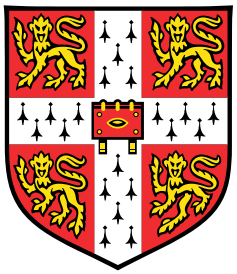
Scalable Gaussian Process Factor Analysis



Jasmine Talia Stone

Supervisor: Dr. Guillaume Hennequin

Advisor: Prof. Máté Lengyel

Department of Engineering
University of Cambridge

This thesis is submitted for the degree of
Master of Philosophy

Churchill College October 2021

This thesis is dedicated to my wonderful partner who stayed by my (zoom) side through stretches of up to 7.5 months apart, to my loving parents without whom I would not be here, and to my housemates who were for many months the only people I saw, and my lifeline.

Declaration

I hereby declare that except where specific reference is made to the work of others, the contents of this dissertation are original and have not been submitted in whole or in part for consideration for any other degree or qualification in this, or any other university. This dissertation is the result my own work and contains nothing which is the outcome of work done in collaboration with others, except where specifically indicated in the text and Acknowledgements. This dissertation contains fewer than 15,000 words excluding appendices, bibliography, and footnotes.

Jasmine Talia Stone
October 2021

Acknowledgements

First and foremost, I thank my supervisor, Guillaume Hennequin for his support and guidance throughout this MPhil. I could not have asked for a more attentive and supportive supervisor, and I am very thankful for your generosity with your time throughout, helpful edits on this thesis, and your patience with me as I learned about Gaussian Processes.

I am grateful for Kris Jensen who has been an invaluable friend and sounding board since we started collaborating, and who helped me figure out how to make better use of compute resources here. I am also grateful for Marine Schimel who was one of the first to welcome me to Cambridge, and who, along with Kris, made every effort to include me if and when any socializing could happen. I thank the rest of the Hennequin lab, and the entire Computational and Biological Learning Group for interesting discussions and talks throughout. I especially thank those of you who organized social events where I was able to meet some of you for the first time at the end of this year.

I would particularly like to thank John Bronskill for generously lending me his monitor, keyboard and mouse for the duration of the year, making those long winter lockdown months, and all the other days I was not able to work in the department at least less taxing ergonomically. While you may insist it was not a big deal for you, it certainly made all the difference for me.

I am grateful to the Churchill Foundation which gave me the opportunity to pursue this research supported by a Churchill Scholarship, without which I would not be here. I am also thankful to Max Shinn who helped me in understanding the application process for the Churchill Scholarship, and encouraged me to apply, and who developed the figure software, CanD, that I use to layout my figures in this thesis. I thank both Max Shinn and Daniel Ehrlich for valuable discussions throughout.

Thank you to Macy Vollbrecht for sitting at the kitchen table with me when I tired of working on my own in my room and for proofreading this acknowledgements section and the dedication.

I would like to acknowledge that Chapter 4, Appendix B, the first paragraph of Section 1.1, and a clause in the first paragraph of Section 1.2 ("and has yielded insights into neural computations...") are adapted from a paper submision by Kristopher T. Jensen*, Ta-Chu Kao*,

myself, and Guillaume Hennequin (*=equal contribution) [34]. The respective contributions to that paper submission are as follows: K.T.J, T.C.K. and G.H. were responsible for project ideation. K.T.J., T.C.K., G.H. and myself were responsible for ideas for analyzing the models run on experimental data. K.T.J. and T.C.K. drafted the text and the derivations respectively. K.T.J. and T.C.K. wrote figure code, and I edited the figure code. The codebase was primarily built by K.T.J. and T.C.K. I contributed the Toeplitz speed-up to the codebase and the GPFA model (run in a different codebase). K.T.J. and I were responsible for fitting the models. G.H., K.T.J., T.C.K. and I edited the manuscript. For that paper submission we are grateful to O'Doherty et al. [52] for making their data publicly available and to Marine Schimel and David Liu for insightful discussions. We thank Marine Schimel, Yashar Ahmadian, Peter Stone, and Jonathan So for helpful comments on the manuscript. We thank David Liu for contributions to the codebase used for our analyses. K.T.J. was funded by a Gates Cambridge scholarship.

I also note that Figure 1.1 is reused from [78] with permission from The American Physiological Society (license number 5087810403273).

Abstract

Recordings of neural activity are key in our quest to understand one of the great mysteries of our time, the brain. While in previous decades experiments recording from single to tens of neurons *a la* Hubel and Wiesel yielded insights at the single-neuron level using trial-averaged analyses, recent advances in technology have enabled simultaneous recordings of up to tens of thousands of neurons over the course of hours or days. These larger datasets provide neuroscientists with new opportunities to study large populations, or even multiple brain areas, together as a whole. For example, neurons that were previously disregarded as noisy or irrelevant due to confusing response properties are now considered important parts of the whole. Additionally, buoyed by computational advances, neuroscientists have begun analyzing single-trial recordings in addition to trial-averaged ones. Single-trial analyses allow neuroscientists to investigate trial-to-trial variability (possibly resulting from changes in attention or mental state) and are necessary when analyzing continuous recordings without well-defined trial structure.

To make sense of this increasingly high-dimensional data, neuroscientists have turned to a variety of supervised and unsupervised dimensionality reduction techniques. One such unsupervised technique is Gaussian Process Factor Analysis (GPFA). GPFA is an interpretable method that outperforms other similarly interpretable techniques (such as principal component analysis and independent component analysis) in modeling spatial and temporal correlations in population recordings. However, its applications have so far been limited mostly due to its prohibitive computational complexity. In particular, it does not scale well to datasets with more than ~ 200 time bins. One way of circumventing this issue is to break trial-structured recordings into shorter constituent trials. However, this ad-hoc strategy limits the types of representations that GPFA can recover, and is not naturally applicable to continuous recordings that do not have a well-defined trial structure.

In this thesis I develop two GPFA implementations with highly tractable time and space complexities that scale near-linearly with the number of time-points. With these methods I demonstrate applicability of GPFA to single stretches of data with upward of 10^5 time bins (10^7 datapoints in total), much larger than previous attempts that were limited to 10^2 time

bins (10^4 datapoints). I compare and demonstrate applications of these methods, and discuss future directions.

Table of contents

List of figures

List of tables

List of code examples

List of Algorithms

Chapter 1

Introduction

1.1 Motivation

The adult human brain contains upwards of 100 billion neurons so its dynamical state space is of proportionally immense dimensionality [4]. Yet many of our day-to-day behaviors such as navigation, motor control and decision making can be described in much lower dimensional subspaces. Accordingly, recent studies across a range of cognitive and motor tasks have shown that neural population activity can often be accurately summarized by the dynamics of a "latent state" evolving in a low-dimensional space [12, 53, 11, 48, 21]. Inferring and investigating these latent processes can therefore help us understand the underlying representations and computations implemented by the brain [31]. To this end, numerous latent variable models have been developed and used to analyze the activity of populations of simultaneously recorded neurons. These models range from simple linear projections in the form of principal component analysis (PCA) to sophisticated non-linear models using modern machine learning techniques [33, 53, 24, 13].

A key goal in dimensionality reduction of population datasets is to infer the collective timecourse of a set of latent processes from observed neural activity. Typical neural population recordings consist of some measure of activity recorded over time in several hundred neurons. Such activity can consist of spike times, voltage traces, or fluctuations in a calcium indicator. In the case of spike times or voltages, preprocessing typically involves binning action potentials with a given bin width, yielding integer spike counts.

Importantly, the continuous nature of behavior and cognitive processes suggests that interpretable latent processes should be smooth in time. Neuroscientists have thus historically first smoothed the binned data on ad-hoc timescales before secondly applying a dimensionality reduction technique (such as PCA, ICA, etc.) to the smoothed data [78]. This allows information to flow across time (e.g. a spike at time t is expected to carry some information

about the value of a latent process at time $t \pm \Delta$ for some range of Δ) and across neurons (a given latent process is typically reflected in the activity of more than one neuron) to yield more accurate, denoised inferences.

1.2 Utility of Gaussian Process Factor Analysis (GPFA)

Gaussian Process Factor Analysis (GPFA) is a dimensionality reduction technique that combines these two steps, simultaneously smoothing and finding the lower dimensional subspace that best encodes the neural activity (Figure 1.1; 79, 78). Combining these steps enables direct learning of inherent dimensionality and the best smoothing parameters from the data itself [78]. GPFA thus outperforms traditional two-step methods, and has yielded insights into neural computations ranging from time tracking to movement preparation and execution [53, 38, 79, 78, 1, 63, 61, 60]. Because GPFA is probabilistic, it allows for extensions incorporating prior knowledge via specific choices of kernels: for example, smoothness of the latent variables can be expressed using a squared exponential kernel, oscillatory dynamics using a spectral mixture or cosine kernel, and dynamical structure using a non-reversible kernel [54, 60].

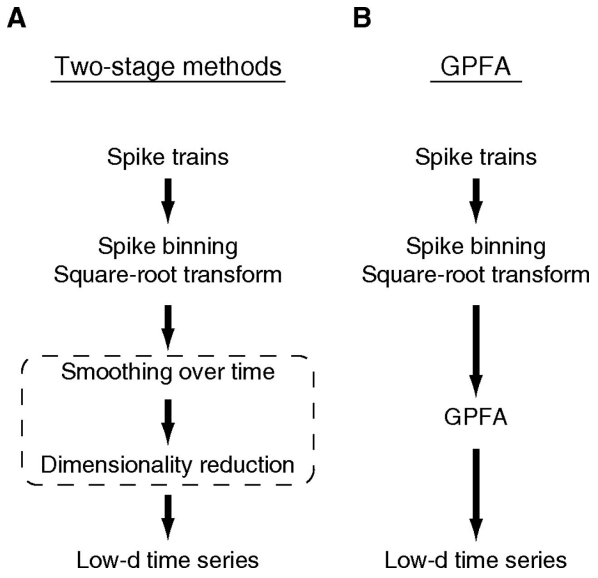


Fig. 1.1 **Two stage dimensionality reduction techniques vs. GPFA.** This figure is reused from [78] with permission from The American Physiological Society. **(A)** Two stage methods (e.g. PCA, ICA) perform smoothing and dimensionality reduction in two sequential steps. **(B)** GPFA performs smoothing and dimensionality reduction simultaneously, allowing the parameters for both steps to be jointly optimized.

Other methods exist for performing simultaneous dimensionality reduction and smoothing. Latent Factor Analysis via Dynamical Systems (LFADS) in particular is a deep network method that can currently outperform GPFA when trying to reconstruct aspects of behavior from the inferred latents [64]. Poisson Linear Dynamical System (PLDS) is a non-kernel and non-deep method that in certain cases can outperform GPFA when predicting spike-rates [46]. LFADS and PLDS both have many more parameters than GPFA models, and so are both more data-hungry and less interpretable [64]. Interpretability has certain benefits (e.g. GPFA assigns each latent a timescale—shorter timescales can correspond to within-trial variations, whereas longer timescales can correspond to slower across-trial drifts reflecting e.g. task engagement). Moreover, explicitly probabilistic methods enable principled quantitative comparison of models with different numbers of latents and different numbers of parameters.

1.3 Mathematical background

1.3.1 Gaussian Processes (GPs)

Scalar Gaussian Processes (GPs) generatively model how output $y \in \mathbb{R}$ varies as a function of some input $t \in \mathbb{R}$:

$$f(\cdot) \sim \mathcal{GP}(\mathbf{0}, k(\cdot,\cdot)) \tag{1.1}$$

$$\boldsymbol{y} = f(\boldsymbol{t}) \sim \mathcal{N}(\mathbf{0}, \boldsymbol{\Sigma}) \text{ where } \Sigma_{ij} = k(t_i, t_j) \tag{1.2}$$

where $\boldsymbol{t} = \{t_1, \ldots, t_T\}$ denotes a set of T input points and $\boldsymbol{y} = \{y_1, \ldots, y_T\}$ denotes the corresponding set of T output points. The function $k(t,t')$, is called the kernel function. There are many possible kernel functions – a good overview is provided by [74, 20]. For the purposes of this thesis, I will only consider stationary kernels, a family of kernel functions $k(t,t')$ that depends only on the difference $t - t'$ between the input points. The squared exponential kernel, also referred to as the Radial Basis Function (RBF) kernel is defined as $k(t,t') = \exp\left(-\frac{(t-t')^2}{2\ell^2}\right)$, where ℓ is a characteristic lengthscale related to the average interval between two zero crossings.

In neuroscience applications, the input dimension is time, and time points (and thus input points) are equally spaced due to spike binning. As a result of this equal spacing and the kernel stationarity, $\boldsymbol{\Sigma}$ has equal values along its diagonals. Thus, $\boldsymbol{\Sigma}$ in Equation 1.2 is a symmetric positive definite matrix with Toeplitz structure (Figure 1.2A).

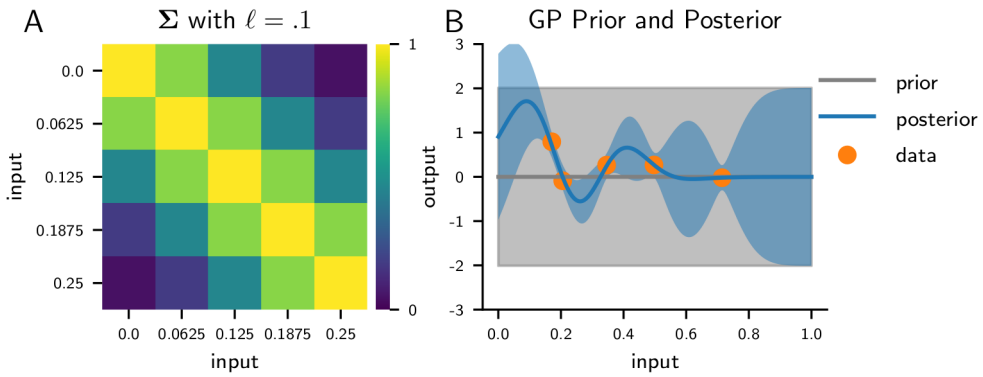


Fig. 1.2 **Gaussian Processes. (A) $\boldsymbol{\Sigma}$** corresponding to the stationary RBF kernel evaluated at a set of $T = 5$ equally spaced input points. **(B)** The Gaussian process prior (gray) encoded by the kernel in A, along with data points (orange) that induce the posterior (blue) (Equations 1.4 and 1.5). Shading shows ± 2 standard deviations from the mean.

Gaussian processes for regression

Gaussian processes can be used in a regression context, to model latent functions which are observed noisily. Specifically, consider a function $f(t)$ observed at a set of T input points $\boldsymbol{t}$ with added Gaussian noise, resulting in data of the form $y_i = f(t_i) + \sigma_n \varepsilon_i$ where $\varepsilon \sim \mathcal{N}(0,1)$. By placing a GP prior (Figure 1.2B) over f, one is able to approach this regression problem in a Bayesian manner. Gaussian processes have the nice property that conditioning can be performed analytically. Specifically, one can infer the values $\boldsymbol{f}_*$ of the underlying function for any set of input points $\boldsymbol{t}_*$ (which may or may not overlap with $\boldsymbol{t}$) as follows. We begin by extending Equation 1.2 to express the joint distribution of $\boldsymbol{y}$ and $\boldsymbol{f}_\star$ as

$$\begin{bmatrix} \boldsymbol{y} \\ \boldsymbol{f}_* \end{bmatrix} \sim \mathcal{N}\left(\boldsymbol{0}, \begin{bmatrix} K(\boldsymbol{t},\boldsymbol{t}) + \sigma_n^2 \boldsymbol{I} & K(\boldsymbol{t},\boldsymbol{t}_*) \\ K(\boldsymbol{t}_*,\boldsymbol{t}) & K(\boldsymbol{t}_*,\boldsymbol{t}_*) \end{bmatrix}\right) \tag{1.3}$$

where the $\sigma_n^2 \boldsymbol{I}$ term added to the kernel matrix $K(\boldsymbol{t},\boldsymbol{t})_{ij} = k(t_i,t_j)$ accounts for the observation noise. Using the standard formula for conditioning in multivariate Gaussian distributions, one can find the posterior mean and covariance of $\boldsymbol{f}_*$ (Figure 1.2B):

$$\mathbb{E}[\boldsymbol{f}_* | \boldsymbol{t}, \boldsymbol{y}, \boldsymbol{t}_*] = K(\boldsymbol{t}_*,\boldsymbol{t})[K(\boldsymbol{t},\boldsymbol{t}) + \sigma_n^2 \boldsymbol{I}]^{-1} \boldsymbol{y} \tag{1.4}$$

$$\operatorname{cov}(\boldsymbol{f}_*) = K(\boldsymbol{t}_*,\boldsymbol{t}_*) - K(\boldsymbol{t}_*,\boldsymbol{t})[K(\boldsymbol{t},\boldsymbol{t}) + \sigma_n^2 \boldsymbol{I}]^{-1} K(\boldsymbol{t},\boldsymbol{t}_*) \tag{1.5}$$

Fitting

Fitting a GP involves optimizing all parameters of the generative model, such as those of the kernel (e.g. the lengthscale ℓ above) and the observation noise σ_n^2, to maximize the marginal likelihood for the data at hand:

$$\log p(\boldsymbol{y}|\boldsymbol{t}) = -\frac{1}{2}\boldsymbol{y}^T(\boldsymbol{K}+\sigma_n^2\boldsymbol{I})^{-1}\boldsymbol{y} - \frac{1}{2}\log|\boldsymbol{K}+\sigma_n^2\boldsymbol{I}| - \frac{T}{2}\log 2\pi \tag{1.6}$$

where $\boldsymbol{K} = K(\boldsymbol{t},\boldsymbol{t})$. Here, $p(\boldsymbol{y}|\boldsymbol{t})$ is called the "marginal likelihood" because it requires marginalizing out the latent function f under its prior.

1.3.2 GPFA

Gaussian Process Factor Analysis (GPFA; 79, 78) is a generalization of the GP regression setup introduced above that models the spatio-temporal structure of a function with more than one output dimension (Figure 1.3). Where classical GP regression as presented above models a scalar output function (e.g. a scalar time series), GPFA instead models a higher-dimensional output $\boldsymbol{y}(t) \in \mathbb{R}^N$.

