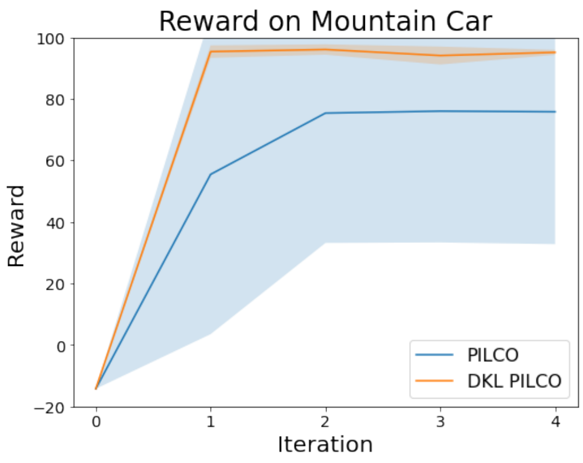


Figure 4-3: This plot depicts the average reward (over ten random seeds) versus the iterations. An iteration is defined as a policy optimization cycle.

mountain-car problem.

4-2-1 Policy Evaluation

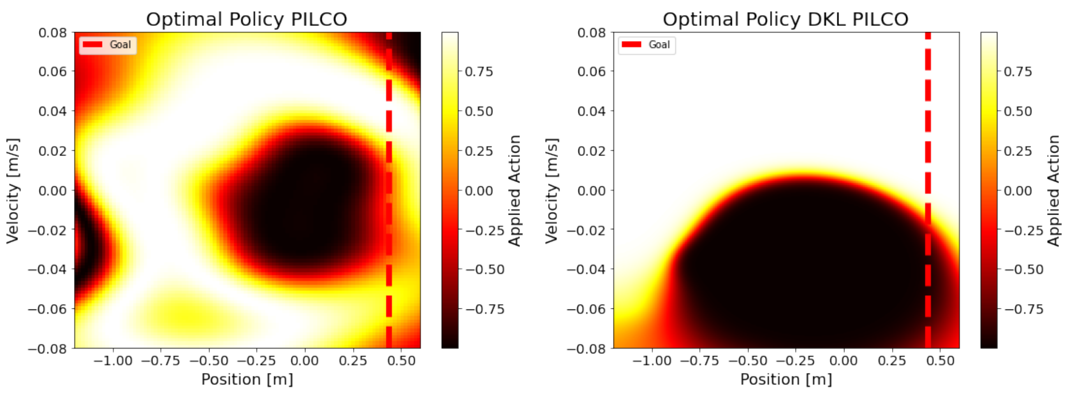
This subsection evaluates the robustness of the RBF controllers originating from both PILCO and DKL PILCO. This evaluation aims to objectively assess these controllers' robustness and quantify the effects of the modifications introduced in the DKL PILCO framework.

Robustness, defined as a system's ability to maintain performance despite uncertainties, disturbances, or parameter variations, is a crucial metric in this analysis [100]. We present a comparative study of the RBF controllers' robustness derived from the established PILCO framework and our extended version DKL PILCO. This comparison allows for an objective identification of each controller's strengths and potential weaknesses, contributing to a more complete understanding of their capabilities. Note that all the presented results in this section, unless explicitly stated, have been obtained on a single random seed but tested on 1000 episodes.

Through this robustness analysis, we seek to quantify and understand how the adaptations in our extended framework contribute to improving policy performance and robustness. We aim to present a balanced discussion, outlining the enhanced capabilities and any potential trade-offs that may arise.

Performance of the policy

In this scenario, we will look at the policy obtained by PILCO and DKL PILCO when the car has an initial position between $-0.1m$ and $0.1m$. Figure 4-4 displays both policies, with PILCO on the left and DKL PILCO on the right. PILCO shows a policy where the car will accelerate to the left for a position roughly between $-0.3m$ and $0.25m$ and velocity between $-0.04\frac{m}{s}$ and $0.02\frac{m}{s}$. DKL PILCO's policy looks significantly different, where the car will almost always accelerate to the left, depending if the velocity is small enough. Comparing both policies, PILCO's policy looks more complex than DKL PILCO, which might result in worse generalization.



(a) Optimal policy obtained by PILCO (b) Optimal policy obtained by DKL PILCO

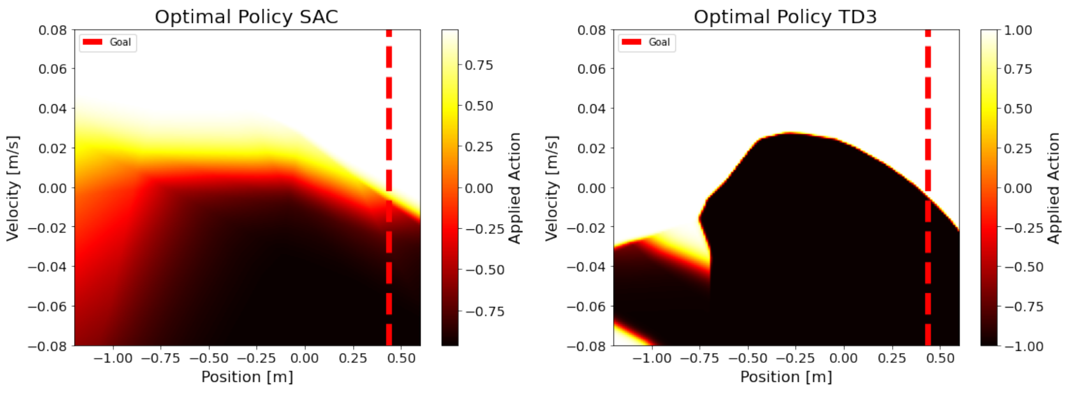
Figure 4-4: Comparison of policies obtained by PILCO and DKL PILCO

Figure 4-5 depict the optimal policies derived from SAC and TD3 (with readily trained agents [82]). Note that SAC shows a smoother policy than TD3, which can be attributed to the entropy regularization used by SAC [38]. Entropy regularization encourages the agent to explore a broad range of actions, which can lead to smoother transitions between the actions [38]. Interestingly, DKL PILCO's policy looks somewhat similar to SAC's, aside from the region where a small amount of control input is used towards the left end (which is larger for SAC). However, DKL PILCO only used a total of 640-time steps to converge to the shown policy in Figure 4-4, while SAC needed around 50k time steps [82].

When comparing the average rewards obtained and the spread (shown in Table 4-5), both policies seem to perform similarly and do not show any practical significant difference.

	Average reward	**Standard deviation**
PILCO	94.13	0.234
DKL PILCO	94.18	0.211

Table 4-5: Comparison of average reward obtained for a single controller after five iterations

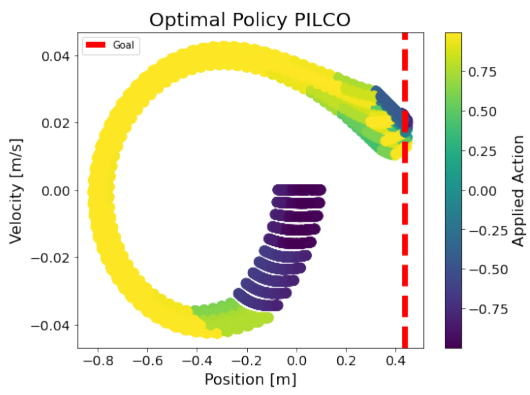


(a) Optimal policy obtained by SAC (b) Optimal policy obtained by TD3

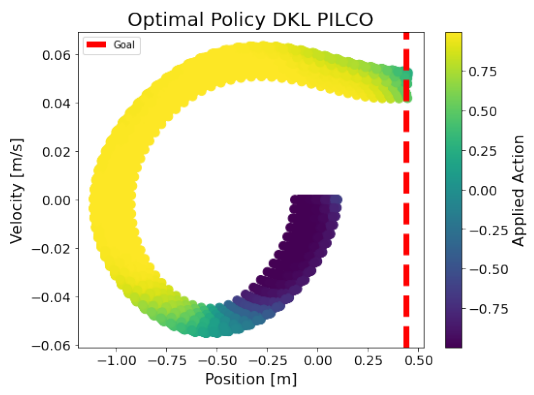
Figure 4-5: Comparison of policies obtained by SAC and TD3

In Figure 4-6, the phase portrait of trajectories obtained starting from an initial position around 0.0m are shown in the phase plane. A trajectory is successful if it ends at the goal line, depicted by the dashed line at 0.45m. Furthermore, the actions are shown as colors, where dark blue corresponds to an action of -1.0 and bright yellow an action of 1.0.

Both frameworks seem to handle slight perturbations on the initial state well since the trajectories depicted in both phase portraits converge to the goal. Interestingly, PILCO reaches the goal position with a lower velocity than DKL PILCO. Specific trajectories also show that the agent only applies a small amount of control action for a sufficient velocity and a position near the goal, indicated by the parts in green. DKL PILCO seems to keep using the maximum or near maximum amount of control action allowed, even near the goal and with a high velocity. Furthermore, it seems that DKL PILCO pulls the car further back to the left before accelerating towards the goal. In this specific case, it seems that PILCO has obtained a more efficient policy since the objective of the environment is to reach the goal and do this with as little control action as possible. However, this was not encoded in our reward signal,



(a) Optimal policy obtained by PILCO



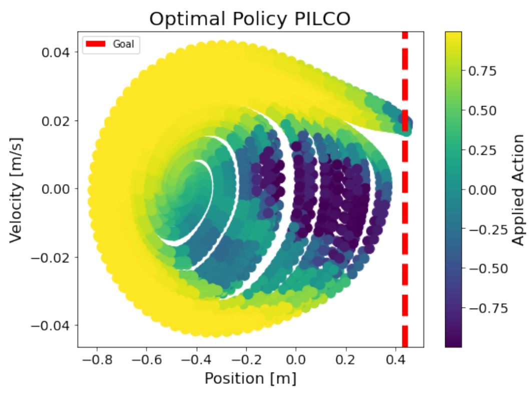
(b) Optimal policy obtained by DKL PILCO

Figure 4-6: Comparison of policies obtained by PILCO and DKL PILCO

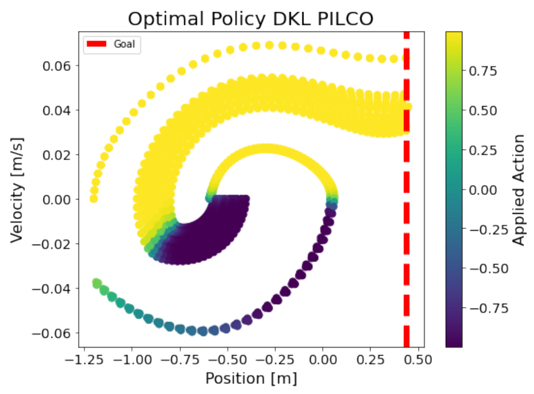
which only took the geometric distance between the goal and the car's position.

Disturbance on the initial state

Figure 4-7 depicts the scenario where the initial position of the car is significantly disturbed. Instead of starting between $-0.1m$ and $0.1m$, the initial position is now between $-0.6m$ and $-0.4m$, corresponding to an initial position at the bottom of the valley.



(a) Optimal policy obtained by PILCO



(b) Optimal policy obtained by DKL PILCO

Figure 4-7: Comparison of policies obtained by PILCOand DKL PILCO

In this case, PILCO struggles to reach its goal. A significant amount of trajectories end up trapped at the bottom of the valley. PILCO seems to try the same strategy as before (Figure 4-6). However, in this case, PILCO seems to prematurely lower the control input resulting in a velocity that is too low to overcome the steep hill near the goal.

On the other hand, DKL PILCO is still able to reach the goal and achieves a similar success rate as in Figure 4-6, suggesting that the agent has learned a policy that generalizes better compared to PILCO. In the previous situation, it seemed wasteful of the agent to reach the goal with a high velocity; in this case, it mitigates the initial state's disturbance and results in a more robust policy with respect to disturbance in the initial state. Note that while the leftmost trajectory seems to have failed, it actually does succeed and continues from velocity $0.0\frac{m}{s}$ onwards. In this specific case, the velocity has been reset to 0 due to the car hitting the wall. Interestingly, this trajectory is just on the boundary between full acceleration to the right or left (shown in Figure 4-8), resulting in the car almost immediately accelerating but then coming back down since it does not have enough momentum. However, after falling back down, the car accelerates to the leftmost boundary and finally reaches the goal, albeit with a slight detour. While this might not have been the most optimal route, it results in a robust policy with no failed trajectories.

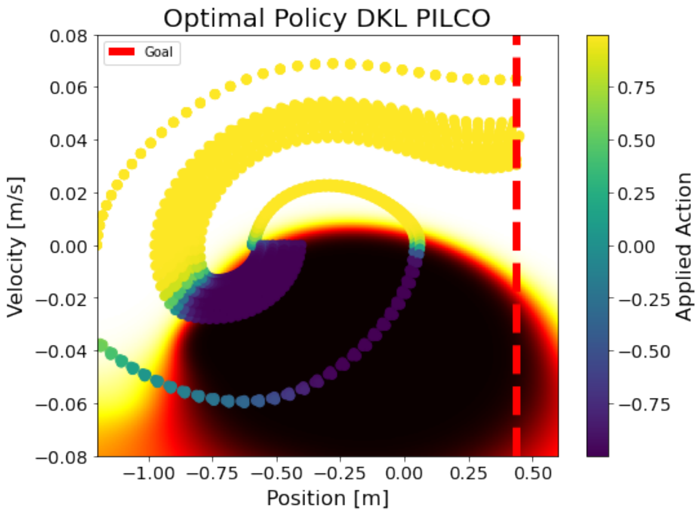


Figure 4-8: DKL PILCO's trajectories with the optimal policy visualized underneath

	Average reward	Standard deviation
PILCO	61.435	24.794
DKL PILCO	93.513	0.519

Table 4-6: Comparions of average reward obtained with initial state disturbance

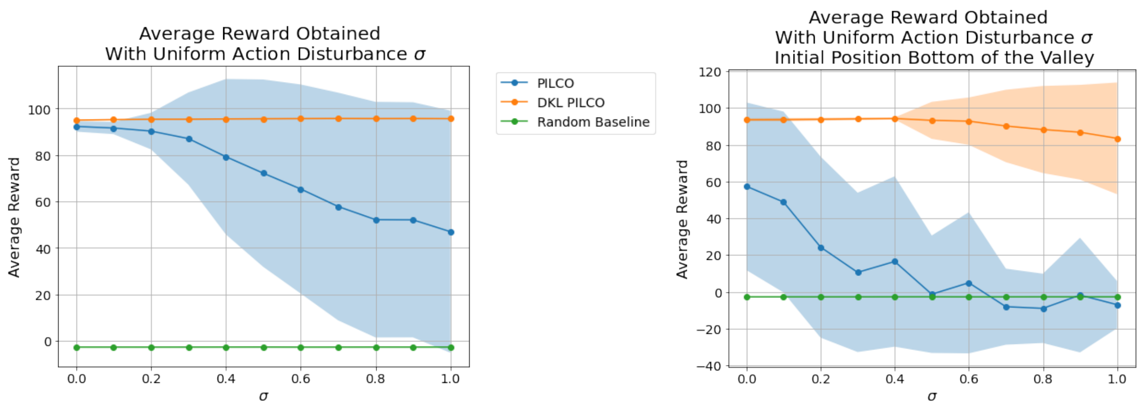
The results are summarized in Table 4-6, where DKL PILCO's average reward shows no practical difference from the previous scenario.

Disturbance on the control action and observations

In this scenario, the control action or the observations will be disturbed using uniform random noise with varying scales $\sigma = \{0.1, 0.2, ..., 1.0\}$. This will result in the following disturbed control inputs or observations:

$$\begin{aligned} a_* &= a + \mathcal{U}_{[-\sigma,\sigma]} \\ \mathbf{x}_* &= \mathbf{x} + \mathcal{U}_{[-\sigma,\sigma]} \end{aligned} \tag{4-3}$$

where a_* is the disturbed control action, $\mathbf{x}_*$ is the disturbed observation and $\mathcal{U}_{[-\sigma,\sigma]}$ is the random noise uniformly drawn between $-\sigma$ and σ. Furthermore, we will test two initial positions, one between $-0.1m$ and $0.1m$ and one at the bottom of the valley.

