UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat de Matemàtiques i Estadística

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Escola Tècnica Superior d'Enginyeria
de Telecomunicació de Barcelona

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Centre de Formació Interdisciplinària Superior

Degree in Mathematics - Degree in Engineering Physics

Bachelor's Degree Thesis

Gaussian Process applied to modeling the dynamics of a deformable material

Ce Xu Zheng

September, 2021

Supervised by:

Luis Sentís (UT at Austin)

Carme Torras (UPC)

Adrià Colomé (UPC)

Abstract

In this thesis, we establish the theoretical basis of dimensional reduction algorithms like the Gaussian Process Latent Variable Model, that captures the best information we can get with the latent dimensions we are given by marginalizing over the reconstruction parameters and optimizing over the latent variables itself. Their application to the reproduction of a time series of observable data via a Markov chain dependency with the Gaussian Process Dynamical Model, and its generalization with control, the Controlled GPDM. Then, we are going to introduce a new model, which is more time efficient and better at generalization, the Mixture of CGPDM applying the mixture of experts' ideas to the problem. And the last section will consist in fine-tuning the model and comparing it to the previous model.
Keywords:

Machine learning, Gaussian Process, Dynamical models, Mixture of Experts

MSC2020: 68T40

Resumen

En esta tesis, estableceremos la base teórica de algunos algoritmos de reducción de la dimensionalidad como el Gaussian Process Latent Variable Model, que captura la mejor información que podemos obtener con las dimensiones del espacio latente que nos son dadas marginalizando los parámetros que usamos para la reconstrucción de los datos y optimizando sobre las propias variables latentes; su aplicación a la reproducción de una serie temporal de observables aplicando la dependencia de una cadena de Markov con el Gaussian Process Dynamical Model, y su generalización con control, el Controlled GPDM. Finalmente, vamos a introducir un nuevo modelo, que es más eficiente en tiempo de cálculo y generaliza mejor, el Mixture of CGPDM aplicando la idea de una mezcla de expertos. La última sección consistirá en afinar el modelo y compararlo con el modelo previo.
Palabras clave:

Aprendizaje automático, Procesos Gaussianos, Modelos dinámicos, mezcla de expertos

MSC2020: 68T40

Resum

En aquesta tesi, establirem la base teòrica d'alguns algoritmes de reducció de la dimensionalitat com el Gaussian Process Latent Variable Model, que captura la millor informació que podem obtenir amb les dimensions de l'espai latent que ens són donades marginalitzant els paràmetres que fem servir per a la reconstrucció de les dades i optimitzant sobre les pròpies

variables latents; seva aplicació a la reproducció d'una sèrie temporal d'observables aplicant la dependència d'una cadena de Markov amb el Gaussian Process Dynamical Model, i la seva generalització amb control, el Controlled GPDM. Finalment, anem a introduir un nou model, que és més eficient en temps de càlcul i generalitza millor, el Mixture of CGPDM aplicant la idea d'una barreja d'experts. L'última secció consistirà en afinar el model i comparar-lo amb el model previ.
Paraules clau: Aprenentatge automàtic, Processos Gaussians, Models dinàmics, barreja d'experts

MSC2020: 68T40

Acknowledgement

I would like to give special thanks to Dr. Luis Sentis for supervising and accompany me along the completion of the thesis even despite the distance that separates us, and also special thanks to Dra. Carme Torras and Adrià Colomé for welcoming me at the IRI.

Thanks to the Interdisciplinary Higher Education Center (CFIS) that allowed me to study two Bachelor Degrees simultaneously, and all the staff that helped all they could every time I needed.

And last but not least, thanks to my parents, that have always supported me, and specially these last days, when my lack of sleep was worrying them, and my friends and relatives.

Contents

1 Introduction 2

2 Theoretical background 3
2.1 Gaussian Process . . . 3
2.2 Gaussian Process Latent Variable Model . . . 7
2.3 Gaussian Process Dynamical Model . . . 9
2.4 Controlled Gaussian Process Dynamical Models . . . 12
2.5 Mixture of Experts . . . 13
2.6 Infinite Mixture of Gaussian Experts . . . 14

3 Mixture of Controlled Gaussian Process Dynamical Model 17
3.1 Background Theory . . . 17
3.2 Designing choices . . . 19
3.2.1 Experts structure and kernels . . . 19
3.2.2 Latent dimensions . . . 22
3.2.3 Training Algorithm . . . 23
3.2.4 Covariance matrix inverse management . . . 27
3.2.5 Predictions Algorithm . . . 30
3.2.6 Observation to latent space . . . 33
3.2.7 Balance hyperparameter . . . 34
3.2.8 PyTorch . . . 35
3.2.9 Optimization algorithms . . . 36

4 Experiments 40
4.1 MoCGPDM tuning . . . 41
4.1.1 Equilibrium betweeen experts training and Gibbs sampling of the indicators . 42
4.1.2 Balance hyperparameter . . . 46
4.1.3 Maximum size of the experts . . . 48
4.1.4 Predictions algorithm . . . 50
4.2 Comparing CGPDM with the proposed MoCGPDM . . . 51

5 Conclusions 54

1. Introduction

In robotics, the control and manipulation of objects is one of the biggest and most prolific fields of investigation. Although significant advances have been done over the manipulation of rigid bodies, the manipulation of non-rigid objects is still challenging and a state-of-the-art problem. The main difficulties come from the high uncertainty that the motion of deformable bodies have, and we cannot parametrize nor control easily. To overcome those inconveniences, we must rely on a robust perception module that can perceive the real state of the object with very little error, and a very effective control policy.

To obtain a good control policy, it has been shown that model-based control performs much better than the standard feedback control, especially when we have to deal with very nonlinear systems. These policies rely on the capacity of knowing how the system will behave given certain control and the state of the system to get ahead and make the appropriate decisions.

The modeling of non-rigid objects typically involves the approximation and prediction of a high number of parameters (sometimes arbitrarily many elements, like in the dynamic finite element methods) which will lead to a great computational cost. And almost always, prior knowledge of some parameters of the materal is needed to infer the objects behaviour.

To avoid these limitations, most of the models make the assumption of quasi-static manipulation (to elude complicated situations like the folding of a cloth), but some other data-driven models have been proposed [1] with only collision-free assumptions, and which are capable of making dynamical predictions by only feeding them with time series of data points. This involves a dimensional reduction of the observable data to make it computationally tractable.

In this thesis, we were motivated to improve the time efficiency of state-of-the-art models like the CGPDM, as those models are not capable to make enough predictions in real time, and they are very limited in the amount of data that they can digest while keeping the usability of the model.

2. Theoretical background

In this section, we introduce all the previous work that formed a path to the development of the model that we are presenting. It starts with the definition of a Gaussian Process and how we can use them to model and predict the dynamics of a cloth, and ends with the definition of the Mixtures of Experts and the special challenges that the Gaussian processes imposes to this idea.

2.1 Gaussian Process

Gaussian processes [7] are a powerful tool in machine learning, they allow us to make predictions about our data by incorporating prior knowledge. Statistically, a Gaussian process is a stochastic process (a collection of random variables indexed), such that every finite collection of those random variables has a multivariate normal distribution.

Firstly, we have to remember that in the multivariate Gaussian distribution is a set of random variables, where each one is distributed normally, and their joint distribution is also defined by a Gaussian with a mean vector μ and a covariance matrix Σ.

$$\mathbf{X} = \begin{bmatrix} X_1 \\ X_2 \\ \vdots \\ X_{n-1} \\ X_n \end{bmatrix} \sim \mathcal{N}(\mu, \Sigma)$$

The μ vector represents the expected value of the multivariable distribution, and the covariance matrix Σ represents in its diagonal the variance of each dimension $\sigma_{i,i}^2$ and the off-diagonal elements describes the correlation between the elements of $\mathbf{X}$ treated as random variables $\sigma_{i,j} = \sigma(X_i, X_j)$. This matrix will always be symmetric and positive semi-definite.

The Gaussian distributions are chosen due to their good properties of marginalization and conditioning. If we split the set of variables into two subsets X and Y, we can use the following notation to describe the distribution.

$$P_{X,Y} = \begin{bmatrix} X \\ Y \end{bmatrix} \sim \mathcal{N}(\mu, \Sigma) = \mathcal{N}\left(\begin{bmatrix} \mu_X \\ \mu_Y \end{bmatrix}, \begin{bmatrix} \Sigma_{XX} & \Sigma_{XY} \\ \Sigma_{YX} & \Sigma_{YY} \end{bmatrix} \right)$$

Through marginalization [8], we can extract partial information from multivariate probability distributions and determine the distributions for each subset as

$$X \sim \mathcal{N}(\mu_X, \Sigma_{XX})$$
$$Y \sim \mathcal{N}(\mu_Y, \Sigma_{YY})$$

where each partition will only depend on its correspondent entries μ and Σ.

The conditioning is used to determine the probability of one variable depending on another variable, and similarly to the marginal distribution, this operation is also closed and yields a modified Gaussian distribution.

$$X|Y \sim \mathcal{N}(\mu_X + \Sigma_{XY}\Sigma_{YY}^{-1}(Y - \mu_Y), \Sigma_{XX} - \Sigma_{XY}\Sigma_{YY}^{-1}\Sigma_{YX})$$
$$Y|X \sim \mathcal{N}(\mu_Y + \Sigma_{YX}\Sigma_{XX}^{-1}(X - \mu_X), \Sigma_{YY} - \Sigma_{YX}\Sigma_{XX}^{-1}\Sigma_{XY})$$

This latter property is the keystone of the Gaussian processes, since it will allow us to introduce and sample new variables in an extended multivariable Gaussian distribution from a given set of data points and the correspondent correlation matrices.

Note that we have not imposed any restriction on the dimensions of the variables, so as long as the dimensions of the X_i variables and μ_i are the same dimensions, and we can define the correlation function for each set of variables, the multivariable Gaussian distribution can be defined and therefore the Gaussian process.

The covariance matrix is usually obtained with a kernel function, that, given two input variables, always returns a scalar representing the correlation. Some of the most widely used kernels are:

$$\text{The white Gaussian Noise } K_{GN}(x.x') = \sigma^2\delta(x, x') \text{ without correlation}$$
$$\text{The linear kernel } K_L(x, x') = \sigma_{base}^2 + \sigma^2(x - c)^T(x' - c)$$
$$\text{The Radial Basis Function (RBF) } K_{RBF}(x, x') = \sigma^2 exp\left(\frac{\|x - x'\|^2}{2 \cdot \ell^2}\right)$$
$$\text{The Ornstein-Uhlenbeck } K_{OU}(x, x') = \sigma^2 exp\left(\frac{\|x - x'\|}{\ell}\right)$$

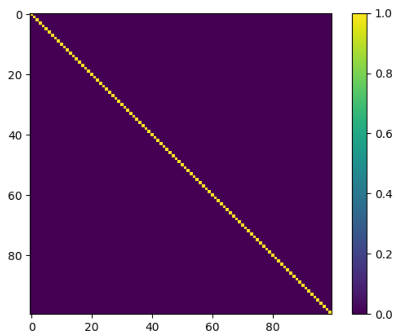
where the σ parameters determine the intensity of the kernel and therefore the average distance of your function away from its mean. It is just a scale factor. The c parameter of the linear kernel is the offset and determines the point where the kernel will reach its minimum variance. The ℓ parameter of the two last kernels represents the length scale and determines the length of the "wiggles" of your function. In general, you won't be able to extrapolate more than ℓ units away from your data. We can see a representation over a one dimensional vector of input points in the figure 1.

To use this tool to predict the behavior of a smooth continuous function, we have to consider the outputs of the function as Normal variables, and the covariance function only dependent on the input variables of the objective function. Once we have the output of some given set of points $f = f(x)$, we can predict the value of the function evaluated in other points $f_* = f(x_*)$ with some variance by the conditioned probability distribution.

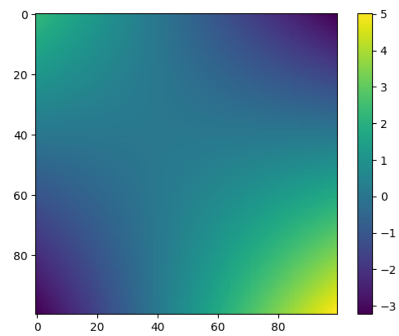
$$f_*|f \sim \mathcal{N}(\mu_{f_*} + \Sigma_{x^*,x}\Sigma_{x,x}^{-1}(f - \mu_f), \Sigma_{x^*,x^*} - \Sigma_{x^*,x}\Sigma_{x,x}^{-1}\Sigma_{x,x^*})$$

To apply this formula, we have to know the expected mean of the distribution μ_{f_*}. It is usually considered the mean of the actual data as a plausible approximation of the distribution's mean, but sometimes it is supposed to have a zero-mean normal, and the predictions will be based on the near data exclusively.

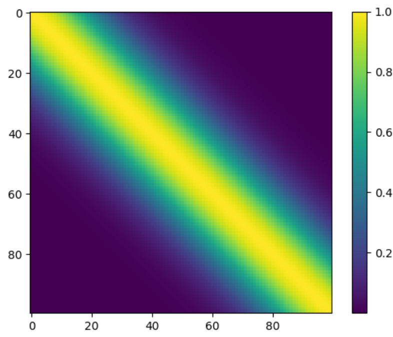
As we can see from the prediction's distribution of the figure 2, the Gaussian noise kernel predictions will not depend on the input and the smoothest predictions comes from the RBF kernel. This is one reason why it is one of the most used and normally chosen by default. The linear kernel only predicts alinear regression over the dataset, and it is usually combined with other kernels instead of employing it roughly.



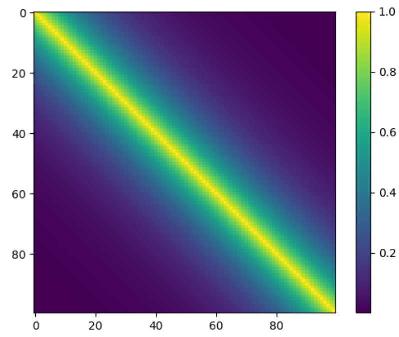
(a) Gaussian noise kernel



(b) Linear kernel

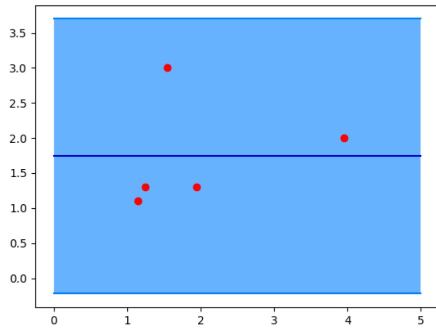


(c) RBF kernel

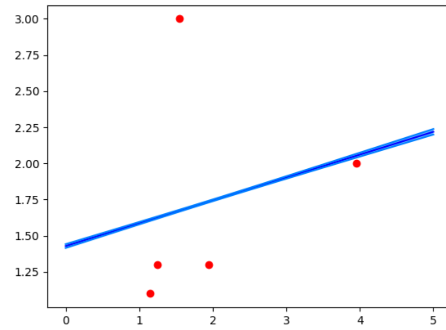


(d) Ornstein-Uhlenbeck kernel

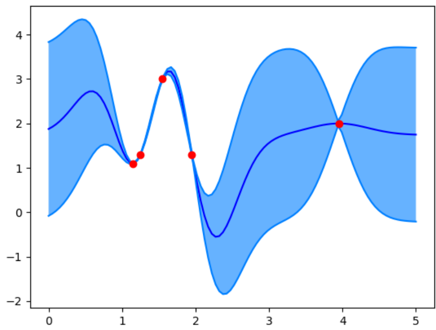
Figure 1: All the covariance matrices are obtained with the same input variables (a hundred points from 0 to 5), and the parameters are $\sigma = 1$ for the Gaussian noise kernel, $\sigma_{base} = 0.3$, $\sigma = 0.55$ and $c = 2$ for the linear kernel and $\ell = 0.8$ for the RBF and OU kernels. The images are obtained with the python *imshow* module.



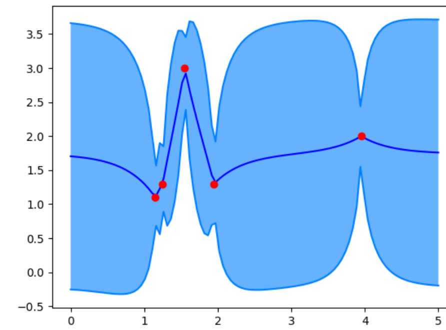
(a) Gaussian noise kernel



(b) Linear kernel



(c) RBF kernel



(d) Ornstein-Uhlenbeck kernel

Figure 2: The data points are represented with a red dots, the dark blue line represents the predicted mean for each input and the blue shaded area is the 95% confidence interval obtained from the standard deviation. The parameters are $\sigma = 1$ for the Gaussian noise kernel, $\sigma_{base} = 0.4$, $\sigma = 0.05$ and $c = 0$ for the linear kernel and $\ell = 0.4$ for the RBF and OU kernels, and the a priori mean estimator is the mean of the ouputs $\mu_{f_*} = \bar{f}$.

2.2 Gaussian Process Latent Variable Model

Machine learning is often split into three categories: supervised learning, where a data set is split into inputs and outputs; reinforcement learning, where typically a reward is associated with achieving a set goal, and unsupervised learning where the objective is to understand the structure of a data set. One approach to unsupervised learning is to represent the data, $\mathbf{Y}$, in some lower dimensional embedded space, $\mathbf{X}$, often called latent variables.

Principal component analysis (PCA) seeks a lower dimensional sub-space and can be used to capture the variance of the data from a deterministic approach by finding the projection according to which the data is best represented in terms of least squares, but has many limitations as the data may not be structured linearly and the dimensional reduction could lead to not very accurate reconstruction of the data (loss of information). To overcome those problems, we can introduce the latent variable models, where a set of latent variables $\mathbf{X} \in \mathcal{R}^{N\times d}$ are related to a set of observed variables $\mathbf{Y} \in \mathcal{R}^{N\times D}$, through a set of parameters. The model is defined probabilistically, and typically the latent variables are then marginalized so the parameters can be found by maximizing the likelihood. Otherwise, we can also marginalize over the parameters and optimize the latent variables.

The Gaussian Process Latent Variable Model [4] is a generalization of the dual version of the Pobabilistic PCA (dual PPCA). To define the probabilistic PCA, let's assume that we are given a set of centred D-dimensional data $\mathbf{Y} = [y_1 \dots y_N]^T$ and denote the d-dimensional latent variable associated to each data point x_i. The relationship between the latent variable and the data point is linear with added noise.

$$y_i = Wx_i + \eta_i$$

where the matrix $W \in \mathcal{R}^{D\times d}$ specifies the linear relationship between the latent space and the data space. The noise values, $\eta_i \in \mathcal{R}^{D\times 1}$, are taken to be an independent sample from a spherical Gaussian distribution with mean zero and covariance $\beta^-1\mathbf{Id}$

$$\eta_i \sim \mathcal{N}(\mathbf{0}, \beta^{-1}\mathbf{Id})$$

The likelihood for a each data point can then be written as

$$p(y_i|x_i, W, \beta) \sim \mathcal{N}(\mathbf{0}, WW^T + \beta^{-1}\mathbf{Id})$$

To find the maximum likelihood solution for the parameters, we have to marginalize the latent varibales by assuming an appropiate prior distribution over them, like the zero mean unit covariance Gaussian distribution

$$x_i \sim \mathcal{N}(\mathbf{0}, \mathbf{Id})$$

And then, the marginalization can be found analitically

$$P(y_i|W, \beta) = \int P(y_i|x_i, W, \beta)P(x_i)dx_i = \mathcal{N}(y_i|\mathbf{0}, WW^T + \beta^{-1}\mathbf{Id})$$

where $\mathcal{N}(y_i|\mathbf{0}, WW^T + \beta^{-1}\mathbf{Id})$ is the likelihood of y_i given that $y_i \sim \mathcal{N}(\mathbf{0}, WW^T + \beta^{-1}\mathbf{Id})$. And the likelihood of the whole set of data can be given by taking advantage of the independence of the data points.

$$P(Y|W, \beta) = \prod_{i=1}^{N} P(y_i|W, \beta)$$

Then the parameters can be optimized to maximize the likelihood and it can be shown that the analytical solution is achieved when the matrix W spans the principal sub-space of the data. So, it can be considered a probabilistic version of the PCA.