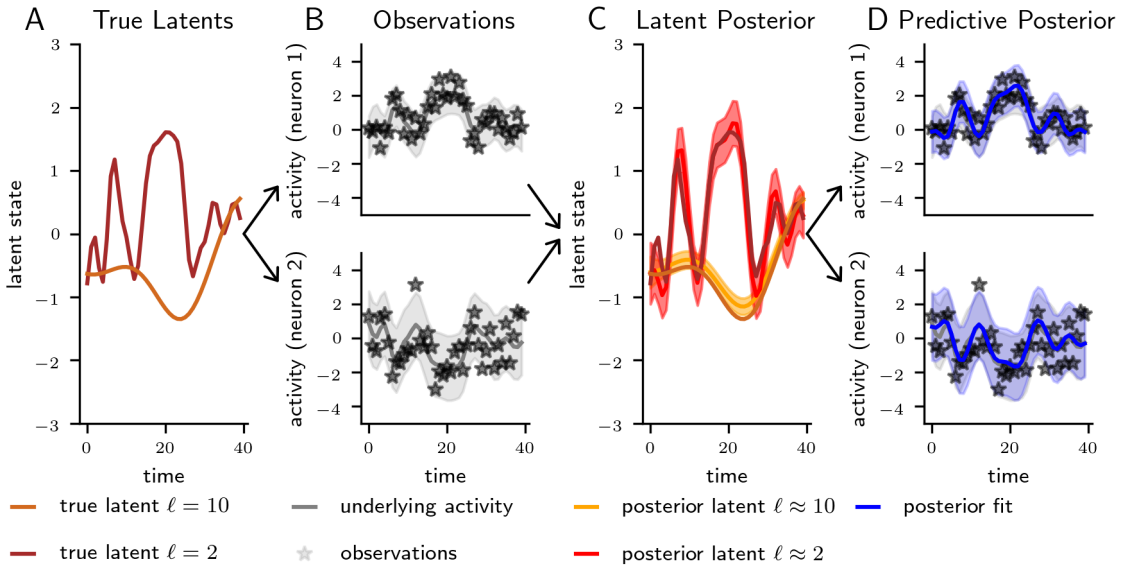


Fig. 1.3 **Gaussian Process Factor Analysis.** **(A)** True generative latents (sampled from Equation 1.7) are combined linearly via mixing matrices $\boldsymbol{C}$ into the **(B)** underlying true activity distribution (Equation 1.8). Observations are sampled from this distribution with variances $\boldsymbol{R}$. **(C-D)** The hyperparameters ($\boldsymbol{C}, \boldsymbol{R}$, and, ℓ_i) are fit, and the **(C)** latent posterior distribution (Equations 1.13 and 1.14) and the resulting **(D)** predictive posterior are inferred. Shading shows ± 2 standard deviations from the mean.

Fundamentally the approach is the same as for scalar GP regression. One uses a GP prior over a set of latent functions, and a linear Gaussian observation model, thereby inducing a joint GP distribution over latents and observations (very much like the GP prior over f coupled with the linear Gaussian noise model of Equation 1.3 induced a joint Gaussian distribution over $\boldsymbol{f}_*$ and $\boldsymbol{y}$).

For GPFA, the notation is as follows. There are T time-points $\boldsymbol{t} \in \mathbb{R}^T$, D latent variables $\boldsymbol{x}(t) \in \mathbb{R}^D$ arising from D a priori independent latent Gaussian processes with underlying kernels $\{k_1, \ldots, k_D\}$, and observations (neural data) from N neurons, $\boldsymbol{y}(t) \in \mathbb{R}^N$. The latents for each time-point can be stacked side-by-side in columns into a matrix $\boldsymbol{X} \in \mathbb{R}^{D \times T}$, and the observations can be stacked into $\boldsymbol{Y} \in \mathbb{R}^{N \times T}$. $\tilde{\boldsymbol{y}} \in \mathbb{R}^{NT}$ is shorthand for $\mathrm{vec}(\boldsymbol{Y})$ where $\mathrm{vec}(\cdot)$ stacks columns vertically. Note that this shorthand is different from that in [60] as I will be using time-space Kronecker products where [60] uses space-time Kronecker products (simply a notational difference in which term is on the lefthand versus the righthand sign of the Kronecker product). $\boldsymbol{I}_N$ denotes the $N \times N$ identity matrix, $\boldsymbol{1}_N$ denotes the ones vector of length N, δ_{ij} is the Kronecker delta (1 when $i = j$, 0 otherwise), $\boldsymbol{e}_i$ is a unit vector of length D with nonzero element at index i, and $\otimes$ is the Kronecker product.

Beyond the individual kernel hyperparameters, there is a mixing matrix, $\boldsymbol{C} \in \mathbb{R}^{N \times D}$, that indicates how latents are combined into observations, a diagonal positive definite matrix of observation noise variances, $\boldsymbol{R} \in \mathbb{R}^{N \times N}$, and a vector of neuron means, $\boldsymbol{\mu} \in \mathbb{R}^N$.

Given the notation above, the observations in GPFA are assumed to arise as follows (Figure 1.3A-B):

$$\boldsymbol{x}_i(\cdot) \sim \mathcal{GP}(\boldsymbol{0}, k_i(\cdot,\cdot)) \tag{1.7}$$

$$\boldsymbol{y}(t) \sim \mathcal{N}(\boldsymbol{\mu} + \boldsymbol{C}\boldsymbol{x}(t), \boldsymbol{R}) \tag{1.8}$$

The latent processes are assumed to be independent of each other:

$$k(x_i(t), x_j(t')) = \delta_{ij} k_i(t,t') \tag{1.9}$$

Thus, the temporal covariance (between two time-points) of the full vector of latents is given by:

$$k_{xx}(t,t') = \sum_{i=1}^{D} k_i(t,t') \otimes (\boldsymbol{e}_i \boldsymbol{e}_i^T), \tag{1.10}$$

leading to the full spatio-temporal covariance of $\tilde{\boldsymbol{x}}$

$$\boldsymbol{K}_{xx} = \sum_{i=1}^{D} K_i(\boldsymbol{t},\boldsymbol{t}) \otimes (\boldsymbol{e}_i \boldsymbol{e}_i^T). \tag{1.11}$$

As promised, this latent covariance coupled with the linear Gaussian observation model induces a multivariate Gaussian marginal over $\tilde{\boldsymbol{y}}$ with mean $\boldsymbol{1}_T \otimes \boldsymbol{\mu}$ and covariance

$$\boldsymbol{K}_{yy} = (\boldsymbol{I}_T \otimes \boldsymbol{C})\boldsymbol{K}_{xx}(\boldsymbol{I}_T \otimes \boldsymbol{C}^T) + (\boldsymbol{I}_T \otimes \boldsymbol{R}) \tag{1.12}$$

GPFA is mathematically identical to the linear model of coregionalization (LMC) which arose from the geostatistics literature [26]. Both LMC and GPFA construct a multi-output kernel as the covariance of a linear combination of multiple latent Gaussian processes. However, while LMC uses latent processes only as intermediate variables in the construction of a multi-output kernel, in GPFA we are actually interested in the posterior over these latents processes (Figure 1.3C).

GPFA for regression

In GPFA, the mean and covariance of the latent variables are inferred as follows (Figure 1.3C):

$$\mathbb{E}(\tilde{\boldsymbol{x}}|\tilde{\boldsymbol{y}}) = \boldsymbol{K}_{xx}(\boldsymbol{I}_T \otimes \boldsymbol{C}^T)\boldsymbol{K}_{yy}^{-1}(\tilde{\boldsymbol{y}} - \boldsymbol{1}_T \otimes \boldsymbol{\mu}) \tag{1.13}$$

$$\mathrm{cov}(\tilde{\boldsymbol{x}}|\tilde{\boldsymbol{y}}) = \boldsymbol{K}_{xx} - \boldsymbol{K}_{xx}(\boldsymbol{I}_T \otimes \boldsymbol{C}^T)\boldsymbol{K}_{yy}^{-1}(\boldsymbol{I}_T \otimes \boldsymbol{C})\boldsymbol{K}_{xx} \tag{1.14}$$

Once the latent variables have been inferred, posterior predictions are made according to Equation 1.8 (Figure 1.3D).

Fitting

The GPFA hyperparameters are fit using gradient descent on the negative marginal log likelihood, as in Equation 1.6

$$\log p(\tilde{\boldsymbol{y}}|\boldsymbol{t}) = -\frac{1}{2}(\tilde{\boldsymbol{y}} - \boldsymbol{1}_T \otimes \boldsymbol{\mu})\boldsymbol{K}_{yy}^{-1}(\tilde{\boldsymbol{y}} - \boldsymbol{1}_T \otimes \boldsymbol{\mu}) - \frac{1}{2}\log|\boldsymbol{K}_{yy}| - \frac{NT}{2}\log 2\pi \tag{1.15}$$

The GPFA hyperparameters include $\boldsymbol{C}$, the diagonal elements of $\boldsymbol{R}$, and the kernel hyperparameters (timescales) of $\{k_1, \ldots, k_D\}$. Because $\boldsymbol{C}$ is learned unconstrained, one can without loss of generality fix the prior variance to be 1 for each latent process to prevent scaling degeneracies. For visualization, one can follow [79] and orthonormalize $\boldsymbol{C}$ using singular value decomposition as $\boldsymbol{CX} = \boldsymbol{U}_C(\boldsymbol{D}_C\boldsymbol{V}_C\boldsymbol{X})$ (leaving the marginal likelihood unaffected). This again eliminates some degeneracies and leads to mixed-timescale latents ordered by variance explained. In Figure 1.3 and throughout this thesis, the latents are not orthonormalized to keep the latents timescale-separated.

1.4 Challenges with traditional implementations

Traditional implementations of GPFA are useful for short, trial-structured datasets, however they scale poorly with a computational complexity of $\mathcal{O}(N^3T^3)$ and a memory footprint of $\mathcal{O}(D^2T^2)$. The computational complexity suggested by Equation 1.15 can be reduced to $\mathcal{O}(D^3T^3)$ using the low rank and diagonal structure of Equation 1.12, however the T^3 factor (and T^2 factor in space complexity) are the limiting factor for large datasets (see Appendix F.1 of ref. 60 for the reduction to $\mathcal{O}(D^3T^3)$). This bottleneck lies in the need to invert (and in traditional implementations, represent) $\boldsymbol{K}_{yy} \in \mathcal{R}^{NT\times NT}$ or $\boldsymbol{K}_{xx} \in \mathcal{R}^{DT\times DT}$, and, in general, inverting a matrix of size $M \times M$ has time complexity $\mathcal{O}(M^3)$.

While D, the number of latents, is typically small, the cubic and square scaling of T in terms of computational complexity and memory respectively are prohibitive for large datasets. In its traditional implementation, GPFA can therefore only be applied to short chunks of data, e.g. $<$ 30s at 10ms resolution for the case of 4 latents. In practice, GPFA has been applied to up to 150 time-points (1.5 seconds at 10ms resolution with up to 15 latents in [79], or 2.6 seconds at 20ms resolution with 9 latents in [38]) or fewer [21].

The traditional solution to this challenge is chunking long datasets from trial-structured experiments into shorter single-trial-length pieces. This solution is problematic because it prevents GPFA from resolving any meaningful longer-timescale latents (Figure 1.4). Longer-timescale latents could represent changes in task engagement, attentional drift, or other relevant information about mental state [21]. While progressive changes across many trials can be captured by non-probabilistic methods such as Williams et. al.'s Tensor Component Analysis (TCA; 72), the trial factors in TCA can only scale the temporal- and neuron-factor activity multiplicatively. This means that, while TCA can capture some changes across trials, it is limited in its expressiveness. Additionally, chunking the data is awkward when a dataset has no inherent trial structure, or has trials of different lengths. This challenge is not unique to GPFA—other approaches such as LFADS also require chunking data [64, 38]. [38] in particular breaks non-trial-structured reaching data into 600ms chunks with 200ms overlaps so that inferred latents could be re-stitched back together as a weighted average.

In this thesis I propose and apply methods for scaling GPFA to these larger datasets by reducing the time and space complexity, overcoming some of the challenges and limitations discussed above.

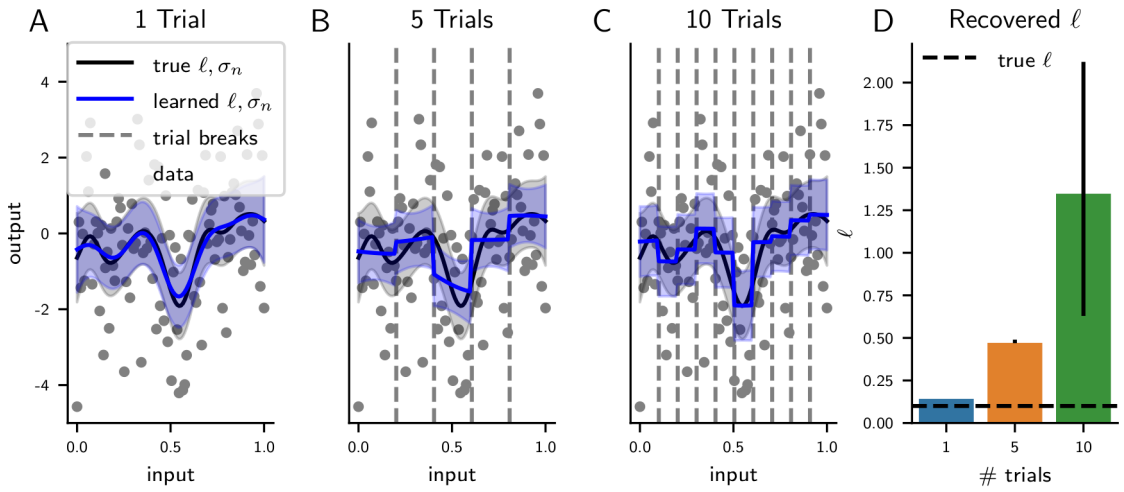


Fig. 1.4 **Resolving timescales on datasets chunked into shorter trials.** **(A-C)** GP models trained on the same data but separated into 1, 5, and 10 smaller trials respectively. Models were initialized with the generative timescale and observation noise hyperparameters ($\ell = .1$ and $\sigma_n = 1.5$). The posteriors were inferred using both the true hyperparameters over the full dataset (blue) and the learned hyperparameters on the smaller trials (black). Shading indicates ± 2 standard deviations from the mean. In the one-trial case, where the trial spans many timescales, the learned posterior is very close to the true posterior. However for models trained on shorter trials, the learned posterior is further from the true posterior, with discontinuities at the trial boundaries. **D)** The recovered timescales for models trained on data split into different numbers of trials across five different initializations ($\ell \in \{0.01, 0.05, 0.1, 0.5, 1.0\}$). Error bars show the 20^{th} and 80^{th} quantiles across these five timescale initializations. Note that for models trained on more trials where each trial spans fewer timescales, the timescale becomes increasingly difficult to resolve—the learned timescale is further from the true timescale, and the variance in recovered timescales increases.

1.5 Self-paced reaching dataset

To demonstrate the efficacy of Scalable GPFA methods described in the following chapters with respect to the challenges discussed in Section 1.4, we aim to apply GPFA to long recordings where chunking would be unnatural or suboptimal.

In particular, we apply GPFA to a biological dataset provided by refs. [52, 47] consisting of spike-sorted data from continuous recordings spanning up to one hour, with ~ 300 neurons recorded in two rhesus macaques. In this dataset, the animals make self-paced reaches to a series of targets in an 8x8 or 8x17 grid. These reaches are self-paced in that as soon as the animal successfully acquires a target, the next target appears. Targets do not time out, so animals could take as much or as little time as they want for each reach. After a target is acquired, there is a 200ms 'lockout interval' during which the next target appears, but no target can be acquired. Data is recorded from both M1 and S1, and hand and cursor kinematics are also recorded. For analyses, neurons with firing rates below 2 Hz are excluded, and data is binned at 25 ms resolution. Notably, this dataset could have longer intrinsic timescales than would be resolvable if the data were artificially chunked into trials.

Scalable GPFA allows us to see if previous findings regarding motor preparation also hold in less structured environments. Applying a scalable version of GPFA introduced in Chapter 4 resolves long timescales that seem to correlate with a measure of task engagement. This dataset additionally can be used to investigate the relationship between representations in M1 and S1 as both areas are recorded from simultaneously.

1.6 Overview

In the remainder of this thesis, I present work that enables GPFA to be applied to data that is multiple orders of magnitude larger than was previously possible. In Chapter 2, I overview existing and previously theorized methods for scaling GPs and GPFA. I then introduce and discuss my implementation of scalable iterative GPFA in Chapter 3. In Chapter 4, I describe a collaboration with Kris Jensen, Ta-chu Kao, and Guillaume Hennequin where I helped develop a scalable variational fully Bayesian extension of GPFA, bGPFA, that infers the matrix $\boldsymbol{C}$ in Equation 1.8 instead of treating it as a set of hyperparameters. I compare these two methods on real data in Chapter 5. Finally I conclude with a discussion of these methods and future directions in Chapter 6.

Chapter 2

Mathematical Preliminary

2.1 Existing methods for scaling GPs

Existing methods for scaling GPs can be categorized into approximate methods, including inducing point and other variational approaches, and "exact" iterative methods that exploit specific structure in the GP covariance function, such as Toeplitz or Kronecker structure [62, 68, 17, 60, 75, 76, 14].

Variational methods lower bound the (sometimes intractable) marginal log likelihood and perform hyperparameter optimization with respect to that more tractable lower bound [62, 68]. In particular, inducing point methods use a set of surrogate input points to construct a low-rank approximation to the covariance matrix, adaptively positioning them in input space so as to maximize the accuracy of the resulting approximate posterior [57, 9]. This works well when the underlying function varies on lengthscales long enough that only a few inducing points are needed to resolve them. However, these methods will require many more inducing points for hour-long neural recordings with typical fluctuation timescales on the order of tens of milliseconds such that they are either not sufficiently low-rank or do not accurately approximate the posterior. If these short timescales were the only contributors to the data's auto-covariance, one could still chunk the data into short trials spanning a few of these timescales only without losing much. Inducing point methods would then be applicable again. However, neural activity is known to vary over a large range of timescales, and chunking would inevitably limit the ability to resolve the timescales at the longer end of the spectrum (Figure 1.4; 6).

In contrast to approximate variational approaches, "exact" iterative methods either compute the exact marginal likelihood or an unbiased stochastic estimate thereof. Scalability is achieved by exploiting specific structure in the kernel to efficiently calculate kernel-vector products, and by using these fast matrix-vector products within the iterative conjugate gra-

dients (CG) algorithm to compute products with the inverse covariance matrix, as required both for training and inference [71, 60, 17]. In particular, it is possible to use CG-based algorithms to compute stochastic estimates of the log determinant term in the log marginal likelihood, as well as its gradient [71, 60]. Chapter 3 is dedicated to the application of such iterative methods to scaling up GPFA, as was suggested in ref. 60 but never implemented. Before that, I now review the various technical components that need to be pieced together to yield exact and scalable iterative GP inference and learning. Some of these components can in fact be integrated into variational inference-based methods, which we exploit in Chapter 4.

2.2 Scaling GPFA via Toeplitz- and Kronecker-structured kernels

In this section I explain the mathematics behind GPFA's prohibitive time and space complexities, and introduce a subset of the solutions proposed in the appendix of ref. 60 for scaling up GPFA.