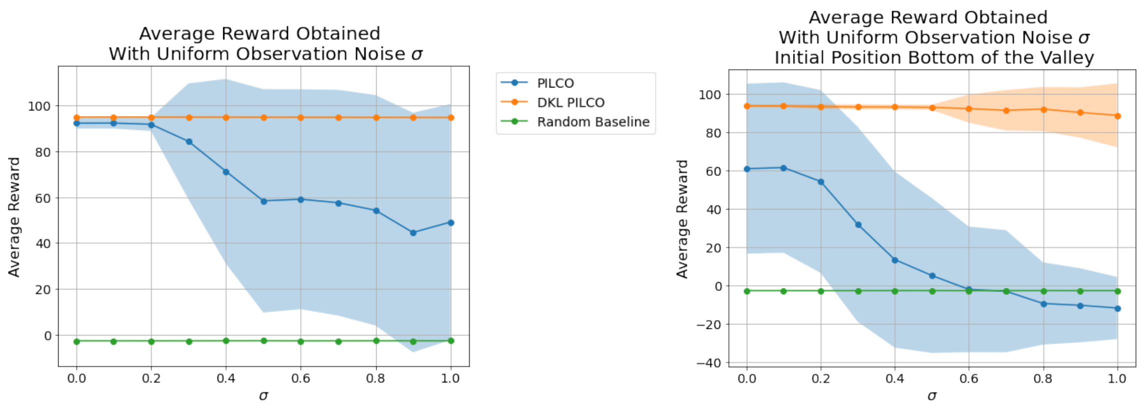


(a) Action disturbance with the initial position between $-0.1m$ and $0.1m$

(b) Action disturbance with the initial position between $-0.6m$ and $-0.4m$

Figure 4-9: Comparison of policies obtained by PILCOand DKL PILCO

Looking at Figure 4-9 DKL PILCO demonstrates more robustness against disturbances on the control action, only showing a drop in performance when the car's initial position is at the bottom of the valley, and the control disturbance is more than 40% of the maximum control action. PILCO, on the other hand, seems to be more sensitive to control disturbances, showing an immediate decrease in performance for small disturbances on the control action when the car starts from the bottom of the valley and eventually showing a similar performance as the random baseline at a control disturbance of $\sigma \geq 0.6$.



(a) Action disturbance with the initial position between $-0.1m$ and $0.1m$

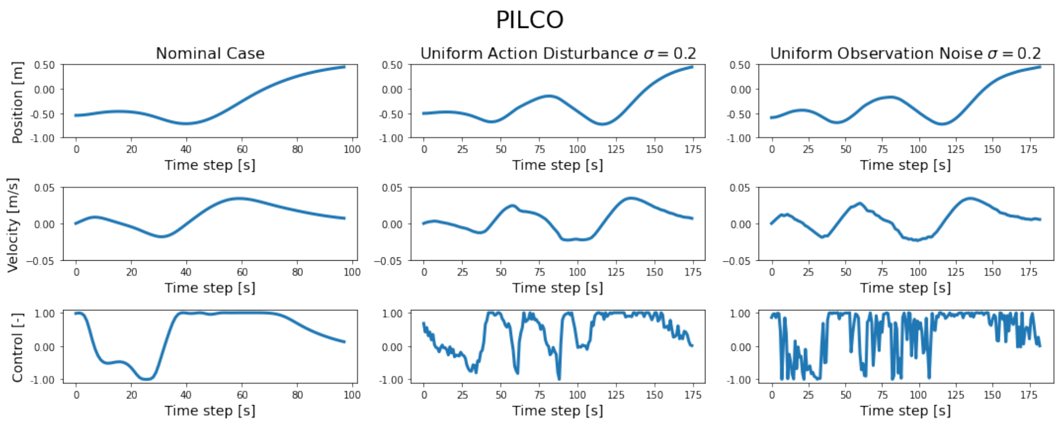
(b) Action disturbance with the initial position between $-0.6m$ and $-0.4m$

Figure 4-10: Comparison of policies obtained by PILCOand DKL PILCO

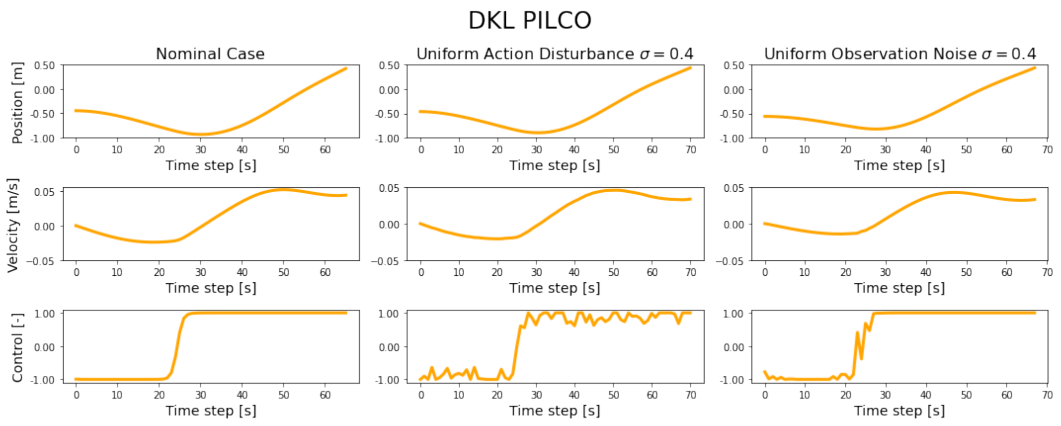
In Figure 4-10 a similar trend can be seen. However, for a small amount of observation noise, PILCO also seems to be unaffected, which can be argued by looking at its policy in

Figure 4-4 where a small amount of noise will generally not result in large changes in the control actions. DKL PILCO demonstrates even more robustness in this aspect, showing only a slight decrease in performance when the observation noise becomes large, from $\sigma \geq 0.7$ onwards. This robustness could be attributed to the policy obtained by DKL PILCO, which effectively divides the state space into two zones (either full acceleration to the left or full acceleration to the right as shown in Figure 4-4). Consequently, only very large noise or state values close to the boundary between left and right acceleration would result in different control actions.

In Figure 4-11, three trajectories for both PILCO and DKL PILCO are shown. Three cases are displayed: the nominal case, action disturbed case, and observation disturbed case.



(a) PILCO's trajectories



(b) DKL PILCO's trajectories

Figure 4-11: Comparison of trajectories, nominal case versus disturbed scenarios

Looking at PILCO's trajectories, it can be seen that its policy is sensitive to both the action and observation disturbance. In the nominal case, the car moves one time back and one time forward and reaches the goal within $100s$. However, in other cases, the car falls back down the hill again and has to repeat the pattern, resulting in a longer time to reach the goal of $180s$. The observation noise seems to affect the control actions of PILCO the most, showing large deviations from the nominal case and reinforcing the previous statement that

PILCO's policy cannot generalize well.

DKL PILCO's policy, on the other hand, demonstrates again its robustness. The disturbed trajectories do not deviate much from the nominal case, even when the amount of noise is doubled compared with PILCO. The car only takes about $5s$ longer to reach the goal. Furthermore, the control actions are only slightly disturbed when the car changes direction, which makes sense since a disturbance on the boundary between two actions will result in deviations in the output.

In general, DKL PILCO showed it could obtain a more robust policy than PILCO. Demonstrating robustness against initial position disturbance, observation noise, and control action disturbance. In the end, DKL PILCO achieved higher rewards and reached the goal faster than PILCO.

4-2-2 Comparison Predictive Power

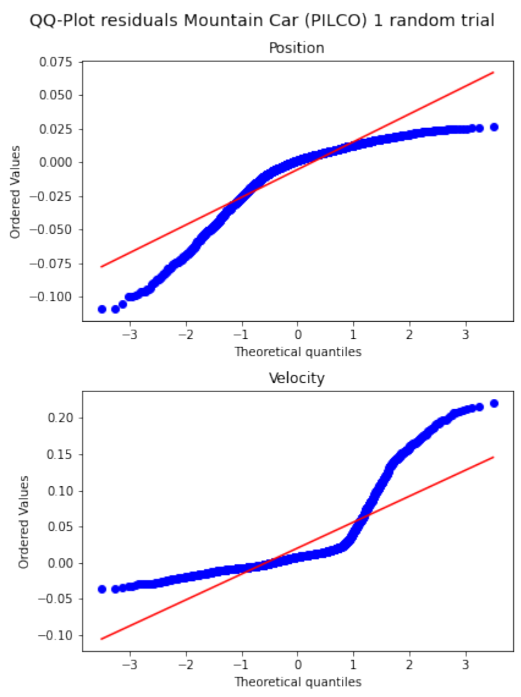
This section will analyze the dynamics model PILCO and DKL PILCO obtained on the mountain car. The experiment involved training both models on one random trial with a length of $T = 80s$. Notably, the policy was excluded in this segment since the primary focus was to draw a comparison between the dynamics models. Consequently, controller actions were derived by randomly sampling from the set of potential controller actions. Following the training, both models were assessed on 100 new random episodes, and for each episode, metrics such as the MAE and MSE were calculated on the one-step-ahead predictions.

Looking at Table 4-7, it can be seen that DKL PILCO achieves a lower MAE and MSE for both the position and velocity compared to PILCO. Furthermore, PILCO's and DKL PILCO's MAE are relatively close to each other (within one order of magnitude). However, PILCO's MSE is several orders of magnitude larger, especially for the velocity. Meaning that, compared to DKL PILCO, PILCO has a higher frequency of relatively large errors, which could be attributed to sensitivity to outliers and skewed underlying distribution. The kurtosis of the observed dataset was determined to be -0.81 and 38.0 for the position and velocity, respectively. As a result, the distribution of the observed velocities has fatter tails and a peaked center, which indicates a significant presence of outliers. PILCO might be struggling to predict these edge cases accurately, suggesting a sensitivity to outliers.

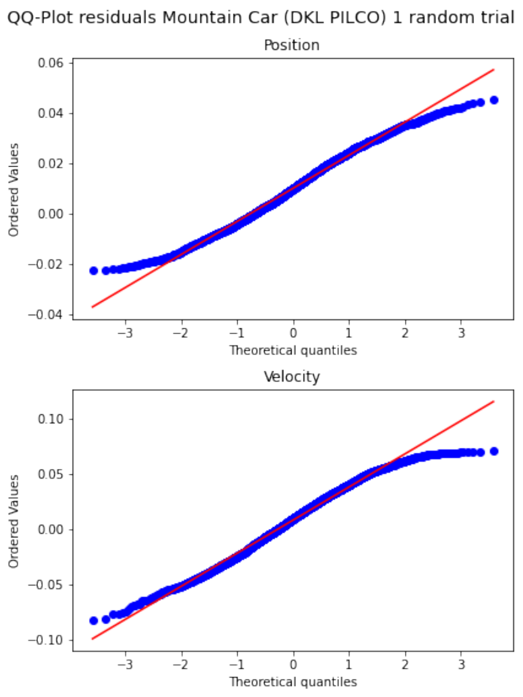
State	**MAE (PILCO)**	**MAE (DKL PILCO)**	**MSE (PILCO)**	**MSE (DKL PILCO)**
Position	$2.28 \cdot 10^{-2} \pm 1.20 \cdot 10^{-4}$	$1.35 \cdot 10^{-2} \pm 3.74 \cdot 10^{-5}$	$8.64 \cdot 10^{-4} \pm 9.59 \cdot 10^{-5}$	$2.77 \cdot 10^{-4} \pm 1.40 \cdot 10^{-5}$
Velocity	$2.89 \cdot 10^{-2} \pm 1.35 \cdot 10^{-2}$	$1.49 \cdot 10^{-3} \pm 2.36 \cdot 10^{-7}$	$1.40 \cdot 10^{-3} \pm 1.39 \cdot 10^{-6}$	$3.21 \cdot 10^{-6} \pm 2.84 \cdot 10^{-12}$

Table 4-7: Statistical analysis of the accuracy of the dynamics models, for both PILCO and DKL PILCO, after being trained on one trial

Figure 4-12 shows the Q-Q plot of the residuals for both models. A Q-Q is a visual tool that can be used to check how well the data aligns with a specific theoretical distribution [46]. An illustrative comparison can be obtained by plotting the dataset's quantiles against the chosen distribution's quantiles. If the data follows the assumed distribution perfectly, then the points should form a straight line at 45 degrees [46]. However, any noticeable divergence from this



(a) PILCO trained on one random trial



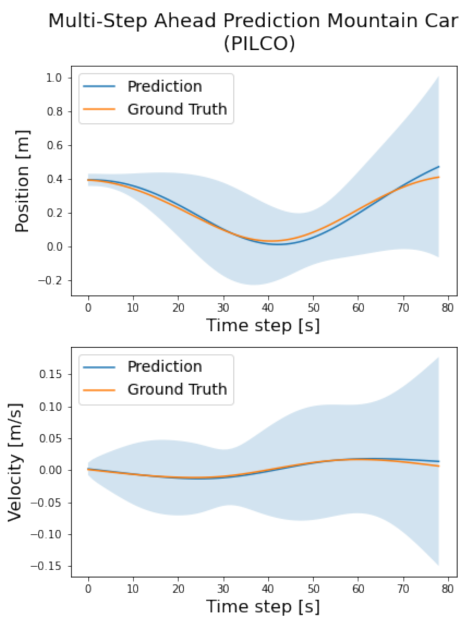
(b) DKL PILCO trained on one random trial

Figure 4-12: Comparison of the QQ plots of the residuals on the mountain car for PILCO and DKL PILCO. Both models have been trained on one random trial.

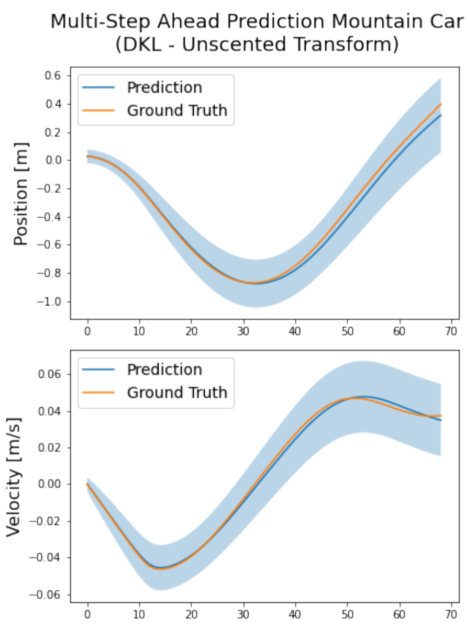
line suggests that the data may adhere to a different distribution than was previously assumed [46]. In this case, a normal distribution is used for reference.

PILCO displays significant left-skew and right-skew for the position and velocity, respectively. This could mean that PILCO tends to underpredict the position and overpredict the velocity. More importantly, the significant skewness of both residuals clearly indicates that PILCO is making systematic errors after one random trial. DKL PILCO, on the other hand, shows residuals that seem to follow a standard normal distribution more closely. This is a good thing since it means that the model is probably well-specified, and any statistical inferences made from DKL PILCO are likely to be more valid than PILCO [78].

Furthermore, in Figure 4-13, the multiple-step ahead (MSA) prediction on the mountain car is shown, which involves a successful policy. The MSA prediction is generated by only using the initial state mean and covariance and then generating the trajectories by propagating these initial conditions for the entire time horizon. First of all, it can be seen that both models can track the ground truth reasonably closely. Secondly, PILCO's confidence intervals of the predictions (shown as a 95% confidence interval) are much broader than DKL PILCO. Wide confidence intervals are not necessarily always a problem. In this case, it does show a lack of precision in the predictive model, which also impacts the policy since the decisions are based on the predictions.



(a) Multi-step ahead prediction from PILCO



(b) Multi-step ahead prediction from DKL PILCO

Figure 4-13: Comparison of the multi-step ahead prediction between both PILCO and DKL PILCO. Note that these trajectories correspond to successful trajectories.

4-3 Numerical Stability

This subsection will examine the numerical stability of PILCO and DKL PILCO. Numerical stability will be assessed through the evaluation of the condition number. The condition number of a square symmetric matrix is often referred to as the spectral condition number [22]. Square symmetric matrices have the property that all their eigenvalues are real (which also needs to hold true for covariance matrices) [89]. The spectral condition number is defined as [4]:

$$\kappa(A) = \frac{\sigma_{\max}(A)}{\sigma_{\min}(A)} \tag{4-4}$$

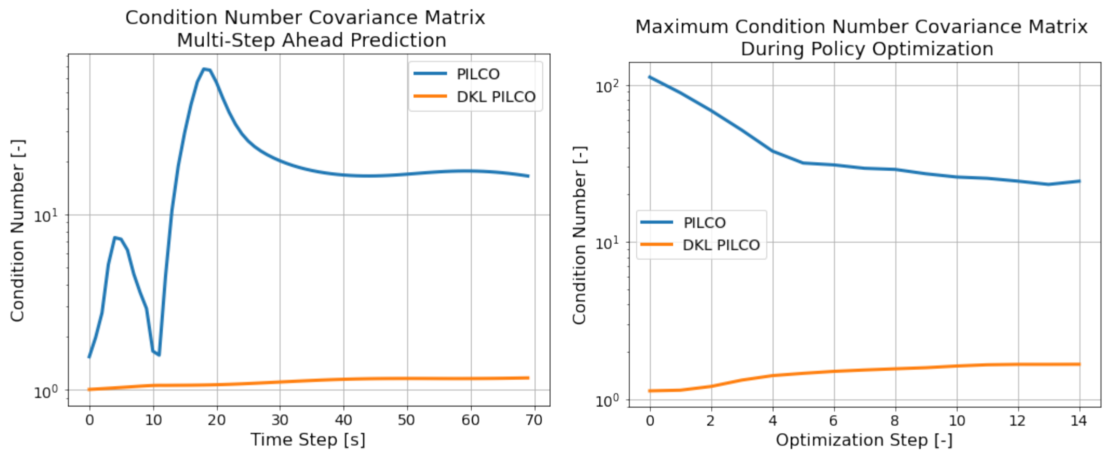
where $\sigma_{\max}(A)$ and $\sigma_{\min}(A)$ are the largest and smallest eigenvalues of A, respectively. The condition number gives a measure of the sensitivity of the system to changes in the system's parameters or to errors in the measurements [15]. If the condition number is large (much greater than 1), small changes in the input can produce large changes in the result - a clear sign of instability. Conversely, the system tends to be stable if the condition number is close to 1. In the context of covariance matrices, a high condition number could result in the following:

- **Sensitivity to small changes**: a large condition number suggests that the uncertainty estimates are sensitive to small changes in the input data. This means that even minor perturbations in the input can result in large fluctuations in the outputs. Therefore, the uncertainty estimates may not be reliable or consistent across different samples [72].