In the other hand, the dual approach to this PPCA is to optimize the latent variables while we marginalize the parameters. To do so, we have to define a prior distribution over the rows of the W parameters. The simplest choice is the spherical Gaussian distribution with unit covariance and zero mean.

$$P(W) = \prod_{i=1}^{D} \mathcal{N}(w_i|\mathbf{0}, \mathbf{Id})$$

where $\mathcal{N}(w_i|\mathbf{0}, \mathbf{Id})$ is the likelihood of w_i given that $w_i \sim \mathcal{N}(\mathbf{0}, \mathbf{Id})$ and w_i are the rows of the W matrix and are considered independent variables. Now we have to marginalize the parameters from the data likelihood

$$P(Y|X, \beta) = \int P(Y|W, X, \beta)P(W)dW$$

With the conjugate prior of W that we have chosen, the marginalization is straightforward and the resulting likelihood takes form

$$P(Y|X, \beta) = \prod_{i=1}^{D} P(Y_{:,i}|X, \beta)$$

where

$$P(Y_{:,i}|X, \beta) = \mathcal{N}(Y_{:,i}|\mathbf{0}, XX^T + \beta^{-1}\mathbf{Id})$$

Then we have to optimize the likelihood with respect to the latent variables X. To simplify the terms, we can optimize the log-likelihood instead thanks to the monotonicity of the logarithm. The resulting objective function will be

$$\mathcal{L} = -\frac{DN}{2}\log 2\pi - \frac{D}{2}\log|K| - \frac{1}{2}tr(K^{-1}YY^T)$$

where

$$K = XX^T + \beta^- 1\mathbf{Id}$$

This latter version is known as the dual probabilistic PCA (dual PPCA).

Now we want to introduce the Gaussian processes. If we condiser a simple Gaussian process prior over the space of functions that are fundamentally linear, but are corrupted by Gaussian noise of variance $\beta^{-1}\mathbf{Id}$. The covariance kernel for such a prior is given by a linear plus Gaussian kernel.

$$k(x, x') = x^T x + \beta^{-1}\delta(x, x')$$

If we apply this kernel over the latent variables, we recover the covariance matrix K that can be recognized as the covariance associated with each factor of the marginal likelihood for dual PPCA. The marginal likelihood for dual PPCA is therefore a product of D independent Gaussian processes (each one for each dimension of the data points).

This interpretation gives us a possibility to generalize the dual PPCA latent variable model to a range of Gaussian Process Latent Variable Models (GPLVM) by breaking the assumptions of linearity, independency and identically distributed. For example, we can break the identically distributed assumption by introducing a different kernel function for each dimension and scaling them, or we can break the linear assumption by using non-linear kernels to define the K matrix like the widely known RBF kernel.

Although originally proposed for dimension reduction, GPLVM has been extended and widely used in many machine learning scenarios for classification and clustering tasks outperforming other Latent Variables Models thanks to the advantages it presents [16]: the non-linear learning characteristic of GPs, and the non-parametric property, as most of the existing LVMs are parametric models in which there is a strong assumption on the projection function or data distribution. However, those improved classification and clustering capabilities are not enough to adjust temporal series of observations, and we will need to introduce new features in order to obtain a dynamical model.

2.3 Gaussian Process Dynamical Model

The Gaussian Process Dynamical Model was firstly introduced by J. M. Wang, A. Hertzmann, and D. J. Fleet in 2005 [3] to model a time-series observations that can capture the non-linearities of the data without overfitting. They took a Bayesian approach to modeling dynamics, averaging over dynamics parameters rather than estimating them leading the group to the Gaussian processes, The high dimensionality of the observable parameters space drove them to the need of a low dimensional representation of the observables, since Gaussian processes don't perform well with high dimensional inputs.

The work was inspired by the unsupervised learning of the GPLVM but without assuming the independency of the observed data by introducing a dependency to subsequent observations. The Gaussian Process Dynamical Model (GPDM) involves two function estimations, the mapping from a latent space to the data space, and a dynamical model in the latent space (mapping from the previous time steps to the following ones). Those functions are typically non-linear and to optimize the likelihood, they proceeded like the GPLVM, they marginalized the parameters of the functions and optimized over the latent variables.

Let's denote the data matrix as before $Y = [y_1, \dots, y_i, \dots, y_N]^T \in \mathcal{R}^{N \times D}$ and the latent variables matrix $X = [x_1, \dots, x_i, \dots, x_N]^T \in \mathcal{R}^{N \times d}$ where D is the dimension of the observable data and d is the dimensions of the latent space. Then we can define the two mapping functions with a discrete-time Markovian dynamics:

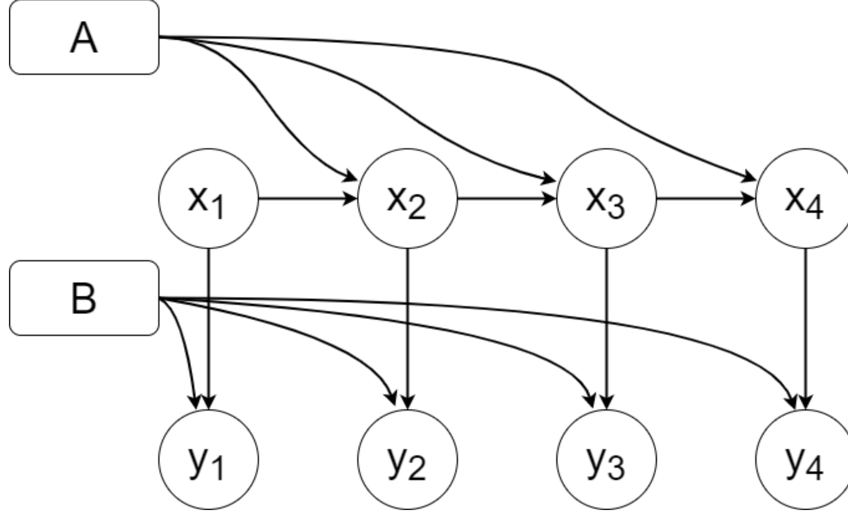


Figure 3: Graphical representation of the dynamical model. There is a time dependency in the latent space parametrized by A and B

$$x_i = f(x_{i-1}; A) + \eta_{x,i}$$
$$y_i = g(x_i; B) + \eta_{y,i}$$

where $\eta_{x,i}$ and $\eta_{y,i}$ are zero-mean Gaussian noise processes, f and g are non-linear mappings parameterized by A and B, respectively. We can see a graphical representation of the time series in the figure 3.

In [3], the authors considered f and g to be a linear combination of basis of non-linear functions

$$f(x_i; A) = \sum_j a_j \phi_j(x_i)$$
$$g(x_i; B) = \sum_j b_j \psi_j(x_i)$$

for weights $A = [a_1, a_2, ...]$, $B = [b_1, b_2, ...]$ and the basis functions ϕ_j and ψ_k. In order to fit the parameters of this model to training data, one must select an appropriate number of basis functions, and ensure that there is enough data to constrain the shape of each basis function. Ensuring both of these conditions can be very difficult in practice.

However, from a Bayesian perspective, we do not know the specific form of those functions and, instead, we have to marginalize out the parameters (considering all the possible combination of them).

For the mapping to the observation function g, the authors in [3] assumed an isotropic Gaussian prior for the columns of B (the correlation matrix is assumed to be the identity matrix), and the marginalization of the function can be done in closed form [11], [10] yielding

$$P(Y|X, \bar{\beta}) = \frac{|W|^N}{\sqrt{(2\pi)^{ND}|K_Y|^D}} exp\left(-\frac{1}{2}tr(K_Y^{-1}YW^2Y^T)\right)$$

where K_Y is the kernel matrix whose elements are defined by a kernel function $(K_Y)_{i,j} = k_y(x_i, x_j)$. In this case, the chosen kernel is the RBF with a Gaussian noise component

$$k_Y(x_i, x_j) = \beta_1 + exp\left(-\frac{\beta_2}{2}\|x_i - x_j\|^2\right) + \beta_3^{-1}\delta_{x_i,x_j}$$

and $\bar{\beta}$ is constituted by all the parameters $\bar{\beta} = \beta_1, \beta_2, ..., W$ including the kernel parameters and the scaling matrix $W = diag(\omega_1, \dots, \omega_D)$ to account for different variances in the different data dimensions. This is equivalent to a GP with kernel function $k_Y(x_i, x_j)/\omega_m^2$ for dimension m.

The dynamic mapping on the latent coordinates X is conceptually similar, but proceeding more carefully. The idea is to marginalize the weights A of the dynamical function g as before.

$$P(X|\bar{\alpha}) = \int P(X, A|\bar{\alpha})dA = \int P(X|\bar{\alpha}, A)P(A|\bar{\alpha})dA$$

But in this case, they had to incorporate the Markov chain dependency that defined the problem in the computation of $P(X|\bar{\alpha}, A)$ at each time step

$$P(X|\bar{\alpha}) = P(x_1)\int\prod_{i=2}^{N} P(x_i|x_{i-1}, \bar{\alpha}, A)P(A|\bar{\alpha})dA$$

Creating two data matrices from the latent variables' matrix, the input one $X_{in} = [x_1, \dots, x_{N-1}]^T$ and the output $X_{out} = [x_2, \dots, x_N]^T$. They finally assumed an isotropic prior to the colums of the A matrix to obtain a closed form of the likelihood as before

$$P(X|\bar{\alpha}) = P(x_1)\frac{1}{\sqrt{(2\pi)^{(N-1)d}|K_X|^d}}exp\left(-\frac{1}{2}tr(K_X^{-1}X_{out}X_{out}^T)\right)$$

where x_i is assumed to have an isotropic Gaussian prior, and the K_X is the $(N-1)\times(N-1)$ kernel matrix is formed with the X_{in} variables and the kernel function. This time, the chosen kernel function is a RBF+linear kernel with some Gaussian noise

$$k_X(x_i, x_j) = \alpha_1 + exp\left(-\frac{\alpha_2}{2}\|x_i - x_j\|^2\right) + \alpha_3 x_i^T x_j + \alpha_4^{-1}\delta_{x_i,x_j}$$

The kernel corresponds to representing g as the sum of a linear term and RBF terms. The inclusion of the linear term is motivated by the fact that linear dynamical models, such as first or second-order autoregressive models, are useful for many systems [3]. And the $\bar{\alpha}$ is constituted only by the kernel functions without any scaling matrix, unlike the mapping one.

It should be noted that, due to the nonlinear dynamical mapping, the joint distribution of the latent coordinates is not Gaussian.

Finally, they set some priors to the $\bar{\alpha}$ and $\bar{\beta}$ hyperparameters to discourage overfitting

$$P(\bar{\alpha}) \propto \prod_i \alpha_i^{-1}$$

$$P(\bar{\beta}) \propto \prod_i \beta_i^{-}1$$

Altogether, the priors, the latent mapping, and the dynamics define a generative model for time-series observations

$$P(X, Y, \bar{\alpha}, \bar{\beta}) = P(Y|X, \bar{\beta})P(X|\bar{\alpha})P(\bar{\alpha})P(\bar{\beta})$$

And learning the GPDM from the observed data entails minimizing the negative log-posterior

$$\mathcal{L} = -\ln P(X, \bar{\alpha}, \bar{\beta}|Y) = -\ln P(X, Y, \bar{\alpha}, \bar{\beta})$$

The last equivalence comes from the fact that the "probability" of the observed data is one, and it is necessary to define correctly the log-likelihood that we want to minimize over the X, $\bar{\beta}$ and $\bar{\alpha}$ parameters numerically.

$$\begin{aligned}\underset{X,\alpha,\beta}{\operatorname{argmin}}\, \mathcal{L} = \underset{X,\alpha,\beta}{\operatorname{argmin}}\, & \frac{d}{2}\ln|K_X| + \frac{1}{2}tr(K^{-1}X_{out}X_{out}^T) + \sum_j \ln \alpha_j \\ & - N\ln|W| + \frac{D}{2}\ln|K_Y| + \frac{1}{2}tr(K_Y^{-1}YW^2Y^T) + \sum_j \ln \beta_j\end{aligned}$$

2.4 Controlled Gaussian Process Dynamical Models

The Controlled GPDM was introduced by Fabio Amadio, Juan Antonio Delgado-Guerrero, Adrià Colomé, and Carme Torras in 2020 [1] in an attempt to capture the high dimensionality and the uncertainty of the dynamics of a cloth during manipulation.

The approach is very similar to the above one, but slightly modifying the first-order Markov dynamics by introducing the control input to the dynamical function

$$\begin{aligned} x_i - x_{i-1} &= f(x_{i-1}, u_{i-1}) + \eta_{x,i} \\ y_i &= g(x_i) + \eta_{y,i} \end{aligned}$$

In this case, the latent dynamical function depends in both the previous latent position x_{i-1} and the previous control instructions u_{i-1}. The output of the latent dynamical function is also modified to the taken latent step $x_i - x_{i-1}$ instead of the next latent position x_i. This was made to improve smoothness of the latent trajectories as suggested in [12].

As before, the observable data will be named as $Y = [y_1, \dots, y_N]^T \in \mathcal{R}^{N\times D}$, where D is the number of dimensions of the observed data (typically a large number). Analogously, the corresponded latent variables will be named as $X = [x_1, \dots, x_N]^T \in \mathcal{R}^{N\times d}$, where d stands for the latent space dimensions. They made the assumption that each one of the j-th columns of the observed matrix (the j-th dimension of all the observed data) has a normal prior with zero mean and $K_y^{(j)}(X)$ covariance matrix

$$Y_{:,j} \sim \mathcal{N}(\mathbf{0}, K_y^{(j)}(X))$$

where the $K_y^{(j)}(X)$ covariance matrix is defined with a kernel $k_y^{(j)}(.,.)$ function over all the latent variables.

$$P(Y_{:,j}|X) = \frac{exp\left(-\frac{1}{2}Y_{:,j}^T(K_y^{(j)}(X))^{-1}Y_{:,j}\right)}{\sqrt{(2\pi)^N|K_y^{(j)}(X)|}}$$

Assuming the independency of the D components of the observations, and that all dimensions have the same kernel function but scales by a factor $k_y^{(j)}(.,.) = \omega_{y,j}^{-2}k_y(.,.)$, they arrive to the same expression for the mapping likelihood as the GPDM

$$P(Y|X) = \frac{|W_y|^N}{\sqrt{(2\pi)^{ND}|K_y(X)|^D}}exp\left(-\frac{1}{2}tr(K_y^{-1}(X)YW_y^2Y^T)\right)$$

where $W_y = diag(\omega_{y,1}, \dots, \omega_{y.D})$, and they used the same RBF with Gaussian noise kernel function for the mapping covariance.

For the latent dynamical model, they proceeded similarly. As the output of the Markov function is the difference between consecutive steps, they defined the Δ variables as $\Delta_{i,:} = x_{i+1,:} - x_{i,:}$ so the whole matrix can be expressed as $\Delta = [X_{2:N,:} - X_{1:N-1,:}]$.

Compactifying the notation of the input of the dynamical function f, they defined $x_t = [x_t^T, u_t^T]^T \in \mathcal{R}^{d+E}$, so the input matrix will be defined as $\tilde{X} = [\tilde{x}_i, \dots, \tilde{x}_{N-1}]^T$.

Now they assumed again that the output prior is a zero mean and $K_x^{(j)}(\tilde{X})$ covariance for each column

$$\Delta_{:,j} \sim \mathcal{N}(\mathbf{0}, K_x^{(j)}(\tilde{X}))$$

Then, they assumed independency between the dimensions and the same kernel function but weighted one more time $k_x^{(j)}(.,.) = \omega_{x,j}^{-2}k_x(.,.)$. to come up with the likelihood of the latent variables (this one is different with respect to the GPDM)

$$P(\Delta|\tilde{X}) = \frac{|W_x|^{N-1}}{\sqrt{(2\pi)^{(N-1)d}|K_x(\tilde{X})|^d}}exp\left(-\frac{1}{2}tr(K_x^{-1}(\tilde{X})\Delta W_x^2\Delta^T)\right)$$

where $W_x = diag(\omega_{x,1}, \dots, \omega_{x,d})$ and the kernel function is the "linear+RBF" kernel with Gaussian noise.

Finally, the likelihood $P(X|Y, U) = P(Y|X)P(\Delta|\tilde{X})$ is maximized without assuming any prior to the parameters by minimizing the negative log-likelihood

$$\mathcal{L} = \frac{D}{2}\ln|K_y(X)| + \frac{1}{2}tr(K_y^{-1}(X)YW_y^2Y^T) - N\ln|W_y| + \\ + \frac{d}{2}\ln|K_x(\tilde{X})| + \frac{1}{2}tr(K_x^{-1}(\tilde{X})\Delta W_x^2\Delta^T) - (N-1)\ln|W_x|$$

2.5 Mixture of Experts

The mixture of Experts is stablished on the divide and conquer principle in which the problem space is divided between a few experts and supervised by a gating network [9]. There are several strategies to divide the problem space between the experts, but they can be classified into two groups according

to the partitioning of the problem. The first one uses a special error function to split the problem stochastically and then the experts specialize in each sub-space. And in the second one, the space is explicitly partitioned before the training and then the experts are assigned.

In the first group, we can define two types of mixtures, the competitive, where all the experts are trained to make good predictions and the gating assigns the correspondent weights to depending on the performance of each one. The associated loss is depicted as

$$\mathcal{L} = \sum_j g_j \|y - O_j\|^2$$

where y is the target output, O_j is the output of each expert and g_j is the gaiting weight assigned to this expert. In this structure , each expert is updated based on their own error so each expert will yield a complete output. But this has a downside, the error function does not ensure the localization of the experts.

And the cooperative structure that will have associated the following error function

$$\mathcal{L} = \|y - \sum_j g_j O_j\|^2$$

The weights of each expert are updated based on the overall ensemble error rather than the errors of each expert. This strong coupling in the process of updating the weights of the experts tends to employ almost all of the experts for each data sample.

As a mixture of Gaussian Process is intrinsically partitioned (we have to assign data points to the experts) and the splitting of the problem space depends on the composition of the experts, we can classify any mixture of Gaussian experts as an implicit mixture. And the fact that each expert will have to output a complete prediction and that the Gaussian Processes can only be reliable to predict the output near the data points that constitutes it means that they should have a competitive gating structure.

Knowing this will help us to determine how we should train the gating model later on. And it is only left to know what stochastics that we are we going to use to split the dataset.

2.6 Infinite Mixture of Gaussian Experts

In traditional Mixture of Experts 2.5, the probability distribution of each prediction will depend on the experts itself and the gating model. We can define the likelihood of some input-output data point as the addition of the likelihood of all the experts weighted by the gating model.

$$P(Y_{i,:}|X_{i,:}) = \sum_{j \in E} P(Y_{i,:}|z_i = j, X_{i,:}, \theta_j) P(z_i = j|X, \phi)$$

where $X_{i,:}$ and $Y_{i,:}$ are the i-th input-output data points, z_i is the discrete indicator variable that assings the i-th data point to one expert, E is the set of all the experts, ϕ are the parameters of the gating model, and the θ_j are the parameters of the j-th expert. The summation is made over all the experts, and $P(z_i = j|X, \phi)$ represents the likelihood of the $X_{i,:}$ input to belong to the j-th expert, what will not only depend on the gating parameters, but also on all the input dataset. Assuming

independency, the likelihood of the whole dataset will be the multiplication of the likelihood of all the dataset points.