2.2.1 The scalability problem

The derivative of GPFA's marginal log likelihood Equation 1.15 with respect to the hyperparameters must be calculated on each optimization step. The portions of Equation 1.15 that will be non-zero when differentiated with respect to the hyperparameters, including $\log|\boldsymbol{K}_{yy}|$ and $\boldsymbol{K}_{yy}^{-1}$, must be calculated for each optimization step. Naively, $\boldsymbol{K}_{yy} \in \mathbb{R}^{NT \times NT}$ is very costly to invert and take the log determinant of $(\mathcal{O}_{\text{space}}(N^2T^2), \mathcal{O}_{\text{time}}(N^3T^3))$. This cost can be reduced to $\mathcal{O}(D^3T^3)$ as described in Appendix F.1 of ref. 60by exploiting the low spatial rank of Kyy, using the Woodbury matrix identity. The scaling of T^3 however is still prohibitive in the case of long datasets. Appendix F.2 of ref. 60 suggests a more efficient method of scaling using efficient $\boldsymbol{K}_{yy}\boldsymbol{v}$ products.

In the Section 2.2.2 I explain how to efficiently calculate $\boldsymbol{K}_{yy}\boldsymbol{v}$ as suggested in ref. 60. Then in Section 3.1 I explain how one can leverage these efficient kernel vector products to iteratively invert and calculate the log determinant of $\boldsymbol{K}_{yy}$ using conjugate gradients and stochastic estimation of the log determinant [15, 60]. In Chapter 4, I explain how one can leverage the Toeplitz portion of these fast kernel vector products to implement a scalable variational version of GPFA.

2.2.2 Fast kernel vector products

If fast $\boldsymbol{K}_{yy}\boldsymbol{v}$ kernel vector products can be calculated, then there are iterative methods (discussed in Section 3.1) that we can take advantage of that do not have the systematic shortcomings of approximate methods discussed in Section 2.1.

Here, I explain mathematically how to achieve fast kernel vector products with GPFA specifically, as proposed but not implemented by [60]. Naively, kernel vector products have $\mathcal{O}(N^2T^2)$ complexity in both space and time because $\boldsymbol{K}_{yy} \in \mathbb{R}^{NT\times NT}$. With the methods proposed by [60] and discussed below, the time complexity is reduced to $\mathcal{O}(NDT + DT\log T)$ and the space complexity is $\mathcal{O}(NT + ND)$.

Exploiting Toeplitz structure

Recall the definition of $\boldsymbol{K}_{xx}$ (Equation 1.11) as a block diagonal matrix composed of blocks of stationary kernels $K_i(\boldsymbol{t},\boldsymbol{t})$. Recall that for typical neural data, spike counts are binned into equally spaced time bins, so the elements of $\boldsymbol{t}$ can be assumed to be evenly spaced. Therefore $K_i(\boldsymbol{t},\boldsymbol{t})$ is structured as illustrated in Figures 1.2 and 2.1A. This structure is referred to as Toeplitz structure—elements are the same along any of the diagonals.

Importantly, Toeplitz matrices can be embedded within circulant matrices as illustrated in Figure 2.1 to enable fast matrix-vector products. Circulant matrices are matrices in which one column contains all the entries of the matrix, and each subsequent column's entries are shifted down by one with the last entry becoming the first entry in the subsequent column. Circulant matrices have the key property that vector products with them can be calculated efficiently in the Fourier domain. Given a circulant matrix $\boldsymbol{F}_c$ entirely characterized by its first column $\boldsymbol{f}$, and a vector, $\boldsymbol{z}$:

$$\boldsymbol{F}_c\boldsymbol{z} = \mathrm{DFT}^{-1}(\mathrm{DFT}(\boldsymbol{f}) \odot \mathrm{DFT}(\boldsymbol{z})) \tag{2.1}$$

where DFT denotes the discrete Fourier transform. To calculate $K_i(\boldsymbol{t},\boldsymbol{t})\boldsymbol{v}$, we can thus embed $K_i(\boldsymbol{t},\boldsymbol{t}) \in \mathbb{R}^{T\times T}$ in a circulant matrix $\boldsymbol{F}_c \in \mathbb{R}^{(2T-1)\times(2T-1)}$, such that

$$\boldsymbol{F}_c \begin{bmatrix} \boldsymbol{v} \\ \boldsymbol{0} \end{bmatrix} = \begin{bmatrix} K_i(\boldsymbol{t},\boldsymbol{t})\boldsymbol{v} \\ \boldsymbol{S}\boldsymbol{v} \end{bmatrix} \tag{2.2}$$

where $\boldsymbol{S} \in \mathbb{R}^{(T-1)\times T}$ is the (irrelevant) lower left corner of $\boldsymbol{F}_c$. Thus, the desired matrix-vector product with $K_i(\boldsymbol{t},\boldsymbol{t})$ can be extracted as the first T elements of the result. Because this multiplication can be computed with fast Fourier transforms, it has $\mathcal{O}(T\log T)$ time

complexity. Because $K_i(\boldsymbol{t},\boldsymbol{t})$ can be stored as the first column of circulant matrix of length $2T-1$, the space complexity is $\mathcal{O}(T)$.

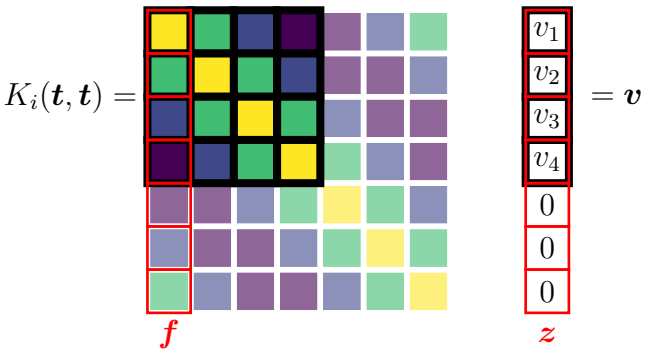


Fig. 2.1 **Schematic of Toeplitz structure.** In this schematic, the color of each element in the matrix indicates the value of that element. To compute $K_i(\boldsymbol{t},\boldsymbol{t})\boldsymbol{v}$ (black outlines), where $\boldsymbol{v} = [v_1, v_2, v_3, v_4]$, first note that $K_i(\boldsymbol{t},\boldsymbol{t})$ has Toeplitz structure—its elements are the same along the diagonal. Because of its Toeplitz structure, $K_i(\boldsymbol{t},\boldsymbol{t})$ can be embedded into a circulant matrix (white outline) which is entirely characterized by its first column (red outline) as follows:

- The first column of the circulant matrix is given by concatenating the first column of $K_i(\boldsymbol{t},\boldsymbol{t})$, $K_i(\boldsymbol{t},t_1) = K_i(t_1,t_1), K_i(t_2,t_1), ..., K_i(t_T,t_1)$ with its reverse, excluding the first element (the last element in the reverse version). The first column of the circulant matrix is thus given by $K_i(t_1,t_1), K_i(t_2,t_1), ..., K_i(t_T,t_1), K_i(t_T,t_1), K_i(t_{T_1},t_1), ..., K_i(t_2,t_1)$.
- The top element of each column of the circulant matrix is given by the bottom element of the previous column (the column immediately to its left).
- The remaining elements of each circulant column are given by previous column's elements, shifted down by one (removing the bottom element).

Multiplying by vector $\boldsymbol{z}$ (red outline) yields the desired result once the first T elements are extracted as described in Equation 2.2. This multiplication can be computed efficiently using fast Fourier transforms (Equation 2.1).

If $\boldsymbol{t}$ is not evenly spaced, these methods can still be applied using KISS-GP, a method that interpolates the kernel onto a regular grid [76].

Exploiting Kronecker structure

The above section demonstrated how to compute fast kernel vector products for stationary, one-dimensional kernels. I now show how to exploit the Kronecker structure of the GPFA kernel (Equation 1.11) to compute fast kernel-vector products with this multi-output covariance. Kronecker products have the property that $(\boldsymbol{A}^T \otimes \boldsymbol{F})\text{vec}(\boldsymbol{V}) = \text{vec}(\boldsymbol{FVA})$ [69]. Applying

this property to $\boldsymbol{K}_{xx}$ (Equation 1.11) and then transposing the inside of the vec $(\cdot)$ gives

$$\boldsymbol{K}_{xx}\boldsymbol{v} = \sum_{i=1}^{D} \left[K_i(\boldsymbol{t},\boldsymbol{t}) \otimes (\boldsymbol{e}_i\boldsymbol{e}_i^T)\right] \boldsymbol{v} = \sum_{i=1}^{D} \text{vec}\left((\boldsymbol{e}_i\boldsymbol{e}_i^T)\boldsymbol{V}K_i(\boldsymbol{t},\boldsymbol{t})\right) = \sum_{i=1}^{D} \text{vec}\left(\left(K_i(\boldsymbol{t},\boldsymbol{t})\boldsymbol{V}^T(\boldsymbol{e}_i\boldsymbol{e}_i^T)\right)^T\right) \tag{2.3}$$

where $\boldsymbol{v} = \text{vec}(\boldsymbol{V})$, $\boldsymbol{V} \in \mathbb{R}^{D\times T}$. Note that $(\boldsymbol{e}_i\boldsymbol{e}_i^T)^T = (\boldsymbol{e}_i\boldsymbol{e}_i^T)$, and $K_i(\boldsymbol{t},\boldsymbol{t})^T = K_i(\boldsymbol{t},\boldsymbol{t})$ because covariance matrices are symmetric.

The time complexity for $\boldsymbol{K}_{xx}\boldsymbol{v}$ thus has a factor of D for the outer sum, and a factor of $T\log T$ for the summand. $\boldsymbol{V}^T(\boldsymbol{e}_i\boldsymbol{e}_i^T)$ selects column i of $\boldsymbol{V}^T$ in $\mathcal{O}(1)$ time. That column, $\boldsymbol{V}_i^T$ can than be multiplied as $K_i(\boldsymbol{t},\boldsymbol{t})\boldsymbol{V}_i^T$ using the methods described in Section 2.2.2 in $\mathcal{O}(T\log T)$ time. Thus the total time complexity for $\boldsymbol{K}_{xx}\boldsymbol{v}$ is $\mathcal{O}(DT\log T)$. The space complexity is $\mathcal{O}(TD)$ due to the circulant representation of $K_i(\boldsymbol{t},\boldsymbol{t})$ as discussed in Section 2.2.2.

Now for $\boldsymbol{K}_{yy}\boldsymbol{v}$ products, the Kronecker product can be leveraged again, twice in the case of the first term:

$$\boldsymbol{K}_{yy}\boldsymbol{v} = (\boldsymbol{I}_T \otimes \boldsymbol{C})\boldsymbol{K}_{xx}(\boldsymbol{I}_T \otimes \boldsymbol{C}^T)\boldsymbol{v} + (\boldsymbol{I}_T \otimes \boldsymbol{R})\boldsymbol{v} \tag{2.4}$$

$$= \text{vec}(\boldsymbol{C}\text{vec}^{-1}(\boldsymbol{K}_{xx}\text{vec}(\boldsymbol{C}^T\boldsymbol{V}\boldsymbol{I}_T))\boldsymbol{I}_T) + \text{vec}(\boldsymbol{R}^T\boldsymbol{V}\boldsymbol{I}_T) \tag{2.5}$$

$$= \text{vec}(\boldsymbol{C}\text{vec}^{-1}(\boldsymbol{K}_{xx}\text{vec}(\boldsymbol{C}^T\boldsymbol{V}))) + \text{vec}(\boldsymbol{R}\boldsymbol{V}) \tag{2.6}$$

Note that here $\boldsymbol{V} \in \mathbb{R}^{N\times T}$ and again $\boldsymbol{v} = \text{vec}(\boldsymbol{V})$.

The time complexity for $\text{vec}(\boldsymbol{R}\boldsymbol{V})$ is $\mathcal{O}(NT)$ because $\boldsymbol{R}$ is a diagonal matrix, the time complexity for $\text{vec}(\boldsymbol{C}^T\boldsymbol{V})$ is $\mathcal{O}(NDT)$, the time complexity of $\text{vec}^{-1}(\boldsymbol{K}_{xx}\text{vec}(\boldsymbol{C}^T\boldsymbol{V}))$ is $\mathcal{O}(DT\log T + NDT)$, and so the time complexity of the entire $\text{vec}(\boldsymbol{C}\text{vec}^{-1}(\boldsymbol{K}_{xx}\text{vec}(\boldsymbol{C}^T\boldsymbol{V})))$ is $\mathcal{O}(DT\log T + NDT)$.

The overall time complexity of $\boldsymbol{K}_{yy}\boldsymbol{v}$, leveraging Toeplitz and Kronecker structure is thus $\mathcal{O}(DT\log T + NDT)$. The overall space complexity is $\mathcal{O}(NT + ND + DT)$. No N^2 term is needed because the diagonal matrices $\boldsymbol{R}$ and $\boldsymbol{I}_T$ can be represented sparsely by vectors of length T. We assume $D < N$ because GPFA is used for dimensionality reduction, so this space complexity reduces to $\mathcal{O}(NT + ND)$ which is no more than the space complexity required to store the vector $\boldsymbol{v}$ and the mixing matrix $\boldsymbol{C}$.

Chapter 3

Scalable Iterative GPFA

3.1 Scaling GPFA assuming fast kernel vector products

This section describes fast iterative methods for computing $\boldsymbol{K}_{yy}^{-1}$ and $\log|\boldsymbol{K}_{yy}|$, the difficult-to-compute and repeatedly needed terms of the loss and its derivative used for fitting GPFA (Equation 1.15). Each iteration makes use of the fast kernel vector products discussed in Chapter 2.

3.1.1 Fast $\mathbf{K}_{yy}^{-1}\mathbf{v}$

Fast $\boldsymbol{K}_{yy}^{-1}\boldsymbol{v}$ products can be computed by using the conjugate gradients (CG) algorithm to solve $\boldsymbol{K}_{yy}\boldsymbol{x} = \boldsymbol{v}$ [30]. CG computes $\boldsymbol{x}$ through a series of $\boldsymbol{K}_{yy}\boldsymbol{a}$ products, without ever requiring the full construction of $\boldsymbol{K}_{yy}$. A naive approach to computing gradients through such solutions would be to backpropagate through every CG step (in our case, the worst case would be NT steps). Appendix F.2 in ref. [60] derives a method to avoid this using an implicit differentiation technique that identifies the desired gradient as another specific CG solve, thus removing the need to store the entire graph of all CG iterations. This enables differentiating through CG in constant memory. In practice, I found this method was unnecessary as CG converged sufficiently in fewer than NT steps. I additionally found that the first software package I began implementing GPFA in (Section 3.2.1) already included a version of CG that dealt with this memory issue in a similar way. While the package I fully implemented GPFA in (Section 3.2.2) does not implement this memory-saving method, in practice I did not find this to be a limiting factor.

3.1.2 Fast $\log|\mathbf{K}_{yy}|$ and its derivative

In this work, the $\log|\mathbf{K}_{yy}|$ is approximated using trace estimation with the Lanczos approximation as described in ref. [18]. [60] introduces an alternative, unbiased method for calculating $\log|\mathbf{K}_{yy}|$ that leverages fast $\boldsymbol{K}_{yy}\boldsymbol{v}$ products that I have not implemented. This unbiased method would be preferable for model comparison where the marginal log likelihood is computed only once after training, and must be precise.

The gradient of the log determinant is what must actually be computed repeatedly during fitting. Fast gradients of the log determinant can be calculated by using Hutchinson trace estimation (Equation 3.2) and rearranging in Equation 3.3 to leverage fast CG-based $\boldsymbol{K}_{yy}^{-1}\boldsymbol{v}$ from Section 3.1.1 [15, 60, 32]:

$$\frac{\partial \log|\boldsymbol{K}_{yy}|}{\partial \theta_i} = \mathrm{Tr}\left[\boldsymbol{K}_{yy}^{-T}\frac{\partial \boldsymbol{K}_{yy}}{\partial \theta_i}\right] \tag{3.1}$$

$$= \left\langle \mathrm{Tr}\left[\boldsymbol{\xi}^T \boldsymbol{K}_{yy}^{-T}\frac{\partial \boldsymbol{K}_{yy}}{\partial \theta_i}\boldsymbol{\xi}\right]\right\rangle_{\boldsymbol{\xi}} \tag{3.2}$$

$$= \left\langle \mathrm{Tr}\left[\boldsymbol{\xi}(\boldsymbol{K}_{yy}^{-1}\boldsymbol{\xi})^T\frac{\partial \boldsymbol{K}_{yy}}{\partial \theta_i}\right]\right\rangle_{\boldsymbol{\xi}} \tag{3.3}$$

Here, θ_i is the hyperparameter one is differentiating with respect to and $\boldsymbol{\xi} \in \mathbb{R}^{NT}$ is a random vector with each entry randomly and evenly sampled from $\{-1,1\}$, the Rademacher distribution.

The expression in Equation 3.3 can be calculated efficiently by defining $\boldsymbol{z} = \boldsymbol{K}_{yy}^{-1}\boldsymbol{\xi}$ which can in turn be calculated efficiently using conjugate gradients, as described in Section 3.1.1. We can then rewrite Equation 3.3 as follows:

$$\left\langle \mathrm{Tr}\left[\boldsymbol{\xi}(\boldsymbol{K}_{yy}^{-1}\boldsymbol{\xi})^T\frac{\partial \boldsymbol{K}_{yy}}{\partial \theta_i}\right]\right\rangle_{\boldsymbol{\xi}} = \left\langle \mathrm{Tr}\left[\boldsymbol{\xi}\boldsymbol{z}^T\frac{\partial \boldsymbol{K}_{yy}}{\partial \theta_i}\right]\right\rangle_{\boldsymbol{\xi}} \tag{3.4}$$

$$= \left\langle \boldsymbol{z}^T\frac{\partial \boldsymbol{K}_{yy}}{\partial \theta_i}\boldsymbol{\xi}\right\rangle_{\boldsymbol{\xi}} \tag{3.5}$$

$$= \left\langle \boldsymbol{z}^T\frac{\partial}{\partial \theta_i}(\boldsymbol{K}_{yy}\boldsymbol{\xi})\right\rangle_{\boldsymbol{\xi}} \tag{3.6}$$

Importantly, Equation 3.6 requires a differentiation of matrix vector products, which can be performed efficiently without explicitly representing $\boldsymbol{K}_{yy}$ as described in Chapter 2. Note that Equation 3.6 reduces the space complexity of differentiating a matrix of size $NT \times NT$ to differentiating a vector of size NT, with respect to $\mathcal{O}(ND)$ parameters ($\mathcal{O}(N^3T^2D) \rightarrow \mathcal{O}(N^2TD)$). However, this memory cost can still be reduced further, by treating $\boldsymbol{z}$ in Equa-

tion 3.6 as a constant w.r.t. $\boldsymbol{\theta}$ and rewriting

$$\frac{\partial \log |\boldsymbol{K}_{yy}|}{\partial \boldsymbol{\theta}} = \frac{\partial}{\partial \boldsymbol{\theta}} \left\langle \boldsymbol{z}^T \boldsymbol{K}_{yy} \boldsymbol{\xi} \right\rangle_{\xi} \tag{3.7}$$

thus reducing the memory cost to $\mathcal{O}(ND)$, i.e. the cost of differentiating a scalar w.r.t. the model's parameters.