- **Inaccuracy**: uncertainty propagation often requires inverting the covariance matrix or computing its determinant. When computing the determinant of an ill-conditioned matrix, these problems can become apparent. Very large or small values can lead to overflow or underflow in floating-point computations, which can become amplified in the calculations due to the ill-conditioning of the matrix [11]. These numerical issues can lead to incorrect results or even algorithm failure.

- **Imprecise confidence intervals**: large condition numbers can lead to larger variances in predictions, which can, in turn, result in wider and potentially less precise confidence intervals [16].

In Figure 4-14 two situations are shown. The plot on the left shows the condition number of the covariance matrix for the multi-step ahead prediction after GP optimization and policy optimization. In comparison, the plot on the right shows the maximum condition number of the covariance matrix encountered during policy optimization. Note that the y-axis on the left plot is on a log scale.



(a) Condition number during multi-step ahead prediction

(b) Condition number during policy optimization

Figure 4-14: Comparison of the condition numbers of the covariance matrix during multi-step ahead prediction and policy optimization. The maximum condition number during one optimization step is denoted during policy optimization.

In both cases, PILCO with moment matching attains a larger condition number than DKL PILCO. Referring back to Figure 4-13, this might explain the wide confidence intervals around its predictions especially compared to DKL PILCO. The sharp rise in the condition number at around $t = 20s$ roughly corresponds to the point where the car changes direction (transitioning from driving backward to forward). At the points where the car changes direction, there is a significant interplay between potential and kinetic energy. A slight variation in the current state (e.g., a small change in velocity) can significantly impact the next state, making the relationship between the current and next states highly non-linear. A small change in position or velocity can lead to a large change in future states. This leads to increased uncertainties when predicting the next state, even just one timestep ahead, causing the condition number of the covariance matrix to spike. When the car changes direction, there is also an abrupt change in the velocity state (from positive to negative or vice versa). The GP might have difficulty capturing this abrupt change, leading to increased uncertainty and a corresponding spike in the condition number. The decrease in condition number following this might be attributed to several factors. First of all, there might be a reduction in action uncertainty. After this point, a successful policy could become near-deterministic near the goal. The only "correct" action will be an acceleration to the right. Furthermore, the car will pass its initial position again and might follow a predictable pattern, resulting in a drop in the uncertainty and thereby reducing the condition number.

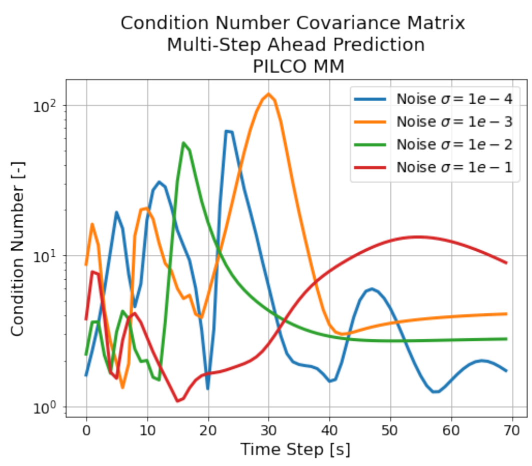
Furthermore, the maximum condition number PILCO attains is even higher during policy optimization, where it starts high and drops after a few optimization steps while still staying significantly higher compared to DKL PILCO. This could indicate sensitivity to the initialization of the parameters of the RBF network. While the condition number drops down with the optimization step, starting from a large condition number can still be cause for concern since it can slow down the convergence. Furthermore, it might result in poor generalization since the model can become sensitive to specific small perturbations in the input data, which was also seen in Figure 4-9 and Figure 4-10, where PILCO's performance was negatively impacted by observation noise or control action disturbance.

DKL PILCO demonstrates a lower and more stable condition number throughout the prediction horizon, which is also seen in the confidence interval around its predictions Figure 4-13. Interestingly, DKL PILCO does not show the same peak as PILCO in its condition number during the critical point, as previously discussed. This could be attributed to the UT used in DKL PILCO, which might be able to handle non-linearities better since it propagates the sigma points directly through the non-linear function. Furthermore, not only does DKL PILCO achieve this during multi-step ahead prediction, but it also shows a similar behavior during policy optimization. This could have been attributed to DKL PILCO's robustness against disturbances as evidenced in Figure 4-9 and Figure 4-10. These results seem encouraging since they indicate that DKL PILCO shows more numerical stability than PILCO.

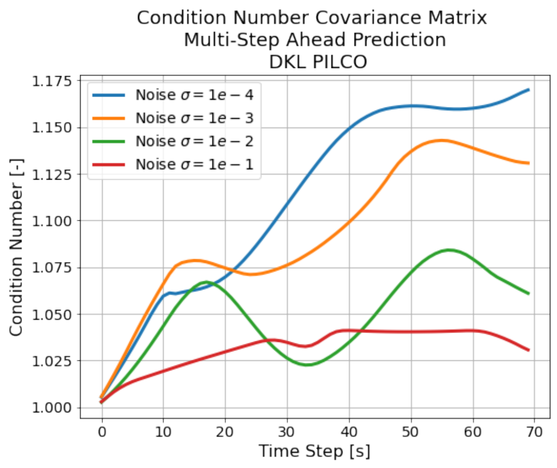
In Figure 4-15, the condition number is shown as a function of the likelihood noise. Generally, they reflect the noise level in the observed data [83]. Note that a lower likelihood noise might be important if the target signal also has a low variance or if the problem requires a very fine-tuned control input.

PILCO demonstrates a large sensitivity to the likelihood noise level, displaying condition numbers up to 100, which can be considered ill-posed. The oscillating peaks in the plots could be attributed to the situation discussed before, where the car changes direction. Multiple peaks could indicate that the car is going back and forth, possibly to build up momentum.

DKL PILCO also shows some sensitivity to the likelihood, but note that the y-axis is not on a log scale in this plot. Hence, DKL PILCO's condition number stays much more bounded and stable compared to PILCO. This is encouraging since it could allow DKL PILCO to learn in more challenging conditions where the target signal variance is low. However, care must still be taken to monitor if the model will not overfit.



(a) Condition numbers PILCO



(b) Condition numbers DKL PILCO

Figure 4-15: Condition number sensitivity analysis

4-4 Swimmer

The swimmer is a robot featuring two joints traversing a two-dimensional plane by swimming through a thick fluid. An actuator manages each joint, and the system earns rewards for progressing along the x-axis. This task is more complex, encompassing an eight-dimensional state space, a two-dimensional control space, and nonlinear dynamics. Moreover, initiating movement in the correct direction necessitates coordination between the two controllers. This makes it challenging to gain a reward signal during the early training phases.

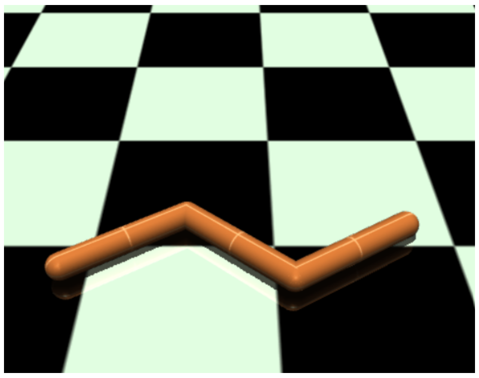


Figure 4-16: The model and environment of the swimmer

In this instance, we will look at a swimmer with three links and two rotors. As a result, the action space will be two-dimensional, representing the torques applied between the links, which are denoted in Table 4-8.

The state (or observation) space consists of 8 dimensions. For each part, i, the angle with respect to the x-axis is measured, θ_i. Furthermore, the angular velocity of each part is measured as well, ω_i. Finally, the velocity of the tip along the x- and y-axis are also measured. A full overview of the state space is shown in Table 4-9.

Action	Control Min	Control Max	Unit
Torque applied on the first rotor	-1	1	Nm
Torque applied on the second rotor	-1	1	Nm

Table 4-8: Action space of the swimmer.

State	Min	Max	Unit
Angle of the front tip	$-\infty$	∞	rad
Angle of the first rotor	$-\infty$	∞	rad
Angle of the second rotor	$-\infty$	∞	rad
Velocity of the tip along the x-axis	$-\infty$	∞	m/s
Velocity of the tip along the y-axis	$-\infty$	∞	m/s
Angular velocity of the front tip	$-\infty$	∞	rad/s
Angular velocity of the first rotor	$-\infty$	∞	rad/s
Angular velocity of the second rotor	$-\infty$	∞	rad/s

Table 4-9: State space of the swimmer.

The reward of the swimmer consists of two parts:

- **Forward moving reward:** A reward for swimming to the right, which is determined as $\frac{x_t - x_{t-1}}{dt}$ where dt is the time between two consecutive actions and depends on the frame skip and the frame time.
- **Control cost:** A cost for penalizing large control actions, calculated as $\sum_{i=1}^{2} a_i^2$.

The total reward is then summarized as $reward = forward\ reward - control\ cost$.

Unfortunately, both DKL PILCO and PILCO could not learn a successful control policy for the swimmer problem. One of the challenges in our approach to optimizing the policy is found in the final part of the trajectory for the angles of the first and second rotors. The angle of the first rotor approaches -1.5 rad, and the second rotor approaches 1 rad; these large angles cause the robot to lock up and impede further motion, shown in Figure 4-17.

An attempt was made to mitigate this issue by extending the planning horizon. However, this resulted in the policy delaying the troubled states' occurrence. Another solution could be using the same control action for multiple timesteps. For example, a system that is simulated for 250-time steps. Then, using the same control action for five-time steps would result in an effective planning horizon of 50-time steps. As a result, the system would appear stiffer, and the control action would be less granular. However, such an approach could be justified if the differences between consecutive states would be sufficiently small. However, this approach had a significant impact on the hyper-parameters and required different priors and constraints on the hyper-parameters. Unfortunately, due to time constraints, such a hyper-parameter search could not be conducted.

Examining the situation from a different perspective, another fundamental factor contributing to the aforementioned problem could be the reward function utilized in this experiment. Specifically, it was an exponential reward function characterized by the following parameters:

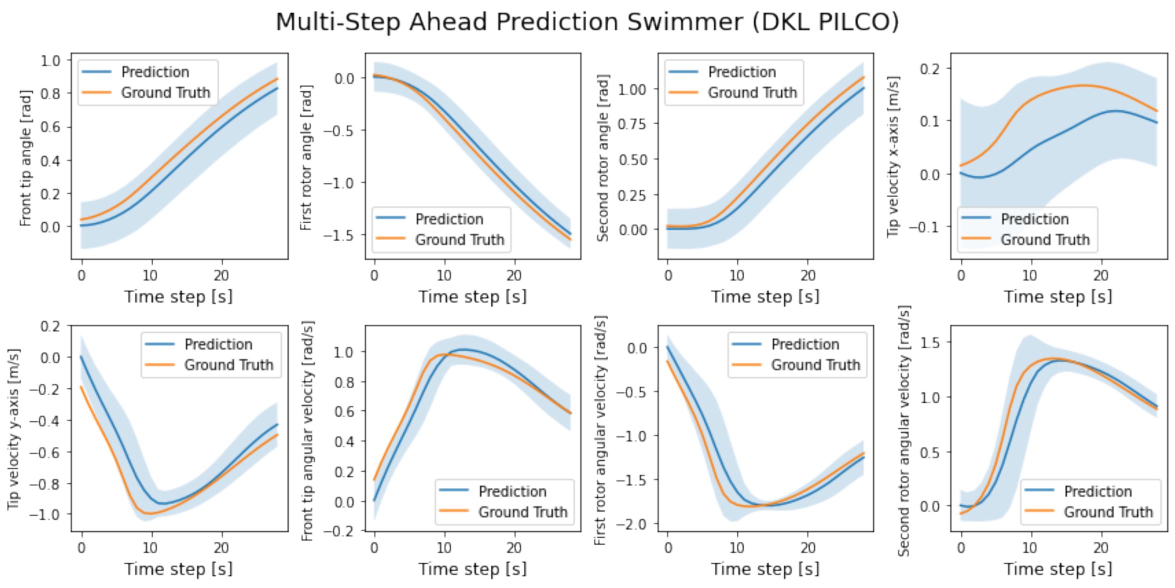


Figure 4-17: Multi-step ahead prediction on the swimmer failed policy, DKL PILCO

$$\boldsymbol{x}_{\text{target}} = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \end{bmatrix} \quad \boldsymbol{w} = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \end{bmatrix}$$

This reward function only places a weight on the velocity of the tip along the x-axis, with the target being $1\frac{m}{s}$. As a result, the robot is only concerned with achieving the targeted velocity instead of also avoiding potential states, which could lead to a terminal state that is difficult to escape. An alternative solution to this issue might involve using a different reward function by incorporating an additional one that penalizes extreme angles, thereby dissuading the agent from gravitating toward problematic states. For instance, a linear reward model could be employed:

$$r = \boldsymbol{x} \cdot \boldsymbol{W} \tag{4-5}$$

where x is the state, $\boldsymbol{W}$ is weight matrix with an entry of 1 only for the fourth state. Subsequently, an exponential cost function might be utilized to impose penalties on large angles. A certain weighting can be applied to determine the ultimate reward. However, the design and tuning of such a combined reward function proved to be non-trivial and was not feasible due to time constraints.

4-4-1 Comparison Predictive Power

This subsection will compare the predictive power of PILCO and DKL PILCO. An RBF network was trained on the input-output data of a trained SAC policy, and the same network will be used for both PILCO and DKL PILCO. About 150 basis functions were necessary to correctly clone the behavior of the SAC policy, further highlighting the complexity of the swimmer problem. The dynamics models of both frameworks will be trained on random interactions with the environment but evaluated on rollouts with the trained policy. The training dataset is the same for both models and consists of 10 rollouts of length $T = 100s$.

The tests in this subsection are done with one random seed but using 100 different rollouts when the mean and standard deviation are reported.

One-step ahead prediction

Figure 4-18 and Figure 4-19 depict the one-step ahead prediction of both PILCO and DKL PILCO. The largest difference between the two seems to be in their uncertainty bounds.

PILCO's larger uncertainty bounds, in conjunction with the wide range of length scales ranging from 2 to 15, may reveal a model that is actively adapting to the complex dynamics of the underlying data. These varying length scales suggest that the model is attempting to tailor its behavior to different regions or dimensions of the input space, accommodating both small-scale noise and larger underlying trends. While this flexibility might enhance the model's ability to represent complex patterns, it could also contribute to increased uncertainty, especially in regions where the data is sparse or the underlying function is more complex.

The wide length scales might indicate a trade-off between fitting noise and capturing underlying trends, making the model adaptable yet sensitive. Compared to DKL PILCO, this sensitivity might result in less accurate ground truth tracking and larger uncertainty bounds, making PILCO more reliant on specific hyper-parameter choices like signal variance and likelihood noise. The model's complexity in tuning might contribute to the observed differences in uncertainty and accuracy in tracking the underlying function.