As stated in [2], the Independently and Identically Distributed (IID) assumption of the indicators is contrary to GP models which solely model the dependencies in the joint distribution. We should then define it for every possible assignment of data points to experts. Let's denote the configuration of the experts as $\mathbf{z} = (z_1, z_2, ...)$. Therefore, the likelihood of all the dataset is a sum over (exponentially many) configurations:

$$\begin{aligned} P(Y|X,\theta) &= \sum_{\mathbf{z}} P(Y|X,\mathbf{z},\theta)P(\mathbf{z}|X,\phi) \\ &= \sum_{\mathbf{z}} \left[\prod_{j\in E} P(Y_{\{i|z_i=j\},:}|X_{\{i|z_i=j\},:},\theta_j) \right] P(\mathbf{z}|X,\phi) \end{aligned} \tag{1}$$

where $Y_{\{i|z_i=j\},:}$ and $X_{\{i|z_i=j\},:}$ are the set of data points assigned to the j-th expert. This time, the gating model has to set the likelihood of the whole configuration $\mathbf{z}$. Whereas the original ME formulation used expectations of assignment variables called responsibilities, this is inadequate for inference in the mixture of GP experts. Consequently, we directly represent the indicators, z_i, and Gibbs sample for them to capture their dependencie.

In statistics, Gibbs sampling [5][17] is a Markov Chain Monte Carlo (MCMC) algorithm for obtaining a sequence of observations which are approximated from a specified multivariate probability distribution, when direct sampling is difficult. As other MCMC algorithms, Gibbs sampling generates a Markov chain of samples, each of which is correlated with nearby samples. As a result, care must be taken if independent samples are desired. Generally, samples from the beginning of the chain (the burn-in period) may not accurately represent the desired distribution and are usually discarded.

This method is especially useful when we know the conditional distribution of one variable with respect to the others since the algorithm consist in iteratively sampling and overwriting all the dimensions of the variable one by one from the previous value. We can see the algorithm in 1.

Algorithm 1 General Gibbs sampling

```
1: procedure GIBBS X^{i+1} SAMPLING FROM X^i
2:     for j ∈ 1 ... N do
3:         x_j^{i+1} ← P(x_j | x_1^{i+1}, ..., x_{j-1}^{i+1}, x_{j+1}^i, ..., x_N^i)
4:     end for
5: end procedure
```

where X^i stands for the i-th sample of the Gibbs algorithm and x_j^i stands for each one of its dimensions. We can loop this sampling as many times as we want, but consecutive samples will be heavily correlated.

Coming back to the problem, we have to define the posterior conditional distribution for each indicator given all the remaining indicators and the data that we needed for the Gibbs sampling:

$$P(z_i = j|\mathbf{z}_{(..,\hat{i},..)}, X, Y, \theta, \phi) \propto P(Y|X, z_i = j, \mathbf{z}_{(..,\hat{i},..)}, \theta)P(z_i = j|\mathbf{z}_{(..,\hat{i},..)}, X, \phi) \tag{2}$$

At this moment, we can compute the probability of the i-th indicator given the input data with the

gating model, and the both likelihood of the output dataset given the input and the new configuration can be computed as in **??**.

Once we have defined this framework, we have to decide what kind of gating model we want to use. In [2], a gating model based in the Dirichlet Process is proposed, which can be defined as the limit of a Dirichlet distribution when the number of classes tends to infinity. They started from a symmetric Dirichlet distribution on proportions:

$$P(\pi_1, ..., \pi_k|\alpha) \sim Dirichlet(\alpha/k) = \frac{\Gamma(\alpha)}{\Gamma(\alpha/k)^k} \prod_j \pi_j^{\alpha/k-1}$$

where α is the (positive) concentration parameter. It can be shown [6] that the conditional probability of a single indicator when integrating over the π_j variables and letting k tend to infinity is given by:

$$\begin{aligned} P(z_i = j|z_{(..,\hat{i},..)}, \alpha) &= \frac{n_{(..,\hat{i},..),j}}{n-1+\alpha}, \forall \text{ components wiht } n_{(..,\hat{i},..),j} > 0 \\ P(z_i \neq z_{i'} \forall i' \neq i|z_{(..,\hat{i},..)}, \alpha) &= \frac{\alpha}{n-1+\alpha}, \text{ for all other (non assigned) components} \end{aligned} \tag{3}$$

where $n_{(..,\hat{i},..),j} = \sum_{i' \neq i} \delta(z_i, j)$ is called the occupation number and represents the number of elements in some particular component excluding the i-th observation, and n is the total number of data points. We have now determined some probability of assigning the data point to each of the existent components (proportional to this occupation factor), and some other probability to create new components if it was necessary. Therefore, we can now introduce the dependency of this distribution on the input by adapting the occupation number employing a local estimator, like a kernel classifier [2]:

$$n_{(..,\hat{i},..),j} = (n-1)\frac{\sum_{i' \neq i} K_\phi(X_{i,:}, X_{i',:})\delta(z_i, j)}{\sum_{i' \neq i} K_\phi(X_{i,:}, X_{i',:})}$$

Where the $\delta(.,.)$ is the Dirac's Delta function to sum up only the points correspondent to each cluster. The kernel function K_ϕ will be parametrized by ϕ and can be defined separately later.

3. Mixture of Controlled Gaussian Process Dynamical Model

3.1 Background Theory

As we have done earlier in section 2.4, we represent the deformations of the material with a high dimensional mesh. This discretization of the object will turn the problem into a prediction of a finite number of features (3 positional dimensions times the number of points of the mesh). The greater the number of points we are using to represent the deformable solid, the better capacity of a precise representation of the object, but it will also come with a greater computational cost in the dynamical prediction and will be unmanageable for building a tractable state-action space policy.

Therefore, it will be mandatory to employ Dimensional Reduction (DR) methods such as the Gaussian Process Latent Variable Model [4] and the dynamical extensions depicted in [3] and [1], where the observables of high dimensionality (for example 3×64 dimensions for a 8 by 8 mesh of a cloth) are transformed to a low dimensional representation latent space, where we can perform the dynamics of the model and define a tractable task space for the control problem.

Here in the Mixture of Controlled Gaussian Process Dynamical Models, we incorporate some concepts of the Infinite Mixture of Gaussian Experts defined above 2.6 to get rid of two important limitations of the GPs. Firstly, the inference requires the inversion and multiplication of $n \times n$ matrix, where the n is the size of the whole dataset, what results in a cubical computational complexity, and therefore, makes it impractical to use it for real time applications with medium large datasets. Secondly, the covariance functions are commonly assumed to be stationary (same parameters) along all the dataset, leading to some lack of flexibility in the model. The use of several experts will allow us to both: define different covariance functions, and train unassociated parameters for each expert, resulting in a better adaptation to the disparate circumstances of separate data points sets and distinct regions of the input space.

As we have seen in the CGPDM section 2.4, we want to maximize the likelihood of the latent variables X and model parameters, α for the latent dynamics mapping and β for the latent space projection given the data:

$$\underset{X,\alpha,\beta}{\operatorname{argmax}}\, P(X,\alpha,\beta|Y,U) = \underset{X,\alpha,\beta}{\operatorname{argmax}}\, P(Y,X,\alpha,\beta,U) = \underset{X,\alpha,\beta}{\operatorname{argmax}}\, P(Y|X,\beta)P(X|\alpha,U)$$

Where $P(Y|X,\beta)$ and $P(X|\alpha,U)$ can be considered two separated Gaussian Process Models that we can convert into two Infinite Mixture of Gaussian Experts. For an easier identification from now on, we are going to name the first term as the **Mapping Model**, because it is related to the capacity of the model to infer the observations from the latent points, and the second term will be named the **Latent Dynamics Model**, because it is related to the capacity of the model to predict the next point in the latent space given the previous positions and the current control. As it can be useful later on, we are going to name D the total number of dimensions of the observable space and d the number of dimensions of the latent space.

The Mapping Model likelihood can be written as a Infinite Mixture of GPs:

$$P(Y|X,\beta) = \sum_{\mathbf{z}^{map}} \left[\prod_{j \in E} P(Y_{\{i|z_i^{map}=j\},:}|X_{\{i|z_i^{map}=j\},:}, \theta_j^{map}) \right] P(\mathbf{z}^{map}|X, \phi^{map}) \tag{4}$$

Following the notation as in eq (1), the $\mathbf{z}^{map}$ is the vector variable that indicates the configuration of the experts, $Y_{\{i|z_i^{map}=j\},:}$ and $X_{\{i|z_i^{map}=j\},:}$ are the set of data points assigned to the j-th expert. Here we have also splitted all the Mapping Model parameters β into the different experts and gating parameters, θ_j^{map} and ϕ^{map} respectively.

For the Latent Dynamics Model, we have to preprocess the data before interpreting $P(X|\alpha, U)$ as a Infinite Mixture of GPs. If we consider the Markov chain dynamics in the latent space only dependent on the two previous latent positions of the same trajectory and the current control action, we can establish this equality:

$$P(X|\alpha, U) = P(X_{1,:}, X_{2,:}) \int \prod_{i=3}^{N} P(X_{i,:}|X_{i-1,:}, X_{i-2,:}, U_{i,:}, \alpha, A) P(A|\alpha) dA$$

Where $X_{i,:}$ is the i-th data point of the latent variables, $P(X_{1,:}, X_{2,:})$ is the probability of the first two latent variables, that we can consider invariant over model parameters and latent variables alterations. Therefore, this term will be ignored from now on in the optimization problem. And finally, A are the weights of a linear combination of basis functions of the dynamical predictor depicted in the GPFM 2.3 [3].

This distribution will be identical if we change the random variable $X_{i,:}$ to $X_{i,:} - X_{i-1,:}$ when the distribution is conditioned to the second term of the substraction.

$$P(X|\alpha, U) = P(X_{1,:}, X_{2,:}) \int \prod_{i=3}^{N} P(X_{i,:} - X_{i-1,:}|X_{i-1,:}, X_{i-2,:}, U_{i,:}, \alpha, A) P(A|\alpha) dA$$

After that, we can rename some of the variables to employ the same notation as the before cited paper, $\Delta_{i,:} := X_{i,:} - X_{i-1,:}$ and $\hat{X_{i,:}} := (X_{i-1,:}, X_{i-2,:}, U_{i,:})$.

$$P(X|\alpha, U) = P(X_{1,:}, X_{2,:}) \int \prod_{i=3}^{N} P(\Delta_{i,:}|\hat{X_{i,:}}, \alpha, A) P(A|\alpha) dA$$

Assuming an isotropic Gaussian prior on the columns of A and using the matrix form of this variables ($\Delta := X_{(3..N),:} - X_{(2..N-1),:}$ and $\hat{X} := (X_{(2..N-1),:}, X_{(1..N-2)}, U_{(3..N),:})$), it can be shown [3] that this expression simplifies to:

$$P(X|\alpha, U) = P(X_{1,:}, X_{2,:}) \frac{1}{\sqrt{(2\pi)^{(N-2)d} |K_{\hat{X}}|^d}} exp\left(-\frac{1}{2} tr(K_{\hat{X}}^{-1} \Delta \Delta^T) \right) =: P(\Delta|\hat{X}, \alpha)$$

Where the kernel ($K_{\hat{X}}$) has yet to be definded, N is the number of data points of this sequence and d is the number of dimensions of the latent space (also the number of colums of the X matrix).

This expression can be extended to as many sequences as we want if each one has at least 3 data points for a correct definition of the Δ and $\hat{X}$ matrices for each one and we can joint them all at the end.

Then, we can introduce the Infinite Mixture of Gaussian Experts to this second term of the likelihood equation.

$$P(\Delta|\hat{X},\alpha) = \sum_{\mathbf{z}^{lat}} \left[\prod_{j \in E} P(\Delta_{\{i|z_i^{lat}=j\},:}|\hat{X}_{\{i|z_i^{lat}=j\},:}, \theta_j^{lat}) \right] P(\mathbf{z}^{lat}|\hat{X}, \phi^{lat}) \tag{5}$$

Following the same notation, the $\mathbf{z}^{lat}$ vector is the configuration variable, $\Delta_{\{i|z_i^{lat}=j\},:}$ and $\hat{X}_{\{i|z_i^{lat}=j\},:}$ are the points assigned to the j-th expert, and we have splitted the Latent Dynamical Model's parameters β into the experts and gating parameters, θ_j^{lat} and ϕ_j^{lat} respectively.

Finally, we have al the terms of the optimization problem properly developed, and it will end up looking like

$$\begin{aligned} &\underset{X,\alpha,\beta}{\text{argmax}}\, P(Y|X,\beta)P(X|\alpha,U) = \\ &\underset{X,\theta_j^{map},\phi^{map},\theta_j^{lat},\phi^{lat}}{\text{argmax}} \left(\sum_{\mathbf{z}^{map}} \left[\prod_{j \in E} P(Y_{\{i|z_i^{map}=j\},:}|X_{\{i|z_i^{map}=j\},:}, \theta_j^{map}) \right] P(\mathbf{z}^{map}|X, \phi^{map}) \right) \\ &\qquad \left(\sum_{\mathbf{z}^{lat}} \left[\prod_{j \in E} P(\Delta_{\{i|z_i^{lat}=j\},:}|\hat{X}_{\{i|z_i^{lat}=j\},:}, \theta_j^{lat}) \right] P(\mathbf{z}^{lat}|\hat{X}, \phi^{lat}) \right) \end{aligned} \tag{6}$$

What includes many not differentiable variables (the inidicators $\mathbf{z}^{map}$ and $\mathbf{z}^{lat}$) and the summation of exponentially many terms, so the optimization cannot be done straightforwardly.

3.2 Designing choices

As in any project, we have had to make many designing choices to obtain better performance in the problem that we want to solve inside the theoretical framework that we have developed earlier. In the following sections, we aim to explain some of the decisions that we have made and some few reasons of why those are the best solutions we have found. But some other hyperparameters had to be chosen empirically and will be chosen in the Experimets section along with some experimental results that we explored.

3.2.1 Experts structure and kernels

One of the first and more important choices that we made are the different kernels that we are going to employ for the experts and the gating models. These are going to determine how the data will relate to the nearby input points giving that notion of locality that the Gaussian Process need for their interpolation.

For the Mapping Experts, we have chosen the kernel proposed in the CGPDM [1] because they have demonstrated a better performance compared to other standard kernels in the single expert configuration. It is based in the widely used Radial Basis Function (RBF) kernel, but with a more

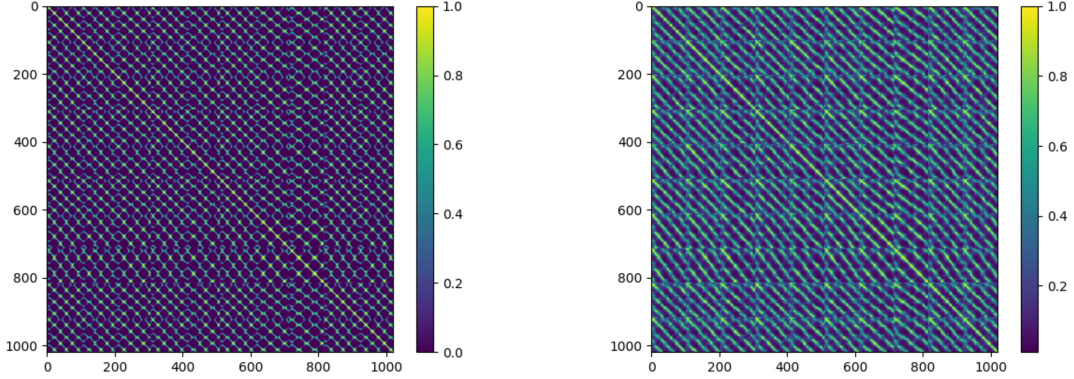


Figure 4: On the left side, we can see the correlation matrix of one expert with traditional RBF kernel and on the right side we can see the correlation matrix of a modified and trained RBF kernel. Images obtained with the *imshow* module inside the *matplotlib* package of python.

flexible structure. The authors splitted the single length-scale parameter into one independet length-scale for each dimension of the input points and integrated them into a weighted computation of the distance (instead of using the regular norm of a vector)

$$k_y(X_{i,:}, X_{j,:}) = exp\left(-\|X_{i,:} - X_{j,:}\|_{\Lambda^{map}}\right) + (\sigma^{map})^2\delta(X_{i,:}, X_{j,:})$$

Where $\|X_{i,:} - X_{j,:}\|_{\Lambda^{map}}$ will be calculated as $\sqrt{(\lambda_1^{map})^2(X_{i,1} - X_{j,1})^2 + ... + (\lambda_d^{map})^2(X_{i,d} - X_{j,d})^2}$, and $\delta(.,.)$ is the Kronecker delta function.

As we can see in the Figure 4, the modified kernel allows a much better connection between the similar points than the standard kernel outside the diagonal. This will translate in more correlation between points of the same and other trajectories and, therefore, having more points as reference to make the predictions.

In addition, for each one of the dimensions of the observable space, we are going to use the same kernel, but scaled by a factor (ω) in the calculation of the likelihood of the data in this expert. This will affect the relevance of each observable dimension in the final likelihood formula:

$$P(Y_{\{i|z_i^{map}=j\},:}|X_{\{i|z_i^{map}=j\},:}, \theta_j^{map}) = \frac{|W_{y,j}|^N exp(-\frac{1}{2}tr((K_y(X_*))^{-1}Y_* W_{y,j}^2 Y_*^T))}{\sqrt{(2\pi)^{(ND)}|K_y(X_*)|^D}} \tag{7}$$

Where θ_j^{map} represents the whole set of parameters $(\omega_{j,1}^{map}, \ldots, \omega_{j,D}^{map}, \lambda_{j,1}^{map}, \ldots, \lambda_{j,d}^{map}, \sigma_j^{map})$, Y_* and X_* are the points assigned to this expert (we have substituted the $(\{i|z_i^{map} = j\}, :)$ subscript with $*$ in the left-hand side of the equation for esthetical purposes), N is the size of this subset of points, $|W_{y,j}|$ is the $diag(\omega_{j,1}^{map}, ..., \omega_{j,D}^{map})$ matrix, and $K_y(X)$ is the covariance matrix built with the kernel function that we have just described.

For the Latent Dynamics Experts, we are going to perform similarly, we have also chosen the kernel proposed in [1] due to its performance characteristics in a single expert configuration. This time it is based in the modified RBF kernel plus a scaled linear term, but adding more parameters,

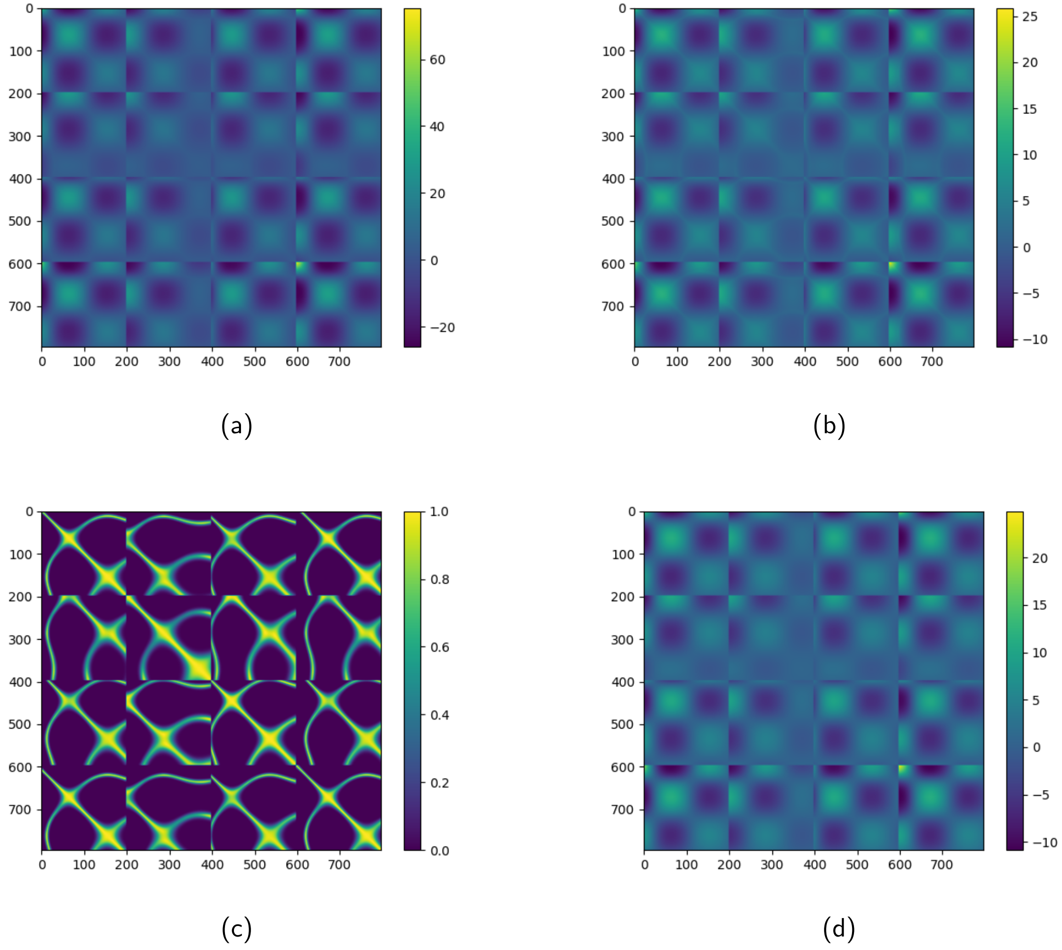


Figure 5: In 5a we have the standard RBF+linear kernel over the points of one expert, in 5b we have the modified version, and 5c and 5d are the RBF and the linear components of the latent kernel respectively.

in the calculus of the norm as in the mapping experts (length-scales) and in the linear components:

$$k_x(\hat{X}_{i,:}, \hat{X}_{j,:}) = exp\left(-\|\hat{X}_{i,:} - \hat{X}_{j,:}\|_{\Lambda^{lat}}\right) + [\hat{X}_{i,:}^T, 1]\Psi^{lat}[\hat{X}_{j,:}^T, 1]^T + (\sigma^{lat})^2\delta(\hat{X}_{i,:}, \hat{X}_{j,:})$$

Where $\|\hat{X}_{i,:} - \hat{X}_{j,:}\|_{\Lambda^{lat}}$ is computed exactly as before, but with $2d + u_dim$ dimensions instead of d, u_dim is the number of dimensions of the control variables, and Ψ^{lat} is a diagonal matrix $diag(\psi_1^2, \dots, \psi_{2d+u_dim+1}^2)$ that can be multiplied and act like a scalar product matrix.