3.2 Implementation

As I learned about GPs and began to implement them, I explored different potential frameworks to use. I started with JAX, an automatic differentiation framework derived from autograd (NumPy-like automatic differentiation in Python) combined with XLA, a compiler for machine learning [8]. After spending some time with JAX, however, we realized that GPyTorch already implemented many of the Toeplitz-structure-based speedups [25]. Moreover, GPyTorch also provides the computational machinery to perform kernel interpolation onto regular grids, which is one of the modern ways of dealing with non-gridded data [76]. Although I did not plan to apply GPFA to such heterogeneous datasets, it appeared important to facilitate these applications for future users. I therefore decided to implement GPFA in GPyTorch to give the resulting implementation more flexibility for others to use.

In hindsight, JAX may have been the better choice, because while GPyTorch has many features already implemented, it has proven relatively opaque in that the precise algorithms used depend dynamically on the problem structure and are therefore harder to predict. Additionally, implementing new features (e.g. new preconditioners) can be quite difficult, especially if no method for adding that type of feature is documented.

3.2.1 JAX

Using JAX, I implemented Toeplitz-accelerated matrix vector products. I used this and the JAX conjugate gradients algorithm, which implements a memory-saving method for differentiating through its CG algorithm, to implement GP regression before moving to GPyTorch.

I found JAX straightforward to work with because it is purposefully implemented such that the user interface is very similar to NumPy – code can mostly be switched from NumPy to JAX by importing JAX in place of NumPy. This similarity also made it transparent for me to understand what my code was doing. It is worth noting, however, that not everything

available in NumPy has been implemented in JAX yet (e.g. some of the functions needed to implement non-reversible GP kernels in [60]).

JAX allows one to perform “just-in-time" (’jit’) compilation using a decorator. Rather than dispatching operations to the GPU one by one, ‘jitting’ a function instructs JAX to compile operations together with access to the full graph for the ‘jitted’ operations, enabling a number of optimizations (e.g. cache efficiency) on the GPU or TPU [8]. While this occasionally resulted in unfamiliar error messages, I found ‘jitting’ relatively easy to learn, and ‘jitted’ JAX provided immense speedups once compiled.

3.2.2 GPyTorch

GPyTorch seemed attractive with its many features, however I found it proved easier to use than to extend. I did not switch back to JAX because learning to use GPyTorch had required a significant time investment, and once something is already implemented, GPyTorch is quite user friendly.

One attractive feature of GPyTorch is its `LazyTensor` class and subclasses. This provides a method for automatically performing efficient computations with structured tensors. For example, `DiagLazyTensor` wraps a diagonal matrix, and automatically inverts it by inverting each diagonal element, and performs matrix multiplications by directly scaling the rows of the matrix or vector to be multiplied with the diagonal matrix. Similarly, GPyTorch offers a `ToeplitzLazyTensor` and a `KroneckerLazyTensor` class that implement efficient computations involving Toeplitz- and Kronecker-structured matrices, respectively. I refer to implementations that leverage this Toeplitz structure as ‘gridded’, and implementations that do not leverage this Toeplitz structure as ‘ungridded.’ These lazy tensors are composable and can greatly accelerate computations when used properly. This part of the library is still under active development, and the available `LazyTensor` subclasses changed as I was using them.

It is worth noting that GPyTorch already provides a multi-output kernel similar to GPFA’s $\boldsymbol{K}_{yy}$ kernel, called the Linear Model of Coregionalization (LMC; 26). However, this class only exposes the marginal covariance $\boldsymbol{K}_{yy}$, but does not expose other components of the joint covariance of $\boldsymbol{x}$ and $\boldsymbol{y}$ that are needed in GPFA to perform inference. Therefore, I wrote an entire GPFA kernel and model class from scratch—code is available in Appendix A.2.

GPyTorch has a settings module which allows different features to be turned on and off as demonstrated in Code Example 3.2. In particular, I use different settings to perform “exact” and iterative GPFA. I detail the differences between the “exact” and iterative GPFA methods used in Section 3.3 in Table 3.1. See Code Example 3.1 for setting up a GPyTorch GPFA model and initializing it; see Code Example 3.2 for training and performing inference using a GPyTorch model, and Algorithm 1 for pseudocode.

3.3 Application of the GPyTorch implementation to synthetic data

I first applied my GPyTorch implementation of GPFA to synthetic data generated from the GPFA model using known hyperparameters. Doing so allowed me to choose training parameters, verify my implementation's correctness, and measure its space and time complexity as discussed below.

For all experiments presented in this section and in Chapter 5, I used the Adam optimizer and ran all computations in double precision [39]. I additionally constrained all elements of R to be > 0.1 as suggested by [71]. I found that without this constraint, the loss diverged.

CG solve accuracy was calculated when solving for $\boldsymbol{b}$ in $\boldsymbol{Ab} = \boldsymbol{c}$ as $\frac{\|\boldsymbol{c}-\boldsymbol{A}\hat{\boldsymbol{b}}\|_2}{\|\boldsymbol{c}\|_2}$. I did not precondition CG—in practice I found that it converged sufficiently for training and inference on synthetic datasets without preconditioning, despite being relatively ill-conditioned. While I did attempt to implement two potential preconditioners, I ran into issues with numerical instabilities that led to worse performance than without the preconditioners. Because iterative GPFA worked on the smaller synthetic datasets I was testing on, I did not prioritize resolving those issues. From experiments I discuss in Chapter 5, however, it seems that a good preconditioner may be critical in the case of large real datasets.

For a comparison of implementation and hyperparameter details between the exact and iterative methods see Table 3.1.

3.3.1 Choosing training parameters

Iterative GPFA is very sensitive to the choice of training parameters. I thus determined parameters to use for training via a grid search (Figure 3.1).

Based on this grid search, I trained iterative and exact GPFA with a learning rate of 0.005 in the remaining experiments. I ran CG to an accuracy tolerance of 10^{-3} during training, a lower tolerance than that suggested by [71], and found that a tolerance of 10^{-5} (lower than ref. 71 suggests) was necessary to obtain accurate latent posteriors after training. I used 10 probe vectors for trace estimation of the gradient of the log determinant. Surprisingly, more probe vectors led to worse performance, possibly because 10 probe vectors leads to a more stochastic gradient descent trajectory which has benefits in many loss landscapes. I did explore whether 1 or 4 probe vectors would outperform 10 probe vectors, however performance suffered with fewer probe vectors.

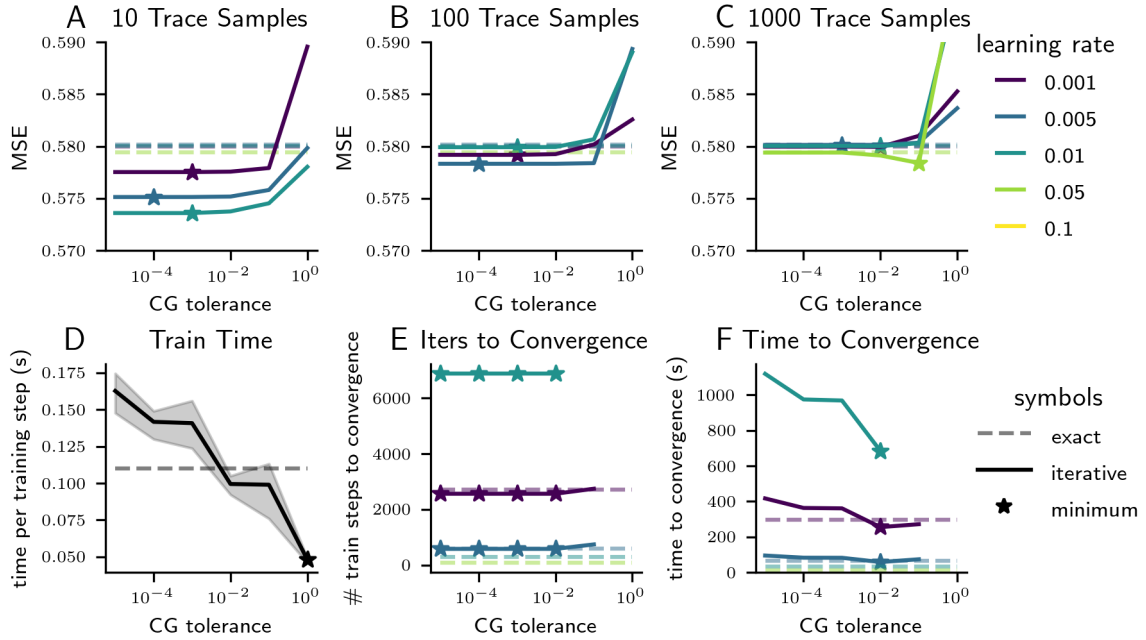


Fig. 3.1 **Training parameters for iterative GPFA. (A-C)** Cross-validated mean square prediction error (MSE) for different CG tolerance values and learning rates with 10, 100, and 1000 trace samples respectively. MSE was calculated for 100 time-points not seen during training, with half of the neurons used to infer the latents. Data not shown has a higher MSE than the range shown. Dashed lines show MSE for the exact training method (Table 3.1) **(D)** Time per training step for 10 trace samples, as a function of CG tolerance, averaged across learning rates. Shadings show the 20^{th} and 80^{th} quantiles across five learning rates. **(E)** Number of steps required to achieve convergence, defined as the exact loss reaching and subsequently staying below $l_{min} + 0.1 \times (l_{init} - l_{min})$, where l_{min} is the minimum loss achieved by the exact method for a given learning rate, and l_{init} is the initial loss. Excluded data did not achieve convergence thus defined. **(F)** Time to convergence computed as the value shown in **D** multiplied by the value shown in **E**. All models in this figure were trained for 7501 training steps on the same data with $T = 100$, $D = 2$, and $N = 20$, and loss was recorded every 20 steps (including the first and last step).

3.3.2 Verifying correctness

To verify that the derivatives used in training iterative GPFA were mathematically correct, I initialized the model with the true generative parameters and trained the model from that initialization. Figure 3.2A demonstrates that the marginal log likelihood does increase, but does not increase too much (less than 2.5 increase from the initial value) during 1000 training steps. Moreover, the posterior latents are more or less conserved, and are relatively consistent across GPFA implementations (Figure 3.2B-C). These small deviations from ground truth parameters and latents are to be expected, because the model was trained on a finite amount of noisy ($R_{i,i} > 0.1$) data.

The difference in loss between exact and iterative implementations (Figure 3.2A) even in the first step is because, in the iterative case, the log determinant term in the loss is approximated using the biased Lanczos method rather than exactly calculated (note however that the stochastic estimation of the gradient of the log determinant is unbiased) [18]. The exact loss is also computed and plotted during iterative training to demonstrate this. Because the exact and iterative methods differ in how the loss is calculated (Table 3.1), the two implementations return different losses, even though at the beginning of the first step, all hyperparameters are equal.

3.3.3 Measuring time and space complexity

To assess the extent to which the iterative method realizes the computational and memory savings predicted by the complexity analysis, I measured both time and peak increase in active memory used by PyTorch during each training step and during inference (Figure 3.3), for both the exact and iterative implementations (see Table 3.1 for details). In particular, note that I did not compute $\log|\boldsymbol{K}_{yy}|$ during these measurements as it is unnecessary for training and inference (it is only needed to monitor the progress of the loss function during training, and for model comparison).

As noted in Table 3.1, the GPyTorch exact implementation does not make use of the Kronecker structure in $\boldsymbol{K}_{yy}$ to compute its Cholesky decomposition. Therefore, the exact implementation depicted here scales suboptimally as N^2 rather than D^2. Regardless, the scaling would still be quadratic in T even with a Kronecker-aware exact implementation. Figure 3.3 demonstrates the scaling with time and memory for exact and iterative implementations with and without leveraging Toeplitz structure. Memory scaling is quadratic in T for all methods except for the iterative gridded method, for which memory scaling is near-linear (Figure 3.3C-D). For the largest T, the iterative gridded method is the only method which does not overflow the memory constraints. While exact methods and iterative

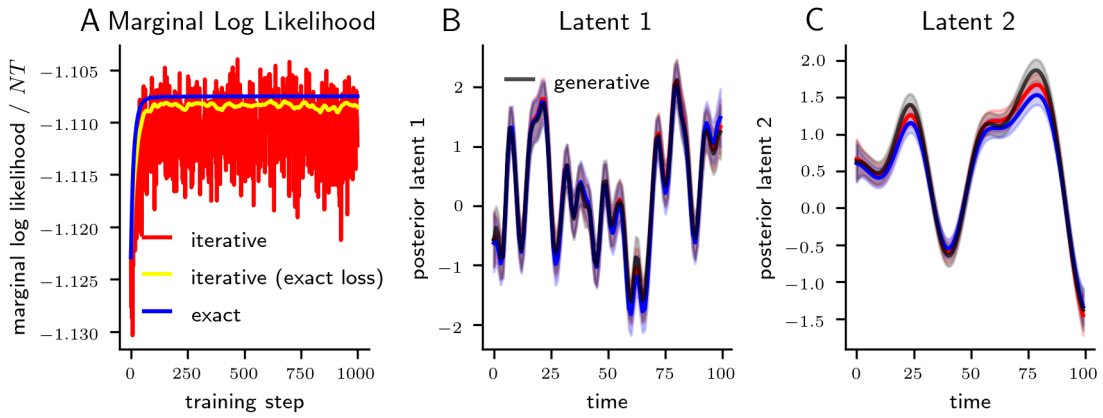


Fig. 3.2 **Correctness of iterative GPFA. (A)** Evolution of the marginal log likelihood for a GPFA model initialized with ground truth generative parameters, and trained using either the exact or the iterative gridded method (see legend and Table 3.1). For the iterative implementation, the loss can be computed exactly (yellow) or stochastically using trace samples (red) during training. **(B-C)** Timecourse of the latent posterior for a model initialized with generative parameters and then either not trained (generative) or trained (iterative, exact). Shading is ± 2 posterior standard deviations (as calculated from the posterior covariance in Equation 1.14). There is good agreement between all three models, with some mismatch between the generative posterior and the iterative and exact posteriors. This is to be expected as there is substantial noise ($R_{i,i} > 0.1$) and finite data. All models in this figure were trained on the same data with $T = 100$, and $N = 20$.

ungridded methods are faster for inference and training with few timepoints, the iterative gridded method is fastest for large T, and enables GPFA on larger T than possible with the other methods (Figure 3.3A-B).

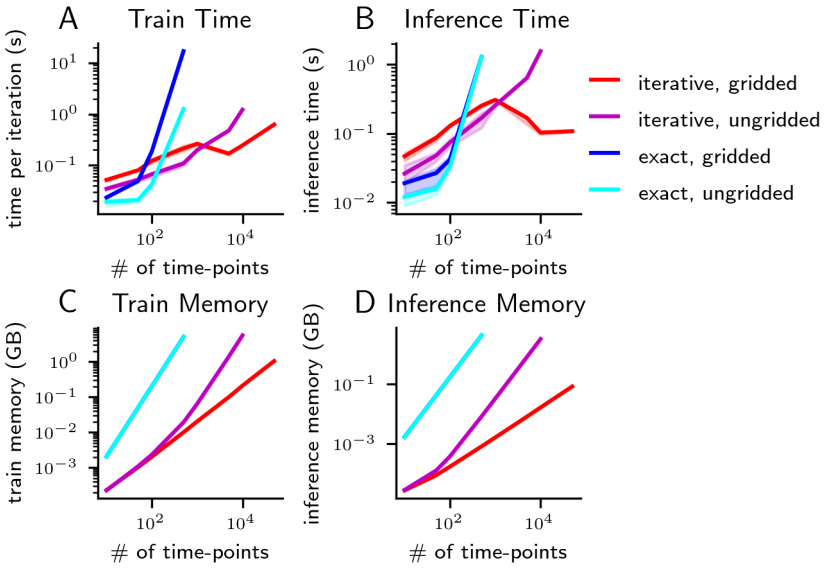


Fig. 3.3 **Performance Scaling of Iterative GPFA with T. (A–B)** Training time per step (A) and inference time (B) both scale quadratically with T in the exact case, and near linearly in the iterative case. **(C–D)** Maximum active memory during training (C) and inference (D) for the various methods. Shadings show the 20^{th} and 80^{th} quantiles across five repeats, average train times are averaged across 20 training steps within each repeat, and train memory is the increase in memory from immediately before training starts to the peak active memory used over the 20 train steps. In all panels, we set $D = 2$ and $N = 20$. Data not shown correspond to unsuccessful runs that overflowed the GPU's memory (11GB).