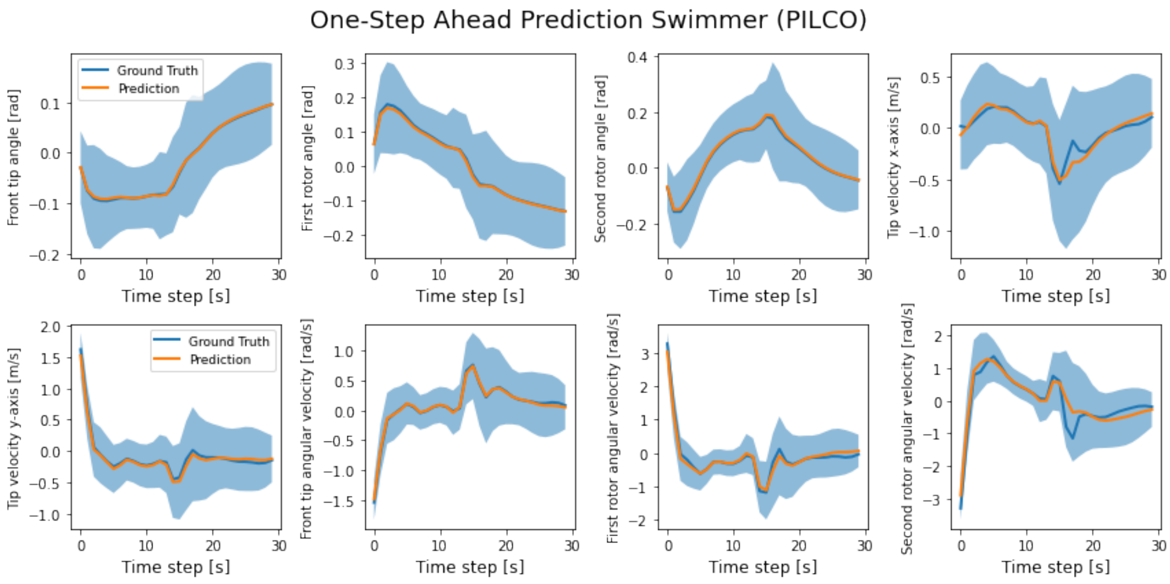


Figure 4-18: One-step ahead prediction on the swimmer, PILCO

DKL PILCO's smaller uncertainty bound could indicate a more confident model. The incorporation of Deep Kernel Learning (DKL) might allow the model to capture complex relationships within the data better, reducing uncertainty. The improved ability to track ground truth might stem from the added complexity and representational power that the DKL component brings to the model.

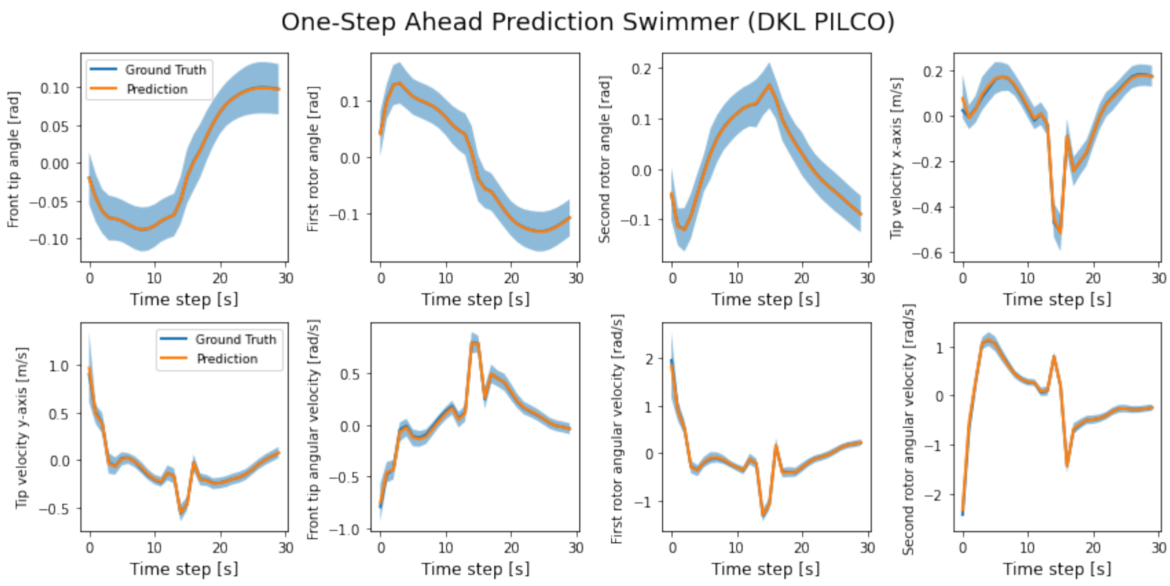
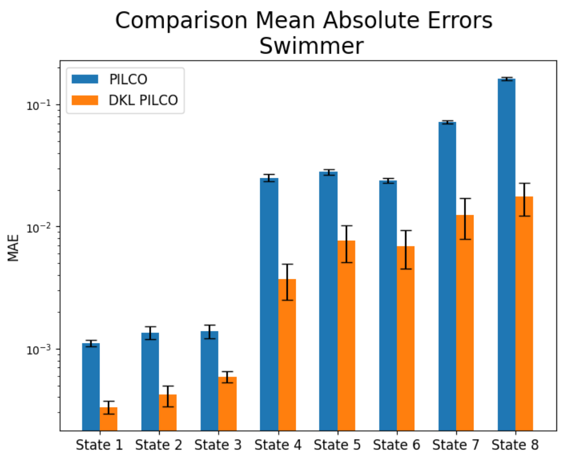
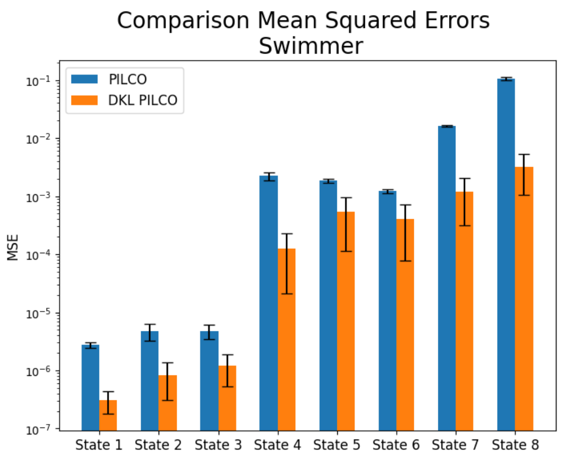


Figure 4-19: One-step ahead prediction on the swimmer, DKL PILCO

Having the length scales centered around 2 without much variation suggests that the DKL PILCO model is applying a more consistent level of smoothness across the input space. This could indicate that the neural network component within DKL is transforming the input features into a space where the relationships are more uniformly captured by the GP. The DKL part might absorb much of the complexity, leaving the GP to model a more consistent, smoothly varying function. This might contribute to the smaller uncertainty bounds and improved tracking.



(a) Mean absolute errors



(b) Mean squared errors

Figure 4-20: Comparison mean absolute and squared errors for the one-step ahead prediction on the swimmer

Figure 4-20 depicts the mean and standard deviation of both the MAE and MSE for each state dimension in a bar plot. The height of the bars indicates the mean, and the error bars the standard deviation. A lower MAE means that the model's predictions are, on average, closer to the actual values [33]. This indicates that the model is generally more accurate. A lower MSE indicates that the model is not only making smaller errors on average but also that it is making fewer large errors [33].

Looking at the Q-Q plots in Figure 4-21 and Figure 4-22, depicting PILCO and DKL PILCO, we see a similar picture as in the mountain car problem. Compared to PILCO. DKL PILCO's residuals tend to follow a normal distribution more closely, which also validates our assumption that DKL could capture more complex relationships in the data without systematic biases.

In conclusion, the normal distribution of residuals in DKL PILCO, combined with the substantially lower MSE and MAE than PILCO, indicates an accurate and well-specified model that is likely capturing the underlying patterns in the data effectively.

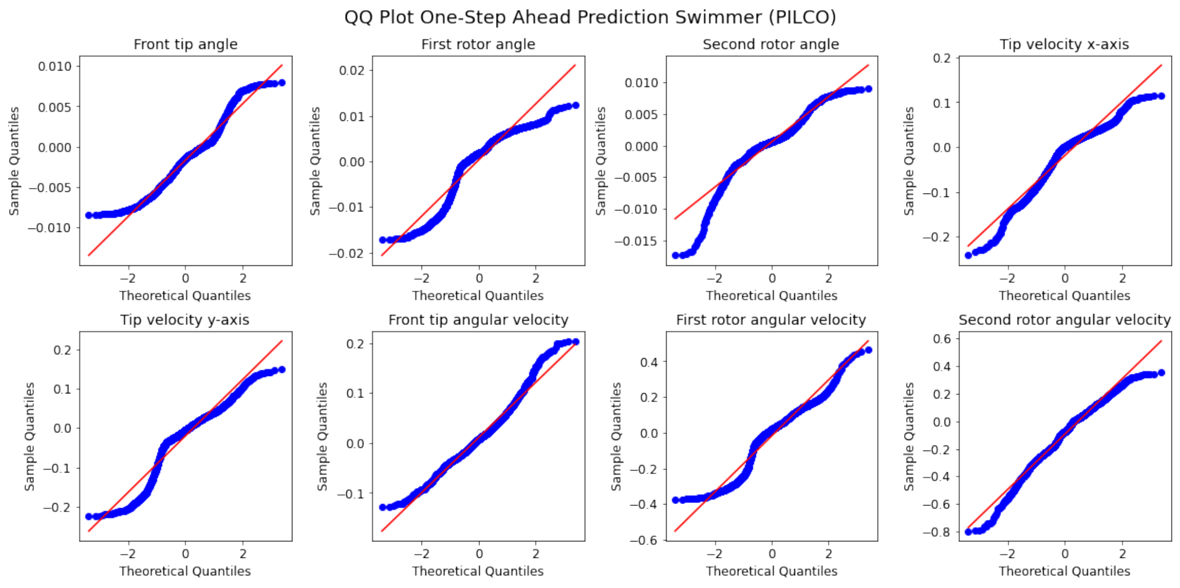


Figure 4-21: QQ-plots of the residuals on the one-step ahead prediction, PILCO

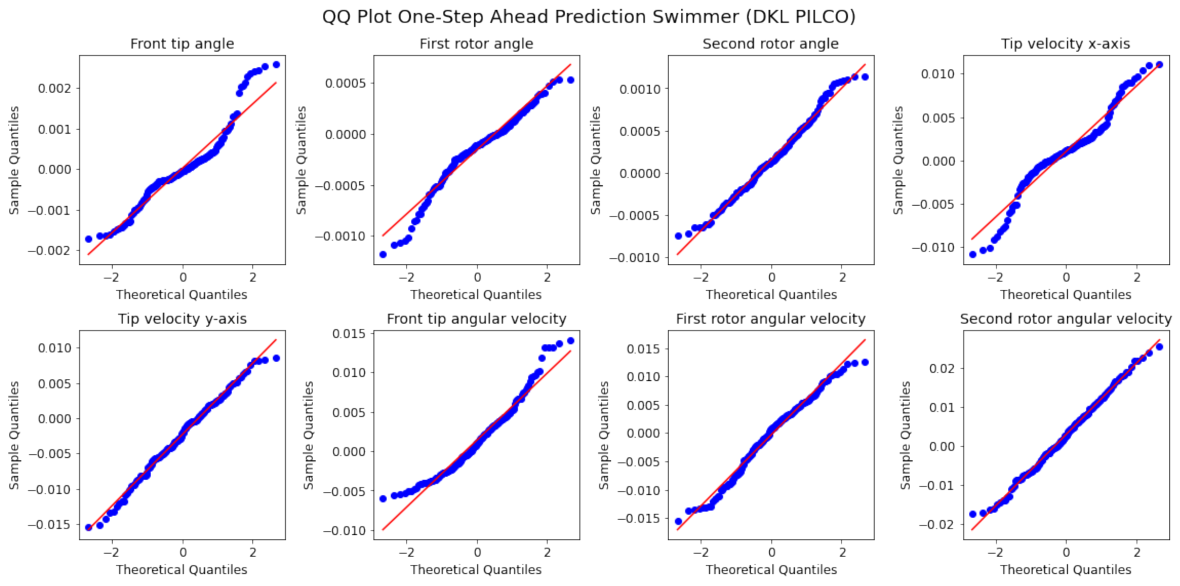


Figure 4-22: QQ-plots of the residuals on the one-step ahead prediction, DKL PILCO

Multi-step ahead predictions

Figure 4-23 and Figure 4-24 show the multi-step ahead predictions of PILCO and DKL PILCO.

Looking at PILCO's predictions, large uncertainty bounds and predictions that generally do not track the ground truth very well can be seen.

The variations in uncertainty for angular velocities might signify a complex interplay between various factors, such as noise, underlying dynamics, and hyper-parameter sensitivity. This fluctuating uncertainty could be indicative of the model's struggle to accurately balance the trade-offs between fitting noise and capturing genuine dynamics, which is also witnessed by the wide range of length scales.

While one might expect the instability in angular velocities to impact angles directly since angular velocities represent the rate of change of angles, the observed behavior suggests a more complex relationship. The lack of direct correspondence between these uncertainties might imply that other factors are at play or a mismatch between the true and learned dynamics.

The instability could also be attributed to the condition number of the covariance matrix during the prediction horizon Figure 4-26. The condition number for PILCO starts at around 30, peaks at around 500 at the 50s mark, and then decays back to 30. Furthermore, some oscillations can be seen between 0 and 50s. The spike and oscillations may suggest numerical instability, and the high condition number could mean that the system is sensitive to noise or errors, which may affect the prediction quality.

The use of moment matching in PILCO might contribute to the observed patterns. This method can lead to approximation errors, particularly in nonlinear systems, which might explain the difficulties in tracking the ground truth closely.

PILCO's multi-step ahead predictions seem to demonstrate issues in stability and accuracy further, possibly stemming from the wide range of length scales and the use of moment

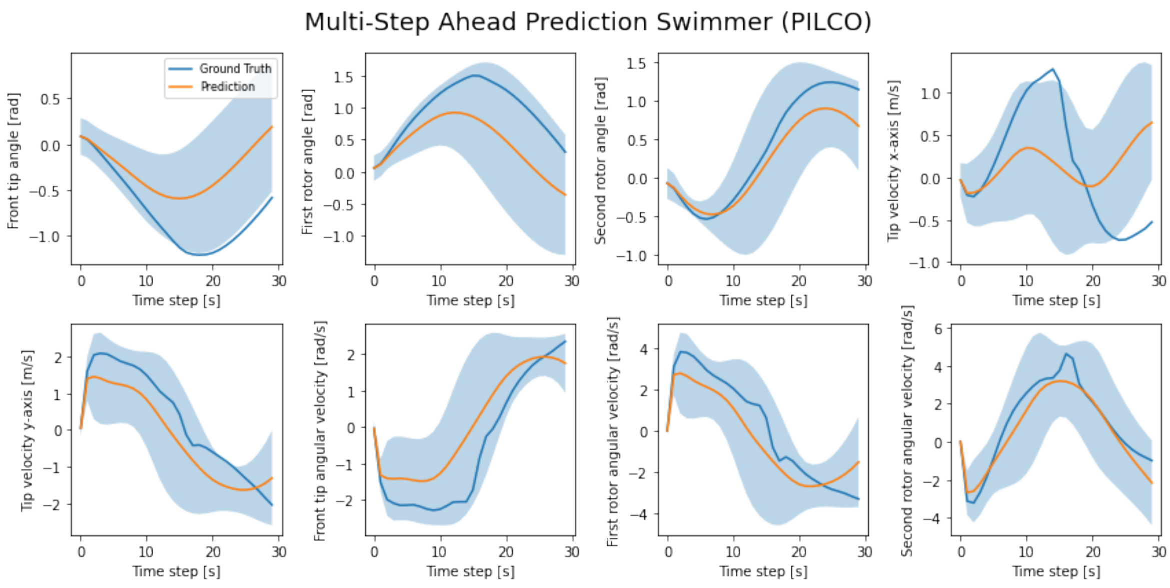


Figure 4-23: Multi-step ahead prediction on the swimmer, PILCO

matching. The model appears sensitive to noise and struggles to track the ground truth closely in specific dimensions.

DKL PILCO demonstrates good performance on the multi-step ahead predictions, marked by small confidence bounds and the ability to track the ground truth almost exactly for extended horizons, even for a prediction horizon of up to 100s Figure 4-25.

The small uncertainty bounds in DKL PILCO's predictions suggest a more confident model. This could be due to the DKL component's ability to capture complex relationships within the data more effectively, reducing uncertainty. The close alignment of the mean predictions with the ground truth over both short and extended horizons points to a model that has learned the underlying dynamics of the swimmer problem well, which was already seen by the smaller MAE and MSE in the previous subsection. It indicates that DKL PILCO is not just fitting the available data but generalizing effectively to unseen scenarios. Maintaining accurate predictions even at a 100-second horizon implies robustness in handling long-term dependencies and forecasting.

Furthermore, the numerical stability of DKL PILCO's predictions is further highlighted when looking at the condition numbers of the covariance matrix in Figure 4-26. Starting with a condition number of around 4, DKL PILCO's condition number slowly increases to about 20 after 100 seconds. This lower and more stable condition number might reflect better numerical stability and a less sensitive system, enhancing prediction accuracy. In contrast to PILCO, DKL PILCO does not exhibit the issues of fluctuating uncertainties. This indicates a more stable and consistent prediction capability, likely supported by the use of the UT.

In conclusion, DKL PILCO's performance in multi-step ahead predictions, characterized by small uncertainty bounds and high accuracy even for long horizons, represents a significant improvement over traditional PILCO. The observations underscore the importance of model extensions like DKL and the incorporation of the UT in capturing complex dynamics and provide insights into the potential applications and strengths of the enhanced model.