As we can see in the Figure 5, there is not a significant and qualitative change in the shape of the kernel after the training of the different components, but there is a very noticeable quantitative change in the magnitude of the covariance values. When we compare the two components of the kernel in the two lower images of the figure, we can also appreciate that the linear kernel takes much more relevance for the total covariance matrix, and it is also much more diffuse than the RBF kernel. This will increase the importance of all the close points without excluding the relatively less

correlated points. The lack of specificity when choosing the reference points will help us to perform predictions even in a sparse input space with more dimensions.

Then, we can also introduce the scaling factor of each dimension to compute the likelihood as done before.

$$P(\Delta_{\{i|z_i^{lat}=j\},:}|\hat{X}_{\{i|z_i^{lat}=j\},:}, \theta_j^{lat}) = \frac{|W_{x,j}|^N exp(-\frac{1}{2}tr((K_x(\hat{X}_*))^{-1}\Delta_* W_{x,j}^2 \Delta_*^T))}{\sqrt{(2\pi)^{(Nd)}|K_x(\hat{X}_*)|^D}} \quad (8)$$

Where θ_j^{lat} represents the whole set of parameters ($\omega_{j,1}^{lat}$, ... , $\omega_{j,d}^{lat}$, $\lambda_{j,1}^{lat}$, ... , $\lambda_{j,d}^{lat}$,$\psi_{j,1}$, ..., $\psi_{j,2d+u_dim+1}$, σ_j^{lat}), Δ_* and $\hat{X}_*$ are the points assigned to this expert (we have again substituted the $(\{i|z_i^{lat} = j\}, :)$ subscript with $*$ in the left-hand side of the equation for esthetical purposes), N is the size of this subset of points, $|W_{x,j}|$ is the $diag(\omega_{j,1}^{lat}, ..., \omega_{j,d}^{lat})$ matrix, and $K_x(\hat{X})$ is the covariance matrix built with the kernel function that we have just described.

For the gating kernels, we are choosing the same one for both, Mapping and Latent Dynamics gating. This will be a simple scaled square distance, that is flexible enough to give more relevance to some dimensions, but without introducing too much complexity. For the Mapping Gating, we define

$$k_\phi^{map}(X_{i,:}, X_{j,:}) = exp\left(-\frac{1}{2}\sum_{k=1}^{d}(X_{i,k}, X_{j,k})^2/(\phi_k^{map})^2\right)$$

And for the Latent Dynamics

$$k_\phi^{lat}(\hat{X}_{i,:}, \hat{X}_{j,:}) = exp\left(-\frac{1}{2}\sum_{k=1}^{2d+u_d im}(\hat{X}_{i,k}, \hat{X}_{j,k})^2/(\phi_k^{lat})^2\right)$$

As we can see, the greater the ϕ parameters get, the narrower will be the influence of some points to others, and therefore, more distinct will be the separation between the points of different experts. This knowledge will be useful later on to evaluate how the model is performing and were the errors can come from.

3.2.2 Latent dimensions

As we suppose that the dimensions of the observable data are correlated in some degree, the idea of reducing the dimensions of the observable into a latent space of lower dimension to obtain better computational performance seems natural, but how many dimensions we have to use at least for a correct representation in normal circumstances is an issue itself (it is obvious that a non-bijective reduction of dimensions will always have some loss of information, or at least representation capabilities).

To make this decision, we have to take into account mainly two things, we want to have enough dimensions, so the reconstruction with the mapping model could be effective at least with the data (enough capacity of representation), but we want as few dimensions as we can because increasing the dimensions will hurt performance both in time and in the capability of generalize the model. The capability of generalizing will be given by the way that we can fill the input space of the experts (both mapping and dynamical) as the Gaussian Process interpolates the output of a new input by

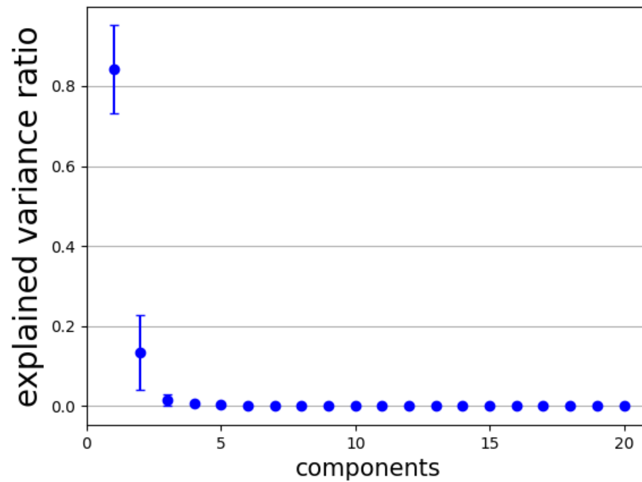


Figure 6: Explained Variance Ratio of the components of a PCA decomposition averaged over 20 different training sequences of the cloth and it's standard deviation.

considering the near input points of the known output dataset. If we have points with enough kernel correlation to any point of the usable input space of the experts (close enough), we can then evaluate any input point and have a reliable output (generalization).

Although it is impractical to fill all the input space of the experts no matter the dimensions we have, we can extend that intuition and say that the more dimensions we have, the worse capabilities of predicting the output will the experts have even for the places close to known points because there will be very few correlated points.

To resolve this trade off between the representation capacity and the experts' performance, we can take as reference the explained variance of the PCA resulting components. This parameter measures the relative amount of variance explained by each component, so taking the few components with the greatest explained variance would be a sensible decision. This will implicitly represent the precision that the mapping experts will have at least while reconstructing the observable data from the latent space , but the expected precision of reconstructing the given data after training is much better.

As we can see (Fig:6), with only three variables we can capture most of the variance that is present in the data, but we will need at least five components to guarantee a good representation of the observed data, and this is the value we are going to use from now on.

3.2.3 Training Algorithm

As the likelihood that we are aiming to optimize now involves the optimization over some discrete variables (the configuration of the infinite mixture of experts), it can't be performed straightforwardly with a regular optimizer (that can only optimize differentiable functions). To do so, we need to split the optimization algorithm in several steps as in [2].

The **First step** will always be to initialize the parameters of the model and the latent variables. An easy and fast way to initialize the latent variables will be to perform a Principal Component Analysis (PCA) decomposition over the observed trajectories and choose the d most significant components

to make sure that we have some ability to recover the previous data from the latent variables since the beginning with a linear projection. The PCA matrices that we have computed here will be saved because we are going to need them later on to transform new observations into latent space points faster and without modification of the previous transformations.

Then, we have to split the input points of the two parts of the model. For the Mapping Model, we wanted to preserve some notion of locality of the experts, so we have chosen to initialize the experts with a k-means algorithm over the latent variables space. For the Latent Dynamics Model, we have prioritized the ability to reproduce complete sequences of points. This means that the points of the same trajectory will need to be in the same expert and therefore, the initial configuration of the latent experts will be done by defining the joint $\hat{X}$ matrix and choosing the trajectories we want to assign instead of assigning individual points.

Once the experts' configurations are determined, we can initialize the parameters that correspond to each one of them. One easy way that we propose is beginning with a standard RBF kernel for the mapping experts by setting to one all the length-scales $(\lambda_{j,1}^{map}, \dots, \lambda_{j,d}^{map}) = ones(d)$, the $\sigma_j^{map} = 1$ and the weights of the different observable dimensions to also one $(\omega_{j,1}^{map}, \dots, \omega_{j,D}^{map}) = ones(D)$. For the latent dynamics' experts, we also propose to initialize with the standard version of the RBF plus linear kernel, $(\lambda_{j,1}^{lat}, \dots, \lambda_{j,2d+u_d im}^{lat}) = ones(2d + u_d im)$, $(\psi_{j,1}^{lat}, \dots, \psi_{j,2d+u_d im+1}^{lat}) = ones(2d + u_d im + 1)$, $\sigma_j^{lat} = 1$ and $(\omega_{j,1}^{lat}, \dots, \omega_{j,d}^{lat}) = ones(d)$. As the likelihood of the model could be not convex, we will need to take care and be able to detect when the optimization reaches a local optimum or the parameters grows very quickly, so we can initialize these parameters with some other reasonable random values (randomly distributed from 0.5 to 1 for example).

And last but not least, we have to initialize the parameters of the gating models. We have found that initializing with some fixed low values (e.g. $\phi = 1$ for all of them) leads to a very poor performance when we have not trained them enough. This will lead to a very poor performance during the Gibbs updating of the indicators at the beginning of the training. One way to avoid this is initializing with greater values, this will solve partially the problem, but will also lead us to a bad Gibbs sampling in later steps due to the narrow influence of the points in the local kernel. The best way that we have found until now is initializing with a greater value of ϕs and pre-train them to optimize the likelihood of assigning each point to the kernel it belongs. We will discuss this method later in this section. Also, if we are going to train similar models but with different data, we have also found that the trained ϕ parameters of one model can be a very good initialization values as long as the hyperparameters like the size of the observation space D, the size of the latent space d and the size of the control space u_{dim} don't change (this provides us some notion of transfer learning), unlike the parameters of the experts.

In the **Second Step** we will maximize the likelihood of all the experts without modifying the configuration variables $\mathbf{z}^{map}$ and $\mathbf{z}^{lat}$ of the mixture. This means that we won't need to perform the exponentially many summation of the eq (6) and focus on the likelihood of the existing experts.

$$
\begin{aligned}
&\underset{X,\theta^{map},\theta^{lat}}{\operatorname{argmax}} P(Y|X,\theta^{map},\mathbf{z}^{map})P(X|\theta^{lat},U,\mathbf{z}^{lat}) \\
&\qquad = \underset{X,\theta^{map},\theta^{lat}}{\operatorname{argmax}} \prod_{j\in E^{map}} P(Y_{\{i|z_i^{map}=j\},:}|X_{\{i|z_i^{map}=j\},:},\theta_j^{map}) \cdot \prod_{j\in E^{lat}} P(\Delta_{\{i|z_i^{lat}=j\},:}|\hat{X}_{\{i|z_i^{lat}=j\},:},\theta_j^{lat}) \\
&\qquad = \underset{X,\theta^{map},\theta^{lat}}{\operatorname{argmin}} \sum_{j\in E^{map}} \mathcal{L}_j^{map} + \sum_{j\in E^{lat}} \mathcal{L}_j^{lat}
\end{aligned}
$$

where θ^{map} and θ^{lat} stands for all the parameters of the mapping experts and the latent dynamics experts jointly, and the $\mathcal{L}_j^{map}$ and $\mathcal{L}_j^{lat}$ are the negative log-likelihood of the experts.

$$\begin{aligned}\mathcal{L}_j^{map} &= \log\left(P(Y_{\{i|z_i^{map}=j\},:}|X_{\{i|z_i^{map}=j\},:},\theta_j^{map})\right)\\ &= \frac{D}{2}\log(|K_y(X_*)|) + \frac{1}{2}tr((K_y(X_*))^{-1}Y_*W_{y,j}^2Y_*^T) - N\log(|W_{y,j}|) + consts\end{aligned}$$

where, as we have done before, X_* and Y_* represents $X_{\{i|z_i^{map}=j\},:}$ and $Y_{\{i|z_i^{map}=j\},:}$ respectively, whose other terms and parameters are described in eq. 7. And

$$\begin{aligned}\mathcal{L}_j^{lat} &= \log\left(P(\Delta_{\{i|z_i^{lat}=j\},:}|\hat{X}_{\{i|z_i^{lat}=j\},:},\theta_j^{lat})\right)\\ &= \frac{d}{2}\log(|K_x(\hat{X}_*)|) + \frac{1}{2}tr((K_x(\hat{X}_*))^{-1}\Delta_*W_{x,j}^2\Delta_*^T)) - N\log(|W_{x,j}|) + consts\end{aligned}$$

whose the parameters are described in eq. 8.

We have taken advantage of the monotonously growing property of the logarithm and the probabilities are non-negative to simplify the expression and make the computation and optimization of the likelihood more tractable.

The **Third Step** consists in a Gibbs sampling of all the indicators of both models. To do so, we will use the equation described in eq. 2 and update the indicators one by one as described in the following algorithm.

Algorithm 2 Gibbs sampling

1: **procedure** Gibbs sample the mapping experts
2:     **for** $i \in random_shuffle_indexes$ **do**
3:         `detach the i-th point from its expert`
4:         **for** $j \in E^{map}$ **do**
5:             **if** $size(\{i|z_i = j\}) < N_{max}^{map}$ **then**
6:                 $p[j] \leftarrow P(Y|X, z_i^{map} = j, \mathbf{z}_{(..,\hat{i},..)}^{map}, \theta^{map})P(z_i^{map} = j|\mathbf{z}_{(..,\hat{i},..)}^{map}, X, \phi^{map})$
7:             **else**
8:                 $p[j] = 0$
9:             **end if**
10:        **end for**
11:        $concatenate(p, \frac{\alpha}{n-1+\alpha})$
12:        `Sample j from the p distribution`
13:        **if** $j \in E^{map}$ **then**
14:            `Attach the i-th point to the expert`
15:        **else**
16:            `Create a new expert`
17:        **end if**
18:    **end for**
19: **end procedure**

where N_{max}^{map} is the maximum size of the experts who will guarantee that there will not be a small number of experts that absorbs all the points resulting in a severe hit on the time performance later on

(remember that the cost of computing the prediction and likelihood of one expert escalates cubically since it involves the inversion and multiplication of the covariance matrix). In our experiments section, we are going to tune this hyperparameter, but the general idea is that it must be big enough to contain some variability inside each expert and to not end up having too many experts, but also be small enough to compute predictions in a very small period of time.

In the line 11 of the Gibbs sampling algorithm 2, we have added an extra term to the probabilities vector p to give the possibility of adding a new expert if the likelihood of belonging to an existing expert is very low. We decided to add this term and compare them to the total likelihood of belonging to existing experts instead of comparing it only to the other gating probabilities as suggested in cite: [2] because we considered it important to know if it was really necessary to create new experts since having too many experts will increase significantly the time we need to perform future iterations of the Gibbs sampling. As the computed probabilities are all proportional to the real probability, they comparable to each other but not to the extra term. We can consider substituting $\frac{\alpha}{n-1+\alpha}$ with a simple variable α that we can easily fine-tunned to obtain the effects that we desire. Since we want to create new experts with a low probability, this value should be small, for example 10 000 times smaller than the probability of one point to belong to the expert it has been detached from, that will be about 0.1 for the mapping and 0.001 for the latent dynamic model following our simple tests.

If the sampling creates new experts, we will need to initialize the parameters associated to this. Ideally, this expert should have similar performance when predicting the data that it contains, so simply initializing the parameters with the standard version of the RBF or the RBF plus linear kernels will not suffice this condition. One slow but sure way to do it is by optimizing the likelihood of this expert modifying only the parameters, but this will lead us to a slow creation of experts that have a high chance of been removed in no time. We have decided that a good heuristic of what parameters it should have is by assigning the mean of the parameters of all the experts, this won't make a perfect initialization, but it is very fast to perform, and the results will try to take advantage of the learned knowledge of the other experts. The **Third Step** consists in a Gibbs sampling of all the indicators of both models. To do so, we will use the equation described in eq. 2 and update the indicators one by one as described in the following algorithm.

We have also found that adding a final step after the for loop where we merge all the expert with fewer points than a given threshold (let's say 5) will grant us a better performance, as we will avoid having many small and not significant experts and also confer us a better time efficient algorithm, as we will not waste time computing the probability of the points to belong to many small clusters after one complete iteration of the algorithm. Note that these small clusters will survive during the looping, but not after it, thanks to this addition.

These experts actually act like "trash bins" that collects all the unwanted points, but, as the experts only has relevance locally, and the gating will give it less local relevance than other specialized experts, the existence of this type of experts should not hurt the performance of the predictions. They only holds the points, so they can be recycled in later iterations without wasting too much computing time.

For the sampling of the latent dynamics experts, the algorithm will look exactly the same, but replacing the elements of the probabilities vector p by the equivalent version in the latent dynamics model $p[j] \leftarrow P(\Delta|\hat{X}, z_i^{lat} = j, \mathbf{z}^{lat}_{(..,\hat{i},..)}, \theta^{lat}) P(z_i^{lat} = j|\mathbf{z}^{lat}_{(..,\hat{i},..)}, \hat{X}, \phi^{lat})$.

Then we need to perform this updating of indicators a few times. We have to find the equilibrium between having a fully uncorrelated sample (by looping this several times) and taking some reasonable time to obtain a good enough sample. In our experiments, we have decided to prioritize the time

efficiency and don't care at all about having uncorrelated samples (we just want to slowly update the indicators), so we loop only one time at each step.

The **Forth Step** is to train the gating parameters ϕ^{map} and ϕ^{lat}. This will be done by maximizing the prediction likelihood of each point when detached from the expert. This means that we are computing the likelihood of the output that each expert will predict, and then multiply it with the probability of assigning this input to those experts of the gating model with a scalar product. That value is what we sum up for all the points and then maximize the value of the summation over the gating parameters. To compute the value that we want to maximize, we are going to go along with the algorithm 3:

Algorithm 3 Gating Objective funtion

1: **procedure** COMPUTE THE OBJECTIVE FUNCTION
2: $Obj \leftarrow 0$
3: **for** $i \leftarrow 1..n$ **do**
4: `detach the i-th point from its expert`
5: **for** $j \in E^{map}$ **do**
6: $p[j] = P(Y_i | X_i, Y_{\{i|z_i=j\},:}, X_{\{i|z_i=j\},:}, \theta_j^{map})$
7: $pz[j] = \frac{\sum_{i' \neq i} K_\phi(X_{i,:}, X_{i',:})\delta(z_i, j)}{\sum_{i' \neq i} K_\phi(X_{i,:}, X_{i',:})}$
8: **end for**
9: $Obj \leftarrow Obj + \langle p, pz \rangle$
10: `reattach the i-th point to the previous expert`
11: **end for**
12: **Return** Obj
13: **end procedure**

The objective function for the latent dynamics model would be analogous.