Table 3.1 **GPyTorch implementation details for exact vs. iterative GPFA.**

Computation	Details	Exact	Iterative
$\boldsymbol{K}_{yy}\boldsymbol{v}$ products	Kronecker	Yes for multiplication, no for inversion (could be, but not implemented in GPyTorch)	Yes
	Toeplitz	Yes if Gridded	Yes if Gridded
$\boldsymbol{K}_{yy}^{-1}\boldsymbol{v}$	How computed	Cholesky	Conjugate Gradients
	CG tolerance	n/a	10^{-3} unless otherwise specified
$\frac{\partial \log \lvert\boldsymbol{K}_{yy}\rvert}{\partial \boldsymbol{\theta}}$	How Computed	Cholesky	Approximated using CG with trace approximation sample vectors
	CG tolerance	n/a	10^{-3} unless otherwise specified
	# trace samples	n/a	10 unless otherwise specified
$\log \lvert\boldsymbol{K}_{yy}\rvert$	How Computed	Cholesky	Lanczos approximation (biased) with trace approximation sample vectors (unbiased)
	Always Computed?	Yes	Not computed when `gpytorch.settings.skip_logdet_forward` is False
	# trace samples	n/a	10 unless otherwise specified
	# Lanczos vectors	n/a	# trace samples
Inference	How Computed	Cholesky	Conjugate Gradients
	CG tolerance	n/a	10^{-5} unless otherwise specified

```
1  import torch
2  import gpytorch
3  import numpy as np
4  from gpfa_model import GPFAModel, gpfa_fa_init # See Appendix A.2
5
6  # the parameters below must be set according to use case
7  grid_mode = ? # Whether to use Toeplitz or not
8  d_fit = ? # Number of latents to fit
9  ell = ? # timescale to initialize with
10 Y = ? # T x N numpy array of training data
11
12 device = torch.device("cuda") # the torch cuda or cpu device
13
14 # Y: np array of training data, T: # of time-points, n: # of neurons
15 T, n = Y.shape
16 T = torch.arange(T).to(device).double() # List of time-points
17
18 # the multitask likelihood has both global and neuron-specific noise,
       here we constrain both > 0.05, ensuring that R > 0.1
19 likelihood = gpytorch.likelihoods.MultitaskGaussianLikelihood(
20   num_tasks=n, noise_constraint = gpytorch.constraints.GreaterThan(.05))
21
22 # Make a list of latent kernels with or without gridding
23 if grid_mode:
24    kernels = [gpytorch.kernels.GridKernel(
25      gpytorch.kernels.RBFKernel(), [T]) for _ in range(d_fit)]
26 else:
27    kernels = [gpytorch.kernels.RBFKernel() for _ in range(d_fit)]
28
29 Ytrain = torch.tensor(Y).to(device).double()
30
31 # Instantiate a model
32 model = GPFAModel(T, Ytrain, likelihood, kernels, d_fit, n)
33
34 # Initialize using factor analysis
35 gpfa_fa_init(model, Y.T[np.new_axis,:], ell, grid_mode)
```

Code Example 3.1 **GPyTorch GPFA initialization**. In line 16, time is measured in arbitrary units. In lines 22, 23 and 25 to 27, note that other kernels are available through GPyTorch.

```
1  # the parameters below must be set according to use case
2  iter_mode = ? # Whether to run the iterative or exact implementation
3  max_cg_iters = ? # When to stop CG even if tolerance not reached.
4  max_steps = ? # number of training steps
5
6  # Make sure everything is in the correct mode
7  model = model.cuda().double().train()
8  likelihood = likelihood.cuda().double().train()
9  T = T.cuda()
10 Ytrain = Ytrain.cuda()
11
12 # Train the model
13 optimizer = torch.optim.Adam(model.parameters(), lr=.005)
14 mll = gpytorch.mlls.ExactMarginalLogLikelihood(likelihood, model)
15
16 with gpytorch.settings.max_cholesky_size(0), gpytorch.settings.
       fast_computations(covar_root_decomposition=iter_mode, log_prob=
       iter_mode, solves=iter_mode), gpytorch.settings.cg_tolerance(1e-3),
       gpytorch.settings.terminate_cg_by_size(True),gpytorch.settings.
       max_cg_iterations(max_cg_iters), gpytorch.settings.
       max_preconditioner_size(0), gpytorch.settings.num_trace_samples(10),
       gpytorch.settings.memory_efficient(True):
17     for i in range(max_steps):
18         optimizer.zero_grad()
19         output = model(T)
20         loss = -mll(output, Ytrain)
21         loss.backward()
22         optimizer.step()
23
24 # Inference
25 model.eval()
26 likelihood.eval()
27
28 with gpytorch.settings.max_cholesky_size(0), gpytorch.settings.
       fast_computations(covar_root_decomposition=iter_mode, log_prob=
       iter_mode, solves=iter_mode), torch.no_grad(), gpytorch.settings.
       fast_pred_var(), gpytorch.settings.cg_tolerance(1e-5), gpytorch.
       settings.terminate_cg_by_size(True),gpytorch.settings.
       max_cg_iterations(max_cg_iters), gpytorch.settings.
       max_preconditioner_size(0), gpytorch.settings.memory_efficient(True):

29         posterior_latents = model.latent_posterior(T)
```

Code Example 3.2 **GPyTorch GPFA Training and Inference.** Parameters used but not defined here are defined in Code Example 3.1. In line 3, `max_cg_iters` is set high enough (around NT) that CG is guaranteed to converge. In lines 16 and 28, `max_cholesky_size(0)` ensures the Cholesky decomposition is not used for small data sizes within the iterative implementation, in contrast to default GPyTorch behavior. `max_preconditioner_size(0)` turns off GPyTorch's default memory-intensive preconditioner.

Chapter 4

Scalable Bayesian GPFA with Automatic Relevance Determination

While iterative GPFA (Chapter 3) provides a method for applying GPFA to larger datasets than previously possible, it still eventually hits memory limits, and cannot be used with intractable noise models such as the Poisson and Negative Binomial noise models which are more appropriate for neuroscientific data. In this chapter, I discuss a project I got involved with partway through developing the "exact" iterative method for implementing scalable GFPA described in Chapter 3, which instead uses an approximate variational approach for implementing scalable GPFA. The variational method enables better scaling without, as it turns out, a loss of performance compared to my iterative method (see Chapter 5). This variational approach also enables noise models more appropriate to neuroscientific data that are otherwise intractable using "exact" methods, such as the Poisson and Negative Binomial noise models. While it is possible that iterative GPFA could be improved to scale better by preconditioning CG, the method described here already scales arbitrarily given data that can fit in memory, because the variational approach allows batching computations by time in addition to other benefits.

In particular, the variational approach allows us to introduce an otherwise intractable prior over the mixing matrix $\boldsymbol{C}$ rather than treating it as a set of hyperparameters to be learned. This results in a fully Bayesian version of GPFA, which we term bGPFA. We also incorporate automatic relevance determination to infer the dimensionality of the latent space directly rather than relying on costly cross-validation. We leverage the Toeplitz and Kronecker structure described in Chapter 2 within this approximate variational approach.

This chapter is adapted from a paper submission by Kristopher T. Jensen*, Ta-Chu Kao*, myself, and Guillaume Hennequin (* = equal contribution) [34].

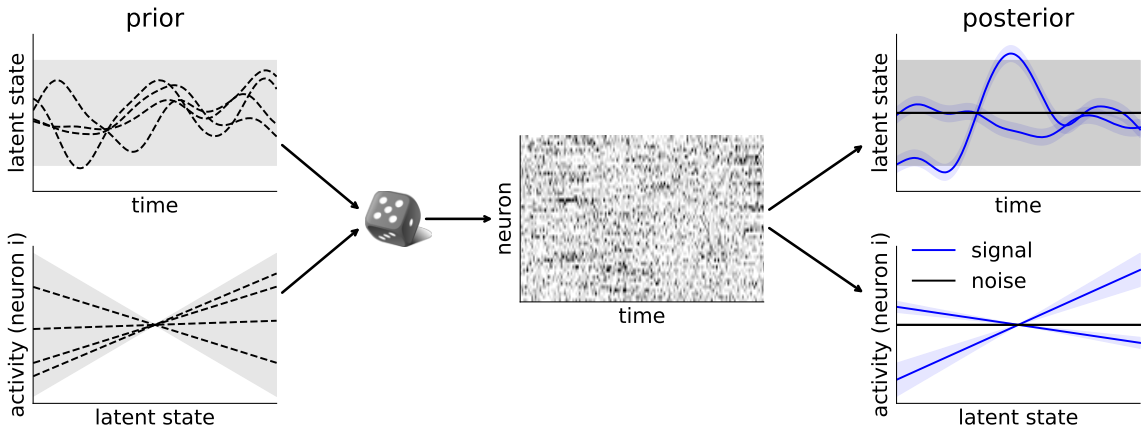


Fig. 4.1 **Bayesian GPFA schematic.** Bayesian GPFA places a Gaussian Process prior over the latent states in each dimension as a function of time $\boldsymbol{t}$ ($p(\boldsymbol{X}|\boldsymbol{t})$; top left) as well as a linear prior over neural activity as a function of each latent dimension ($p(\boldsymbol{F}|\boldsymbol{X})$; bottom left). Together with a stochastic noise process $p(\boldsymbol{Y}|\boldsymbol{F})$, which can be discrete for electrophysiological recordings, this forms a generative model that gives rise to observations $\boldsymbol{Y}$ (middle). From the data and priors, bGPFA infers posterior latent states for each latent dimension ($p(\boldsymbol{X}|\boldsymbol{Y})$; top right) as well as a posterior predictive observation model for each neuron ($p(\boldsymbol{Y}_{test}|\boldsymbol{X}_{test},\boldsymbol{Y})$; bottom right). When combined with automatic relevance determination, the model learns to automatically discard superfluous latent dimensions by maximizing the log marginal likelihood of the data (right, black vs. blue).

My contributions to this work include the Toeplitz speed-up, the GPFA model, running the bGPFA model on a version of a continuous reaching dataset that excluded a period where the monkey was resting, as well as contributions to analysis ideas, figure edits, and text edits.

4.1 Introduction

Canonical GPFA is not scalable to time series longer than a few hundred time bins and assumes a Gaussian noise model which is often inappropriate for discrete and non-negative electrophysiological recordings [24]. Here, we address these challenges by formulating a scalable and fully Bayesian version of GPFA (bGPFA; Figure 4.1) with a computational complexity of $\mathcal{O}(D^2T + DT\log T)$ and a memory cost of $\mathcal{O}(D^2T)$. To do this, we introduce an efficiently parameterized variational inference strategy that ensures scalability to long recordings and facilitates the use of non-Gaussian noise models. Additionally, the Bayesian formulation provides a framework for principled model selection based on approximate marginal likelihoods [65]. This allows us to perform automatic relevance determination and thus fit a single model without prior assumptions about the underlying dimensionality, which is instead inferred from the data itself [50, 7].

We validate our method on a small synthetic dataset with Gaussian noise where canonical GPFA is tractable, and we show that bGPFA has comparable performance without requiring cross-validation to select the latent dimensionality. bGPFA also naturally extends to non-Gaussian data where it recovers ground truth parameters and latent trajectories. We then apply bGPFA to longitudinal, multi-area recordings from primary motor (M1) and somatosensory (S1) areas in a monkey self-paced reaching task spanning 30 minutes. bGPFA readily scales to such datasets, and the inferred latent trajectories improve decoding of kinematic variables compared to the raw data. This decoding improves further when taking into account the temporal offset between motor planning encoded by M1 and feedback encoded by S1. We also show that the latent trajectories for M1 converge to consistent regions of state space for a given reach direction at the onset of each individual reach. Importantly, the distance in latent space to this preparatory state from the state at target onset is predictive of reaction times across reaches, similar to previous results in a task that includes an explicit 'motor preparation epoch' where the subject is not allowed to move [1]. This illustrates the functional relevance of such preparatory activity and suggests that motor preparation takes place even when the task lacks well-defined trial structure and externally imposed delay periods, consistent with findings by Lara et al. [42] and Zimnik and Churchland [81]. Finally, we analyze the task relevance of slow latent processes identified by bGPFA which evolve on timescales of several seconds, much larger than the timescales that can be resolved by methods designed for trial-structured data. We find that some of these slow processes are also predictive of reaction time across reaches, and we hypothesize that they reflect task engagement which varies over the course of several reaches.

4.2 Method

In the following, we use the notation $\boldsymbol{A}$ to refer to the matrix with elements a_{ij}. We use $\boldsymbol{a}_k$ to refer to the k$^{\text{th}}$ row *or* column of $\boldsymbol{A}$ with an index running from 1 to K, represented as a column vector.

4.2.1 Generative model

Latent variable models for neural recordings typically model the neural activity $\boldsymbol{Y} \in \mathbb{R}^{N\times T}$ of N neurons at times $\boldsymbol{t} \in \mathbb{R}^{T}$ as arising from shared fluctuations in D latent variables $\boldsymbol{X} \in \mathbb{R}^{D\times T}$. Specifically, the probability of a given recording can be written as

$$p(\boldsymbol{Y}|\boldsymbol{t}) = \int p(\boldsymbol{Y}|\boldsymbol{F})\, p(\boldsymbol{F}|\boldsymbol{X})\, p(\boldsymbol{X}|\boldsymbol{t})\, d\boldsymbol{F}\, d\boldsymbol{X}, \tag{4.1}$$

where $\boldsymbol{F} \in \mathbb{R}^{N \times T}$ are intermediate, neuron-specific variables that can often be thought of as firing rates or a similar notion of noise-free activity. For example, GPFA [78] specifies

$$p(\boldsymbol{Y}|\boldsymbol{F}) = \prod_{n,t} \mathcal{N}(y_{nt}; f_{nt}, \sigma_n^2) \tag{4.2}$$

$$p(\boldsymbol{F}|\boldsymbol{X}) = \delta(\boldsymbol{F} - \boldsymbol{C}\boldsymbol{X}) \tag{4.3}$$

$$p(\boldsymbol{X}|\boldsymbol{t}) = \prod_{d} \mathcal{N}(\boldsymbol{x}_d; \boldsymbol{0}, \boldsymbol{K}_d) \qquad \text{with } \boldsymbol{K}_d = k_d(\boldsymbol{t}, \boldsymbol{t})) \tag{4.4}$$

That is, the prior over the d^{th} latent function $x_d(t)$ is a Gaussian process [73] with covariance function $k_d(\cdot,\cdot)$ (usually a radial basis function), and the observation model $p(\boldsymbol{Y}|\boldsymbol{X})$ is given by a parametric linear transformation with independent Gaussian noise.

In this work, we additionally introduce a prior distribution over the mixing matrix $\boldsymbol{C} \in \mathbb{R}^{N \times D}$ with hyperparameters specific to each latent dimension. This allows us to *learn* an appropriate latent dimensionality for a given dataset using automatic relevance determination (ARD) similar to previous work in Bayesian PCA (Appendix B.8; 7) rather than relying on cross-validation or ad-hoc thresholds of variance explained. Unlike in standard GPFA, the log marginal likelihood (Equation 4.1) becomes intractable with this prior. We therefore develop a novel variational inference strategy [70] which also (i) provides a scalable implementation appropriate for long continuous neural recordings, and (ii) extends the model to general non-Gaussian likelihoods better suited for discrete spike counts.

In this new framework, which we call Bayesian GPFA (bGPFA), we use a Gaussian prior over $\boldsymbol{C}$ of the form $c_{nd} \sim \mathcal{N}(0, s_d^2)$, where s_d is a scale parameter associated with latent dimension d. Integrating $\boldsymbol{C}$ out in Equation 4.3 then yields the following observation model (recalling that $\boldsymbol{F}$ can be thought of as firing-rates or similar noise-free activity, and $\boldsymbol{X}$ are the latent variables):

$$p(\boldsymbol{F}|\boldsymbol{X}) = \prod_{n} \mathcal{N}(\boldsymbol{f}_n; 0, \boldsymbol{X}^T \boldsymbol{S}^2 \boldsymbol{X}), \qquad \text{with } \boldsymbol{S} = \text{diag}(s_1, \ldots, s_D). \tag{4.5}$$

Moreover, we use a general noise model $p(\boldsymbol{Y}|\boldsymbol{F}) = \prod_{n,t} p(y_{nt}|f_{nt})$ where $p(y_{nt}|f_{nt})$ is any distribution for which we can evaluate its density.

4.2.2 Variational inference and learning

To train the model and infer both $\boldsymbol{X}$ and $\boldsymbol{F}$ from the data $\boldsymbol{Y}$, we use a nested variational approach. It is intractable to compute $\log p(\boldsymbol{Y}|\boldsymbol{t})$ (Equation 4.1) analytically for bGPFA, and we therefore introduce a lower bound on $\log p(\boldsymbol{Y}|\boldsymbol{t})$ at the outer level and another one on

$\log p(\boldsymbol{Y}|\boldsymbol{X})$ at the inner level. These lower bounds are constructed from approximations to the posterior distributions over latents ($\boldsymbol{X}$) and noise-free activity ($\bar{\boldsymbol{F}}$) respectively.

Distribution over latents At the outer level, we introduce a variational distribution $q(\boldsymbol{X})$ over latents and construct an evidence lower bound (ELBO; 70) on the log marginal likelihood of Equation 4.1:

$$\log p(\boldsymbol{Y}|\boldsymbol{t}) \geq \mathcal{L} := \mathbb{E}_{q(\boldsymbol{X})}\left[\log p(\boldsymbol{Y}|\boldsymbol{X})\right] - \mathrm{KL}\left[q(\boldsymbol{X})||p(\boldsymbol{X}|\boldsymbol{t})\right]. \tag{4.6}$$

Conveniently, maximizing this lower bound is equivalent to minimizing $\mathrm{KL}\left[q(\boldsymbol{X})||p(\boldsymbol{X}|\boldsymbol{Y})\right]$ and thus also yields an approximation to the posterior over latents in the form of $q(\boldsymbol{X})$. We estimate the first term of the ELBO using Monte Carlo samples from $q(\boldsymbol{X})$ and compute the KL term analytically.

Here, we use a so-called whitened parameterization of $q(\boldsymbol{X})$ [29] that is both expressive and scalable to large datasets:

$$q(\boldsymbol{X}) = \prod_{d=1}^{D} \mathcal{N}(\boldsymbol{x}_d; \boldsymbol{\mu}_d, \boldsymbol{\Sigma}_d) \quad \text{with} \quad \boldsymbol{\mu}_d = \boldsymbol{K}_d^{\frac{1}{2}} \boldsymbol{\nu}_d \quad \text{and} \quad \boldsymbol{\Sigma}_d = \boldsymbol{K}_d^{\frac{1}{2}} \boldsymbol{\Lambda}_d \boldsymbol{\Lambda}_d^T \boldsymbol{K}_d^{\frac{1}{2}^T} \tag{4.7}$$

where $\boldsymbol{K}_d^{\frac{1}{2}}$ is any square root of the prior covariance matrix $\boldsymbol{K}_d$, and $\boldsymbol{\nu}_d \in \mathbb{R}^T$ is a vector of variational parameters to be optimized. $\boldsymbol{\Lambda}_d \in \mathbb{R}^{T \times T}$ is a positive semi-definite (to reduce degeneracy) variational matrix whose structure is chosen carefully so that its squared Frobenius norm, log determinant, and matrix-vector products can all be computed efficiently which facilitates the evaluation of Equations 4.8 and 4.9. This whitened parameterization has several advantages. First, it does not place probability mass where the prior itself does not. In addition to stabilizing learning [49], this also guarantees that the posterior is temporally smooth for a smooth prior. Second, the KL term in Equation 4.6 simplifies to

$$\mathrm{KL}[q(\boldsymbol{X})||p(\boldsymbol{X}|\boldsymbol{t})] = \frac{1}{2} \sum_d \left(||\boldsymbol{\Lambda}_d||_{\mathrm{F}}^2 - 2 \log |\boldsymbol{\Lambda}_d| + ||\boldsymbol{\nu}_d||^2 - T \right). \tag{4.8}$$

Third, $q(\boldsymbol{X})$ can be sampled efficiently via a differentiable transform (i.e. the reparameterization trick) provided that fast differentiable $\boldsymbol{K}_d^{\frac{1}{2}} \boldsymbol{\nu}$ and $\boldsymbol{\Lambda}_d \boldsymbol{\nu}$ products are available for any vector $\boldsymbol{\nu}$:

$$\boldsymbol{x}_d^{(m)} = \boldsymbol{K}_d^{\frac{1}{2}} (\boldsymbol{\nu}_d + \boldsymbol{\Lambda}_d \boldsymbol{\eta}_d) \qquad \text{with} \qquad \boldsymbol{\eta}_d \sim \mathcal{N}(\boldsymbol{0}, \boldsymbol{I}), \tag{4.9}$$

where $\boldsymbol{x}_d^{(m)} \sim q(\boldsymbol{x}_d)$. This is important to form a Monte Carlo estimate of $\mathbb{E}_{q(\boldsymbol{X})}\left[\log p(\boldsymbol{Y}|\boldsymbol{X})\right]$.