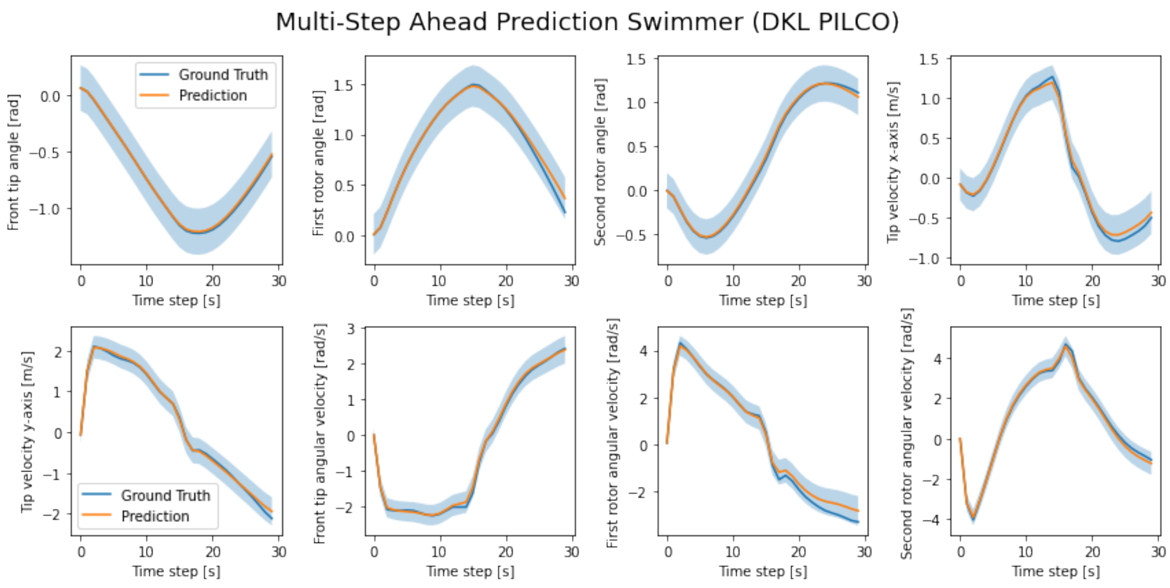


Figure 4-24: Multi-step ahead prediction on the swimmer, PILCO

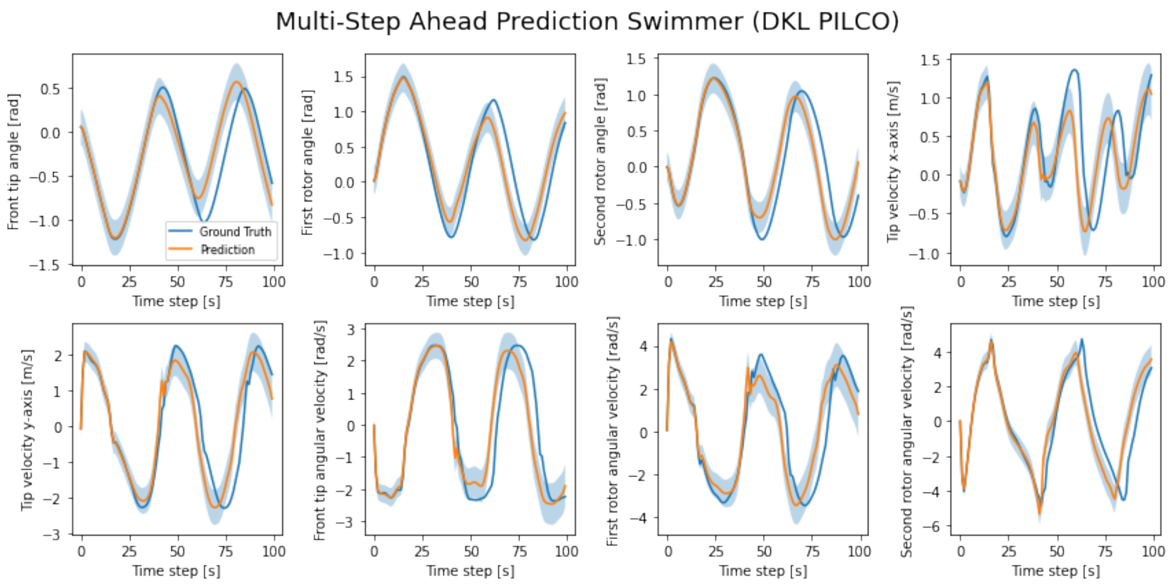


Figure 4-25: Multi-step ahead prediction on the swimmer with a long horizon of 100s, DKL PILCO

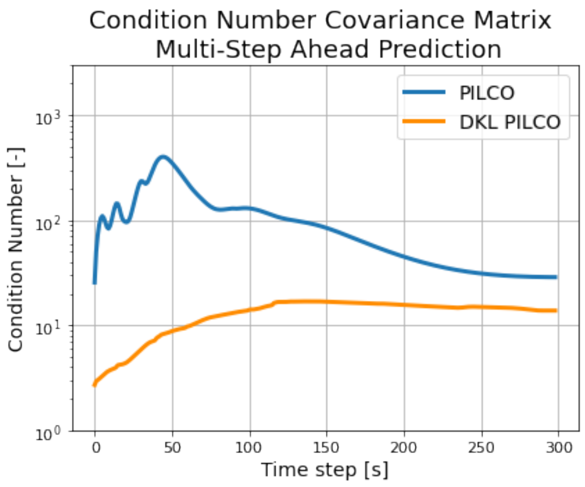


Figure 4-26: Condition numbers of the covariance matrix during multi-step ahead prediction on the swimmer problem

4-5 The Case For Stochastic Mini-Batching and Batch Normalization Layers

In Figure 4-27, the loss curves of the dynamics model for the one-step ahead predictions are shown for the mountain car problem. The curves depict six scenarios: the first two columns represent DKL with and without stochastic mini-batching; and with and without batch normalization. The last columns show the regular GP, with and without stochastic mini-batching. The stochastic mini-batching scenarios use a learning rate of $\gamma = 1e^{-3}$ and a batch size of $m = 64$, while the full batch optimization uses a learning rate of $\gamma = 1e^{-2}$ (except when no batch normalization is used in DKL).

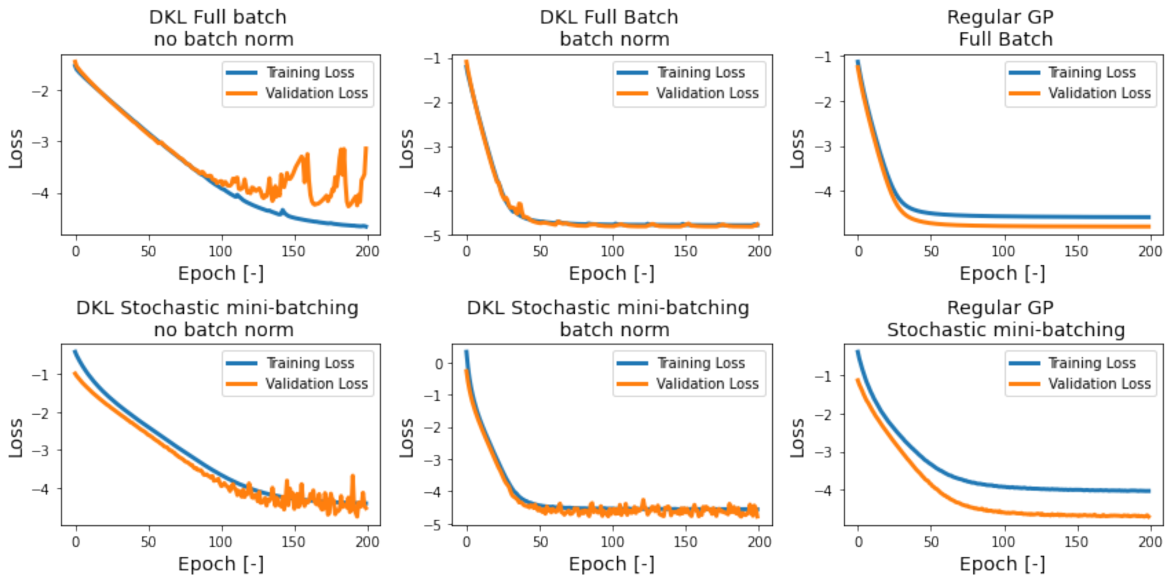


Figure 4-27: Loss curves comparison for the mountain car problem

Interestingly, when looking at the loss curve of DKL without batch normalization, we see overfitting behavior, where the validation loss oscillates while the training loss is still going down. The regular GP does not suffer from the same behavior, confirming the statements by Ober et al. (2021) [75] that DKL can suffer from overfitting. Looking at DKL without batch normalization but with stochastic mini-batching, the problem seems to be partly mitigated, with some noise still present toward the end of optimization. However, the most significant impact seems to come from the batch normalization layers when looking at the full batch case. First, we can use a higher learning rate, showing a similar loss curve to the regular GP. Furthermore, using stochastic mini-batching on top does not seem to improve the loss curve much. The largest effect might be the output stabilization for each hidden layer in the NN, resulting in stabilization of the gradient as well (when comparing the bottom left and middle plots).

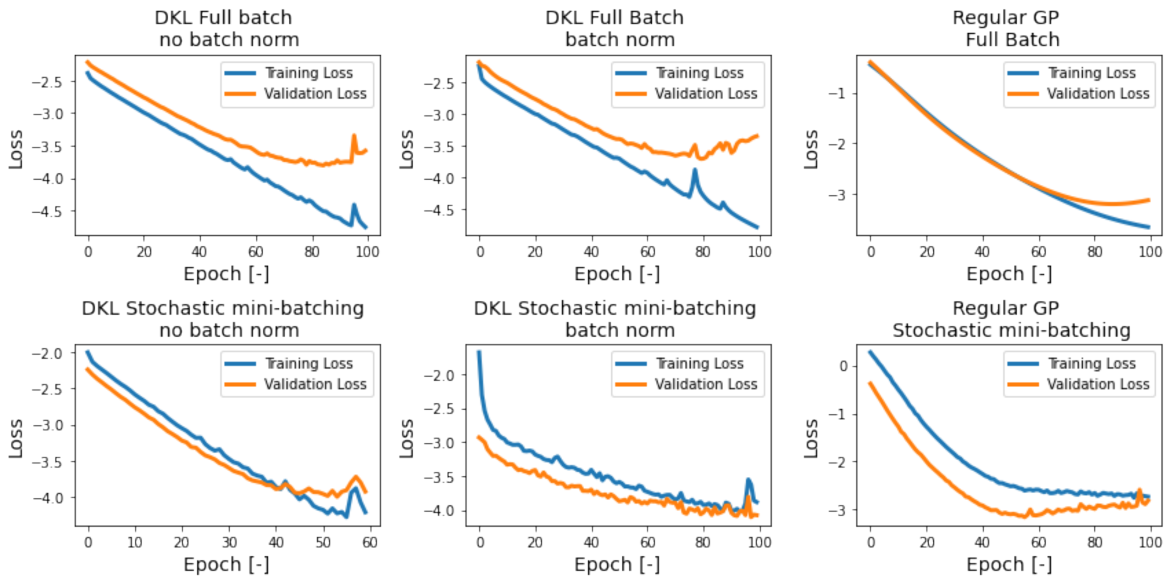


Figure 4-28: Loss curves comparison for the swimmer problem

However, when looking at the loss curves for the swimmer problem in Figure 4-28, it can be seen that a combination of batch normalization and stochastic mini-batching is required to prevent overfitting. Interestingly, the regular GP is also overfitting in this case. This could be attributed to the larger dimensionality of the problem, which could result in test data being further removed from the training data (compared to the mountain car problem).

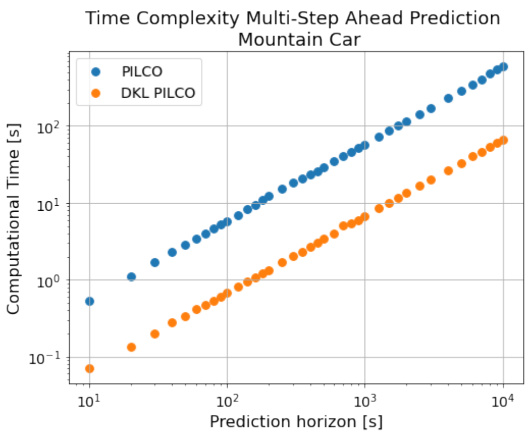
These plots indicate that DKL can be used within the PILCO framework, but care must be taken to prevent overfitting, either through using batch normalization, stochastic mini-batching, or both.

4-6 Computational Complexity Comparison

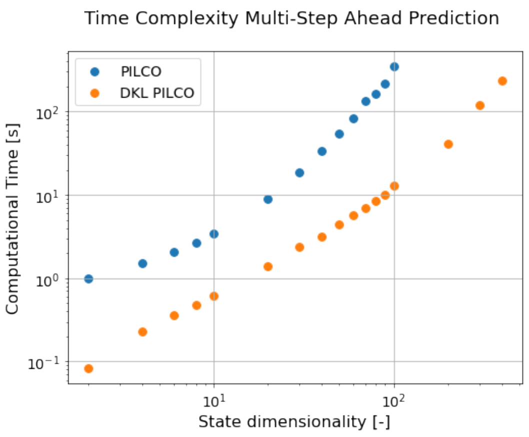
This section will discuss the time complexities of PILCO and DKL PILCO. In the first experiment, the mountain car environment was chosen. Both cases used the entire data set of

400 data points and an RBF controller with 25 basis functions. The aim was to estimate the time complexity as a function of the prediction horizon. Hence, multi-step ahead prediction was performed with a variable prediction horizon (ranging between $10s$ and 10^4s) for a fixed data set. Furthermore, the run was repeated five times for each time horizon, and the mean is depicted on the left in Figure 4-29. The variance in the results was negligible relative to the magnitude of the reported means and is therefore not shown. Note that the computational time for the multi-step ahead predictions directly impacts the computational time for the policy optimization since a multi-step ahead prediction must be performed for every policy optimization step.

The second experiment considers a fixed time horizon of $T = 25s$ and one-dimensional control action. In this experiment, the dimension of the state space is taken as the variable of interest (ranging between 2 and 400 dimensions). The training targets were transformed through a sine wave to introduce non-linearity. Note that the goal of this experiment was not to obtain accurate multi-step ahead predictions; hence the GPs were not trained in this setup. Finally, the plots in Figure 4-29 are depicted on a log-log scale.



(a) Time complexity as a function of prediction horizon T



(b) Time complexity as a function of state space dimensionality

Figure 4-29: Comparison of time complexities PILCO and DKL PILCO, as a function of a fixed time horizon and as a function of the state space dimension.

Looking at the left in Figure 4-29, it can be seen that both DKL PILCO and PILCO share the same time complexity. However, the time complexity line of DKL PILCO is shifted down with respect to PILCO, indicating that it consistently performs 10 times more efficiently compared to PILCO.

To summarize, both figures in Figure 4-29 indicate that DKL PILCO is consistently more efficient on multi-step ahead predictions for varying input sizes as well as dimensionalities, compared to PILCO.

Chapter 5

Conclusion

This study shows that Deep Kernel Learning (DKL) can be used to control unknown systems by modifying the existing Probabilistic Inference for Learning COntrol (PILCO) framework. We proposed and developed Deep Kernel PILCO (DKL PILCO), an extension of the original PILCO framework. This extension incorporated an Neural Network (NN), used in sequence with a Gaussian Process (GP), and employed the Unscented Transform (UT) for uncertainty propagation instead of moment matching.

Our results indicated that DKL PILCO outperformed the original PILCO, on data efficiency, by requiring 50% fewer random trials to achieve a successful policy. Additionally, DKL PILCO achieved more robust policies against action, observation, and initial state disturbances on the mountain car problem.

Moreover, DKL PILCO was also able to improve the predictive accuracy of PILCO as well. On average, DKL PILCO achieved an Mean Squared Error (MSE) at least 10 times lower than PILCO on both the mountain car and the swimmer problem, indicating its scalability to higher dimensions. In addition, the residuals of DKL PILCO indicated that the model was making less systematic errors than PILCO. DKL PILCO's multi-step ahead predictions also yielded a higher precision than PILCO's, which can be potentially beneficial in planning with unsafe states.

Also, our findings suggest that DKL PILCO offers more scalability potential for complex problems than PILCO, as it exhibited a computational time of its multi-step ahead predictions that were, on average, about 10 times less than PILCO, regardless of the length of the prediction horizon or dimensionality.

Finally, DKL PILCO demonstrated superior numerical stability compared to PILCO, maintaining condition numbers of the covariance matrix that were generally not higher than 20, in contrast to PILCO which showed, in the worst case, large peaks of $300 - 400$.

While our findings demonstrate substantial improvements in various areas, it is important to acknowledge certain limitations that may affect the interpretation of our results.