Finally, we loop from the second step to fourth until we are satisfied with the prediction capabilities of the model.

To summarize, the training algorithm will consist in some few steps were we optimize and sample different parts of the models sequentially but always trying to maximize the final likelihood of the complete model:

1. Initialize all the parameters, from the latent varaibles to the experts configuration and parameters.
2. Maximize the likelihood of the experts.
3. Gibbs sample the discrete configuration indicators.
4. Maximize the likelihood of the predictions with the gating models
5. Loop from step 2 to 4 until satisfactory performance

3.2.4 Covariance matrix inverse management

To increase the time efficiency during the computation of the likelihood of the experts, we have to store the inverse of the covariance matrix of every expert throughout the Gibbs sampling. This

will allow us to get rid of the costly computation of inverses that will not change in general terms, since the experts' parameters and the latent variables values are frozen. The only changes that will modify these matrices are the changes of the configuration of the experts during the detachment and attachments of points to experts. Both of them involve attaching or detaching one line and one column on the covariance matrices of the affected experts, so updating the inverses of the matrices can be done with rank one updates on the inverses of the matrix (one for the column and one for the row) instead of the expensive recalculation of the covariance matrices (at least one detachment at the beginning and one attachment per expert to obtain the probabilities vector p).

This can be achieved with the Sherman-Morrison formula [13]:

$$\left(\mathbf{A}^{-1} + uv^T\right)^{-1} = \mathbf{A}^{-1} + \frac{\mathbf{A}^{-1}uv^T\mathbf{A}^{-1}}{1 + v^T\mathbf{A}^{-1}u}$$

where u and v are vertical vectors of the same size as the matrix $\mathbf{A}$, and $\mathbf{A}^{-1}$ its inverse. To detach one point (let's say the i-th row and column) from the covariance matrix inverse, we will have to perform two one rank matrix updates.

$$\mathbf{A}'^{-1} = ((\mathbf{A} - W_1) - W_2)^{-1}$$

where

$$W_1 = \begin{bmatrix} 0 & \dots & q_1 & \dots & 0 \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ & & q_{i-1} & & \\ 0 & \dots & 0 & \dots & 0 \\ & & q_{i+1} & & \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ 0 & \dots & q_n & \dots & 0 \end{bmatrix} = \begin{pmatrix} q_1 \\ \vdots \\ q_{i-1} \\ 0 \\ q_{i+1} \\ \vdots \\ q_n \end{pmatrix} \begin{pmatrix} 0 & \dots & \overset{\hat{i}}{1} & \dots & 0 \end{pmatrix}$$

(the column of W_1 containing the q values is column $\hat{i}$)

and

$$W_2 = \hat{i} \begin{bmatrix} 0 & \dots & & 0 & & \dots & 0 \\ \vdots & \ddots & & \vdots & & \ddots & \vdots \\ q_1 & \dots & q_{i-1} & 0 & q_{i+1} & \dots & q_n \\ \vdots & \ddots & & \vdots & & \ddots & \vdots \\ 0 & \dots & & 0 & & \dots & 0 \end{bmatrix} = \hat{i} \begin{pmatrix} 0 \\ \vdots \\ 1 \\ \vdots \\ 0 \end{pmatrix} \begin{pmatrix} q_1 & \dots & q_{i-1} & 0 & q_{i+1} & \dots & q_n \end{pmatrix}$$

where $q_* = k(X_i, X_*)$ the correspondent covariance between the i-th input and the *-th input of the expert. Doing it like this, we have turned the updating of the inverse from a cubical complexity ($\mathcal{O}(n^3)$) to a squared complexity ($\mathcal{O}(n^2)$) only in the multiplication of a matrix with a vector and

the summation of two matrices but also with a much smaller cost even with small n. After obtaining the $\mathbf{A}'^{-1}$ matrix, we will obtain the inverse of the submatrix by excluding the i-th row and column from it. This can be done when the correspondent row and column are zeros except for the (i, i) position and easily demonstrated by preforming operations with submatrices of $\mathbf{A}'$

Proof.

$$\begin{bmatrix} A_{1,1} & \mathbf{0} & A_{1,2} \\ \mathbf{0} & a_{i,i} & \mathbf{0} \\ A_{2,1} & \mathbf{0} & A_{2,2} \end{bmatrix}^{-1} = \left(S^T \begin{bmatrix} A_{1,1} & A_{1,2} & \mathbf{0} \\ A_{2,1} & A_{2,2} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & a_{i,i} \end{bmatrix} S \right)^{-1} = S^T \begin{bmatrix} A_{1,1} & A_{1,2} & \mathbf{0} \\ A_{2,1} & A_{2,2} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & a_{i,i} \end{bmatrix}^{-1} S =$$

$$= S^T \begin{bmatrix} B_{1,1} & B_{1,2} & \mathbf{0} \\ B_{2,1} & B_{2,2} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & a_{i,i}^{-1} \end{bmatrix} S = \begin{bmatrix} B_{1,1} & \mathbf{0} & B_{1,2} \\ \mathbf{0} & a_{i,i}^{-1} & \mathbf{0} \\ B_{2,1} & \mathbf{0} & B_{2,2} \end{bmatrix}$$

where S is the change-of-basis matrix that will help us permute the elements and is defined as

$$S = \begin{bmatrix} \mathbf{Id}_{1,1} & \mathbf{0} & \mathbf{0}_{1,2} \\ \mathbf{Id}_{1,2} & \nvdash & \mathbf{Id}_{2,2} \\ \mathbf{0} & 1 & \mathbf{0} \end{bmatrix}$$

and the B submatrices are defined by the inverse of the joint submatrices of A

$$\begin{bmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{bmatrix} = \begin{bmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{bmatrix}^{-1}$$

We have used the subscripts of the matrices to help us keep track of the dimensions of the block wise operations and taken advantage of the inversion of a block wise diagonal matrix tajes the following shape

$$\begin{bmatrix} A_{1,1} & \mathbf{0} \\ \mathbf{0} & A_{2,2} \end{bmatrix}^{-1} = \begin{bmatrix} A_{1,1}^{-1} & \mathbf{0} \\ \mathbf{0} & A_{2,2}^{-1} \end{bmatrix}$$

□

To obtain the inverse of a supermatrix (attaching a point to the expert's covariance matrix), we will start adding a column and a row of zeros in the i-th position where the point would be, and the (i, i) position of the new inverse have to start with $1/k(X_i, X_i)$ as the inverse of this "submatrix" of one by one. Then, the updates will be performed similarly as detaching an element.

$$\mathbf{A}^{-1} = ((\mathbf{A}' + W_1) + W_2)^{-1}$$

To compute the likelihood of the experts, we also need to know the logarithm of the determinant of the covariance matrix. As performing these calculations also involves cubical complexity operations, it would also be pretty natural to store and update efficiently the value of the log-determinant of the covariance matrices. To do so, we can introduce and demonstrate the effects on the determinant of a matrix with a rank one perturbation.

$$det(\mathbf{A} + \mathbf{u}\mathbf{v}^T) = det(\mathbf{A})(1 + \mathbf{v}\mathbf{A}^{-1}\mathbf{u})$$

Proof. To proof the equality, we will use the distributive property of the determinant $det(AB) = det(A)det(B)$ and the fact that the determinant of a triangular matrix is the product of the elements of the diagonal. First we can have that:

$$(\mathbf{A} + \mathbf{u}\mathbf{v}^T) = \mathbf{A}(\mathbf{Id} + \mathbf{A}^{-1}\mathbf{u}\mathbf{v}^T)$$

Then, we only have to obtain the determinant of the second multiplicand.

$$\begin{aligned}
det(\mathbf{Id} + \mathbf{A}^{-1}\mathbf{u}\mathbf{v}^T) &= det\begin{pmatrix}\mathbf{Id} + \mathbf{A}^{-1}\mathbf{u}\mathbf{v}^T & \mathbf{A}^{-1}\mathbf{u} \\ \mathbf{0} & 1\end{pmatrix} \\
&= det\left(\begin{pmatrix}\mathbf{Id} & \mathbf{0} \\ \mathbf{v}^T & 1\end{pmatrix}\begin{pmatrix}\mathbf{Id} + \mathbf{A}^{-1}\mathbf{u}\mathbf{v}^T & \mathbf{A}^{-1}\mathbf{u} \\ \mathbf{0} & 1\end{pmatrix}\begin{pmatrix}\mathbf{Id} & \mathbf{0} \\ -\mathbf{v}^T & 1\end{pmatrix}\right) \\
&= det\left(\begin{pmatrix}\mathbf{Id} + \mathbf{A}^{-1}\mathbf{u}\mathbf{v}^T & \mathbf{A}^{-1}\mathbf{u} \\ \mathbf{v}^T + \mathbf{v}^T\mathbf{A}^{-1}\mathbf{u}\mathbf{v}^T & \mathbf{v}^T\mathbf{A}^{-1}\mathbf{u} + 1\end{pmatrix}\begin{pmatrix}\mathbf{Id} & \mathbf{0} \\ -\mathbf{v}^T & 1\end{pmatrix}\right) \\
&= det\begin{pmatrix}\mathbf{Id} + \mathbf{A}^{-1}\mathbf{u}\mathbf{v}^T - \mathbf{A}^{-1}\mathbf{u}\mathbf{v}^T & \mathbf{A}^{-1}\mathbf{u} \\ \mathbf{v}^T + \mathbf{v}^T\mathbf{A}^{-1}\mathbf{u}\mathbf{v}^T - (\mathbf{v}^T\mathbf{A}^{-1}\mathbf{u} + 1)\mathbf{v}^T & \mathbf{v}^T\mathbf{A}^{-1}\mathbf{u} + 1\end{pmatrix} \\
&= det\begin{pmatrix}\mathbf{Id} & \mathbf{A}^{-1}\mathbf{u} \\ \mathbf{0} & \mathbf{v}^T\mathbf{A}^{-1}\mathbf{u} + 1\end{pmatrix} \\
&= (\mathbf{v}^T\mathbf{A}^{-1}\mathbf{u} + 1)
\end{aligned}$$

And finally, we substitute the terms in the equality

$$det(\mathbf{A} + \mathbf{u}\mathbf{v}^T) = det\left(\mathbf{A}(\mathbf{Id} + \mathbf{A}^{-1}\mathbf{u}\mathbf{v}^T)\right) = det(\mathbf{A})det(\mathbf{Id} + \mathbf{A}^{-1}\mathbf{u}\mathbf{v}^T) = det(\mathbf{A})(1 + \mathbf{v}\mathbf{A}^{-1}\mathbf{u})$$

□

And the rank one updates to detach and attach elements to the experts will be the same as the updating of the inverse.

As all of these operations involves the product of big matrices, the numerical errors cannot be ignored. Specially if we are going to concatenate several modifications of the experts' compositions, the compounding of the errors can grow very fast, leading to bad calculations of the likelihood or even fatal errors like dividing by zero. As we cannot detect those errors fast, we cannot rely on the compounded updates, and we are using these new matrices just to compute the probabilities to assign different experts, but the stored and updated matrices at the end of each assignment will be computed with traditional methods.

3.2.5 Predictions Algorithm

We have at least two ways to make predictions with the Mixture of Gaussian Experts. The first one is by considering the predictions of each expert as separate, uncorrelated random variables with normal distributions. Then, the mixture of experts can be considered as a weighted sum of all the random variables. As all the variables are uncorrelated and the covariance matrices we generate are diagonal (each dimension of the output is also uncorrelated), the outcome will also have a normal distribution.

$$Y_{model} = \sum_{j \in E} P(j) Y_j \sim \sum_{j \in E} P(j) \mathcal{N}(\mu_j, \sigma_j^2) \sim \mathcal{N}\left(\sum_{j \in E} P(j)\mu_j, \sum_{j \in E} P(j)^2 \sigma_j^2 \right)$$

Proof. The demonstration of the avobe equation can be done with each dimension of the prediction individually. Let's consider two normal variables with unit variance $X.Y \sim \mathcal{N}(0,1)$ and the sumation of the two variables $Z = X + Y$. We denote the probability distribution function of each one as

$$f(x) = \frac{1}{\sqrt{2\pi}} e^{-x^2/2}, \quad g(y) = \frac{1}{\sqrt{2\pi}} e^{-y^2/2},$$

Then, the Cumulative Distribution Function of Z looks like

$$z \longmapsto \int_{x+y \leq z} f(x)g(y)dxdy$$

The key observation is that

$$f(x)g(y) = \frac{1}{2\pi} e^{-(x^2+y^2)/2}$$

is radially simetric, so we rotate the coordinate plane about the origin such that the line $x+y=z$ is described by the equation $x' = c$, where $c = c(z)$ can be determined geometrically, and thanks to the radial simetry, $f(x)g(y) = f(x')g(y')$, and the y' can be integrated out from the CDF of Z

$$\int_{x' \leq c, y' \in \mathcal{R}} f(x')g(y')dx'dy' = \int_{-\inf}^{c(z)} f(x')dx' = \Phi(c(z))$$

We then have that the $c(z)$ has a normal distribution of zero mean and unit variance. To determine the function, we find that after the rotation, the x' coordinates must be perpendicular to the $x+y=z$ line, so $c(z)$ will be the distance of that line to the origin $c(z) = \sqrt{(z/2)^2 + (z/2)^2} = z/\sqrt{2}$, and therefore

$$z/\sqrt{2} \sim \mathcal{N}(0,1) \Rightarrow z \sim \mathcal{N}(0,2)$$

For any a, b, we can obtain the distribution of $Z = aX + bY$ with the same argument but using the correspondent plane $ax + by = z$ leading to $c(z) = \frac{z}{\sqrt{a^2+b^2}}$

$$aX + bY \sim \mathcal{N}(0, a^2 + b^2)$$

And the same argumet follows with N variables in a N-dimensional space

$$\sum_{i=1}^{N} a_i X_i \sim \mathcal{N}(0, \sum_{i=1}^{N} a_i^2)$$

Finally, applying the following equivalence

$$X_i \sim \mathcal{N}(\mu_i, \sigma_i^2) \Leftrightarrow \frac{1}{\sigma_i}(X_i - \mu) \sim \mathcal{N}(0,1)$$

We can get the desired formula

$$\sum_{i=1}^{N}(a_i\sigma_i)\frac{1}{\sigma_i}(X_i - \mu_i) \sim \mathcal{N}\left(0, \sum_{i=1}^{N}(a_i\sigma_i)^2\right) \Rightarrow \sum_{i=1}^{N} a_i X_i \sim \sum_{i=1}^{N} a_i \mu_i \mathcal{N}\left(0, \sum_{i=1}^{N}(a_i\sigma_i)^2\right)$$

□

where Y_{model} denotes the random variable that is the output of the model, Y_j the ones of each expert and $P(j)$ is the probability of choosing the j-th expert, and employs the gating models:

$$P(j) = P(j|X_{input}, X, \phi) = \frac{\sum_i K_\phi(X_{i,:}, X_{input})\delta(z_i, j)}{\sum_i K_\phi(X_{i,:}, X_{input}}$$

Although this will give us a very simple way of the sample the predictions, and the flexibility to use only the mean output instead of sampling for a fast execution of the predictions, this can result in pretty bad results as we could fall in between of several experts' predictions with an overall low likelihood.

The other way to sample the predictions is by generating a new distribution as the result of the weighted sum of all the experts' distributions (the Mixture of Gaussian Experts distribution). This might not have a simple shape and the direct sampling could be pretty hard to obtain, as we have to sample each dimension one by one and integrating over the space to obtain the Cumulative Probability Distribution.

$$P(Y_{pred}|X_{input}, Y, X) = \sum_{j \in E} P(j|X_{input}, X, \phi) P(Y_{pred}|X_{input}, Y_{(i|z_i=j),:}, X_{(i|z_i=j),:}, \theta_j)$$

If we choose to use the Mixture of Gaussian Experts Distribution, we can also sample first which expert we are going to use, and then sample the prediction with this expert (or use a simple predictor like the mean output instead of sampling). As a result, we will obtain one sample of the distribution, but this has the risk of choosing the wrong expert and maybe end up having inconsistent predictions.

As we can see in figure 7, there are several advantages and disadvantages of choosing one method over the other when we try to make the predictions. In the left image, the two experts are very separated, and the predictor will have to choose either one or the other expert to have a plausible prediction, so it will have several local maxima. If we average the two experts' predictions, it will generate a distribution with the maxima in a very low likelihood point, but it will be more conservative than choosing only one expert. When we perform the mean predictor (choose one mean of the distribution instead of sampling) for a faster performance, the two different methods will clearly cast very disparate results.

In the right image, the experts have an overlapped distribution, and the average distribution will be very similar to the Mixture of Gaussian Experts distribution, so probably choosing this predictor will have much better performance than choosing only one expert if we are going to perform a sampling or a mean predictor. We will discuss later in the experiments section 4 which option has

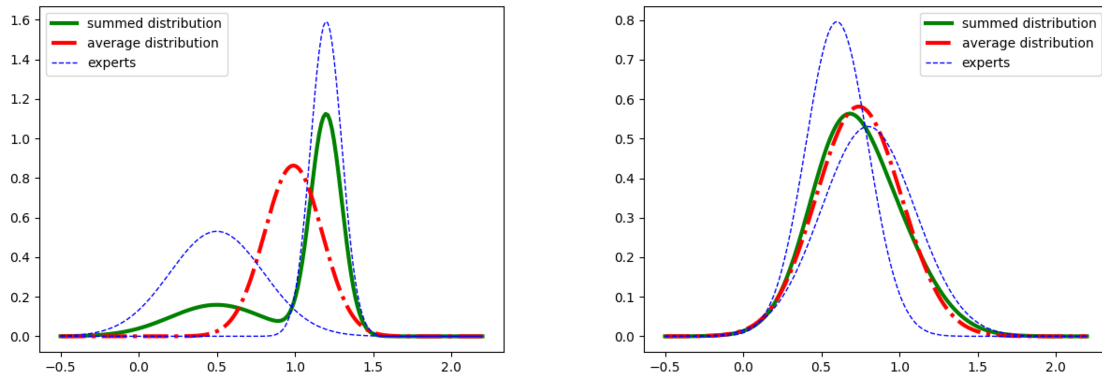


Figure 7: Distribution of the Mixture of Experts, in green and continuous line the Mixture of Gaussian Experts (weighted sum of the two Experts Distribution), and in red and dashdot line, the distribution of the average predictor (single normal distribution with weighted average mean and variance).

better performance. This will mostly depend on the composition of the experts and the compounding of the distributions of the experts, as well as the relative relevance of the critical points when we have to choose one or the other.

3.2.6 Observation to latent space

If we want to use the model to predict the dynamics of a deformable object given the known control and the training data, we need to be able to translate the currently observed position to the latent space. As we have seen, the model does not provide us any straightforward way to obtain the latent representation of new data, we have to find the way to obtain the desired latent variable.

This method should fulfill at least two conditions, it has to be fast, as we will need to perform it before initializing the rollout of the model. And it has to be accurate, as any little error will be compounded once we have started the predictions. To decide how we do it, we are going to propose a few options and then choose the one that fits better our needs.

The first and fastest option is by computing the distance of the new observation to the observations of the training set and choose the latent variable of the closest point (that already has a representation in the latent space built in the model). This will have a very fast linear complexity and may give pretty good results if the observations are very close to the training set.

Another option will be to choose a few closest points instead of one and averaging their latent representations. This option would be almost as fast as the previous one and might be more robust when it comes to assigning the representation as there is no chance that in a bunch of points, all of them has a bad latent variable counterpart. But it also comes with the downside that we have to choose the threshold distance that we are going to consider or the minimum and maximum number of points that we are going to average.

The next option will be saving the PCA matrices that we have created during the initialization of the latent variables and fit the new observed data to the dimensional reduction. This operation could be very efficient in time, as it only involves the multiplication of a small matrix with the observable. But it is very likely that the latent space has shifted from the one created by the PCA.