To avoid the challenging computation of $\boldsymbol{K}_d^{\frac{1}{2}}\boldsymbol{v}$ for general $\boldsymbol{K}_d$ [2], we directly parameterize $\boldsymbol{K}_d^{\frac{1}{2}}$, the positive definite square root of $\boldsymbol{K}$, which implicitly defines the prior covariance function $k_d(\cdot,\cdot)$. In this work we use an RBF kernel for $\boldsymbol{K}_d$ and give the expression for $\boldsymbol{K}_d^{\frac{1}{2}}$ in Appendix B.5. Additionally, we use Toeplitz acceleration methods to compute $\boldsymbol{K}_d^{\frac{1}{2}}\boldsymbol{v}$ products in $\mathcal{O}(T\log T)$ time and with $\mathcal{O}(T)$ memory cost [75, 60]. We implement and compare different choices of $\boldsymbol{\Lambda}_d$ in Appendix B.5. For the experiments in this work, we use the following parameterization:

$$\boldsymbol{\Lambda}_d = \boldsymbol{\Psi}_d \boldsymbol{C}_d \tag{4.10}$$

where $\boldsymbol{\Psi}_d$ is diagonal with positive entries and $\boldsymbol{C}_d$ is circulant, symmetric, and positive definite. This parameterization enables cheap computation of KL divergences and matrix-vector products while maintaining sufficient expressiveness (Appendix B.5).

Distribution over neural activity Evaluating $\log p(\boldsymbol{Y}|\boldsymbol{X}) = \sum_n \log p(\boldsymbol{y}_n|\boldsymbol{X})$ for each sample drawn from $q(\boldsymbol{X})$ is intractable for general noise models. Thus, we further lower-bound the ELBO of Equation 4.6 by introducing an approximation $q(\boldsymbol{f}_n|\boldsymbol{X})$ to the posterior $p(\boldsymbol{f}_n|\boldsymbol{y}_n,\boldsymbol{X})$:

$$\log p(\boldsymbol{y}_n|\boldsymbol{X}) \geq \mathbb{E}_{q(\boldsymbol{f}_n|\boldsymbol{X})}\left[\log p(\boldsymbol{y}_n|\boldsymbol{f}_n)\right] - \mathrm{KL}\left[q(\boldsymbol{f}_n|\boldsymbol{X})||p(\boldsymbol{f}_n|\boldsymbol{X})\right]. \tag{4.11}$$

We repeat the whitened variational strategy described at the outer level by writing

$$q(\boldsymbol{f}_n|\boldsymbol{X}) = \mathcal{N}(\boldsymbol{f}_n;\hat{\boldsymbol{\mu}}_n,\hat{\boldsymbol{\Sigma}}_n) \quad \text{with} \quad \hat{\boldsymbol{\mu}}_n = \hat{\boldsymbol{K}}^{\frac{1}{2}}\hat{\boldsymbol{v}}_n \quad \text{and} \quad \hat{\boldsymbol{\Sigma}}_n = \hat{\boldsymbol{K}}^{\frac{1}{2}}\boldsymbol{L}_n\boldsymbol{L}_n^T\hat{\boldsymbol{K}}^{\frac{1}{2}}, \tag{4.12}$$

where $\hat{\boldsymbol{v}}_n \in \mathbb{R}^D$ is a neuron-specific vector of variational parameters to be optimized along with a lower-triangular matrix $\boldsymbol{L}_n \in \mathbb{R}^{D\times D}$; and $\hat{\boldsymbol{K}}$ denotes the covariance matrix of $p(\boldsymbol{f}|\boldsymbol{X})$, whose square root $\hat{\boldsymbol{K}}^{\frac{1}{2}} = \boldsymbol{X}^T\boldsymbol{S}$ follows from Equation 4.5. The low-rank structure of $\hat{\boldsymbol{K}}$ enables cheap matrix-vector products and KL divergences:

$$\mathrm{KL}[q(\boldsymbol{f}_n|\boldsymbol{X})||p(\boldsymbol{f}_n|\boldsymbol{X})] = \frac{1}{2}\left(||\boldsymbol{L}_n||_{\mathrm{F}}^2 - 2\log|\boldsymbol{L}_n| + ||\hat{\boldsymbol{v}}_d||^2 - D\right). \tag{4.13}$$

Note that the KL divergence does not depend on $\boldsymbol{X}$ in this whitened parameterization (Appendix B.7). Moreover, $q(\boldsymbol{f}_n|\boldsymbol{X})$ in Equation 4.12 has the form of the exact posterior when the noise model is Gaussian (Appendix B.6), and it is equivalent to a stochastic variational inducing point approximation [28] for general noise models (Appendix B.7).

Finally, we need to compute the first term in Equation 4.11:

$$\mathbb{E}_{q(\boldsymbol{f}_n|\boldsymbol{X})}\left[\log p(\boldsymbol{y}_n|\boldsymbol{f}_n)\right] = \sum_t \mathbb{E}_{q(f_{nt}|\boldsymbol{X})}\left[\log p(y_{nt}|f_{nt})\right]. \tag{4.14}$$

Each term in this sum is simply a 1-dimensional Gaussian expectation which can be computed analytically in the case of Gaussian or Poisson noise (with an exponential link function), and otherwise approximated efficiently using Gauss-Hermite quadrature (Appendix B.10; 28).

4.2.3 Summary of the algorithm

Putting Section 4.2.1 and Section 4.2.2 together, optimization proceeds at each iteration by drawing M Monte Carlo samples $\{\boldsymbol{X}_m\}_1^M$ from $q(\boldsymbol{X})$ and estimating the overall ELBO as:

$$\begin{aligned}\mathcal{L} = \frac{1}{M}\sum_{\boldsymbol{X}_m\sim q(\boldsymbol{X})}&\left[\sum_{n,t}\mathbb{E}_{q(f_{nt}|\boldsymbol{X}_m)}\left[\log p(y_{nt}|f_{nt})\right]\right] \\ &-\sum_n \mathrm{KL}\left[q(\boldsymbol{f}_n)||p(\boldsymbol{f}_n)\right] - \sum_d \mathrm{KL}\left[q(\boldsymbol{x}_d)||p(\boldsymbol{x}_d)\right],\end{aligned} \tag{4.15}$$

where the expectation over $q(f_{nt}|\boldsymbol{X})$ is evaluated analytically or using Gauss-Hermite quadrature depending on the noise model (Appendix B.10). We maximize $\mathcal{L}$ with respect to $\theta = \{\{s_d\}_1^D, \{\tau_d\}_1^D, \{\boldsymbol{\nu}_d\}_1^D, \{\tilde{\boldsymbol{c}}_d\}_1^D, \{\boldsymbol{\Psi}_d\}_1^D, \{\boldsymbol{L}_n\}_1^N, \{\hat{\boldsymbol{\nu}}_n\}_1^N$ and the parameters for the noise model of choice using stochastic gradient ascent with Adam [39]. This has a total computational time complexity of $\mathcal{O}(MNTD^2 + MDT\log T)$ and memory complexity of $\mathcal{O}(MNTD^2)$ where N is the number of neurons, T the number of time points, and D the latent dimensionality. For large datasets such as the monkey reaching data in Section 4.3.2, we compute gradients with respect to θ and the noise model parameters using mini-batches across time to mitigate the memory costs; that is, gradients for the sum over t in Equation 4.15 are computed in multiple passes. The algorithm is described in pseudocode with further implementation and computational details in Appendix B.11, and is available on github at https://github.com/tachukao/mgplvm-pytorch/tree/bgpfa. The model learned by bGPFA can subsequently be used for predictions on held-out data by conditioning on partial observations as used for cross-validation in Section 4.3.1 and discussed in Appendix B.12. Latent dimensions that have been 'discarded' by automatic relevance determination will automatically have negligible contributions to the resulting posterior predictive distribution since the prior scale parameters s_d are approximately zero for these dimensions (see Appendix B.8 for details).

4.3 Experiments and results

In this section we apply bGPFA with automatic relevance determination to synthetic and biological data in order to validate the method and highlight its utility for neuroscience research.

4.3.1 Synthetic data

We first generated an example dataset from the GPFA generative model (Equations 4.2-4.4) with a true latent dimensionality of 3. We proceeded to fit both factor analysis (FA), GPFA, and bGPFA with different latent dimensionalities $D \in [1, 10]$. Here, we fitted bGPFA without automatic relevance determination such that $s_d = s \forall d$. As expected, the marginal likelihoods increased monotonically with D for both FA and GPFA (Figure 4.2A; Appendix B.8). In contrast, the bGPFA ELBO reached its optimum value at the true latent dimensionality $D^\star = 3$. This is a manifestation of "Occam's razor", whereby fully Bayesian approaches favor the simplest model that adequately explains the data $\boldsymbol{Y}$ [45]. This is also confirmed by the cross-validated predictive performance which was optimal at $D = 3$ for all methods (Figure 4.2B). Notably, the introduction of ARD parameters $\{s_d\}$ in bGPFA allowed us to fit a single model with large $D = 10$. This model simultaneously achieved both the maximum ELBO and minimum test error obtained by bGPFA without ARD at $D^\star = 3$ (Figure 4.2A and B, blue) without *a priori* assumptions about the latent dimensionality or the need to perform extensive cross-validation. Consistent with the ground truth generative process, only 3 of the scale parameters s_d remained well above zero after training (Figure 4.2B, inset).

We then proceeded to apply bGPFA ($D = 10$) to an example dataset drawn using Equations 4.4 and 4.5 with a ground truth dimensionality $D^\star = 2$, and either Gaussian, Poisson, or negative binomial noise. For all three datasets, the learned parameters clustered into a group of two latent dimensions with high information content (Appendix B.9) and a group of eight uninformative dimensions, consistent with the generative process (Figure 4.2C). In each case, we extracted the inferred latent trajectories corresponding to the informative dimensions and found that they recapitulated the ground truth up to a linear transformation (Figure 4.2D). Fitting flexible noise models such as the negative binomial model is important because neural firing patterns are known to be overdispersed in many contexts [66, 23, 5]. However, it is often unclear how much of that overdispersion should be attributed to common fluctuations in hidden latent variables ($\boldsymbol{X}$ in our model) compared to private noise processes in single neurons [44]. In our synthetic data with negative binomial noise, we could accurately recover the single-neuron overdispersion parameters (Figure 4.2E; Appendix B.10), suggesting that

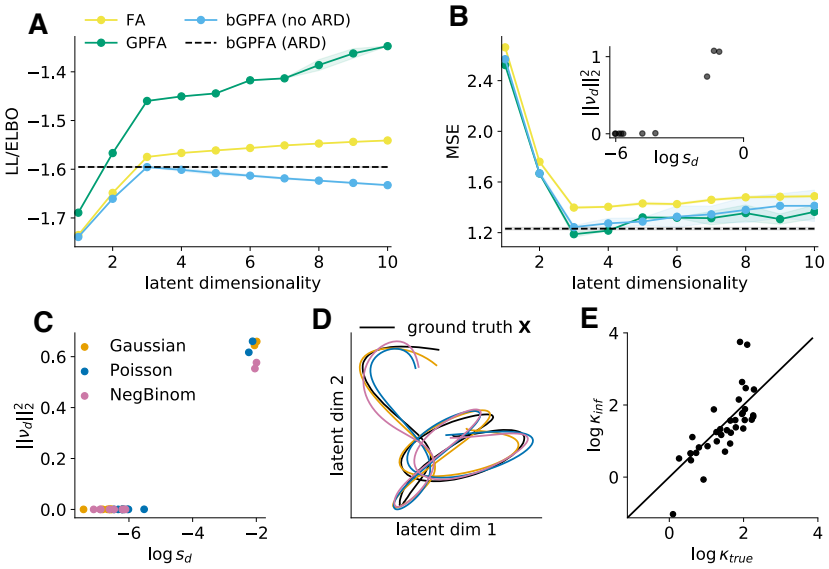


Fig. 4.2 **Bayesian GPFA applied to synthetic data. (A)** Log likelihoods of factor analysis (yellow) and GPFA (green) and ELBO of Bayesian GPFA without ARD (blue) fitted to synthetic data with a ground truth dimensionality of three for different model dimensionalities. FA and GPFA exhibit monotonically increasing marginal likelihoods while the ELBO of Bayesian GPFA has a maximum corresponding to the true latent dimensionality. bGPFA with ARD recovered this three-dimensional latent space as well as the optimum ELBO of bGPFA without ARD (black dashed line). **(B)** Cross-validated prediction errors for the models in (a) (Appendix B.12). The minimum is at $D^\star = 3$ for all methods, consistent with the maximum of the bGPFA ELBO without ARD in (A). bGPFA with ARD recovered the performance of the optimal bGPFA model without requiring a search over latent dimensionalities. Inspection of the learned prior scales $\{s_d\}$ and posterior mean parameters $||\boldsymbol{v}_d||_2^2$ (inset) indicates that ARD retained only $D^\star = 3$ informative dimensions (top right) and discarded the other 7 dimensions (bottom left). Shadings in (A) and (B) indicate ± 2 stdev. across 10 model fits. **(C)** Learned hyperparameters of bGPFA with ARD and either Gaussian, Poisson or negative binomial noise models fitted to two-dimensional synthetic datasets with observations drawn from the corresponding noise models (Appendix B.10). The hyperparameters clustered into two groups of informative (top right) and non-informative (bottom left) dimensions (Appendix B.9). **(D)** Latent trajectory in the space of the two most informative dimensions (c.f. (C)) for each model with the ground truth shown in black. **(E)** The overdispersion parameter κ_n for each neuron learned in the negative binomial model, plotted against the ground truth (Appendix B.10). Solid line indicates $y = x$; note that $\kappa_n \to \infty$ corresponds to a Poisson noise model.

such unsupervised models have the capacity to resolve overdispersion due to private and shared processes.

In summary, bGPFA provides a flexible method for inferring both latent dimensionalities, latent trajectories, and heterogeneous single-neuron parameters in an unsupervised manner.

In the next section, we show that the scalability of the model and its interpretable parameters facilitate the analysis of large neural population recordings.

4.3.2 Primate recordings

In this section, we apply bGPFA to biological data recorded from a rhesus macaque during a self-paced reaching task with continuous recordings spanning 30 minutes (52, 47; Figure 4.3A). The continuous nature of these recordings as one long trial makes it a challenging dataset for existing analysis methods that explicitly require the availability of many trials per experimental condition [53], and poses computational challenges to Gaussian process-based methods that cannot handle long time series [78]. While the ad-hoc division of continuous recordings into surrogate trials can still enable the use of these methods [38], here we show that our formulation of bGPFA readily applies to long continuous recordings. We fitted bGPFA with a negative binomial noise model to recordings from both primary motor cortex (M1) and primary somatosensory cortex (S1). For all analyses, we used a single recording session (`indy_20160426`, as in 38), excluded neurons with overall firing rates below 2 Hz, and binned data at 25 ms resolution. This resulted in a data array $\boldsymbol{Y} \in \mathbb{R}^{200\times 70482}$ (130 M1 neurons and 70 S1 neurons).

We first fitted bGPFA independently to the M1 and S1 sub-populations with $D = 25$ latent dimensions. In this case, ARD retained 20 (M1) and 13 (S1) dimensions (Figure 4.3B). We then proceeded to train a linear decoder to predict hand kinematics in the form of x and y hand velocities from either the inferred firing rates or the raw data convolved with a 50 ms Gaussian kernel (38; Appendix B.12). We found that the model learned by bGPFA predicted kinematics better than the convolved spike trains, suggesting that (i) the latent space accurately captures kinematic representations, and (ii) the denoising and data-sharing across time in bGPFA aids decodability beyond simple smoothing of neural activity. Interestingly, by repeating this decoding analysis with an artificially imposed delay between neural activity and decoded behavior, we found that neurons in S1 predominantly encoded current behavior while neurons in M1 encoded a motor plan that predicted kinematics 100-150 ms into the future (Figure 4.3B). This is consistent with the motor neuroscience literature suggesting that M1 functions as a dynamical system driving behavior via downstream effectors [12].

We then fitted bGPFA to the entire dataset including both M1 and S1 neurons and found that kinematic predictions improved over individual M1- and S1-based predictions (Figure 4.3B). In this analysis, the decoding performance as a function of delay between neural activity and behavior exhibited a broader peak than for the single-region decoding. We hypothesized that this broad peak reflects the fact that these neural populations encode both *current* behavior in S1 as well as *future* behavior in M1 (Figure 4.3C). Indeed when

we took this offset into account by shifting all M1 spike times by +100 ms and retraining the model, decoding performance increased ($69.22\% \pm 0.06$ vs. $67.84\% \pm 0.07$ mean $\pm$ sem variance explained across ten model fits; Appendix B.12). Additionally, the shifted data exhibited a narrower decoding peak now attained for near-zero delay between kinematics and latent trajectories (Figure 4.3D). Consistent with the improved kinematic decoding, we also found that shifting M1 spikes by 100 ms increased the ELBO per neuron ($-34\,882.77 \pm 0.39$ vs. $-34\,893.18 \pm 0.71$) and approximately minimized the linear dimensionality of the data (Appendix B.4; 58).

We next wondered if bGPFA could be used to reveal putative motor preparation processes, which is non-trivial due to the lack of trial structure and well-defined preparatory epochs. We partitioned the data post-hoc into individual 'reaches', each consisting of a period of time where the target location remained constant. For these analyses, we only considered 'successful' reaches where the monkey eventually moved to the target location (Appendix B.3), and we defined movement onset as the first time during a reach where the cursor speed exceeded a low threshold (Appendix B.1). We began by visualizing the latent processes inferred by bGPFA as they unfolded prior to movement onset in each reach epoch. For visualization purposes, we ranked the latent dimensions based on their learned prior scales (a measure of variance explained; Appendix B.9) and selected the first two. Prior to movement onset, the latent trajectories tended to progress from their initial location at target onset towards reach-specific regions of state space (see example trials in Figure 4.3E for leftward and rightward reaches). To quantify this phenomenon, we computed pairwise similarities between latent states across all 681 reaches, during (i) stimulus onset and (ii) 75 ms before movement onset (chosen such that it is well before any detectable movement; Appendix B.1). We defined similarity as the negative Euclidean distance between latent states and restricted the analysis to 'fast' latent dimensions with timescales smaller than 200 ms to study this putatively fast process. When plotted as a function of reach direction, the latent similarities at target onset showed little discernable structure (Figure 4.3F, left). In contrast, the pairwise similarities became strongly structured 75 ms before movement onset where neighboring reach directions were associated with similar preparatory latent states (Figure 4.3F, right). Similar, albeit noisier, results were found when using factor analysis instead of bGPFA (Appendix B.1). These findings are consistent with previous reports of monkey M1 partitioning preparatory and movement-related activity into distinct subspaces [22, 42], as well as with the analogous finding that a 'relative target' subspace is active before a 'movement subspace' in previous analyses of this particular dataset [38].

