Learning unknown ODE models with Gaussian processes
SUPPLEMENTARY DOCUMENT

Markus Heinonen[1,2,*] MARKUS.O.HEINONEN@AALTO.FI

Çagatay Yıldız[1,*] CAGATAY.YILDIZ@AALTO.FI

Henrik Mannerström[1] HENRIK.MANNERSTROM@AALTO.FI

Jukka Intosalmi[1] JUKKA.INTOSALMI@AALTO.FI

Harri Lähdesmäki[1] HARRI.LAHDESMAKI@AALTO.FI

[1]Aalto University, Finland; [2]Helsinki Institute of Information Technology HIIT, Finland
[*]Joint first author

1. Sensitivity Equations

In the main text, the sensitivity equation is formulated using matrix notation

$$\dot{S}(t) = J(t)S(t) + R(t). \tag{1}$$

Here, the time-dependent matrices are obtained by differentiating the vector valued functions with respect to vectors i.e.

$$S(t) = \begin{bmatrix} \frac{dx_1(t,U)}{du_1} & \frac{dx_1(t,U)}{du_2} & \dots & \frac{dx_1(t,U)}{du_{MD}} \\ \frac{dx_2(t,U)}{du_1} & \frac{dx_2(t,U)}{du_2} & \dots & \frac{dx_2(t,U)}{du_{MD}} \\ \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots \\ \frac{dx_D(t,U)}{du_1} & \frac{dx_D(t,U)}{du_2} & \dots & \frac{dx_D(t,U)}{du_{MD}} \end{bmatrix}_{D\times MD} \tag{2}$$

$$J(t) = \begin{bmatrix} \frac{\partial f(\mathbf{x}(t),U)_1}{\partial x_1} & \frac{\partial f(\mathbf{x}(t),U)_1}{\partial x_2} & \dots & \frac{\partial f(\mathbf{x}(t),U)_1}{\partial x_D} \\ \frac{\partial f(\mathbf{x}(t),U)_2}{\partial x_1} & \frac{\partial f(\mathbf{x}(t),U)_2}{\partial x_2} & \dots & \frac{\partial f(\mathbf{x}(t),U)_2}{\partial x_D} \\ \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots \\ \frac{\partial f(\mathbf{x}(t),U)_D}{\partial x_1} & \frac{\partial f(\mathbf{x}(t),U)_D}{\partial x_2} & \dots & \frac{\partial f(\mathbf{x}(t),U)_D}{\partial x_D} \end{bmatrix}_{D\times D} \tag{3}$$

$$R(t) = \begin{bmatrix} \frac{\partial f(\mathbf{x}(t),U)_1}{\partial u_1} & \frac{\partial f(\mathbf{x}(t),U)_1}{\partial u_2} & \dots & \frac{\partial f(\mathbf{x}(t),U)_1}{\partial u_{MD}} \\ \frac{\partial f(\mathbf{x}(t),U)_2}{\partial u_1} & \frac{\partial f(\mathbf{x}(t),U)_2}{\partial u_2} & \dots & \frac{\partial f(\mathbf{x}(t),U)_2}{\partial u_{MD}} \\ \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots \\ \frac{\partial f(\mathbf{x}(t),U)_D}{\partial u_1} & \frac{\partial f(\mathbf{x}(t),U)_D}{\partial u_2} & \dots & \frac{\partial f(\mathbf{x}(t),U)_D}{\partial u_{MD}} \end{bmatrix}_{D\times MD} \tag{4}$$

2. Optimization

Below is the explicit form of the log posterior. Note that we introduce $\mathbf{u} = vec(U)$ and $\Omega = \text{diag}(\omega_1^2, \dots, \omega_D^2)$ for notational simplicity.

$$\log \mathcal{L} = \log p(U|\boldsymbol{\theta}) + \log p(Y|\mathbf{x}_0, U, \boldsymbol{\omega}) \tag{5}$$

$$= \log \mathcal{N}(\mathbf{u}|\mathbf{0}, \mathbf{K}_{\boldsymbol{\theta}}(Z,Z)) + \sum_{i=1}^{N} \log \mathcal{N}(\mathbf{y}_i|\mathbf{x}(t_i, U), \Omega) \tag{6}$$

$$= -\frac{1}{2}\mathbf{u}^T \mathbf{K}_{\boldsymbol{\theta}}(Z,Z)^{-1}\mathbf{u} - \frac{1}{2}\log|\mathbf{K}_{\boldsymbol{\theta}}(Z,Z)| - \frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{D}\frac{(y_{ij} - x_j(t_i, U, \mathbf{x}_0))^2}{\omega_j^2} - \sum_{i=1}^{N}\frac{1}{2}\log|\Omega| \tag{7}$$

$$= -\frac{1}{2}\mathbf{u}^T \mathbf{K}_{\boldsymbol{\theta}}(Z,Z)^{-1}\mathbf{u} - \frac{1}{2}\log|\mathbf{K}_{\boldsymbol{\theta}}(Z,Z)| - \frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{D}\frac{(y_{i,j} - x_j(t_i, U, \mathbf{x}_0))^2}{\omega_j^2} - N\sum_{j=1}^{D}\log \omega_j \tag{8}$$