And the last option is the hardest to implement and slower to execute, but it should return better latent representations. This one consists in initializing with some latent representation of the observed data (with any of the previous methods) and maximize the likelihood of predicting this observable by the mapping model.

$$\underset{X_{new}}{\text{argmax}}\, P(Y_{new}|X_{new}, Y, X, \beta)$$

In general, we will consider enough to average the few closest points setting a maximum of 10 points, and a maximum relative distance of 0.5. The relative distance will be computed as

$$d_i = \frac{\|Y_{new} - Y_{i,:}\|}{\|Y_{new}\|}$$

3.2.7 Balance hyperparameter

During the training of experts, we are optimizing the summation of the negative log-likelihood of the mapping experts and the latent experts. This induces us to consider the two likelihood terms equivalent during the calculus of the gradient using back propagation algorithms, although both terms could be of a very different scale of magnitude because the mapping have multipliers of the size of the output and the latent only have the multipliers of the magnitude of the latent space.

$$\mathcal{L} = \mathcal{L}^{map} + \mathcal{L}^{lat} = \sum_{j \in E^{map}} \frac{D}{2}\log(|K_y(X_*)|) + \frac{1}{2}tr((K_y(X_*))^{-1} Y_* W_{y,j}^2 Y_*^T) - N\log(|W_{y,j}|)$$
$$+ \sum_{j \in E^{lat}} \frac{d}{2}\log(|K_x(\hat{X}_*)|) + \frac{1}{2}tr((K_x(\hat{X}_*))^{-1} \Delta_* W_{x,j}^2 \Delta_*^T)) - N\log(|W_{x,j}|) + consts$$

If the two loss functions had separate variables to optimize, we could ignore this fact and just wait until both functions reach their minimum independently, but in our case, the latent space variables determine the evaluation of the two functions. As one of the functions could be overrepresented due to the nature of the composition, with an *a posteriori* thought, we considered adding an extra balance parameter that can equilibrate the two terms weight during the optimization of the latent variables. By default, the balance parameter is set to one, since this value recovers the original function we were optimizing.

The addition of this parameter shifts the likelihood that we are maximizing, therefore changing the optimal solution of all the parameters, so we have to proceed with care when we change this value.

$$\underset{X,\theta^{map},\theta^{lat}}{\text{argmin}}\ \mathcal{L}^{map} + balance \cdot \mathcal{L}^{lat} = \underset{X,\theta^{map},\theta^{lat}}{\text{argmax}}\ P(Y|X, \theta^{map}, \mathbf{z}^{map})P(X|\theta^{lat}, U, \mathbf{z}^{lat})^{balance}$$

Later on, we are also going to discuss how different values will affect to the final predictors' capabilities and several problems that we encountered when we impose too small or big balance parameters.

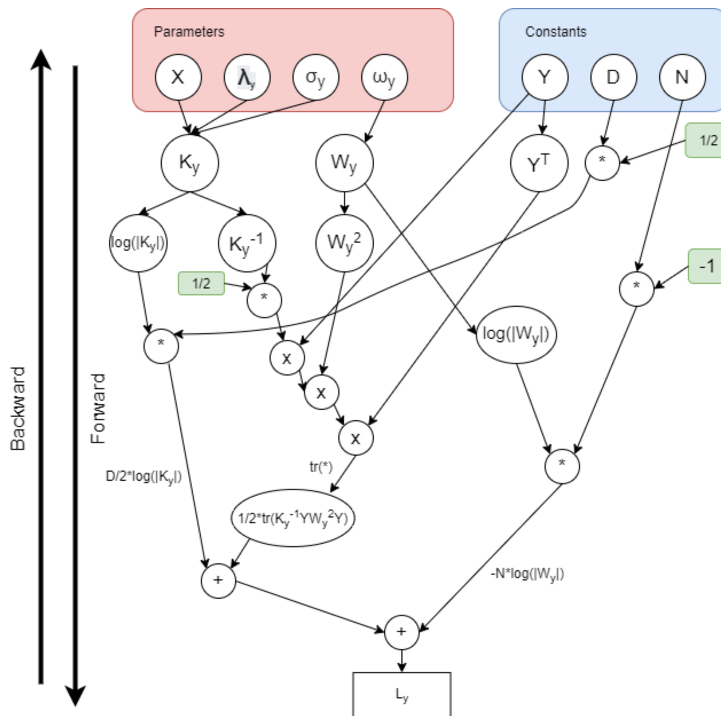


Figure 8: Simplification of the mapping negative log-likelihood computational graph

3.2.8 PyTorch

For the implementation of the model, we decided to employ python as the main programming language because it is widely adopted on machine learning applications and its easiness of use thanks to the huge quantity of libraries available like numpy, sklearn and specially the pytorch library. This last library is based on torch library and developed by the Facebook AI Research lab (FAIR) but with free and open-source modified BSD (Berkeley Software Distribution) license, what imposes minimal restrictions on the use and distribution of covered software.

It is also implemented on C, what has maximum relevance, because interpreted languages as python cannot reach the computation efficiency that compiled languages has when we involve complex operations with matrices and repetitive loops when searching the gradient of functions. Of the modules that are built in the pytorch library, we are going to use the torch.nn module as it has very useful functions as backward(), that automatically computes the gradient over the parameters, and optimizers, that manages those gradients over different optimization steps and situations where the simple gradient descent will not be effective enough.

To compute the gradient of the loss function, these oriented to machine learning and deep learning libraries usually have implemented the back propagation algorithm to compute the gradient for optimization purposes. To do so, they take advantage of the chain rule and form computation graphs to organize the propagation of the gradient similar to the one represented in 8 applying the correspondent derivation rules to each operation one by one and compound them altogether at the end. This will allow us to compute analytically the gradient of any function, no matter the complexity, as long as the function is differentiable (at least piece wise).

One of the main advantages of the torch library over the TensorFlow one is how the computational

graph is built. In TensorFlow, you have to predefine all the structure of the graph before you can perform any real computation, so if you are going to use any external data, you are going to have the structure of them predefined in some placeholders that you need to fill during the runtime. On the other hand, the PyTorch library believes in dynamic graphs, that means that the graph can be defined/manipulated on-the-go during the forward propagation. The aforementioned property allows us a much more flexible way to calculate the loss function since we can be free of introducing and detaching variables from the graph, make many conditional operations without having to worry about the stability or the efficiency of the graph (avoiding unnecessary back propagation of gradients that are not going to intervene later) and the construction of fancier graphs structures. The dynamical approach will also give us a better debugging experience, as the graphic is defined at runtime you can use any Python debugging tools or even old trusty print statements.

3.2.9 Optimization algorithms

The PyTorch library has many optimizers available, most of them are specialized in very specific cases. The sparse optimizers are particularly efficient when we are trying to train models that involve sparse tensors (with many zeros) and only back propagates through the non-zero elements, The stochastic gradient descent algorithms are particularly good at managing the cases when we cannot encompass all the dataset in one iteration of the back propagation (generally due to memory limitations) this may result in a saw tooth shape of the loss function along the training (as the loss value will depend on the data we are evaluating). Most of these optimizers will handle the problem by averaging the gradients over several iterations (also called with momentum) or other heuristics to end up decreasing the loss values in average.

In our case, we have a pretty tractable amount of data, so we won't need to split the data, and we don't expect having very sparse tensors along the model, hence, we can use pretty standard algorithms. Here, we are going to delve into the three that we considered more relevant for our model.

The **Gradient Descent** is an optimization algorithm that consists in performing consecutive updates following the gradient direction in the parameters space and taking steps of the size of the learning rate (usually denoted as α). This is one of the simplest algorithms to train machine learning models, and it is the foundation of most of those algorithms, like a dynamical learning rate (that reduces the length of the learning rate as we are training the model and therefore approaching to the optima).

If we consider the loss function that we want to minimize as a function of the parameters $f(\theta)$, the updating of the parameters at step t can be written as

$$\theta(t+1) = \theta(t) - \alpha * \nabla f(\theta(t))$$

The **Adam** optimizer implements the Adaptive Moment Estimator algorithm [14], it is an extension to gradient descent and automatically adapts a learning rate for each input variable for the objective function and further smooths the search process by using an exponentially decreasing moving average of the gradient to make updates to variables.

Each learning rate is automatically adapted throughout the search process based on the gradients encountered for each variable, maintaining a first and second moment of the gradient. First, we must maintain a moment vector and exponentially weighted infinity norm for each parameter being optimized as part of the search, referred to as m and ν respectively. They are initialized to 0.0 at

the start of the search. Then, at each time step t, we update the first moment with the gradient and the hyperparameter β_1:

$$m(t+1) = \beta_1 m(t) + (1-\beta_1)\nabla f(\theta(t))$$

and then we update the second moment with the squared gradient and the hyperparameter β_2:

$$\nu(t+1) = \beta_2 \nu(t) + (1-\beta_2)\nabla f(\theta(t))^2$$

As they are initialized with zero, the moments will be biased during the first steps (previous values of the moments are influenced by the initialization), and the moments need to be corrected.

$$\hat{m}(t) = m(t)/(1-\beta_1)$$

$$\hat{\nu}(t) = \nu(t)/(1-\beta_2)$$

The updating of the parameters will finally look like this

$$\theta(t+1) = \theta(t) + \alpha\hat{m}(t)/(\sqrt{\hat{\nu}(t)} + \epsilon)$$

where the ϵ is some small value (like $1e-8$) that grant that we don't divide by zero. The decaying values will be typically $\beta_1 = 0.9$ and $\beta_2 = 0.999$, values that will make the previous gradients still relevant in the few following steps (the greater, the more relevant up to a maximum of one), but can always be modified or even have dynamical values.

The **L-BFGS** (Limited memory Broyden–Fletcher–Goldfarb–Shanno) [15] algorithm is an optimization algorithm in the family of quasi-Newton methods that approximates the BFGS using a limited amount of computer memory.

The Newton's method iteratively updates the parameters with a quadratic Taylor expansion of the function near the parameters at each time step:

$$f(\theta_n + \Delta\theta) \approx f(\theta_n) + \Delta\theta^T \nabla f(\theta_n) + \frac{1}{2}\Delta\theta^T(\nabla^2 f(\theta_n))\Delta\theta$$

were we can simplify the equation by renaming some terms $h_n(\Delta\theta) := f(\theta_n + \Delta\theta)$, $g_n := \nabla f(\theta_n)$ and $\mathcal{H}_n := \nabla^2 f(\theta_n)$. Then, we want to choose the $\Delta\theta$ that minimizes the $h_n(\Delta\theta)$ function.

$$\frac{\partial h_n(\Delta\theta)}{\partial \Delta\theta} = g_n + \mathcal{H}_n \Delta\theta$$

If we assume that the hessian $\mathcal{H}_n$ is positive definite, any local extreme of $h_n(.)$ is a local minimum, so we only need to find a $\Delta\theta$ that evaluates the left-hand side of the previous equation.

$$\Delta\theta = -\mathcal{H}_n^{-1} g_n$$

As in previous optimization algorithms, this will determine some direction to optimize the function, but typically we will choose some value $0 < \alpha$ to perform the parameters update.

$$\theta_{n+1} = \theta_n + \alpha(-\mathcal{H}_n^{-1} g_n)$$

As the function that we need to optimize grows in the number of parameters, the computation of the hessian and its inverse ends up been not tractable, so new methods emerge to solve these problems like the quasi-Newton methods, where instead of computing it at each step, we try to recover the information of the hessian (or an approximation) by updating the initial hessian matrix (or some guess of the hessian) at each optimization step.

Some of the properties that we can impose to a good hessian update could be the secant condition, what means that the gradient of the quadratic Taylor approximation ($h_n(\Delta\theta) = f(\theta_n) + \Delta\theta^T g_n + \Delta\theta^T \mathcal{H}_n \Delta\theta$) and the gradient of the actual function are equal in the current parameters and previous parameters point.

$$\begin{cases} \nabla h_n(0) = \nabla f(\theta_n) =: g_n \\ \nabla h_n(\theta_{n-1} - \theta_n) = \nabla f(\theta_{n-1}) =: g_{n-1} \end{cases} \Rightarrow \nabla h_n(0) - \nabla h_n(\theta_{n-1} - \theta_n) = g_n - g_{n-1}$$

$$\Rightarrow f(\theta_n) + g_n + \mathcal{H}_n \cdot 0 - f(\theta_n) - g_n - \mathcal{H}_n(\theta_{n-1} - \theta_n) = g_n - g_{n-1} \Rightarrow \mathcal{H}_n(\theta_n - \theta_{n-1}) = (g_n - g_{n-1})$$

This ensures that the updated version of the hessian behaves like a hessian, at least for the difference $(\theta_n - \theta_{n-1})$. Assuming the hessian is invertible, renaming $y_n := g_n - g_{n-1}$ and $s_n = \theta_n - \theta_{n-1}$ we obtain the secant condition:

$$\mathcal{H}_n^{-1} y_n = s_n$$

The other essential condition that the hessian must satisfy is the symmetrical condition, since we assume that the loss function is at least order two derivable and the order of differentiation doesn't matter.

Given these conditions, we would like to find the best approximation to the real $\mathcal{H}_{n+1}$ hessian matrix that can satisfy them.

$$\begin{aligned} &\min_{\mathcal{H}^{-1}} \|\mathcal{H}^{-1} - \mathcal{H}_n\|^2 \\ &s.t. \mathcal{H}^{-1} y_n = s_n \\ &\quad \mathcal{H}^{-1} \texttt{is symetric} \end{aligned}$$

The norm used here is the weighted frobenius norm. The solution to this problem is:

$$\mathcal{H}_{n+1} = (\mathbf{Id} - \rho_n y_n s_n^T) \mathcal{H}_n^{-1} (\mathbf{Id} - \rho_n y_n s_n^T) + \rho_n s_n s_n^T$$

where $\rho_n = (y_n^T s_n)^{-1}$. This is the BFGS update, and it keeps the positive definite property (if $\mathcal{H}_n$ is psd, $\mathcal{H}_{n+1}$ also is), and we only need to have stored the initial $\mathcal{H}_0$ and the successive s_n and y_n vectors to construct at each step the approximation of the hessian. As we only need to be able to compute the product of the matrix with a vector $\mathcal{H}_n^{-1} g_n$ and obtain the direction of optimization, now we have a procedural algorithm to obtain the direction and without computing any real hessian 4.

As we keep iterating the algorithm, we need to keep track of all the vectors ending with a potentially infinite memory requirement. To solve this, the limited memory version of the BFGS

Algorithm 4 Compute the product $\mathcal{H}_n^{-1} g_n$

```
 1: procedure BFGS MULTIPLICATION(H_0^{-1}, {s_k}, {y_k}, g_n)
 2:     d ← g_n
 3:     // Compute the right product
 4:     for i ← 1..n do
 5:         α_i ← ρ_i s_i^T d
 6:         d ← d − α_i y_i
 7:     end for
 8:     // Compute center
 9:     d ← H_0 d
10:     // Compute the left product
11:     for i ← 1..n do
12:         β ← ρ_i y_i d
13:         d ← d + (α_{n−i+1} − β) s_i
14:     end for
15:     Return d
16: end procedure
```

(L-BFGS) only stores the last m vectors of y_k and s_k, and usually starts every time with the same not dense hessian (e.g $\mathcal{H}_{n-m}$ is always **Id**). Then computes the direction of optimization with the same algorithm as before.

Even with these limited-memory variant, the requirements still pretty high and can easily fulfil the 16 GB of RAM memory that we have available in the Google Colab environment resulting in crashes of the kernel during the trainings. Specially if we don't take care of detaching the branches that are irrelevant to us during the forward calculation graph (e.g. during the training of the gating parameters, we need to detach the latent variables every time we use them as we are not going to modify them, but they will fill memory space). Despite those inconveniences, this method has shown to outperform the Gradient descent variants specially where the surface of the loss function is not flat, both in number of iterations and the final loss function.

In general, we will prefer to use the more sensitive algorithms, that are capable of fine tune the parameters better, but in situations where there are many small local minims, we need a rough algorithm like the simple Gradient Descent with a fixed α parameter, otherwise, we take the risk of falling in suboptimal parameters without the capacity of reaching the optima. This will be especially relevant when we are training the parameters of the gating model, as the predictions' likelihood can be full of local minima for each data point and in the experiments, we found that in most of the cases, the gradient was erratically enough that the Adam and LBFGS algorithms were not able to even change the parameters, where the simpler version of the Gradient Descent successfully trained the parameters in the long term.

4. Experiments

In this section, we want to make an overview of the experimentation process that guided us on deciding how to create the best models that we can get with this theoretical frame, and make qualitative comparisons against the previous model (Controlled GPDM) both in time efficiency and predictions reliability.

To compare the models' performance, we have to introduce some parameters. Firstly, we will measure the accuracy of the observable space prediction with the relative error along one complete trajectory (or some subsequence of the trajectory):

$$\epsilon^k = 100 \cdot \sum_{t=1}^{N} \frac{\|y_t^k - y_t^{*k}\|}{\|y_t^k\|}$$

where the k superscript indicates that it corresponds to the k-th test trajectory, the y_t represents the observation of the t-th time step of the reference trajectory and the y_t^* is the analogous but for the predicted trajectory. This latter trajectory is obtained by mapping from the latent trajectory x^* with the mapping model, and the latent trajectory is obtained by compounding the predictions of the latent dynamical model.

If we perform this measurement over several test trajectories, we can get an idea of how it will perform predicting those long sequences and even know the confidence interval in which we can expect the errors to happen by computing the standard deviation of the relative error of those test trajectories.

Before we start using this error function, we have to clarify that the error is not constant along the whole sequence, as we introduce the first steps to start the rollout, the error near them will be very small (if any, it will come from the mapping model, not from the dynamical model) but as we keep going, we will be compounding the error in the latent space leading to very big and not representative trajectory error as we can see in the figure 9. In general, the mean of the error will grow monotonously with the number of steps until we reach regions where there is not any near point and the predictions generates random samples without correlation with the training data.

To simplify the measure of the relative error, we will use the mean over all the sequence as a reliability parameter, but we have to have in mind that the rollout over shorter sequences in general will have smaller relative error values than the longer ones. We will specially need to keep this in mind if we use different datasets with different lengths, and we should know that this value is only comparable inside the same dataset.

To measure the execution time, we will measure the time it takes to predict each step during the rollout of a complete trajectory and the parameters that we are going to consider are the mean and the standard deviation that exhibits the variability of this parameter.

$$\bar{t} = \frac{\sum_{i=1}^{N} t_i}{N}$$

$$std = \sqrt{\frac{\sum_{i=1}^{N}(t_i - \bar{t})^2}{N}}$$

To measure the training time, we will only measure the mean of a few training steps because

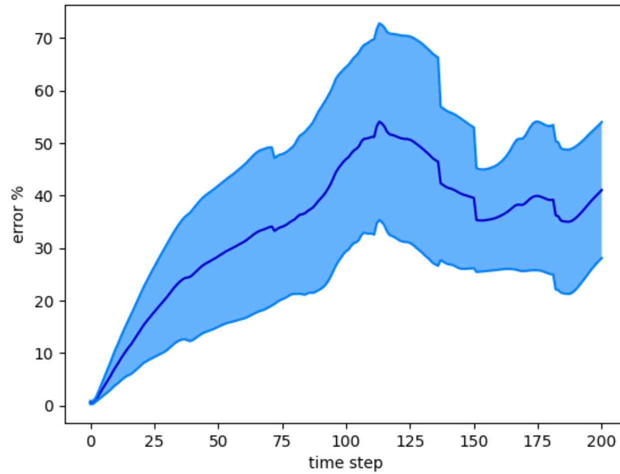


Figure 9: The error evolution plot has been obtained averaging 20 rollouts with a trained MoGPDM. The dark blue line is the mean relative error, and the blue shaded area is the 95% confidence interval computed with the standar deviation.

training some of the greatest models requires hours, and it would take days to complete the plot otherwise.

To keep coherence for the time performance, we will use the same hardware set up for all the tests. This is composed by a laptop with 32 GB of DDR4 RAM memory with 3200 MHz of clock speed, a mobile AMD Ryzen 9 5900HX CPU with 3,30 GHz of base clock speed and 4,80 GHz of max boost speed, and 8 cores with 16 threads.