Previous work on delayed reaches has shown that monkeys start reaching earlier when the neural state attained at the time of the go cue – which marks the end of a delay period

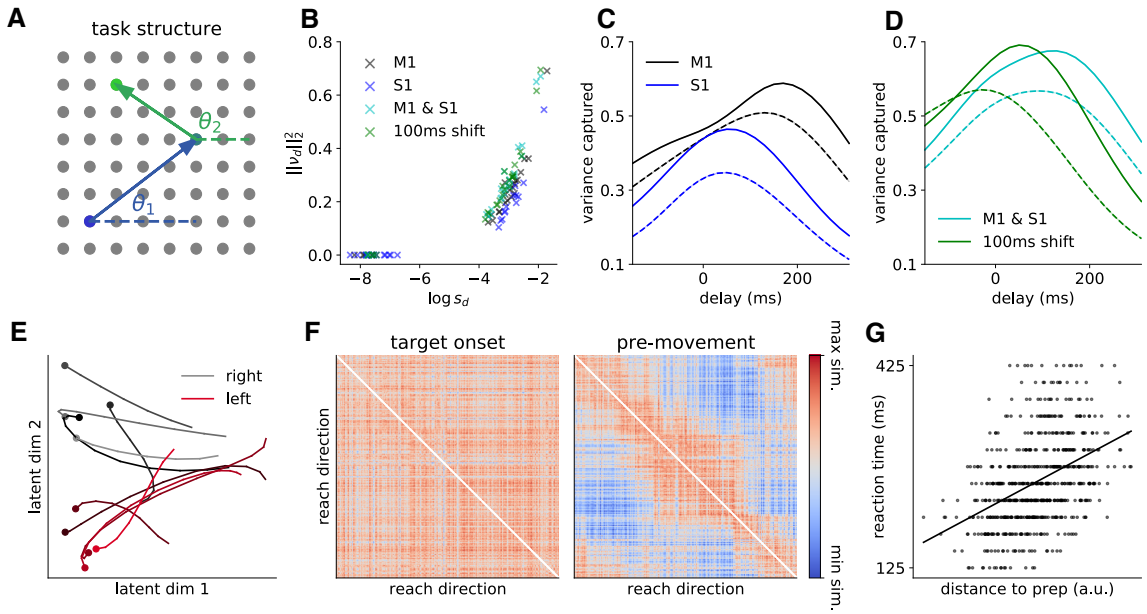


Fig. 4.3 **Bayesian GPFA applied to primate data.** **(A)** Schematic illustration of the self-paced reaching task. When a target on a 17x8 grid is reached (arrows), a new target lights up on the screen (colours), selected at random from the remaining targets (8x8 grid shown for clarity). In several analyses, we classify movements according to reach angle measured relative to horizontal (θ_1, θ_2). **(B)** Learned mean and scale parameters for the bGPFA models. Small prior scales s_d and posterior mean parameters ($||\boldsymbol{v}_d||_2^2$) indicate uninformative dimensions (Appendix B.9). **(C)** We applied bGPFA to monkey M1 and S1 data during the task and trained a linear model to decode kinematics from firing rates predicted from the inferred latent trajectories with different delays between latent states and kinematics. Neural activity was most predictive of future behavior in M1 (black) and current behavior in S1 (blue). Dashed lines indicate decoding from the raw data convolved with a Gaussian filter. **(D)** Decoding from bGPFA applied to the combined M1 and S1 data (cyan). Performance improved further when decoding from latent trajectories inferred from data where M1 activity was shifted by 100 ms relative to S1 activity (green). **(E)** Example trajectories in the two most informative latent dimensions for five rightward reaches (grey) and five leftward reaches (red). Trajectories are plotted from the appearance of the stimulus until movement onset (circles). During 'movement preparation', the latent trajectories move towards a consistent region of latent state space for each reach direction. **(F)** Similarity matrix of the latent state at stimulus onset showing no obvious structure (left) and 75 ms prior to movement onset showing modulation by reach direction (right). **(G)** Reaction time plotted against Euclidean distance between the latent state at target onset and the mean preparatory state for the corresponding reach direction ($\rho = 0.424$).

with a known reach direction – is close to an "optimal subspace" [1, 35]. We wondered if a similar effect takes place during continuous, self-initiated reaching in the absence of explicit delay periods. Based on Figure 4.3E, we hypothesized that the monkey should start

moving earlier if, at the time the next target is presented, its latent state is already close to the mean preparatory state for the required next movement direction. To test this, we extracted the mean preparatory state 75 ms prior to movement onset (as above) for each reach direction in the dataset. We found that the distance between the latent state at target onset and the corresponding mean preparatory state was strongly predictive of reaction time (RT; Figure 4.3G, Pearson $\rho = 0.424$, $p = 3 \times 10^{-29}$). Such a correlation was also weakly present with factor analysis ($\rho = 0.21$, $p = 9 \times 10^{-8}$) but not detectable in the raw data ($\rho = 0.02$, $p = 0.6$). We also verified that the strong correlation found with bGPFA was not an artifact of the temporal correlations introduced by the prior (Appendix B.2). Taken together, our results suggest that motor preparation is an important part of reaching movements even in an unconstrained self-paced task. Additionally, we showed that bGPFA captures such behaviorally relevant latent dynamics better than simpler alternatives, and our scalable implementation enables its use on the large continuous reaching dataset analysed here.

Finally we noted that some latent dimensions had long timescales on the order of 2 seconds, which is longer than the timescale of individual reaches (1-2 seconds; Appendix B.2). We hypothesized that these slow dynamics might reflect motivation or task engagement. Consistent with this hypothesis, we found that the slowest latent process ($\tau = 2.1$ s) was correlated with reaction time during successful reaches (Pearson $\rho = 0.383$, $p = 1.1 \times 10^{-23}$) and strongly modulated during a longer period of time where the monkey did not reach to any targets (Appendix B.3). Interestingly, the information contained about reaction time in this long timescale latent dimension was largely complementary to that encoded by the distance to preparatory states in the 'fast' dimensions (Appendix B.2). We thus find that bGPFA is capable of capturing not only single-reach dynamics and preparatory activity but also processes on longer timescales that would be difficult to identify with methods designed for the analysis of many shorter trials.

4.4 Discussion

Related work The generative model of bGPFA can be considered an extension of the canonical GPFA model proposed by Yu et al. [78] to include a Gaussian prior over the loading matrix $\boldsymbol{C}$ (Section 4.2.1). In this view, bGPFA is to GPFA what Bayesian PCA is to PCA [7]; in particular, it facilitates automatic relevance determination to infer the dimensionality of the latent space from data [7, 50, 65]. Additionally, we utilize advances in variational inference [40, 59] to make the algorithm scalable to the large datasets recorded in modern neuroscience. In particular, we contribute a novel form of circulant variational

GP posterior that is both accurate and scalable. Similar to previous work by Duncker and Sahani [19] and Zhao and Park [80], variational inference also facilitates the use of arbitrary observation noise models, including non-Gaussian models more appropriate for electrophysiological recordings. Furthermore, our method is an extension of work on Gaussian process latent variable models (GPLVMs) [43, 65] which have recently found use in the neuroscience literature as a way of modelling flexible, nonlinear tuning curves [77, 33]. This is because integrating out the loading matrix in $p(\boldsymbol{Y}|\boldsymbol{X})$ with a Gaussian prior gives rise to a Gaussian process, here with a linear kernel. The low-rank structure of this linear kernel yields computationally cheap likelihoods, and our variational approach is equivalent to the sparse inducing point approximation used in the stochastic variational GP (SVGP) framework [27, 28]. In particular, our variational posterior is the same as that which would arise in SVGP with at least D inducing points irrespective of where those inducing points are placed (Appendix B.7). We also note that for a Gaussian noise model, the resulting low-rank Gaussian posterior is in fact the form of the exact posterior distribution (Appendix B.6). Additionally, since in bGPFA both the prior over latents and the observation model are GPs, bGPFA is an example of a deep GP [16], in this case with two layers that use an RBF kernel and a linear kernel respectively. Finally, our parameterizations of the posteriors $q(\boldsymbol{x}_d)$ and $q(\boldsymbol{f}_n)$ can be viewed as variants of the 'whitening' approach introduced by Hensman et al. [29] which both facilitates efficient computation of the KL terms in the ELBOs and also stabilizes training (Section 4.2.2).

Conclusion In summary, bGPFA is an extension of the popular GPFA model in neuroscience that allows for regularized, scalable inference and automatic determination of the latent dimensionality as well as the use of non-Gaussian noise models more appropriate for neural recordings. Importantly, the hyperparameters of bGPFA are efficiently optimized based on the ELBO on training data which alleviates the need for cross-validation or complicated algorithms otherwise used for hyperparameter optimization in overparameterized models [33, 77, 78, 37, 38, 24]. Our approach can also be extended in several ways to make it more useful to the neuroscience community. For example, replacing the spike count-based noise models with a point process model would provide higher temporal resolution [19], and facilitate inference of optimal temporal delays across neural populations [41] which will likely be useful as multi-region recordings become more prevalent in neuroscience [36]. Additionally, by substituting the linear kernel in $p(\boldsymbol{Y}|\boldsymbol{X})$ for an RBF kernel in Euclidean space [77] or on a non-Euclidean manifold [33], we can recover scalable versions of recent GPLVM-based tools for neural data analyses with automatic relevance determination.

Chapter 5

Application and Comparison of Scalable GPFA Methods

In this chapter, I compare the methods discussed in Chapters 3 and 4 on real neural data.

5.1 Self-paced reaching in non-human primates

In Section 4.3.1 we showed that, with synthetic data, bGPFA with ARD recovers the correct dimensionality. Here I repeat this analysis on real primate reaching data on models with Gaussian likelihoods, and compare the performance of iterative GPFA (Section 3.2.2) to that of bGPFA. Overall, bGPFA performs well both with and without ARD, and ARD retains a few more dimensions than could be inferred by model selection of the non-ARD version based on cross-validated mean squared prediction error (Figure 5.1B). Iterative GPFA, however, does not perform as well as either bGPFA models.

5.1.1 Details

I trained models on 125 seconds of data with 25 ms resolution ($T = 5000$). I used the first 125 seconds of the "indy_20160627_01" dataset which consists of an hour-long recording. I subselected neurons with a firing rate of 2Hz or more over the course of the recording, resulting in 106 neurons. I took the square root of the spike counts to make the data more appropriate for a Gaussian noise model and centered the data for each neuron before extracting the first 125 seconds of activity.

I initialized all models using factor analysis, as follows. For iterative GPFA, the initial $\boldsymbol{C}$ mixing matrix was set to a randomly rotated version of the loading matrix found by factor analysis. For bGPFA, which does not represent the $\boldsymbol{C}$ matrix explicitly, factor analysis was

used to initialize both the variational distribution over the latents and the ARD scale factors. I trained five instances of each model with different random seeds, affecting the random unitary transform of $\boldsymbol{C}$, Monte Carlo sampling of the variational posterior in bGPFA, and the probe vectors used for trace estimation in iterative GPFA.

The mean squared error in Figure 5.1B was computed on the subsequent 125 seconds of data from the recording, computing the posterior over latents based on half of the neurons, and calculating the mean squared error on posterior predictions for the other half of the neurons.

bGPFA models with and without ARD were trained using 7501 training steps while iterative GPFA was trained for 2001 training steps for reasons discussed in Section 5.1.3. Inference was performed by re-optimizing the latent variational mean and covariance ($\hat{\boldsymbol{\mu}}_n$ and $\hat{\boldsymbol{\sigma}}_n^2$ respectively in Algorithm 2) over 3751 optimization steps, having frozen all other model parameters. In contrast, iterative GPFA enables direct inference involving a single CG solve, without requiring further gradient-based optimization.

Note that I ran bGPFA with ARD with $D = 30$.

5.1.2 Results

Iterative GPFA had a marginal log likelihood that increased with the number of dimensions, whereas bGPFA without ARD had a marginal log likelihood that was maximized at $D = 7$. This is to be expected as bGPFA is Bayesian over the mixing matrix, whereas iterative GPFA is not. bGPFA also had a higher marginal log likelihood overall—this result was unexpected as GPFA had a higher marginal log likelihood than bGPFA in Section 4.3.1 and the marginal log likelihood shown for bGPFA is actually the lower bound on the marginal log likelihood. This discrepancy result from the biased approximation to the log determinant used in iterative GPFA (Table 3.1).

Iterative GPFA was significantly slower than bGPFA (Figure 5.1C) and had a higher cross-validated mean squared prediction error relative to either bGPFA with or without ARD (Figure 5.1B). The speed difference is due to the expensive repeated iterative calculations made within the CG algorithm at each training step of iterative GPFA. While bGPFA and iterative GPFA scaled similarly with D, the scaling factor is much larger for iterative GPFA. The difference in mean squared prediction error is likely due to poor convergence in iterative GPFA (discussed further in Section 5.1.3).

The difference in memory usage shown in Figure 5.1D between bGPFA with and without ARD can be explained by the use of $D = 30$ in models with ARD, although those models did eventually retain 11-12 dimensions only (as given by the learned ARD scale parameters; Figure 5.1A-B).

While the methods did not all recover the exact same number of latents as in Section 4.3.1, they recovered similar numbers of latents, and some mismatch is to be expected as there was model mismatch between the generative model and the true data.

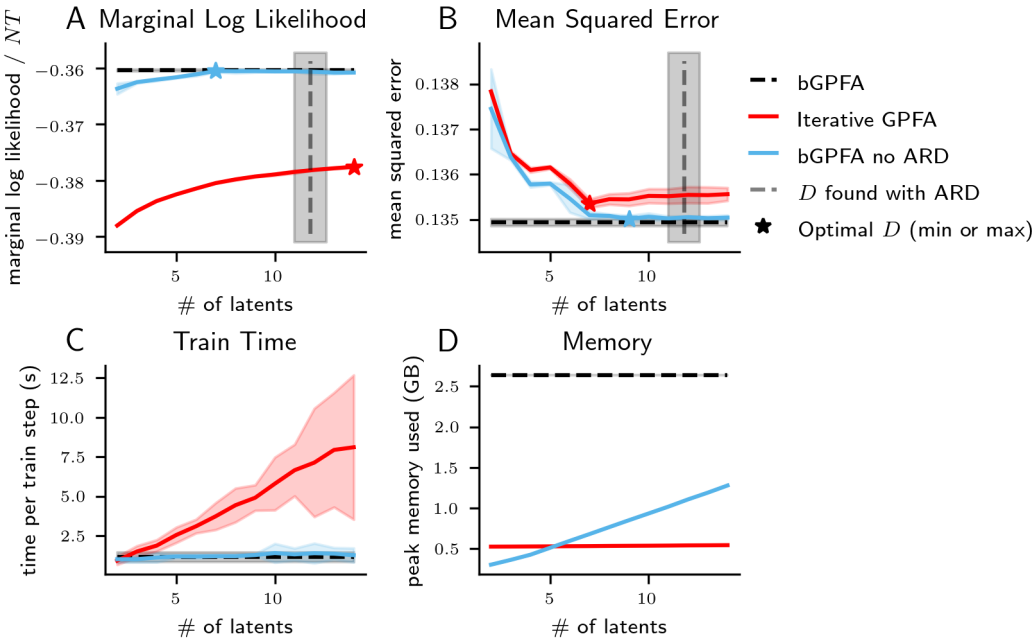


Fig. 5.1 **Performance of iterative GPFA and bGPFA on primate data across D. (A-B)** Marginal log likelihood and mean squared error of iterative GPFA, bGPFA, and bGPFA without ARD on a self-paced primate reaching dataset ($T = 5000, N = 106$). **(C-D)** Train time and peak active memory usage during training and inference. In contrast to Figure 3.3, training time was measured less precisely over a larger range of computations, including printing some periodic updates, and peak active memory usage was computed less precisely over the full training and inference set of computations, without subtracting preexisting memory cost. The computations are, however, comparable across methods and still serve as a good comparison between iterative and Bayesian GPFA.

5.1.3 Discussion

Iterative GPFA models were trained for only 2001 training steps because the loss eventually diverged for iterative GPFA models trained for 7501 steps. The reason for this divergence is uncertain, but is likely related to an inappropriate setting of the learning rate. In particular, a bug in the version of GPyTorch that I used (that has since been corrected) meant that the loss was not scaled by the number of time-points. While I fixed this by dividing by T in all results and figures shown, I did not correct this during training, so the learning

rate found in Figure 3.1 with $T = 100$ likely did not generalize well here with $T = 5000$. I thus ran iterative GPFA for 2001 steps which kept it in a realm where it had converged but not yet diverged. Although I did not have time to look into this instability in detail, it will be worth investigating in the future. Another source of error could be inaccurate gradients—GPyTorch uses in-place updating through CG iterations in a 'jitted' PyTorch function, something PyTorch documentation warns can lead to incorrect gradients [55].

A good CG preconditioner and the resolution of this iterative GPFA instability could improve the performance of iterative GPFA. Such a preconditioner could potentially also reduce the time per training step by reducing the number of CG iterations needed to reach a given accuracy tolerance. Nevertheless, I think this is unlikely to lead to significant speedups given that CG currently converges to the specified tolerance within 100 CG iterations (whereas naively it could take up to $DT = 5000D$ iterations with this data).

bGPFA is more flexible than iterative GPFA overall. bGPFA is more flexible in terms of the noise model—I used the Gaussian noise model in these experiments because the Negative Binomial model applied in Section 4.3.2 is not tractable with the non-variational GPFA implementations. bGPFA is also much more flexible in terms of memory usage because the KL approximation enables batching by time points and by Monte Carlo samples—one can set these batch sizes such that the computation fits within the available memory (Algorithm 2). In contrast, the "exact" derivative of the loss computed by iterative GPFA here requires all time points to be handled simultaneously (Algorithm 1).