Our goal is to compute the gradients with respect to the initial state $\mathbf{x}_0$, latent vector field $\widetilde{U}$, kernel parameters $\boldsymbol{\theta}$ and noise variables $\boldsymbol{\omega}$. As explained in the paper, we compute the gradient of the posterior with respect to inducing vectors U and project them to the white domain thanks to noncentral parameterisation. The analytical forms of the partial derivatives are as follows:

$$\frac{\partial \log \mathcal{L}}{\partial u_k} = \sum_{i=1}^{N}\sum_{j=1}^{D}\frac{y_{i,j} - x_j(t_i, U, \mathbf{x}_0)}{\omega_j^2}\frac{\partial x_j(t_i, U, \mathbf{x}_0)}{\partial u_k} - \mathbf{K}_{\boldsymbol{\theta}}(Z,Z)^{-1}\mathbf{u} \tag{9}$$

$$\frac{\partial \log \mathcal{L}}{\partial (x_0)_d} = \sum_{i=1}^{N}\sum_{j=1}^{D}\frac{y_{i,j} - x_j(t_i, U, \mathbf{x}_0)}{\omega_j^2}\frac{\partial x_j(t_i, U, \mathbf{x}_0)}{\partial (x_0)_d} \tag{10}$$

$$\frac{\partial \log \mathcal{L}}{\partial \omega_j} = \frac{1}{\omega_j^3}\sum_{i=1}^{N}(y_{i,j} - x_j(t_i, U, \mathbf{x}_0))^2 - \frac{N}{\omega_j} \tag{11}$$

Seemingly hard to compute terms, $\frac{\partial x_j(t_i, U, \mathbf{x}_0)}{\partial u_k}$ and $\frac{\partial x_j(t_i, U, \mathbf{x}_0)}{\partial (x_0)_d}$, are computed using sensitivities. The lengthscale parameter $\boldsymbol{\ell}$ is considered as a model complexity parameter and is chosen from a grid using cross-validation. We furthermore need the gradient with respect to the other kernel variable, i.e., the signal variance σ_f^2. Because $\mathbf{K}_{\boldsymbol{\theta}}(Z,Z)$ and $\mathbf{x}(t_i, U)$ are the functions of kernel, computing the gradients with respect to σ_f^2 is not trivial and we make use of finite differences:

$$\frac{\partial \log \mathcal{L}}{\partial \sigma_f} = \frac{\log \mathcal{L}(\sigma_f + \delta) - \log \mathcal{L}(\sigma_f)}{\delta} \tag{12}$$

We use $\delta = 10^{-4}$ to compute the finite differences.

One problem of using gradient-based optimization techniques is that they do not ensure the positivity of the parameters being optimized. Therefore, we perform the optimization of the noise standard deviations $\boldsymbol{\omega} = (\omega_1, \ldots, \omega_D)$ and signal variance σ_f with respect to their logarithms:

$$\frac{\partial \log \mathcal{L}}{\partial \log c} = \frac{\partial \log \mathcal{L}}{\partial c}\frac{\partial c}{\partial \log c} = \frac{\partial \log \mathcal{L}}{\partial c}c \tag{13}$$

where $c \in (\sigma_f, \boldsymbol{\omega})$. The training algorithm is given in Algorithm 1.

3. Implementation Details

We initialise the inducing vectors $U = (\mathbf{u}_1, \ldots \mathbf{u}_M)$ by computing the empirical gradients $\dot{\mathbf{y}}_i = \mathbf{y}_i - \mathbf{y}_{i-1}$, and conditioning as

$$U_0 = \mathbf{K}(Z,Y)\mathbf{K}(Y,Y)^{-1}c\dot{\mathbf{y}}, \tag{14}$$

where we optimize the scale c against the posterior. The whitened inducing vector is obtained as $\widetilde{U}_0 = \mathbf{L}_{\boldsymbol{\theta}}^{-1}U_0$. This procedure produces initial vector fields that partially match the trajectory already. We then do 100 restarts of the optimization from random perturbations $\widetilde{U} = \widetilde{U}_0 + \varepsilon$.

Algorithm 1: NPODE training algorithm

Initialize $U, Z, \mathbf{x}_0, \boldsymbol{\theta}, \boldsymbol{\omega}$
Compute $\mathbf{K}_{\boldsymbol{\theta}}(Z, Z)$ and $\mathbf{L}_{\boldsymbol{\theta}}$
while *not converged* **do**
  Integrate the system to compute the path $\mathbf{x}(t)$ and the sensitivities $S(t)$
  Compute the gradients by (9), (10), (11), (12) and (13),
  Apply the noncenteral parameterization trick: $\nabla_{\widetilde{U}} \log \mathcal{L} = \mathbf{L}_{\boldsymbol{\theta}}^T \nabla_U \log \mathcal{L}$
  Update $\mathbf{x}_0, \widetilde{U}, \sigma_f, \boldsymbol{\omega}$ based on L-BFGS update rule
  Set $U = \mathbf{L}_{\boldsymbol{\theta}} \widetilde{U}$

We use L-BFGS gradient optimization routine in Matlab. We initialise the inducing vector locations Z on a equidistant fixed grid on a box containing the observed points. We select the lengthscales $\ell_1, \ldots, \ell_D$ using cross-validation from values $\{0.5, 0.75, 1, 1.25, 1.5\}$. In general large lengthscales induce smoother models, while lower lengthscales cause overfitting.