The dataset that we are going to use to perform the experiments comes from a finite elements' simulation of a cloth developed at the Institut de Robòtica i Informàtica Industrial (IRII) [18]. It is a modified version of the dataset that the investigation group used to evaluate the CGPDM [1] and it is divided into four subsets with different range of movements $R \in \{30, 60, 90, 120\}$. In our dataset, the upper right corner always stays in the origin of coordinates, so the model only has to learn the deformations of the cloth and not the spatial position.

Once we have defined the variables that we want to compare, and the experimental framework, we can start to make the experiments and show some results.

4.1 MoCGPDM tuning

We have previously defined the training algorithm that we are going to use 3.2.3, but we have not specified yet some of the hyperparameters that we are going to employ like the number of training loops, the balance hyperparameters or the experts' optimization steps that we need to perform in each step of the algorithm. This is because those parameters cannot be optimally chosen for all the models, and therefore they are subject to be chosen empirically.

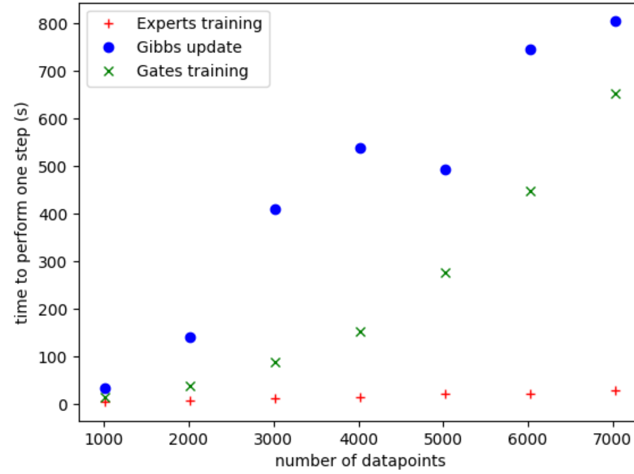


Figure 10: Here we can see how much time can it take to perform one training step of the different steps of the training algorithm.

4.1.1 Equilibrium betweeen experts training and Gibbs sampling of the indicators

Firstly, we need to balance between the optimization of the experts' parameters plus latent variables (second step of the algorithm) and the update of the experts' components (third step). The need to find that consideration comes from the high computational cost that the Gibbs update has compared to the computational cost of performing one step of parameters and latent variables' optimization (figure 10).

But more importantly, the balance is needed due to the fact that optimizing the latent variables with separated experts may result in the specialization of those latent variables to its experts. This means that if one latent point belongs to one expert, the optimization steps may drag it to another direction as it belonged to another expert, and then, the latent space could lose meaning as one joint latent space because each expert would have its own latent space representation. It entails the futility of performing the dynamics in a latent space and using a gating model to choose the experts that are going to predict both de dynamic and the mapping.

As the latent variable model is supposed to be continuous (proximal data point will also have proximal latent representation), one way to know if we have fallen into a non-representative latent space is by visually inspecting the different trajectories in the latent space (see figure 11). If there are many trajectories with a saw tooth shape, it means that consecutive observations (proximal data points) have very different representation in the latent space. To automate the inspection, we can also define some parameter that determines the "quality" of the trajectories based on the continuity of the curvature approximation of each point inside each trajectory computed, for example as

$$\kappa_i = \frac{2\sin\varphi_i}{\|x_{i+1} - x_{i-1}\|}$$

where φ_i is the angle between the segments $\overline{x_{i-1}x_i}$ and $\overline{x_i x_{i+1}}$.

To achieve this equilibrium, we need to employ several methods. One is to ensure that all the

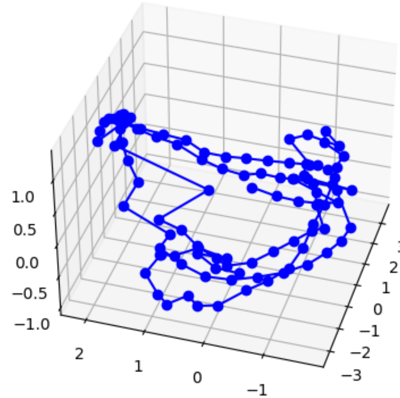


Figure 11: One smooth latent trajectory that has been bad trained fixing the points to different experts and overfitting them on purpose.

latent variables are free enough to keep moving between different experts and, therefore, averaging the gradient that comes from different experts keeping the meaning of the whole latent space. This means that we need to alternate between training the experts and Gibbs sampling the indicators with a reasonable ratio so the specialization of the latent variables does not occur.

Another important thing we have to be careful about is to train the latent variables with a small learning rate. This will avoid making big turns every time we update them inside one expert. As the LBFGS optimizer does not admit separate learning rates for different parameters, we will have to choose either training with a small learning rate or employing another optimizer, like Adam.

At first, we can expect a similar performance with the Adam and the LBFGS optimizer as both of them make second order estimations to search the training direction, but there are fundamental differences when we try to apply the optimizers to this the problem. The LBFGS will perform several training steps each time we call it as it needs to fill the memory with the gradient approximations to estimate the Hessian, and the learning rate is not modular, so it is mandatory to have small learning rates for all the parameters. In the other hand, the Adam optimizer makes small gradient updates each time we call it (faster, but each step is less relevant), and the flexibility of the learning rates will allow us to set more aggressive learning rates to the expert's parameters while we keep been conservative training the latent variables. Those characteristics of the Adam optimizer translate into a greater capacity keeping a higher rate between experts training steps and Gibbs updates without overfitting (the latent variables to the experts) using the Adam optimizer compared to the LBFGS.

To compare how the two optimizers behave, we are setting two model training loops with the same 10 sequences of training data. The first one with the Adam optimizer performing 50 training steps with $lr = 0.01$ for the parameters and $lr = 0.0001$ for the latent variables against each Gibbs updating. And the second model with the LBFGS optimizer performing only 2 training steps with a $lr = 0.0001$ for each Gibbs update. As we can see in figure 12, both of the optimizers have a saw tooth outline of the loss function during training. The descents happen during the experts training

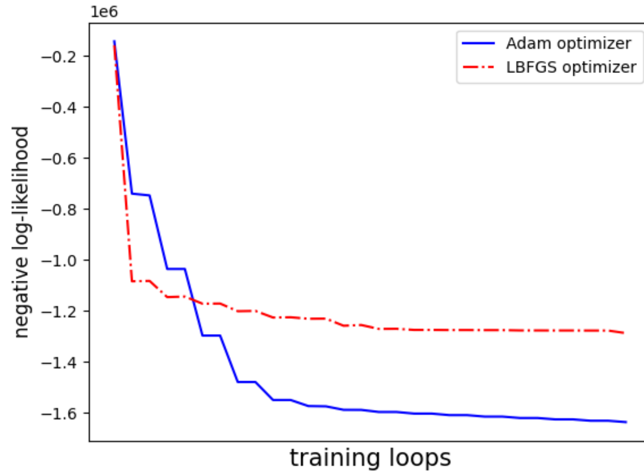


Figure 12: The evolution of the negative log-likelihood value during the training process comparing Adam and the LBFGS optimizer.

and the loss increments takes place when we Gibbs update the experts [1]. The minimum loss will be reached when the random loss increments during the Gibbs update and the ability of optimizers to decrease the loss with the given constrains reach an equilibrium. And, as we expected, Adam will outperform the LBFGS in the long term thanks to the number of steps that it is allowed to take. Note that in general, the LBFGS performs better with the same number of training steps (but with a worse computational cost) and usually can reach similar losses with less time if the number of variables is moderate. It was also the default optimizer that they used to train the CGPDM [3], and so the comparison between the optimizers.

In the figure 13 we can see a comparison of the same latent trajectory in both models. The represented trajectory in the Adam trained model is noticeably smoother, and the same happens with the rest of the trajectories of the training set. We have found that indeed, smoother trajectories tend to come with a better capacity of prediction (figure 14) as the experts will learn to reproduce a more consistent behavior in the latent dynamics instead of an erratically one.

Note that we can always make the learning rate smaller for the LBFGS optimizer and then be able to train more times at each loop step. There will be always ways to fine-tune the hyperparameters to make this training methods plausible, but it requires much more work than straight using the Adam optimizer. And in any case, the learning rates of the latent variables will always be bound to the learning rate of the latent variables. The only way to "modify" the learning rate using the LBFGS is by detaching the latent variables from the computational graph (see figure 8) at some steps, resulting in a slower alteration of the latent variables, but we are not going to explore this option during this thesis project.

[1]This might sound counterintuitive as we performed the sampling giving more probabilities to enhance the loss, but we also need to take into account the effect of the gating model and the randomness of the process.

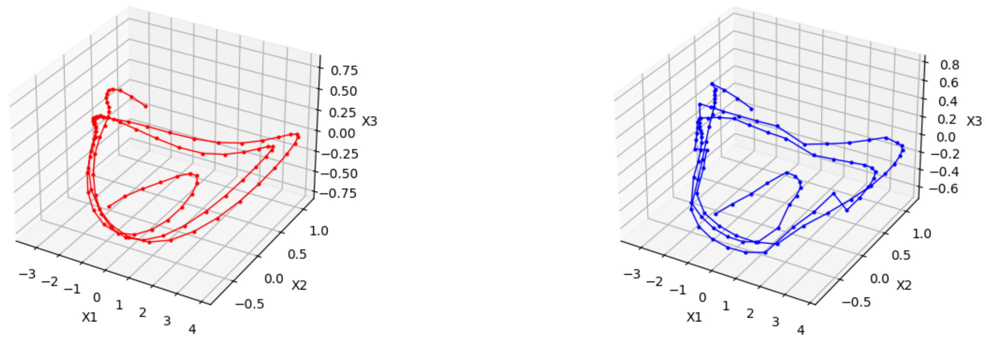


Figure 13: Representation of the three more relevant latent variables of the same trajectory of the training set of the two optimizers. On the left side, in blue, the one corresponding to the Adam optimizer, and on the right side, in red, the corresponding to the LBFGS optimizer.

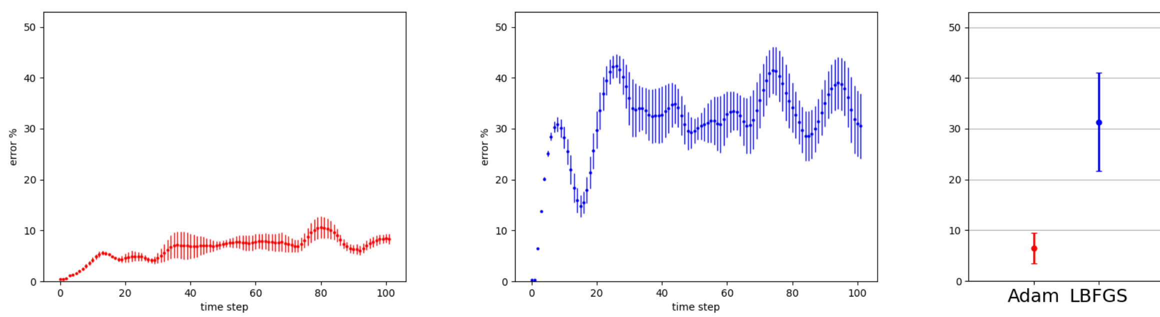


Figure 14: Here we can see the time evolution of the error with different trajectories. On the left side, in blue, the error committed by the model trained with Adam, in the middle the analogous by the model trained with the LBFGS, and on the right side, the mean error and the standard deviation of all the error committed by each model.

4.1.2 Balance hyperparameter

Another parameter we have to consider and evaluate the performance of is the balance hyperparameter. As we have seen before, this is introduced to give more relevance to the likelihood related to the latent dynamical mapping, as its negative log-likelihood is about a hundred times smaller. As the dynamical model is the one that will compound the error by iterating over the previously predicted steps, it seems pretty reasonable to give it more weight in order to shift the optimal result to a more suited to predict the dynamics.

Before performing the experiments to compare the behavior of models trained with different balance hyperparameters, we can make some guesses of how the models will change when we modify this. Firstly, we are going to assume that the experts' parameters are always going to reach the optimum for whatever latent variables that we throw in, so the main doubt will come from how we should train the latent variables and how much relevance should any of the experts have while training them.

What we can see about the log-likelihood is that the one corresponding to the mapping is about a hundred times greater than the one corresponding to the latent dynamics. This is because there are much more dimensions in the observable space than in the latent space. This magnitude discrepancy could turn into a better optimization of the mapping experts as most of the gradient will be driven to those experts instead of the latent experts. In this scenario, incrementing the balance hyperparameter could result in a great advantage.

In the other hand, the portion of the gradient of the latent variables that belongs to the latent experts usually behaves more erratically (small changes in the parameters may return very different gradients) due to the fact that it goes through both the input and the output of those experts. In addition, if we train the latent space to achieve a better mapping, we will have a more representative latent space, and therefore, smoother trajectories at least at first. Those could constitute a downside of incrementing the balance hyperparameter.

To make the experiments, we have chosen the same 10 training sequences of 102 observations each (a total of 1020 data points) for all the models. The maximum number of points for each expert is 450, to fit at least 4 sequences per expert and have at least 3 experts. And the training loop is designed with the Adam optimizer of the previous section. The balance hyperparameters that we are going to check with the experiments will be 0.1, 1, 10 and 100.

At first sight, we can see in the figure 15 that there is not a significant discrepancy in the loss evolution, but the model with the lower balance values seems to have discontinuous latent trajectories. This might indicate a worse performance for this model.

As we can see in the figure 16, neither incrementing nor decrementing the value of the balance hyperparameter will enhance the performance of the model (at least, the optimal balance will be of the same order of magnitude). In general terms, we can also guarantee the lower values of the balance tend to have better performance than greater values, so we can say that the arguments that favor a lower balance are more relevant than the others at the end.

If we want to know how the predictions' error would look, we can check the following link to a video where it is represented the movement of the predicted observable and the reference observable for the current four models. https://youtu.be/sDvdAyr4qWg

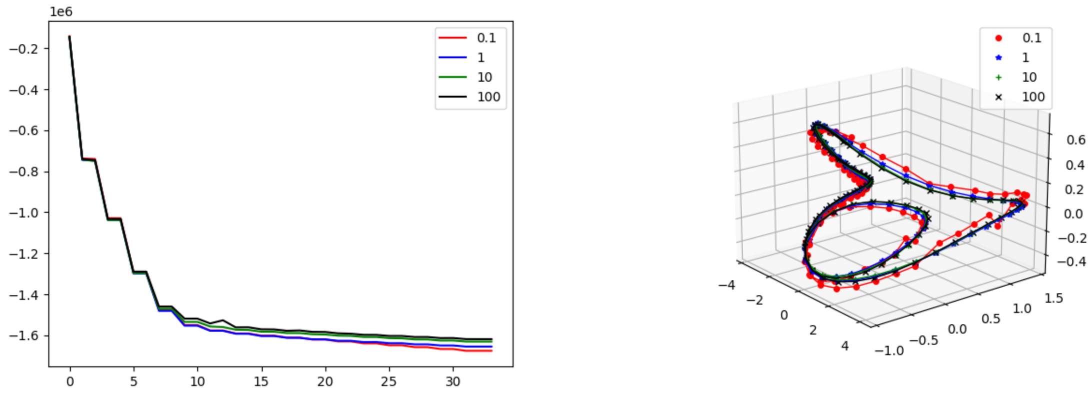


Figure 15: On the left side, we have the loss outline during the training, and at the right size, we have represented the same trajectory in the latent variables of the different models. The red color is assigned to the 0.1 balance parameter, blue for the balance = 1, yellow for the 10 and black for the 100.

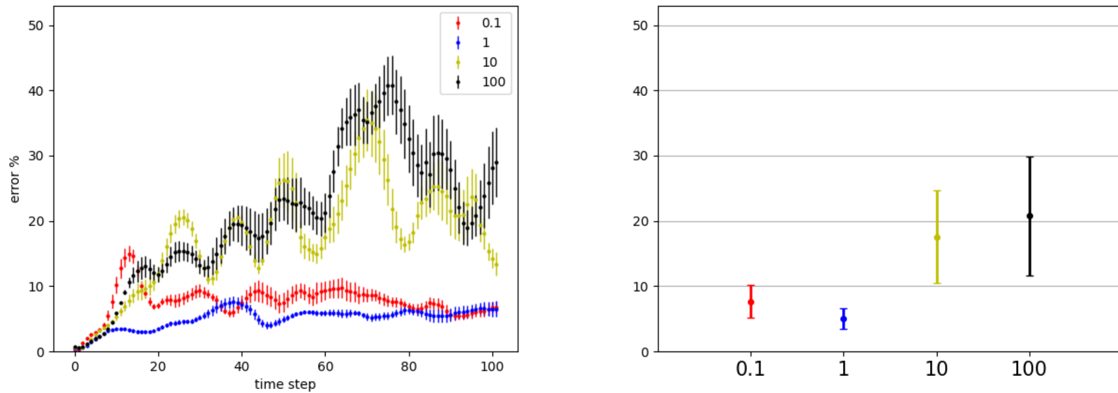


Figure 16: On the left side, we can see the relative error evolution along the time series of the four balance hyperparameter models. And on the right side, we have the relative error averaged along the time series with the 95% confidence interval.

4.1.3 Maximum size of the experts

This is the most difficult hyperparameter to tune, since we do not know *a priori* neither the number of data points that the model is going to have in total nor the shape of the latent variables. We can only expect certain behavior and then choose the N_{max} that we consider that it fits our needs.

In general, the predictions time performance will increase (it will take less time) if we choose smaller experts, as the predictions performed by each expert depends cubically on the number of points per expert and only linearly over the number of experts. If we suppose that we will always fill the experts up to their maximum number, we can make some computational cost approximations. If we have N data points, we will have $\lceil N/N_{max} \rceil$ experts, each one with cubical complexity at prediction time $\mathcal{O}(N/N_{max} \cdot N_{max}^3) = \mathcal{O}(NN_{max}^2)$, and the computation of the kernel function to each data point in the gating model will have linear complexity. This results in a combined complexity for the prediction algorithm that will depend on the number of points N and N_{max}

$$\mathcal{O}(NN_{max}^2) + \mathcal{O}(N)$$

As we reduce the maximum number of points that the experts can have, we expect a performance reduction. This is because each expert will dispose of fewer points to interpolate the output of some data, especially next to the borders of each expert, but this effect should also be counteracted by the gating model. We will see how the performance is hit by this hyperparameter.

The variation of this hyperparameter will also affect the training time. The expert's complexity will be given by the cubic scaling of each expert and linearly with the number of experts resulting in

$$\mathcal{O}(N/N_{max} \cdot N_{max}^3) = \mathcal{O}(NN_{max}^2)$$

similar to the prediction algorithm, but this time it will take more time as we need to also inverse the covariance matrix and not only multiply matrices.

During the Gibbs sampling, we loop along all the training set. For each point, we have to detach the point from the expert ($\mathcal{O}(N_{max}^2)$), and compute the total likelihood if the point belonged to each expert(assuming $\lceil N/N_{max} \rceil$ experts, the cost is $\mathcal{O}(N/N_{max} \cdot NN_{max}^2) = \mathcal{O}(N^2 N_{max})$ and finally reattach the point to an expert (also $\mathcal{O}(N_{max}^2)$). The resulting total cost will scale as

$$\mathcal{O}(N(2 * N_{max}^2 + N^2 N_{max})) = \mathcal{O}(N^2 N_{max}^2 + N^3 N_{max})$$

This is the most expensive step of the training algorithm and would also benefit of smaller N_{max}.

The gating training is done also by looping over all the training set indicators. We have to detach one indicator ($\mathcal{O}(N_{max}^2)$ and compute the gating weights for each expert ($\mathcal{O}(N)$) and the likelihood of each expert individually ($\mathcal{O}(NN_{max}^2)$. The resulting total cost complexity can be calculated as

$$\mathcal{O}(N(N_{max}^2 + NN_{max}^2)) = \mathcal{O}(NN_{max}^2 + N^2 N_{max}^2) = \mathcal{O}(N^2 N_{max}^2)$$

Knowing this, we are going to design the experiments by looking to how many experts we expect to have. With 10 sequences of the dataset of 102 points each, we will have 1020 points. The N_{max} values that we are going to try are $N_{max} = 110$ to expect 10 experts, $N_{max} = 230$ to expect 5 experts, $N_{max} = 450$ to have 3 experts and $N_{max} = 1100$ to have only one expert (this is the degenerated case).