Interaction between the GP and NN

Regarding model learning, we employed stochastic mini-batching to jointly train the network and GP. However, the batch size and shuffling usage could influence the GP's length scales, warranting further sensitivity analysis. In addition, it might also be interesting to see how restricting the length scales of the GP to a small specific range would affect the performance of the ensemble. One possible scenario could be that when restricting the GP to longer length scales, the NN might be forced to act as a low-pass filter, which could reduce the risk of overfitting. Furthermore, we utilized maximum likelihood for GP's hyper-parameter optimization, which requires appropriate hyper-parameter priors. A fully Bayesian approach might offer a more efficient alternative with less tuning [29] [41]. Moreover, the NN we used for feature extraction was a basic, Feedforward Neural Network (FNN). Future studies could explore the benefits of other network architectures, such as a Recurrent Neural Network (RNN), which could provide a more comprehensive system description by utilizing states from multiple steps back. Such an implementation would probably require more careful tuning but could be beneficial in cases where Markovian dynamics cannot describe the system [6].

Scalability of DKL PILCO

Furthermore, we did not use the sparse GP approximation of the original proposed DKL framework [104]. While using this approach will improve the scalability of the DKL to larger datasets, it would also introduce some risks, such as feature collapse [99]. These risks could be addressed with a bi-Lipschitz constraint or by replacing the Kernel Interpolation for Scalable Structured Gaussian Processes (KISS-GP) with Structured Kernel Interpolation of Products (SKIP) [32].

Uncertainty Propagation

Moreover, our version of the UT assumes a Gaussian state distribution. Significant deviation from this assumption could lead to approximation errors [26]. Therefore, future research could consider employing Generalized UT or particle filtering. Generalized UT, which also considers skewness and kurtosis tensors, has shown better accuracy for non-Gaussian distributions [26]. As implemented by Amadio et al. (2021) [2], particle filtering showed a significant improvement over PILCO. Particle filtering provides the flexibility of manual particle number setting to suit problem complexity [2], in contrast with UT, where the amount of sigma points is limited by the state dimension. However, particle filtering can introduce significant computational costs compared to UT; thus, careful tuning might be required. Furthermore, introducing stochastic sampling methods would likely require implementing the reparameterization trick [2]. However, using the reparameterization trick might result in exploding gradients and random directions [77], especially for long chains of nonlinear computations. Parmas et al. (2018) [77] developed the probabilistic inference for particle-based policy search (PIPPS) framework, which used likelihood ratio gradients and demonstrated more robust gradients compared to the gradients obtained with the reparameterization trick [77]. Hence, an implementation that combines the approach of Amadio et al. (2021), Parmas et al. (2018), and DKL would be worth exploring.

Another interesting option to propagate uncertainty would be Polynomial Chaos Expansion (PCE). PCE is a mathematical technique that represents a random variable or stochastic process by expanding it into a series of orthogonal polynomials chosen to match the statistical properties of the underlying uncertainty [14]. One of the benefits of PCE in the context of PILCO would be its ability to handle a wide variety of probability distributions, not just Gaussian distributions [14]. Successful implementations have already been made where the mean function of a GP is replaced by PCE and where the ensemble is used in the context of Model Predictive Control (MPC) [13].

Policy Parameterization

Furthermore, our policy is parameterized by an Radial Basis Function (RBF) network. While RBF networks can be highly effective in specific scenarios due to their universal approximation properties and their ability to handle multi-dimensional input data [76], RBF networks have some limitations that might impact their performance. The most prominent one is the curse of dimensionality [80]. As the number of dimensions in the input data increases, the number of RBF centers required for good performance grows exponentially [66]. As a result, the computational complexity can be high, especially for long prediction horizons and high-dimensional state spaces, since the distance from each data point to each basis function center needs to be computed. This can make RBF networks impractical for very high-dimensional problems [80]. An extension on the RBF network has been proposed, specifically to mitigate the curse of dimensionality, through an adaptive kernel incorporating the cosine distance measure alongside the Euclidian distance measure [55].

Another interesting option would be to replace the RBF network with a Deep Neural Network (DNN). One of the significant advantages of DNNs is their ability to learn hierarchical representations from raw input data [70]. This could potentially address the "curse of dimensionality" that RBF networks can suffer from as the DNN learns to extract and use only the most important features in its internal layers [10]. This could make DNNs more scalable than RBF networks. However, implementing DNNs could bring its own set of challenges. Generally, they need more data and longer training times, and due to their black-box nature, they are harder to interpret than RBF networks. Furthermore, using DNNs might increase the risk of catastrophic forgetting, where a previously learned task is forgotten due to fine-tuning the existing policy or learning a new task [27]. RBF networks are known to provide local approximations to the function they are learning [49], with each basis function typically only responding to inputs that are close to its center. Therefore, learning a new task (or new data) is less likely to drastically affect what the network has learned about older tasks (or older data).

Policy Optimization

In our framework, we use the ADAM optimizer for both model optimization and policy optimization. However, there have been cases where ADAM failed to converge [84]. Extensions on ADAM have been proposed that could offer improved convergence speed and stability, e.g., AMSGrad [98] or Nesterov ADAM [24]. Exploring these optimizers might be beneficial, especially when the problem's complexity grows.

Appendix A

Mathematical Background

A-1 Gaussian Distributions

Gaussian distributions are the cornerstone of GPs. Let us first start with a formal definition of the Gaussian distribution [101]:

Definition 4. *A random variable X has a Gaussian (or normal) distribution if its probability density function (pdf) is*

$$f(x) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left\{ -\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2 \right\}, \quad \textit{for } -\infty < x < \infty \tag{A-1}$$

The parameters μ and σ^2 are the mean and variance of X, respectively. Often the distribution of X is noted as $N\left(\mu, \sigma^2\right)$

Gaussian distributions play a significant role in many natural phenomena because most of these phenomena are approximately Gaussian [59]. This property is described in the Central Limit Theorem, which is stated as follows [43]:

Theorem 3. *Central Limit Theorem Let $X_1, X_2, \ldots, X_n$ be independent and identically (i.i.d.) distributed random variables with $E\left(X_i\right) = \mu$ and $Var\left(X_i\right) = \sigma^2 < \infty$. Define*

$$U_n = \frac{\sum_{i=1}^{n} X_i - n\mu}{\sigma\sqrt{n}} = \frac{\bar{X} - \mu}{\sigma/\sqrt{n}} \quad \textit{where } \bar{X} = \frac{1}{n}\sum_{i=1}^{n} X_i \tag{A-2}$$

Then the distribution of U_n converges to the standard normal distribution function as $n \to \infty$.

This theorem implies that if we make n large enough, we can approximate the probabilities of U_n by the corresponding probabilities of the standard normal distribution. In short [101]: asymptotically, $\bar{X}$ is normally distributed with mean μ and variance $\frac{\sigma^2}{n}$.

So far, we have discussed the Gaussian distribution in which we have one random variable (univariate). However, we can go one step further and treat multiple random variables (multivariate). A multivariate Gaussian distribution is a distribution in which the random variables are jointly Gaussian. Formally a multivariate Gaussian distribution is defined as follows [43]:

Definition 5. *A random vector* $\mathbf{X} \in \mathbb{R}^n$ *has a multivariate Gaussian (or normal) distribution if its probability density function (pdf) is*

$$f(\mathbf{x}; \mu, \mathbf{\Sigma}) = \frac{1}{(2\pi)^{k/2}|\mathbf{\Sigma}|^{1/2}} \exp\left\{-\frac{1}{2}(\mathbf{x}-\mu)^T \mathbf{\Sigma}^{-1}(\mathbf{x}-\mu)\right\} \tag{A-3}$$

where μ *is the mean vector of length* n *and* $\mathbf{\Sigma}$ *the* $n \times n$ *symmetric, positive definite covariance matrix.*

Gaussian distributions have properties which make them attractive for modelling. One of the nice features is their affine property [88]:

$$x \sim \mathcal{N}(\mu, \Sigma) \Leftrightarrow (Ax+b) \sim \mathcal{N}\left(A\mu + b, A\Sigma A^T\right) \tag{A-4}$$

which states that an affine transformation of a Gaussian distribution is Gaussian as well.

Another nice property is the fact that the marginal distribution of a multivariate Gaussian is Gaussian as well, which means it is closed under marginalization [10]. Let us consider the bivariate case:

Theorem 4. *Given a two random variables* X_a *and* X_b *which are jointly Gaussian according to* $f(X_a, X_b) \sim \mathcal{N}(\mu, \Sigma) = \mathcal{N}\left(\begin{bmatrix}\mu_a \\ \mu_b\end{bmatrix}, \begin{bmatrix}\Sigma_{aa} & \Sigma_{ab} \\ \Sigma_{ba} & \Sigma_{bb}\end{bmatrix}\right)$. *Then the marginal distribution of* X_a *is Gaussian as well:*

$$f(X_a) = \int f(X_a, X_b)\, dX_b \tag{A-5}$$

with its mean and covariance defined as follows:

$$\begin{aligned} E[X_a] &= \mu_a \\ \operatorname{cov}[X_a] &= \Sigma_{aa} \end{aligned} \tag{A-6}$$

We can also compare observations drawn from a normal distribution with different a different μ and σ between them [10]. We can standardize our observations by subtracting the mean and dividing it by its standard deviation [10]:

$$Z = \frac{X - \mu}{\sigma} \tag{A-7}$$

An additional useful property is that Gaussian distributions are closed under conditioning as well [10]. Conditional probabilities are defined as follows [88]:

Definition 6. *Given two events* A *and* B *with* $P(B) > 0$. *The conditional probability of event* A *given event* B *is defined as:*

$$P(A \mid B) = \frac{P(A \cap B)}{P(B)} \tag{A-8}$$

which means the probability that A *will occur given that* B *has already occurred.* $P(A \cap B$ *is the probability that both* A *and* B *will occur.*

This means that if we have a joint Gaussian distribution, then the conditional probability distribution is Gaussian as well, more specifically [10]:

Theorem 5. *Given a joint Gaussian distribution $\mathcal{N}(\mathbf{X} \mid \boldsymbol{\mu}, \boldsymbol{\Sigma})$ with $\boldsymbol{\Lambda} \equiv \boldsymbol{\Sigma}^{-1}$ and defined as follows:*

$$\begin{aligned} \mathbf{X} &= \begin{pmatrix} \mathbf{X}_a \\ \mathbf{X}_b \end{pmatrix}, \quad \boldsymbol{\mu} = \begin{pmatrix} \boldsymbol{\mu}_a \\ \boldsymbol{\mu}_b \end{pmatrix} \\ \boldsymbol{\Sigma} &= \begin{pmatrix} \boldsymbol{\Sigma}_{aa} & \boldsymbol{\Sigma}_{ab} \\ \boldsymbol{\Sigma}_{ba} & \boldsymbol{\Sigma}_{bb} \end{pmatrix}, \quad \boldsymbol{\Lambda} = \begin{pmatrix} \boldsymbol{\Lambda}_{aa} & \boldsymbol{\Lambda}_{ab} \\ \boldsymbol{\Lambda}_{ba} & \boldsymbol{\Lambda}_{bb} \end{pmatrix} \end{aligned} \tag{A-9}$$

Then the condtional distribution $p\left(\mathbf{X}_a \mid \mathbf{X}_b\right)$ is described by:

$$\begin{aligned} p\left(\mathbf{X}_a \mid \mathbf{X}_b\right) &= \mathcal{N}\left(\mathbf{X} \mid \boldsymbol{\mu}_{a|b}, \boldsymbol{\Lambda}_{aa}^{-1}\right) \\ \boldsymbol{\mu}_{a|b} &= \boldsymbol{\mu}_a - \boldsymbol{\Lambda}_{aa}^{-1}\boldsymbol{\Lambda}_{ab}\left(\mathbf{X}_b - \boldsymbol{\mu}_b\right) \end{aligned} \tag{A-10}$$

Bayes' theorem To help our understanding of the next property, it can be useful to treat Bayes' theorem first. Bayes' theorem is stated as follows [88]:

Theorem 6. *Given two events A and B with $P(B) > 0$. The conditional probability of event A given event B is defined as:*

$$P(A \mid B) = \frac{P(B \mid A)P(A)}{P(B)} \tag{A-11}$$

We call $P(A)$ and $P(B)$ the prior probabilities. The likelihood is denoted by $P(B \mid A)$ and the posterior by $P(A \mid B)$.

Finally, Gaussian distributions are conjugate distributions, meaning that starting with a Gaussian likelihood function and choosing a Gaussian prior will result in a Gaussian posterior [88]. More generally, conjugate distributions belong to the same probability distribution family if the prior and posterior distributions belong to the same probability distribution family. According to Bishop [10], the result can be summarized as follows:

Theorem 7. *Given a marginal Gaussian distribution for x and a conditional Gaussian distribution of y given x:*

$$\begin{aligned} p(\mathbf{x}) &= \mathcal{N}\left(\mathbf{x} \mid \boldsymbol{\mu}, \boldsymbol{\Lambda}^{-1}\right) \\ p(\mathbf{y} \mid \mathbf{x}) &= \mathcal{N}\left(\mathbf{y} \mid \mathbf{A}\mathbf{x} + \mathbf{b}, \mathbf{L}^{-1}\right) \end{aligned} \tag{A-12}$$

Then the marginal distribution of y and the conditional Gaussian distribution of x given y are:

$$\begin{aligned} p(\mathbf{y}) &= \mathcal{N}\left(\mathbf{y} \mid \mathbf{A}\boldsymbol{\mu} + \mathbf{b}, \mathbf{L}^{-1} + \mathbf{A}\boldsymbol{\Lambda}^{-1}\mathbf{A}^{\mathrm{T}}\right) \\ p(\mathbf{x} \mid \mathbf{y}) &= \mathcal{N}\left(\mathbf{x} \mid \boldsymbol{\Sigma}\left\{\mathbf{A}^{\mathrm{T}}\mathbf{L}(\mathbf{y} - \mathbf{b}) + \boldsymbol{\Lambda}\boldsymbol{\mu}\right\}, \boldsymbol{\Sigma}\right) \end{aligned} \tag{A-13}$$

where

$$\boldsymbol{\Sigma} = \left(\boldsymbol{\Lambda} + \mathbf{A}^{\mathrm{T}}\mathbf{L}\mathbf{A}\right)^{-1} \tag{A-14}$$

A-2 Cholesky Decomposition

Cholesky Decomposition is a mathematical technique used to decompose a Hermitian, positive-definite matrix into the product of a lower triangular matrix and its conjugate transpose [58]. Given a matrix $\mathbf{A} \in \mathbb{R}^{n \times n}$, then $\mathbf{A}$ can be decomposed as:

$$\mathbf{A} = \mathbf{L}\mathbf{L}^* \tag{A-15}$$

where $\mathbf{L}$ is a lower triangular matrix with real and positive diagonal entries, and $\mathbf{L}^*$ is the conjugate transpose of $\mathbf{L}$. The Cholesky Decomposition is used to compute the covariance matrix's square root efficiently, and the resulting lower triangular matrix is then used to generate the sigma points [58].

A-3 Tower Rule and Law of Total Variance

The tower rule, also known as the law of iterated expectations, is a fundamental property of conditional expectation. It can be stated as follows: Suppose X is a random variable with a defined $\mathrm{E}(X)$, and Y is a random variable on the same probability space. Then, the tower rule is defined as [88]:

$$\mathrm{E}(X) = \mathrm{E}(\mathrm{E}(X \mid Y)) \tag{A-16}$$

This means that the expectation of the conditional expectation of X given Y is equal to the expectation of X.

The law of total variance, sometimes referred to as the conditional variance formula or Eve's law, provides a way to express the variance of a random variable in terms of its conditional variance. It can be stated as follows [88]:

$$\mathrm{Var}(X) = \mathbb{E}(\mathrm{Var}(X \mid Y)) + \mathrm{Var}(\mathbb{E}(X \mid Y)) \tag{A-17}$$

The law of total variance partitions the variance of X into two components: the expected conditional variance and the variance of the conditional expectation.

A-4 Taylor Approximation

The Taylor approximation is a method used to represent a function as an infinite sum of terms calculated from the values of its derivatives at a single point. The Taylor series approximation of a function $f(x)$ at a is given by [93]:

$$f(x) \approx f(a) + f'(a)(x-a) + \frac{f''(a)}{2!}(x-a)^2 + \frac{f'''(a)}{3!}(x-a)^3 + \cdots \tag{A-18}$$

In general, the n-th degree Taylor polynomial of $f(x)$ about a is [34]:

$$P_n(x) = f(a) + f'(a)(x-a) + \frac{f''(a)}{2!}(x-a)^2 + \cdots + \frac{f^{(n)}(a)}{n!}(x-a)^n \tag{A-19}$$

Taylor's formula is then defined as [34]:

$$f(x) = f_n(x) + R_n(x) \tag{A-20}$$

where $R_n(x)$ is the remainder term.

Appendix B

Approximation Errors of the Unscented Transform

The following chapter will outline the approximation errors the Unscented Transform (UT) made when the underlying distribution is not Gaussian. All the derivations shown are based on the paper of Ebeigbe et al. (2021) [26]. Furthermore, they have also proposed an extension to the UT to mitigate this issue called the generalized UT. For more information on the generalized UT, please refer to the paper [26].