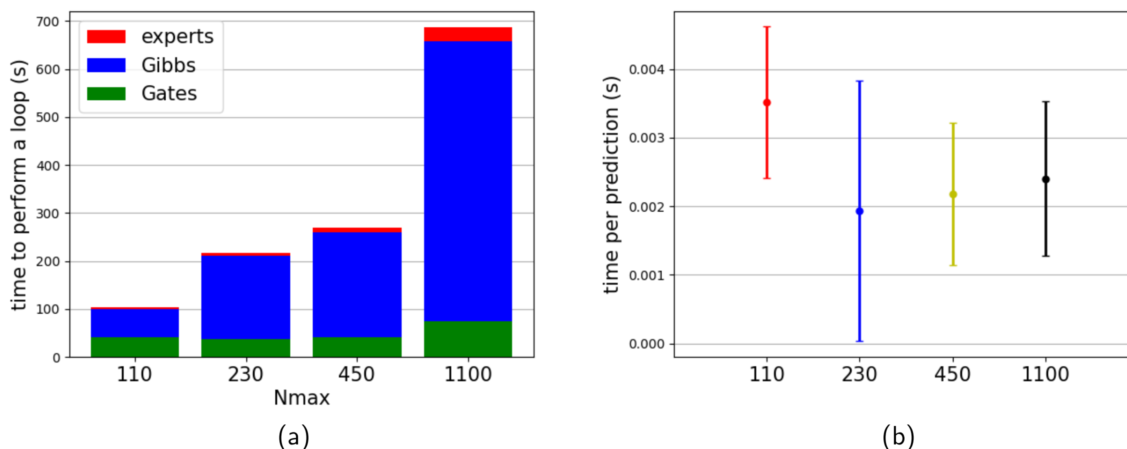


Figure 17: On the left side, we have portrayed the composition of the training time that each loop takes in function of the N_{max}. And on the right side, we have the time that it takes to perform a latent prediction for the model with the averaged prediction.

As we can see in the figure 17a, the greater the N_{max}, the more time we are going to need at each training loop. We can also notice how the composition changes as the experts' training time starts to grow and become noticeable as the experts' size grow.

In the figure 17b, the prediction time that the model with $N_{max} = 110$ was the greatest, which was unexpected and will be investigated as a future work, but the rest of the models followed the evolution we anticipated.

In the figures 18, we can recognize that the model with smaller experts made unexpectedly good predictions despite having the latent variables very fragmented. There is a possible explanation to this case, as the mapping model generally performs pretty well when the latent variables are close to the training data, the errors of the trajectory come from the compounding of latent dynamics predictions. In our training algorithm, the experts of the latent space are initialized with whole sequences of data, and in the $N_{max} = 110$ case, only one sequence can be filled to each latent expert. If the Gibbs sampling does not mix the experts, or only moves very few data points, the sequences of the experts will stay intact and there will be a specialization to reproduce the given sequences both in the gating model and the experts itself.

On the other hand, the rest of the models perform accordingly to what we anticipated, where the degenerated case was able to make the best predictions and the more fragmented the latent variables, the worse performance we can expect in general.

If we choose the N_{max} parameter in function of the number of points, we will have a reduction in the training and predicting time by a factor (the square of the number of expected experts in the predictions), but we will not obtain a complexity reduction. If we fix this parameter to a certain number, the prediction complexity will become linear (this is necessary to obtain "usable" dynamical models) with respect to the data points. But the training algorithms will keep been cubical unless we modify the Gibbs sampling algorithm.

As we have shown, there are many advantages and disadvantages of incrementing the size of the experts, as we need to train several models, we want to choose the parameters that allow us to train

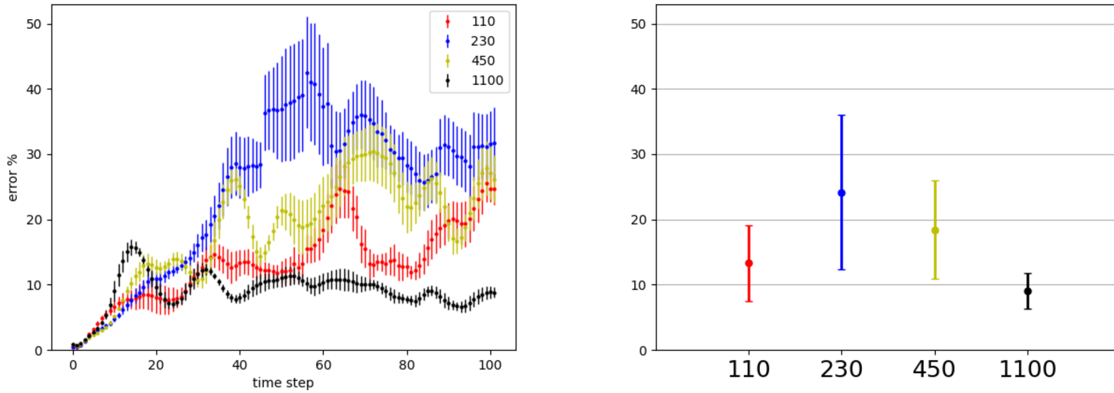


Figure 18: On the left side, we can see the relative error evolution along the time series of the four models. And on the right side, we have the relative error parameter averaged along the time series.

relatively fast but preserving as much performance as possible. To this end, we have chosen to use the parameter $\mathbf{N_{max}} = \mathbf{450}$ from now on, as we consider that is might be in a sweet point.

4.1.4 Predictions algorithm

In the 3.2.5, we have shown that there are two possible interpretations of how the gating model should average the predictions of the experts. The weighted averaging of independent Normals (from now on, we are going to call it the "averaging" prediction) and the joint likelihood of the distributions of a mixture of Gaussian process (from now on, "mixture" prediction). This latter one has to be sampled by first sampling the expert and then sampling the output using this one expert.

This not only entails two different possible results if we encounter difficult situations, as we have described before. But also has different computational complexity. The averaging prediction needs to compute the guesses of all the experts $\mathcal{O}(NN_{max}^2)$ and the gating $\mathcal{O}(N)$, but this last one is much faster despite having the same marginal complexity.

However, in the mixture predictions, we only need to compute the gating model $\mathcal{O}(N)$ and then sample one expert and calculate the prediction $\mathcal{O}(N_{max}^3)$.

Given that the number of points will always be larger than the largest expert (with more points associated), the mixture predictions will always be faster or at least as fast as the averaging predictions.

To check which one has better chances to perform better in general, we will train one model with 10 different sequences as we have done before and compare the committed error in the training sequences and in 10 similar test sequences.

In the figure 17b we can see that the average error is very similar between the two prediction techniques, but the average method tends to perform slightly better than the mixture of experts. This is because by averaging the predictions, we ensure that the error is going to be smaller than choosing a bad expert. We can say that the averaging predictions are more conservative and robust in the long term.

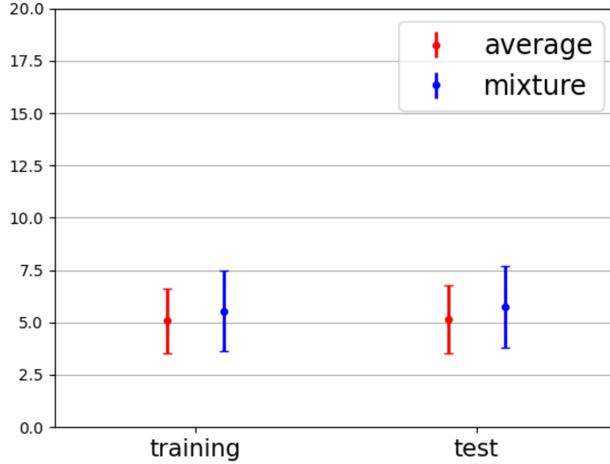


Figure 19: Average error committed by the model comparing the two interpretations of the prediction algorithm. The average time that the averaging prediction took to make one dynamical prediction is $t = 0.00206s$, and the mixture prediction needed $0.00084s$ to make each step.

4.2 Comparing CGPDM with the proposed MoCGPDM

Summarizing the results of the previous experiments, the best Mixture of Controlled Gaussian Process Dynamical Models we can get is with a balance hyperparameter equals one, as the original likelihood. With a maximum size of the experts of 450. The experts must be trained with the Adam optimizer with a learning rate $lr = 0.01$ for the experts' parameters (big enough to allow a fast adaptation to new data points) and a learning rate $lr = 0.0001$ for the latent variables (small enough, so we don't overfit the latent space to the experts). The Gating parameters should be pretrained at the beginning to avoid senseless Gibbs updates later on, and the optimizer must be a simple gradient descent, as higher order optimizers don't work well. The ratio of training will be 50 steps of the Adam optimizer, one Gibbs update and two Gates training steps at each training loop.

The size of the latent space will depend on the dataset we are using, in the cloth simulations dataset that we are using, $d = 5$ dimensions are enough to represent the observed data. And for the predictions, we are going to adopt the averaging method, as it has showed to perform slightly better.

The first thing that we can compare is the time required to train and evaluate the model. We have already described the computational complexity of the MoCGPDM, and the training of the CGPDM involves the computation of the log-likelihood. This implies the calculation of the covariance matrix, its inversion and its multiplication with other matrices, so the marginal complexity is cubical over all the training set $\mathcal{O}(N^3)$.

To make predictions, we have already accelerated the process by saving the inverse of the covariance matrix after the training loops (saving us the time needed to compute and inverse the covariance matrix), but it still involves the multiplication of $N \times N$ matrices, so the marginal complexity will also be cubical $\mathcal{O}(N^3)$. Note that there are algorithms with a better marginal complexity, but with the size of these matrices, they do not compensate the effort of implementing them and can even downgrade the overall performance.

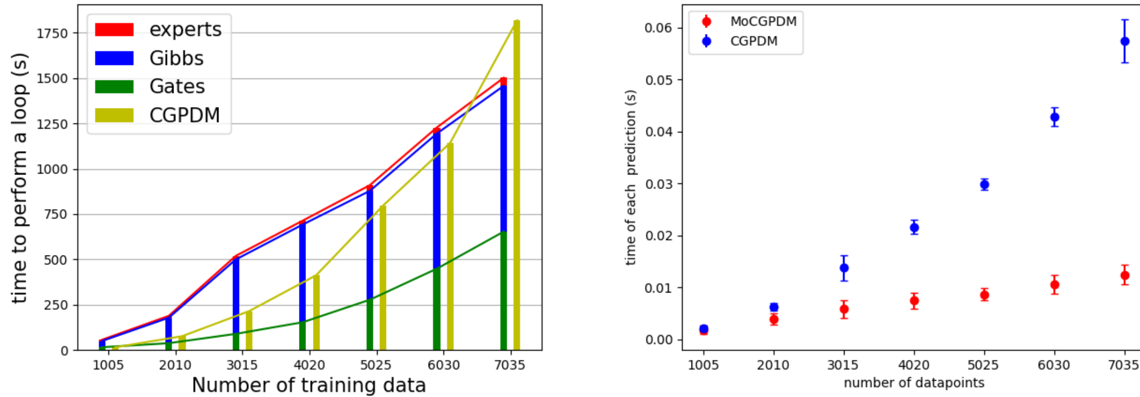


Figure 20: On the left side, the time we needed to perform one optimization loop as function of the number of data points. We are confronting the time needed to make one training step of the CGPDM against the time that the MoCGPDM takes to train the experts, Gibbs sample the experts' indicators and train the gating parameters stacked. On the right side, the average time needed to perform one prediction step by the two models as function of the dataset size.

As we can see in the figure 20, with a fixed N_{max}, the training of the Mixture will grow at a slower rate than the original CGPDM, and the predictions time will grow linearly for the Mixture model, against the cubical evolution of the CGPDM.

Note that we typically we will need more training loops for the CGPDM than for the MoCGPDM, this means that the needed time to train satisfactorily the GPDM will surpass the total time needed to train the Mixture much earlier.

From the performance perspective, we expect a reduction in the predicting accuracy as none of the experts dispose of the whole dataset to make the predictions, but this should be counteracted by the gating model and the more flexibility that having multiple experts parameters should bring us. This flexibility could even become a benefit in certain cases, where the there are places with a great density of points and other places where there are very sparse points. The CGPDM would need to try to equilibrate the lengths of the kernel, but the MoGPDM could handle it by introducing more experts, one with short length for the dense part and other with long lengths for the sparse space.

In the figure 21 we can see the results of both models in different situations and datasets. As we expected, in optimal conditions, the CGPDM will outperform the proposed model, but in situations with a lack of information (5 sequences for greater movement ranges) the averaging of the predictions can bring us better results than one single expert. The MoGPDM will also outperform the CGPDM in situations where there are very concentrated spaces (20 seq) where the redundancy of data points hurts the performance of this latter model.

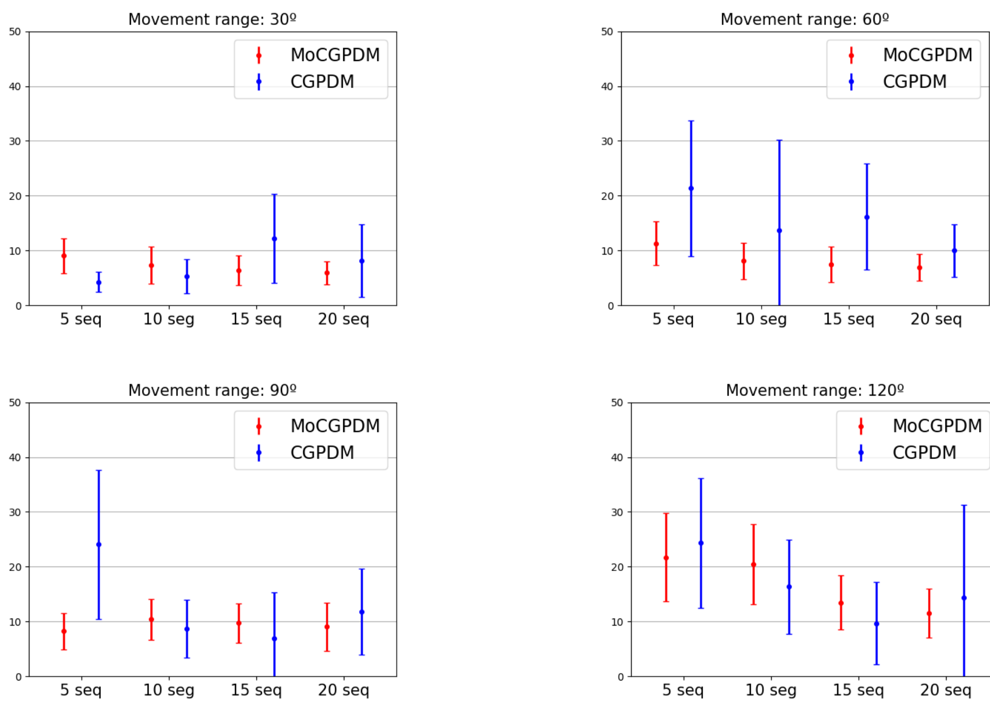


Figure 21: Mean relative errors (with 95% C.I.) obtained by different setups in the considered movement ranges. The red color is assigned to the results of the MoCGPDM and the blue results correspond to the CGPDM

5. Conclusions

The objective of this project was to improve the computational efficiency of one state-of-the-art model that was able to learn the dynamics of a deformable object only by feeding data to it.

To reach this goal, we have introduced the divide and conquer idea by implementing a mixture of experts with the special considerations that the Gaussian Processes requires to be considered as viable independent experts.

After shaping the mixture of Gaussian experts to model the data, we had to fine-tune the training algorithm and all the hyperparameters related with the log-likelihood to avoid the degeneration of the mixture of experts. This was the part that took most of the time, as it was not easy to find some proper combination of them that worked well.

This resulted in a usable dynamical model that doesn't need any previous knowledge of the system and is fully data driven. It was also capable to generalize the dynamics of high dimensional observations, like the mesh of points of a cloth under the manipulation of an agent that we are feeding to it.

This new model has much better computational performance than the previous CGPDM model, and is capable to make real time predictions even with a high number of trainings data points accomplishing the goal that we had set at the beginning of the project.

One consideration that we need to have, is that this model, as any Gaussian Process, is only capable to interpolate the data between training points. This means that is never going to reach the performance of a real simulation, no matter the number of sequences you feed him. But it is also able to make some predictions only by giving a few sequences to it if the range of motion is close to the given ones.

As future work, we will try to train the model in a much general environment, with no restrictions of movements (but always with the quasi-static manipulation assumption) and see how it performs.

We could also fit this model in a learning from demonstration application, where you show the robot how to move a cloth, and the machine has to be able to extract the needed data through a visual module, and then reproduce similar movements with a model-based control or set new objectives but near the trained dataset.

In anyone wants to see the code behind the model, you can visit the public GitHub repository MoGPDM that I made with the source code of the torch class object and some examples of usage.

References

[1] Fabio Amadio, Juan Antonio Delgado-Guerrero, Adrià Colomé and Carme Torras, *Controlled Gaussian Proccess Dynamical Model*, Arxiv:2103.06615, 2020

[2] Carl Edward Rasmussen and Zoubin Ghahramani, *Infinite Mixtures of Gaussian Process Experts*, Advances in Neural Information Processing Systems 14, MIT Press, 2002.

[3] J. M. Wang, A. Hertzmann, and D. J. Fleet, *Gaussian Process Dynamical Models*, Advances in neural information processing systems, vol. 18, pp. 1441–1448, 2005.

[4] N. Lawrence and A. Hyvärinen, *Probabilistic non-linear principal component analysis with gaussian process latent variable models.* Journal of machine learning research, vol. 6, no. 11, 2005.

[5] Cori Maklin *Gibbs Sampling. Yet another MCMC method.*

[6] C. E. Rasmussen, *The Infinite Gaussian Mixture Model, NIPS 12, S.A. Solla, T.K.Leen and K.-R. Müller (eds.), pp. 554–560, MIT Press*, 2000

[7] C.E. Rasmussen and C.K. I. Williams, *Gaussian Processes for Machine Learning*, the MIT Press, 2006

[8] C.E. Rasmussen, *Advanced Lectures on Machine Learning*, pp. 63–71. Springer Berlin Heidelberg, 2004

[9] Saeed Masoudnia and Reza Ebrahimpour, *Mixture of experts: a literature survey*. Springer Science+Business Media B.V. 2012

[10] D. MacKay, *Information Theory, Inference, and Learning Algorithms*, 2003

[11] R. M. Neal, *Bayesian Learning for Neural Networks*, Springer-Verlag, 1996

[12] J. M. Wang, D. J. Fleet, and A. Hertzmann, *Gaussian process dynamical models for human motion*, IEEE transactions on pattern analysis and machine intelligence, vol. 30, no. 2, pp. 283–298, 2007.

[13] Press, William H., Teukolsky, Saul A., Vetterling, William T. and Flannery, Brian P. *Numerical Recipes: The Art of Scientific Computing (3rd ed.)*, "Section 2.7.1 Sherman–Morrison Formula", New York: Cambridge University Press, 2007

[14] Diederik P. Kingma and Jimmy Lei Ba, *ADAM: A Method for Stochastic Optimization*, International Conference for Learning Representations, Board 11, May 9 2015

[15] D. C. Liu and J. Nocedal, *On the Limited Memory Method for Large Scale Optimization*, Mathematical Programming B, 45, 3, pp. 503-528, 1989

[16] Ping Li and Songcan Chen, *A review on Gaussian Process Latent Variable Models*, CAAI Transactions on Intelligence Technology, Volume 1, Issue 4, October 2016, Pages 366-376

[17] S. Geman and D. Geman, *Stochastic Relaxation, Gibbs Distributions, and the Bayesian Restoration of Images*, IEEE Transactions on Pattern Analysis and Machine Intelligence, 6, pages 721–741. (1989)

[18] Franco Coltraro, Jaume Amorós, Maria Alberich-Carramiñana and Carme Torras, *An Inextensible Model for Robotic Simulations of Textiles*, Arxiv:2103.09586, 2021