Suppose $\boldsymbol{x} \in \mathbb{R}^D$ is a random vector. Then the following can be defined, the mean $\overline{\boldsymbol{x}} \in \mathbb{R}^D$, covariance matrix $\boldsymbol{P} \in \mathbb{R}^{D\times D}$, skewness tensor $\boldsymbol{S} \in \mathbb{R}^{D\times D\times D}$, and kurtosis tensor $\boldsymbol{K} \in \mathbb{R}^{D\times D\times D\times D}$ as [26]:

$$\begin{aligned} \overline{\boldsymbol{x}} &= \mathbb{E}[\boldsymbol{x}] \\ \boldsymbol{P} &= \mathbb{E}\left[(\boldsymbol{x}-\overline{\boldsymbol{x}})(\boldsymbol{x}-\overline{\boldsymbol{x}})^T\right] \\ \boldsymbol{S}_{ijk} &= \mathbb{E}\left[(\boldsymbol{x}-\overline{\boldsymbol{x}})_i(\boldsymbol{x}-\overline{\boldsymbol{x}})_j(\boldsymbol{x}-\overline{\boldsymbol{x}})_k\right] \\ \boldsymbol{K}_{ijkl} &= \mathbb{E}\left[(\boldsymbol{x}-\overline{\boldsymbol{x}})_i(\boldsymbol{x}-\overline{\boldsymbol{x}})_j(\boldsymbol{x}-\overline{\boldsymbol{x}})_k(\boldsymbol{x}-\overline{\boldsymbol{x}})_l\right] \end{aligned} \tag{B-1}$$

for $i, j, k, l \in \{1, \cdots, D\}$.

Furthermore, define the nonlinear transformation $\boldsymbol{f}(\boldsymbol{x})$, with $\boldsymbol{x} \in \mathbb{R}^D$ a random vector.

Approximation Errors in the Mean

The true mean $\overline{\boldsymbol{y}}$ can be calculated, using a Taylor series expansion around $\overline{\boldsymbol{x}}$, with $\tilde{\boldsymbol{x}} = \boldsymbol{x} - \overline{\boldsymbol{x}}$ [26]:

$$\begin{aligned} \overline{\boldsymbol{y}} &= \mathbb{E}[\boldsymbol{f}(\boldsymbol{x})] \\ &= \boldsymbol{f}(\overline{\boldsymbol{x}}) + \mathbb{E}\left[D_{\tilde{\boldsymbol{x}}}\boldsymbol{f} + \frac{D^2_{\tilde{\boldsymbol{x}}}\boldsymbol{f}}{2!} + \frac{D^3_{\tilde{\boldsymbol{x}}}\boldsymbol{f}}{3!} + \frac{D^4_{\tilde{\boldsymbol{x}}}\boldsymbol{f}}{4!} + \cdots\right] \end{aligned} \tag{B-2}$$

where $D^k_{\tilde{x}} f$ is the kth total differential of $f(\boldsymbol{x})$ when perturbed by $\tilde{\boldsymbol{x}}$ around $\overline{\boldsymbol{x}}$, which can be written as [26]:

$$D^k_{\tilde{\boldsymbol{x}}}\boldsymbol{f} = \left(\sum_{i=1}^{D} \tilde{\boldsymbol{x}}_i \frac{\partial}{\partial \boldsymbol{x}_i}\right)^k \boldsymbol{f}(\boldsymbol{x})\Bigg|_{x=\overline{\boldsymbol{x}}} \tag{B-3}$$

Consequently, the true mean can then be evaluated as follows [26]:

$$\begin{aligned}\overline{\boldsymbol{y}} =& \boldsymbol{f}(\overline{\boldsymbol{x}}) + \Bigg\{\sum_{i,j=1}^{D} \frac{\boldsymbol{P}_{ij}}{2!}\frac{\partial^2 \boldsymbol{f}}{\partial \boldsymbol{x}_i \partial \boldsymbol{x}_j} + \sum_{i,j,k=1}^{D} \frac{\boldsymbol{S}_{ijk}}{3!}\frac{\partial^3 \boldsymbol{f}}{\partial \boldsymbol{x}_i \partial \boldsymbol{x}_j \partial \boldsymbol{x}_k} \\ &+ \sum_{i,j,k,l=1}^{D} \frac{\boldsymbol{K}_{ijkl}}{4!}\frac{\partial^4 \boldsymbol{f}}{\partial \boldsymbol{x}_i \partial \boldsymbol{x}_j \partial \boldsymbol{x}_k \partial \boldsymbol{x}_l}\Bigg\}_{\boldsymbol{x}=\overline{\boldsymbol{x}}} + \cdots\end{aligned} \tag{B-4}$$

where $\boldsymbol{P}_{ij} = \mathbb{E}\left[\tilde{\boldsymbol{x}}_i\tilde{\boldsymbol{x}}_j\right], \boldsymbol{S}_{ijk} = \mathbb{E}\left[\tilde{\boldsymbol{x}}_i\tilde{\boldsymbol{x}}_j\tilde{\boldsymbol{x}}_k\right]$, and $\boldsymbol{K}_{ijkl} = \mathbb{E}\left[\tilde{\boldsymbol{x}}_i\tilde{\boldsymbol{x}}_j\tilde{\boldsymbol{x}}_k\tilde{\boldsymbol{x}}_l\right]$. Now the approximated mean using the sigma points, using the Gaussian assumptions is then [26]:

$$\begin{aligned}\overline{\boldsymbol{y}}_u =& \boldsymbol{f}(\overline{\boldsymbol{x}}) + \frac{1}{2}\sum_{i,j=1}^{D} \boldsymbol{P}_{ij}\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{x}_i \partial \boldsymbol{x}_j}\Bigg|_{\boldsymbol{x}=\overline{\boldsymbol{x}}} \\ &+ \frac{1}{2(D+\kappa)}\sum_{i=1}^{2D}\left(\frac{D^4_{\sigma_i}\boldsymbol{f}}{4!} + \frac{D^6_{\sigma_i}\boldsymbol{f}}{6!} + \cdots\right)\end{aligned} \tag{B-5}$$

Now comparing Equation B-4 and Equation B-5, we can see that the odd-powered moments are not present in the approximated mean. This is a consequence of the Gaussian assumption used, which results in odd-powered moments always being zero due to the symmetry [88]. Furthermore, the fourth-order term cannot capture a part of the true kurtosis.

Approximation Errors in the Covariance Matrix

Similarly, the true covariance of y can be written as [26]:

$$\boldsymbol{P}_y = \mathbb{E}\left[(\boldsymbol{y}-\overline{\boldsymbol{y}})(\boldsymbol{y}-\overline{\boldsymbol{y}})^T\right] \tag{B-6}$$

Then, $\boldsymbol{y}-\overline{\boldsymbol{y}}$ is determined as [26]:

$$\boldsymbol{y}-\overline{\boldsymbol{y}} = D_{\tilde{\boldsymbol{x}}}\boldsymbol{f} + \frac{D^2_{\tilde{\boldsymbol{x}}}\boldsymbol{f}}{2!} + \frac{D^3_{\tilde{\boldsymbol{x}}}\boldsymbol{f}}{3!} - \mathbb{E}\left[\frac{D^2_{\tilde{\boldsymbol{x}}}\boldsymbol{f}}{2!} + \frac{D^3_{\tilde{\boldsymbol{x}}}\boldsymbol{f}}{3!}\right] + \cdots \tag{B-7}$$

Substituting this expression back into the original definition of the covariance, we obtain [26]:

$$\begin{aligned}\boldsymbol{P}_y =& \mathbb{E}\Bigg[D_{\tilde{\boldsymbol{x}}}\boldsymbol{f}\left(D_{\tilde{\boldsymbol{x}}}\boldsymbol{f}\right)^T + \frac{D^2_{\tilde{\boldsymbol{x}}}\boldsymbol{f}\left(D_{\tilde{\boldsymbol{x}}}\boldsymbol{f}\right)^T}{2!} + \frac{D_{\tilde{\boldsymbol{x}}}\boldsymbol{f}\left(D^2_{\tilde{\boldsymbol{x}}}\boldsymbol{f}\right)^T}{2!} \\ &+ \frac{D^3_{\tilde{\boldsymbol{x}}}\boldsymbol{f}\left(D_{\tilde{\boldsymbol{x}}}\boldsymbol{f}\right)^T}{3!} + \frac{D_{\tilde{\boldsymbol{x}}}\boldsymbol{f}\left(D^3_{\tilde{\boldsymbol{x}}}\boldsymbol{f}\right)^T}{3!} + \frac{D^2_{\tilde{\boldsymbol{x}}}\boldsymbol{f}\left(D^2_{\tilde{\boldsymbol{x}}}\boldsymbol{f}\right)^T}{2! \times 2!}\Bigg] \\ &+ \mathbb{E}\left[\frac{D^2_{\tilde{\boldsymbol{x}}}\boldsymbol{f}}{2!}\right]\mathbb{E}\left[\frac{D^2_{\tilde{\boldsymbol{x}}}\boldsymbol{f}}{2!}\right]^T + \cdots\end{aligned} \tag{B-8}$$

where the first term of the expression can be written as [26]:

$$\mathbb{E}\left[D_{\tilde{x}}\boldsymbol{f}\left(D_{\tilde{x}}\boldsymbol{f}\right)^T\right] = \left.\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{x}}\right|_{\boldsymbol{x}=\overline{\boldsymbol{x}}} \mathbb{E}\left[\tilde{\boldsymbol{x}}\tilde{\boldsymbol{x}}^T\right] \left.\frac{\partial \boldsymbol{f}^T}{\partial \boldsymbol{x}}\right|_{\boldsymbol{x}=\overline{\boldsymbol{x}}} = \boldsymbol{f}\boldsymbol{P}\boldsymbol{f}^T \tag{B-9}$$

Then, the true covariance matrix can written as [26]:

$$\begin{aligned} P_y =& \boldsymbol{f}\boldsymbol{P}\boldsymbol{f}^T + \Bigg\{ \sum_{i,j,k=1}^{D} \frac{\boldsymbol{S}_{ijk}}{2!}\left[\frac{\partial^2 \boldsymbol{f}}{\partial \boldsymbol{x}_i \partial \boldsymbol{x}_j}\frac{\partial \boldsymbol{f}^T}{\partial \boldsymbol{x}_k} + \frac{\partial \boldsymbol{f}}{\partial \boldsymbol{x}_i}\frac{\partial^2 \boldsymbol{f}^T}{\partial \boldsymbol{x}_j \partial \boldsymbol{x}_k}\right] \\ & + \sum_{i,j,k,l=1}^{D} \boldsymbol{K}_{ijkl}\Bigg[\frac{1}{3!}\frac{\partial^3 \boldsymbol{f}}{\partial \boldsymbol{x}_i \partial \boldsymbol{x}_j \partial \boldsymbol{x}_k}\frac{\partial \boldsymbol{f}^T}{\partial \boldsymbol{x}_l} \\ & + \frac{1}{3!}\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{x}_i}\frac{\partial^3 \boldsymbol{f}^T}{\partial \boldsymbol{x}_j \partial \boldsymbol{x}_k \partial \boldsymbol{x}_l} + \frac{1}{4}\frac{\partial^2 \boldsymbol{f}}{\partial \boldsymbol{x}_i \partial \boldsymbol{x}_j}\frac{\partial^2 \boldsymbol{f}^T}{\partial \boldsymbol{x}_k \partial \boldsymbol{x}_l}\Bigg] \\ & + \left[\sum_{i,j=1}^{D}\frac{\boldsymbol{P}_{ij}}{2}\frac{\partial^2 \boldsymbol{f}}{\partial \boldsymbol{x}_i \partial \boldsymbol{x}_j}\right]\left[\sum_{i,j=1}^{D}\frac{\boldsymbol{P}_{ij}}{2}\frac{\partial^2 \boldsymbol{f}}{\partial \boldsymbol{x}_i \partial \boldsymbol{x}_j}\right]^T \Bigg\}_{\boldsymbol{x}=\overline{\boldsymbol{x}}} + \cdots \end{aligned} \tag{B-10}$$

Now, the approximated covariance matrix calculated with the sigma points is then given as [26]:

$$\begin{aligned} \boldsymbol{P}_u =& \boldsymbol{f}P\boldsymbol{f}^T + \frac{1}{2(n+\kappa)}\sum_{i}^{2D}\Bigg(\frac{D_{\sigma_i}\boldsymbol{f}\left(D^3_{\sigma_i}\boldsymbol{f}\right)^T}{3!} \\ & + \frac{D^3_{\sigma_i}\boldsymbol{f}\left(D_{\sigma_i}\boldsymbol{f}\right)^T}{3!} + \frac{D^2_{\sigma_i}\boldsymbol{f}\left(D^2_{\sigma_i}\boldsymbol{f}\right)^T}{2! \times 2!}\Bigg) \\ & + \left[\frac{1}{2}\sum_{i,j=1}^{D}\boldsymbol{P}_{ij}\left.\frac{\partial^2 \boldsymbol{f}}{\partial \boldsymbol{x}_i \partial \boldsymbol{x}_j}\right|_{\boldsymbol{x}=\overline{\boldsymbol{x}}}\right]\left[\frac{1}{2}\sum_{i,j=1}^{D}\boldsymbol{P}_{ij}\left.\frac{\partial^2 \boldsymbol{f}}{\partial \boldsymbol{x}_i \partial \boldsymbol{x}_j}\right|_{\boldsymbol{x}=\overline{\boldsymbol{x}}}\right]^T + \cdots \end{aligned} \tag{B-11}$$

Again, we have a similar situation as for the approximated mean. When the underlying distribution of $\boldsymbol{x}$ is not Gaussian, then the approximation is only accurate up to the second order [26].

Bibliography

[1] C.C. Aggarwal. *Neural Networks and Deep Learning A Textbook.* Springer, 2018.

[2] Fabio Amadio, Juan Antonio Delgado-Guerrero, Adrià Colomé, and Carme Torras. Controlled Gaussian Process Dynamical Models with Application to Robotic Cloth Manipulation. 2021.

[3] Mohammadreza Amirian and Friedhelm Schwenker. Radial Basis Function Networks for Convolutional Neural Networks to Learn Similarity Distance Metric and Improve Interpretability. *IEEE Access*, 8:123087–123097, 2020.

[4] Haim Avron, Alex Druinsky, and Sivan Toledo. Spectral condition-number estimation of large sparse matrices. *Numerical Linear Algebra with Applications*, 26(3):1–25, 2019.

[5] Randall Balestriero and Richard G. Baraniuk. Batch Normalization Explained. (2):1–21, 2022.

[6] Leonardo Banchi, Edward Grant, Andrea Rocchetto, and Simone Severini. Modelling non-markovian quantum processes with recurrent neural networks. *New Journal of Physics*, 20(12):1–11, 2018.

[7] Bastien Batardière, Julien Chiquet, and Joon Kwon. Finite-sum optimization: Adaptivity to smoothness and loopless variance reduction. 2023.

[8] Matthias Bauer, Mark Van Der Wilk, and Carl Edward Rasmussen. Understanding Probabilistic Sparse Gaussian Process Approximations. (Nips), 2016.

[9] D.P. Bertsekas. *Dynamic Programming and Optimal Control Vol I.* Athena Scientific, 2005.

[10] C. Bishop. *Pattern Recognition and Machine Learning.* Springer, 1 edition.

[11] Åke Björck. *Numerical Methods in Matrix Computations*, volume 59. 2015.

[12] Raghu Bollapragada, Dheevatsa Mudigere, Jorge Nocedal, Hao Jun Michael Shi, and Ping Tak Peter Tang. A Progressive Batching L-BFGS Method for Machine Learning. *35th International Conference on Machine Learning, ICML 2018*, 2:989–1013, 2018.

[13] E. Bradford and L. Imsland. Combining Gaussian processes and polynomial chaos expansions for stochastic nonlinear model predictive control. pages 1–39, 2021.

[14] Paul Christian Bürkner, Ilja Kröker, Sergey Oladyshkin, and Wolfgang Nowak. A fully Bayesian sparse polynomial chaos expansion approach with joint priors on the coefficients and global selection of terms. *Journal of Computational Physics*, 488:1–56, 2023.

[15] Halisson Alberdan Cavalcanti Cardoso, Silvio de Barros Melo, Ricardo Martins de Abreu Silva, Sidartha Azevedo Lobo de Carvalho, Silas Garrido Teixeira de Carvalho Santos, and Carlos Costa Dantas. Spectral analysis and optimization of the condition number problem. *Computer Physics Communications*, 258:107587, 2021.

[16] Hannah E Clapham, Vianney Tricou, Nguyen Van Vinh Chau, Cameron P Simmons, and Neil M Ferguson. Identifiability analysis. *Journal of the Royal Society, Interface / the Royal Society*, 11(2):2–4, 2014.

[17] G. Cybenko. Approximation by Superpositions of a Sigmoidal Function. *Mathematics of Control, Signals and Systems*, 1989.

[18] Renato De Mori. Recent advances in automatic speech recognition. *Signal Processing*, 1(2):95–123, 1979.

[19] Marc Deisenroth. *Efficient Reinforcement Learning using Gaussian Processes.* Number November 2010. 2010.

[20] Marc Peter Deisenroth, Dieter Fox, and Carl Edward Rasmussen. Gaussian processes for data-efficient learning in robotics and control. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 37(2):408–423, 2015.

[21] Marc Peter Deisenroth, Carl Edward Rasmussen, and Jan Peters. Gaussian process dynamic programming. *Neurocomputing*, 72(7-9):1508–1524, 2009.

[22] Felipe Bottega Diniz. Condition number and matrices. pages 1–13, 2017.

[23] Hao Dong, Zihan Ding, and Shanghang Zhang. *Deep Reinforcement Learning: Fundamentals, Research and Applications.* Springer, 2020.

[24] Timothy Dozat. Incorporating Nesterov Momentum into Adam. *ICLR Workshop*, (1):2013–2016, 2016.

[25] David Kristjanson Duvenaud. Automatic Model Construction with Gaussian Processes. *PhD Thesis, University of Cambridge*, (June):144, 2014.

[26] Donald Ebeigbe, Tyrus Berry, Michael M Norton, Andrew J Whalen, Dan Simon, Timothy Sauer, and Steven J Schiff. A Generalized Unscented Transformation for Probability Distributions. *ArXiv*, pages 1–15, 2021.

[27] Sami Ede, Serop Baghdadlian, Leander Weber, An Nguyen, Dario Zanca, Wojciech Samek, and Sebastian Lapuschkin. Explain to Not Forget: Defending Against Catastrophic Forgetting with XAI. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 13480 LNCS:1–18, 2022.

[28] Magnus Ekman. *Learning Deep Learning.* Addison-Wesley, 2022.

[29] Peter I. Frazier. A Tutorial on Bayesian Optimization. (Section 5):1–22, 2018.

[30] Scott Fujimoto, Herke Van Hoof, and David Meger. Addressing Function Approximation Error in Actor-Critic Methods. *35th International Conference on Machine Learning, ICML 2018*, 4:2587–2601, 2018.

[31] Juan Camilo Gamboa Higuera, David Meger, and Gregory Dudek. Synthesizing Neural Network Controllers with Probabilistic Model-Based Reinforcement Learning. *IEEE International Conference on Intelligent Robots and Systems*, pages 2538–2544, 2018.

[32] Jacob R. Gardner, Geoff Pleiss, Ruihan Wu, Kilian Q. Weinberger, and Andrew Gordon Wilson. Product kernel interpolation for scalable gaussian processes. *International Conference on Artificial Intelligence and Statistics, AISTATS 2018*, 84:1407–1416, 2018.

[33] Aurélien Géron. *Hands-on Machine Learning whith Scikit-Learing, Keras and Tensorfow.* 2019.

[34] Agathe Girard, Carl Edward Rasmussen, Joaquin Quiñonero Candela, and Roderick Murray-Smith. Gaussian process priors with uncertain inputs application to multiple-step ahead time series forecasting. *Advances in Neural Information Processing Systems*, 2003.

[35] Ian J. Goodfellow. *Deep Learning.* 2016.

[36] Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. *3rd International Conference on Learning Representations, ICLR 2015 - Conference Track Proceedings*, pages 1–11, 2015.

[37] Atılım Güneş Baydin, Barak A. Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. Automatic differentiation in machine learning: A survey. *Journal of Machine Learning Research*, 18:1–43, 2018.

[38] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. *35th International Conference on Machine Learning, ICML 2018*, 5:2976–2989, 2018.

[39] D Harrison. A Brief Introduction to Automatic Differentiation for Machine Learning. *arXiv preprint arXiv:2110.06209*, pages 1–8, 2021.

[40] Kohei Hayashi, Masaaki Imaizumi, and Yuichi Yoshida. On Random Subsampling of Gaussian Process Regression: A Graphon-Based Analysis. 2019.

[41] Ali Hebbal, Loic Brevault, Mathieu Balesdent, El-Ghazali Talbi, and Nouredine Melab. Bayesian Optimization using Deep Gaussian Processes. 2019.

[42] N. Hensman, J., Fusi, N., Lawrence. Gaussian Processes for Big Data. In *Proceedings of the Twenty-Ninth Conference on Uncertainty in Artificial Intelligence (UAI2013)*, 2013.

[43] R.V. Hogg, J.W. McKean, and A.T. Craig. *Introduction to Mathematical Statistics.* Pearson, 8 edition, 2018.

[44] Roger A. Horn and Charles R. Johnson. *Matrix Analysis.* 2013.

[45] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5):359–366, 1989.

[46] Ke-Wei Huang, Mengke Qiao, Xuanqi Liu, Siyuan Liu, and Mingxi Dai. Computer Vision and Metrics Learning for Hypothesis Testing: An Application of Q-Q Plot for Normality Test. pages 1–14, 2019.

[47] Jan Hückelheim, Harshitha Menon, William Moses, Bruce Christianson, Paul Hovland, and Laurent Hascoët. Understanding Automatic Differentiation Pitfalls. pages 1–14, 2023.

[48] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *32nd International Conference on Machine Learning, ICML 2015*, 1:448–456, 2015.

[49] Aysu Ismayilova and Muhammad Ismayilov. On the universal approximation property of radial basis function neural networks. pages 1–11, 2023.

[50] John Jackson, Luca Laurenti, Eric Frew, and Morteza Lahijanian. Safety Verification of Unknown Dynamical Systems via Gaussian Process Regression. *Proceedings of the IEEE Conference on Decision and Control*, 2020-Decem:860–866, 2020.

[51] Sanket Kamthe and Marc Peter Deisenroth. Data-efficient reinforcement learning with probabilistic model predictive control. *International Conference on Artificial Intelligence and Statistics, AISTATS 2018*, 84:1701–1710, 2018.

[52] Motonobu Kanagawa, Philipp Hennig, Dino Sejdinovic, and Bharath K Sriperumbudur. Gaussian Processes and Kernel Methods: A Review on Connections and Equivalences. pages 1–64, 2018.

[53] Karl Johan Åström. *Introduction to stochastic control theory.* Academic Press, 1970.

[54] Sydney M. Katz, Kyle D. Julian, Christopher A. Strong, and Mykel J. Kochenderfer. Generating Probabilistic Safety Guarantees for Neural Network Controllers. pages 0–2, 2021.

[55] Shujaat Khan, Imran Naseem, Roberto Togneri, and Mohammed Bennamoun. A Novel Adaptive Kernel for the RBF Neural Networks. *Circuits, Systems, and Signal Processing*, 36(4):1639–1653, 2017.

[56] Diederik P. Kingma and Jimmy Lei Ba. Adam: A method for stochastic optimization. *3rd International Conference on Learning Representations, ICLR 2015 - Conference Track Proceedings*, pages 1–15, 2015.

[57] D. Kirk. *Optimal control theory: An introduction.* Dover Publications, Inc., 2004.

[58] Juho Kokkala, Arno Solin, and A. S. Arkkä. Sigma-Point filtering and smoothing based parameter estimation in nonlinear dynamic systems. *Journal of Advances in Information Fusion*, 11(1):15–30, 2016.

[59] Larry Wasserman. All of Statistcs, A Concies Course in Statistical Inference. *Springer*, pages 327–348, 2000.

[60] Aristotelis Lazaridis and Ioannis Vlahavas. REIN-2: Giving birth to prepared reinforcement learning agents using reinforcement learning agents. *Neurocomputing*, 497:86–93, 2022.

[61] Johannes Lederer. Activation Functions in Artificial Neural Networks: A Systematic Overview. pages 1–42, 2021.

[62] Xiangru Lian and Ji Liu. Revisit batch normalization: New understanding and refinement via composition optimization. *AISTATS 2019 - 22nd International Conference on Artificial Intelligence and Statistics*, 2020.

[63] Haitao Liu, Jianfei Cai, Yew Soon Ong, and Yi Wang. Understanding and comparing scalable Gaussian process regression for big data. *Knowledge-Based Systems*, 164:324–335, 2019.

[64] D MacKay. Introduction to Gaussian Processes. *Book Neural Networks and Machine Learning, Springer-Verlag*, pages 84–92, 1998.

[65] Charles C. Margossian. A review of automatic differentiation and its efficient implementation. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 9(4):1–32, 2019.

[66] Momcilo Markus. Artificial Neural Networks. *Hydroinformatics: Data Integrative Approaches in Computation, Analysis, and Modeling*, pages 275–320, jan 2007.

[67] Rowan Thomas McAllister and Carl Edward Rasmussen. Data-efficient reinforcement learning in continuous state-action Gaussian-POMDPs. *Advances in Neural Information Processing Systems*, 2017-Decem:2041–2050, 2017.

[68] Andrew McHutchon. Nonlinear Modelling and Control using Gaussian Processes. *Thesis*, (August), 2014.

[69] Charles A. Micchelli, Yuesheng Xu, and Haizhang Zhang. Universal kernels. *Journal of Machine Learning Research*, 7:2651–2667, 2006.

[70] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing Atari with Deep Reinforcement Learning. pages 1–9, 2013.

[71] Thomas M. Moerland, Joost Broekens, Aske Plaat, and Catholijn M. Jonker. Model-based Reinforcement Learning: A Survey. *Foundations and Trends in Machine Learning*, 16(1):1–118, 2022.

[72] Cleve B. Moler. *Numerical Computing with Matlab.* Society for Industrial and Applied Mathematics, 2004.

[73] K. Murphy. *Machine Learning A Probabilistic Perspective.* The MIT Press, 2012.

[74] Kristin Nielsen, Caroline Svahn, Hector Rodriguez-Deniz, and Gustaf Hendeby. UKF Parameter Tuning for Local Variation Smoothing. *IEEE International Conference on Multisensor Fusion and Integration for Intelligent Systems*, pages 1–8, 2021.

[75] Sebastian W. Ober, Carl E. Rasmussen, and Mark van der Wilk. The Promises and Pitfalls of Deep Kernel Learning. (Uai), 2021.

[76] J Park and I W Sandberg. Universal approximation using radial-basis-function networks. *Neural Computation*, 3(2):246–257, 1991.

[77] Paavo Parmas, Carl Edward Rasmussen, Jan Peters, and Kenji Doya. PIPPS: Flexible Model-Based Policy Search Robust to the Curse of Chaos BT - International Conference on Machine Learning. *International Conference on Machine Learning*, pages 4065–4074, 2018.

[78] Gustavo H.A. Pereira. On quantile residuals in beta regression. *Communications in Statistics: Simulation and Computation*, 48(1):302–316, 2019.

[79] Quang Pham, Chenghao Liu, and Steven C.H. Hoi. Continual Normalization: Rethinking Batch Normalization for Online Continual Learning. *ICLR 2022 - 10th International Conference on Learning Representations*, 2022.

[80] D. Popovic. *Intelligent Control with Neural Networks.* Academic Press, 2000.

[81] Joaquin Qui. A Unifying View of Sparse Approximate Gaussian Process Regression. 6:1939–1959, 2005.

[82] Antonin Raffin. RL Baselines3 Zoo, 2020.

[83] C.E. Rasmussen and C.K.I. Williams. *Gaussian Processes for Machine Learning.* Number 5. 2006.

[84] Sashank J. Reddi, Satyen Kale, and Sanjiv Kumar. On the convergence of Adam and beyond. *6th International Conference on Learning Representations, ICLR 2018 - Conference Track Proceedings*, pages 1–23, 2018.

[85] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 2016-Decem:779–788, 2016.

[86] James Reinders, Ben Ashbaugh, James Brodman, Michael Kinsner, John Pennycook, and Xinmin Tian. *Data parallel C++: Mastering DPC++ for programming of heterogeneous systems using C++ and SYCL.* 2020.

[87] F. Rosenblatt. The Perceptron - A Perceiving and Recognizing Automaton, 1957.

[88] Sheldon M. Ross. *Introduction to Probability Models.* Academic Press, 2019.

[89] Denis Serre. *Matrices: Theory and Applications*, volume 96. Springer, Lyon, France, 2002.

[90] Hidetoshi Shimodaira. Improving predictive inference under covariate shift by weighting the log-likelihood function. *Journal of Statistical Planning and Inference*, 90(2):227–244, 2000.

[91] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George Van Den Driessche, Thore Graepel, and Demis Hassabis. Mastering the game of Go without human knowledge. *Nature*, 550(7676):354–359, 2017.

[92] Edward Snelson and Zoubin Ghahramani. Sparse Gaussian Processes using Pseudo-inputs. pages 1–24, 2006.

[93] James Stewart, Daniel Clegg, and Saleem Watson. *Calculus: Early Transcendentals*. 2021.

[94] Richard S. Sutton and Andrew G. Barto. *Reinforcement Leaning*. 2018.

[95] Phillip Swazinna, Steffen Udluft, Daniel Hein, and Thomas Runkler. Comparing Model-free and Model-based Algorithms for Offline Reinforcement Learning. *IFAC-PapersOnLine*, 55(15):19–26, 2022.

[96] M.K. Titsias. Variational Learning of Inducing Variables in Sparse Gaussian Processes. 2009.

[97] Omar G. Towers, Mark and Terry, Jordan K. and Kwiatkowski, Ariel and Balis, John U. and Cola, Gianluca de and Deleu, Tristan and Goulão, Manuel and Kallinteris, Andreas and KG, Arjun and Krimmel, Markus and Perez-Vicente, Rodrigo and Pierré, Andrea and Schulhoff,. Gymnasium, 2023.

[98] Phuong Thi Tran and Le Trieu Phong. On the Convergence Proof of AMSGrad and a New Version. *IEEE Access*, 7:61706–61716, 2019.

[99] Joost van Amersfoort, Lewis Smith, Andrew Jesson, Oscar Key, and Yarin Gal. On Feature Collapse and Deep Kernel Learning for Single Forward Pass Uncertainty. 2021.

[100] Alexander Von Rohr, Friedrich Solowjow, and Sebastian Trimpe. Improving the Performance of Robust Control through Event-Triggered Learning. *Proceedings of the IEEE Conference on Decision and Control*, 2022-Decem:3424–3430, 2022.

[101] D. Wackerly, W. Mendenhall III, and R. Scheaffer. *Mathematical statistics with applications, 4th edition*. Thomson Learning, 2008.

[102] Eric Wan and Rudolph van der Merwe. The Unscented Kalman filter for non-linear estimation. In *Proceedings of the IEEE 2000 Adaptive Systems for Signal Processing, Communications, and Control Symposium*, 2000.

[103] Andrew Gordon Wilson, Christoph Dann, and Hannes Nickisch. Thoughts on Massively Scalable Gaussian Processes. pages 1–25, 2015.

[104] Andrew Gordon Wilson, Zhiting Hu, Ruslan Salakhutdinov, and Eric P. Xing. Deep kernel learning. *Proceedings of the 19th International Conference on Artificial Intelligence and Statistics, AISTATS 2016*, (1998):370–378, 2016.

[105] Andrew Gordon Wilson and Hannes Nickisch. Kernel interpolation for scalable structured Gaussian processes (KISS-GP). *32nd International Conference on Machine Learning, ICML 2015*, 3:1775–1784, 2015.

[106] R. Wisniewski and M. Bujorianu. Safe Dynamic Programming. 2021.

[107] Fengji Yi, Wenlong Fu, and Huan Liang. Model-based reinforcement learning: A survey. *Proceedings of the International Conference on Electronic Business (ICEB)*, 2018-Decem:421–429, 2018.

[108] Yiqun Zhang and Houbiao Li. A new distance measurement and its application in K-Means Algorithm. pages 1–12, 2022.

Glossary

List of Acronyms

Adam	Adaptive Moment Estimation
ARD	Automatic Relevance Determination
auto-diff	Automatic Differentiation
BFGS	Broyden–Fletcher–Goldfarb–Shanno
BNN	Bayesian Neural Networks
DNN	Deep Neural Network
DKL	Deep Kernel Learning
DKL PILCO	Deep Kernel PILCO
FNN	Feedforward Neural Network
GP	Gaussian Process
iid	Independent and Identically Distributed
KISS-GP	Kernel Interpolation for Scalable Structured Gaussian Processes
L-BFGS	Limited-Memory BFGS
MAE	Mean Absolute Error
MPC	Model Predictive Control
MSE	Mean Squared Error
ML	Machine Learning
MLP	Multi-Layer Perceptron
NN	Neural Network
PCE	Polynomial Chaos Expansion
PILCO	Probabilistic Inference for Learning COntrol
PSD	Positive Semi-Definite
Q-Q	Quantile-Quantile
ReLU	Rectified Linear Unit

RBF	Radial Basis Function
RL	Reinforcement Learning
RNN	Recurrent Neural Network
SAC	Soft Actor-Critic
SE	Squared Exponential
SGD	Stochastic Gradient Descent
SKI	Structured Kernel Interpolation
SKIP	Structured Kernel Interpolation of Products
TLU	Threshold Logic Unit
TD3	Twin Delayed Deep Deterministic Policy Gradients
UT	Unscented Transform