I attempted to run iterative GPFA with the full hour-long dataset ($T = 134517$), however doing so required mini-batching over probe vectors to alleviate memory load. This provided memory savings but nearly quadrupled the duration of each training step to over 9 minutes, because I could only compute three trace samples in memory at a time, so I had to perform all CG computations four times. As mentioned in ref. 71, while CG does scale near-linearly with T, it is still a very time-intensive computation. Due to limited compute resources, combined with the suboptimal performance of iterative GPFA and the inability to use more appropriate noise models with iterative GPFA, I decided not to fully train any iterative GPFA models on the full dataset. I was able to train a bGPFA model with ARD and the Gaussian noise model on the full dataset, anticipating comparing these results with iterative GPFA. I did not present those results here because it would be more appropriate to perform the analysis with a Negative Binomial noise model, and this was already done for a similar, slightly shorter, dataset in Section 4.3.2.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

In summary, I introduced two methods for scaling GPFA to many time-points for neural data. Both methods scale near-linearly in time due to Toeplitz and Kronecker structure that can be leveraged in binned neuroscientific data (Chapter 2). I demonstrated these methods with T on the order of 10^5, i.e. three orders of magnitude larger than previous applications ($T \sim 10^2$). This scalability enabled the application of GPFA to long (up to hour-long) non-trial structured recordings, and yielded latents that varied with timescales approximately as long as individual reaches. These methods are not, however, restricted to datasets with regularly spaced inputs—datasets with irregularly spaced inputs can still make use of the methods introduced here using the methods described in ref. 76.

I began by introducing a scalable version of "exact" GPFA that I termed iterative GPFA following refs. 17 and 3 (Chapter 3). I then introduced an approximate variational GPFA method with additional features including a Bayesian prior over the mixing matrix C, automatic relevance determination, and support for otherwise intractable likelihood models that are more appropriate to neuroscience, such as the Negative Binomial noise model Chapter 4. I then compared these two methods, concluding that bGPFA scales more readily without a loss in performance compared to iterative GPFA, although proposed future work could potentially shift this balance (Chapter 5 and section 6.2).

6.2 Future Work

The additional benefits afforded by the approximate variational approach for neuroscientific purposes make me doubtful that iterative GPFA even with future work would become preferable to bGPFA, however future work is needed to verify this.

In order to make iterative GPFA more feasible on extremely large datasets with $T > 10^5$, the issues with numerical instabilities in convergence as discussed in Section 5.1 must first be resolved. Suggestions by [3] for improving stopping criterion for CG and for better initialization (from the value at the previous training step) could significantly reduce the time required by CG methods by reducing the number of iterations each CG solve requires. Additionally, a non-memory intensive preconditioner could also reduce the time cost of CG significantly and potentially mitigate the training instabilities encountered with iterative GPFA.

I look forward to seeing what neuroscientific insights bGPFA and related scalable methods yield when applied to a broader array of large neural datasets.

References

[1] Afshar, A., Santhanam, G., Byron, M. Y., Ryu, S. I., Sahani, M., and Shenoy, K. V. (2011). Single-trial neural correlates of arm movement preparation. *Neuron*, 71(3):555–564.

[2] Allen, E., Baglama, J., and Boyd, S. (2000). Numerical approximation of the product of the square root of a matrix with a vector. *Linear Algebra and its Applications*, 310(1-3):167–181.

[3] Artemev, A., Burt, D. R., and van der Wilk, M. (2021). Tighter Bounds on the Log Marginal Likelihood of Gaussian Process Regression Using Conjugate Gradients. *arXiv:2102.08314 [cs, stat]*. arXiv: 2102.08314.

[4] Azevedo, F. A., Carvalho, L. R., Grinberg, L. T., Farfel, J. M., Ferretti, R. E., Leite, R. E., Filho, W. J., Lent, R., and Herculano-Houzel, S. (2009). Equal numbers of neuronal and nonneuronal cells make the human brain an isometrically scaled-up primate brain. *Journal of Comparative Neurology*, 513(5):532–541.

[5] Azouz, R. and Gray, C. M. (1999). Cellular mechanisms contributing to response variability of cortical neurons in vivo. *J. Neurosci.*, 19(6):2209–2223.

[6] Bernacchia, A., Seo, H., Lee, D., and Wang, X.-J. (2011). A reservoir of time constants for memory traces in cortical neurons. *Nature neuroscience*, 14(3):366–372.

[7] Bishop, C. M. (1999). Bayesian PCA. *Advances in neural information processing systems*, pages 382–388.

[8] Bradbury, J., Frostig, R., Hawkins, P., Johnson, M. J., Leary, C., Maclaurin, D., Necula, G., Paszke, A., VanderPlas, J., Wanderman-Milne, S., and Zhang, Q. (2018). JAX: composable transformations of Python+NumPy programs.

[9] Bui, T., Hernandez-Lobato, D., Hernandez-Lobato, J., Li, Y., and Turner, R. (2016). Deep gaussian processes for regression using approximate expectation propagation. In Balcan, M. F. and Weinberger, K. Q., editors, *Proceedings of The 33rd International Conference on Machine Learning*, volume 48 of *Proceedings of Machine Learning Research*, pages 1472–1481, New York, New York, USA. PMLR.

[10] Challis, E. and Barber, D. (2013). Gaussian kullback-leibler approximate inference. *Journal of Machine Learning Research*, 14(8).

[11] Chaudhuri, R., Gerçek, B., Pandey, B., Peyrache, A., and Fiete, I. (2019). The intrinsic attractor manifold and population dynamics of a canonical cognitive circuit across waking and sleep. *Nature neuroscience*, 22(9):1512–1520.

[12] Churchland, M. M., Cunningham, J. P., Kaufman, M. T., Foster, J. D., Nuyujukian, P., Ryu, S. I., and Shenoy, K. V. (2012). Neural population dynamics during reaching. *Nature*, 487(7405):51–56.

[13] Cunningham, J. P. and Byron, M. Y. (2014). Dimensionality reduction for large-scale neural recordings. *Nature neuroscience*, 17(11):1500–1509.

[14] Cunningham, J. P., Shenoy, K. V., and Sahani, M. (2008). Fast gaussian process methods for point process intensity estimation. In *Proceedings of the 25th International Conference on Machine Learning*, ICML '08, page 192–199, New York, NY, USA. Association for Computing Machinery.

[15] Cutajar, K., Osborne, M., Cunningham, J., and Filippone, M. (2016). Preconditioning Kernel Matrices. In *International Conference on Machine Learning*, pages 2529–2538. PMLR. ISSN: 1938-7228.

[16] Damianou, A. and Lawrence, N. D. (2013). Deep Gaussian processes. In *Artificial intelligence and statistics*, pages 207–215. PMLR.

[17] Davies, A. (2015). *Effective implementation of Gaussian process regression for machine learning*. PhD thesis, University of Cambridge.

[18] Dong, K., Eriksson, D., Nickisch, H., Bindel, D., and Wilson, A. G. (2017). Scalable Log Determinants for Gaussian Process Kernel Learning. *arXiv:1711.03481 [cs, stat]*. arXiv: 1711.03481.

[19] Duncker, L. and Sahani, M. (2018). Temporal alignment and latent Gaussian process factor inference in population spike trains. In *Advances in Neural Information Processing Systems*, volume 31.

[20] Duvenaud, D. K. (2014). *Automatic Model Construction with Gaussian Processes*. PhD thesis, University of Cambridge.

[21] Ecker, A., Berens, P., Cotton, R., Subramaniyan, M., Denfield, G., Cadwell, C., Smirnakis, S., Bethge, M., and Tolias, A. (2014). State Dependence of Noise Correlations in Macaque Primary Visual Cortex. *Neuron*, 82:235–248.

[22] Elsayed, G. F., Lara, A. H., Kaufman, M. T., Churchland, M. M., and Cunningham, J. P. (2016). Reorganization between preparatory and movement population responses in motor cortex. *Nat. Commun.*, 7:1–15.

[23] Fenton, A. A. and Muller, R. U. (1998). Place cell discharge is extremely variable during individual passes of the rat through the firing field. *Proceedings of the National Academy of Sciences*, 95(6):3182–3187.

[24] Gao, Y., Archer, E. W., Paninski, L., and Cunningham, J. P. (2016). Linear dynamical neural population models through nonlinear embeddings. In *Advances in Neural Information Processing Systems*, volume 29.

[25] Gardner, J. R., Pleiss, G., Bindel, D., Weinberger, K. Q., and Wilson, A. G. (2018). Gpytorch: Blackbox matrix-matrix gaussian process inference with gpu acceleration. In *Advances in Neural Information Processing Systems*.

[26] Goulard, M. and Voltz, M. (1992). Linear coregionalization model: Tools for estimation and choice of cross-variogram matrix. *Mathematical Geology*, 24(3):269–286.

[27] Hensman, J., Fusi, N., and Lawrence, N. D. (2013). Gaussian processes for Big data. In *Proceedings of the Twenty-Ninth Conference on Uncertainty in Artificial Intelligence*, UAI'13, pages 282–290, Arlington, Virginia, USA. AUAI Press.

[28] Hensman, J., Matthews, A., and Ghahramani, Z. (2015a). Scalable variational Gaussian process classification. In *Artificial Intelligence and Statistics*, pages 351–360. PMLR.

[29] Hensman, J., Matthews, A. G., Filippone, M., and Ghahramani, Z. (2015b). MCMC for variationally sparse Gaussian processes. In *Advances in Neural Information Processing Systems*, volume 28.

[30] Hestenes, M. R., Stiefel, E., et al. (1952). Methods of conjugate gradients for solving linear systems. 49(1).

[31] Humphries, M. D. (2020). Strong and weak principles of neural dimension reduction. *arXiv preprint arXiv:2011.08088*.

[32] Hutchinson, M. F. (1990). A stochastic estimator of the trace of the influence matrix for laplacian smoothing splines. *Communications in Statistics - Simulation and Computation*, 19(2):433–450. Publisher: Taylor & Francis _eprint: https://doi.org/10.1080/03610919008812866.

[33] Jensen, K., Kao, T.-C., Tripodi, M., and Hennequin, G. (2020). Manifold GPLVMs for discovering non-euclidean latent structure in neural data. In *Advances in Neural Information Processing Systems*, volume 33, pages 22580–22592.

[34] Jensen, K. T., Kao, T.-C., Stone, J. T., and Hennequin, G. (2021). Scalable Bayesian GPFA with automatic relevance determination and discrete noise models. *bioRxiv*.

[35] Kao, T.-C., Sadabadi, M. S., and Hennequin, G. (2021). Optimal anticipatory control as a theory of motor preparation: A thalamo-cortical circuit model. *Neuron*, 109:1567–1581.

[36] Keeley, S. L., Zoltowski, D. M., Aoi, M. C., and Pillow, J. W. (2020). Modeling statistical dependencies in multi-region spike train data. *Current Opinion in Neurobiology*.

[37] Keshtkaran, M. R. and Pandarinath, C. (2019). Enabling hyperparameter optimization in sequential autoencoders for spiking neural data. In Wallach, H., Larochelle, H., Beygelzimer, A., d'Alché-Buc, F., Fox, E., and Garnett, R., editors, *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc.

[38] Keshtkaran, M. R., Sedler, A. R., Chowdhury, R. H., Tandon, R., Basrai, D., Nguyen, S. L., Sohn, H., Jazayeri, M., Miller, L. E., and Pandarinath, C. (2021). A large-scale neural network training framework for generalized estimation of single-trial population dynamics. *bioRxiv*.

[39] Kingma, D. P. and Ba, J. (2015). Adam: A method for stochastic optimization. In *3rd International Conference on Learning Representations, ICLR*.

[40] Kingma, D. P. and Welling, M. (2014). Auto-encoding variational bayes. In *2nd International Conference on Learning Representations, ICLR*.

[41] Lakshmanan, K. C., Sadtler, P. T., Tyler-Kabara, E. C., Batista, A. P., and Yu, B. M. (2015). Extracting low-dimensional latent structure from time series in the presence of delays. *Neural computation*, 27(9):1825–1856.

[42] Lara, A. H., Elsayed, G. F., Zimnik, A. J., Cunningham, J. P., and Churchland, M. M. (2018). Conservation of preparatory neural events in monkey motor cortex regardless of how movement is initiated. *eLife*, 7:e31826.

[43] Lawrence, N. and Hyvärinen, A. (2005). Probabilistic non-linear principal component analysis with Gaussian process latent variable models. *Journal of machine learning research*, 6(11).

[44] Low, R. J., Lewallen, S., Aronov, D., Nevers, R., and Tank, D. W. (2018). Probing variability in a cognitive map using manifold inference from neural dynamics. *BioRxiv*.

[45] MacKay, D. J. (2003). *Information theory, inference and learning algorithms*. Cambridge university press.

[46] Macke, J. H., Buesing, L., Cunningham, J. P., Yu, B. M., Shenoy, K. V., and Sahani, M. (2012). Empirical models of spiking in neural populations. In *Advances in Neural Information Processing Systems 24: 25th conference on Neural Information Processing Systems (NIPS 2011)*, pages 1350–1358.

[47] Makin, J. G., O'Doherty, J. E., Cardoso, M. M., and Sabes, P. N. (2018). Superior arm-movement decoding from cortex with a new, unsupervised-learning algorithm. *Journal of neural engineering*, 15(2):026010.

[48] Minxha, J., Adolphs, R., Fusi, S., Mamelak, A. N., and Rutishauser, U. (2020). Flexible recruitment of memory-based choice representations by the human medial frontal cortex. *Science*, 368(6498).

[49] Murray, I. and Adams, R. P. (2010). Slice sampling covariance hyperparameters of latent Gaussian models. *arXiv preprint arXiv:1006.0868*.

[50] Neal, R. M. (2012). *Bayesian learning for neural networks*, volume 118. Springer Science & Business Media.

[51] Opper, M. and Archambeau, C. (2009). The variational Gaussian approximation revisited. *Neural computation*, 21(3):786–792.

[52] O'Doherty, J. E., Cardoso, M., Makin, J., and Sabes, P. (2017). Nonhuman primate reaching with multichannel sensorimotor cortex electrophysiology. *Zenodo http://doi.org/10.5281/zenodo*, 583331.

[53] Pandarinath, C., O'Shea, D. J., Collins, J., Jozefowicz, R., Stavisky, S. D., Kao, J. C., Trautmann, E. M., Kaufman, M. T., Ryu, S. I., Hochberg, L. R., et al. (2018). Inferring single-trial neural population dynamics using sequential auto-encoders. *Nature methods*, 15(10):805–815.

[54] Parra, G. and Tobar, F. (2017). Spectral mixture kernels for multi-output gaussian processes. In Guyon, I., Luxburg, U. V., Bengio, S., Wallach, H., Fergus, R., Vishwanathan, S., and Garnett, R., editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc.

[55] Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S. (2019). Pytorch: An imperative style, high-performance deep learning library. In Wallach, H., Larochelle, H., Beygelzimer, A., d'Alché-Buc, F., Fox, E., and Garnett, R., editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc.

[56] Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., et al. (2011). Scikit-learn: Machine learning in python. *the Journal of machine Learning research*, 12:2825–2830.

[57] Quinonero-Candela, J. and Rasmussen, C. E. (2005). A unifying view of sparse approximate Gaussian process regression. *The Journal of Machine Learning Research*, 6:1939–1959.

[58] Recanatesi, S., Ocker, G. K., Buice, M. A., and Shea-Brown, E. (2019). Dimensionality in recurrent spiking networks: global trends in activity and local origins in connectivity. *PLoS computational biology*, 15(7):e1006446.

[59] Rezende, D. J., Mohamed, S., and Wierstra, D. (2014). Stochastic backpropagation and approximate inference in deep generative models. In *International conference on machine learning*, pages 1278–1286. PMLR.

[60] Rutten, V., Bernacchia, A., Sahani, M., and Hennequin, G. (2020). Non-reversible gaussian processes for identifying latent dynamical structure in neural data. In Larochelle, H., Ranzato, M., Hadsell, R., Balcan, M. F., and Lin, H., editors, *Advances in Neural Information Processing Systems*, volume 33, pages 9622–9632. Curran Associates, Inc.

[61] Sauerbrei, B. A., Guo, J.-Z., Cohen, J. D., Mischiati, M., Guo, W., Kabra, M., Verma, N., Mensh, B., Branson, K., and Hantman, A. W. (2020). Cortical pattern generation during dexterous movement is input-driven. *Nature*, 577(7790):386–391.

[62] Snelson, E. and Ghahramani, Z. (2006). Sparse gaussian processes using pseudo-inputs. In Weiss, Y., Schölkopf, B., and Platt, J., editors, *Advances in Neural Information Processing Systems*, volume 18. MIT Press.

[63] Sohn, H., Narain, D., Meirhaeghe, N., and Jazayeri, M. (2019). Bayesian computation through cortical latent dynamics. *Neuron*, 103(5):934–947.

[64] Sussillo, D., Jozefowicz, R., Abbott, L. F., and Pandarinath, C. (2016). LFADS - Latent Factor Analysis via Dynamical Systems. *arXiv:1608.06315 [cs, q-bio, stat]*. arXiv: 1608.06315.

[65] Titsias, M. and Lawrence, N. D. (2010). Bayesian Gaussian process latent variable model. In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, pages 844–851. JMLR Workshop and Conference Proceedings.

[66] Tomko, G. J. and Crapper, D. R. (1974). Neuronal variability: non-stationary responses to identical visual stimuli. *Brain research*, 79(3):405–418.

[67] Tosi, A., Hauberg, S., Vellido, A., and Lawrence, N. D. (2014). Metrics for probabilistic geometries. *arXiv preprint arXiv:1411.7432*.

[68] Tran, D., Ranganath, R., and Blei, D. M. (2016). Variational gaussian process. In *ICLR*.

[69] Van Loan, C. F. (2000). The ubiquitous Kronecker product. *Journal of computational and applied mathematics*, 123(1-2):85–100.

[70] Wainwright, M. J. and Jordan, M. I. (2008). *Graphical models, exponential families, and variational inference*. Now Publishers Inc.

[71] Wang, K., Pleiss, G., Gardner, J., Tyree, S., Weinberger, K. Q., and Wilson, A. G. (2019). Exact Gaussian Processes on a Million Data Points. *Advances in Neural Information Processing Systems*, 32:14648–14659.

[72] Williams, A. H., Kim, T. H., Wang, F., Vyas, S., Ryu, S. I., Shenoy, K. V., Schnitzer, M., Kolda, T. G., and Ganguli, S. (2018). Unsupervised Discovery of Demixed, Low-Dimensional Neural Dynamics across Multiple Timescales through Tensor Component Analysis. *Neuron*, 98(6):1099–1115.e8.

[73] Williams, C. K. and Rasmussen, C. E. (1996). Gaussian processes for regression.

[74] Williams, C. K. and Rasmussen, C. E. (2006). *Gaussian processes for machine learning*, volume 2. MIT press Cambridge, MA.

[75] Wilson, A. G., Dann, C., and Nickisch, H. (2015). Thoughts on massively scalable Gaussian processes. *arXiv preprint arXiv:1511.01870*.

[76] Wilson, A. G. and Nickisch, H. (2015). Kernel interpolation for scalable structured Gaussian processes (KISS-GP). In *Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37*, ICML'15, pages 1775–1784, Lille, France. JMLR.org.

[77] Wu, A., Roy, N. A., Keeley, S., and Pillow, J. W. (2017). Gaussian process based nonlinear latent structure discovery in multivariate spike train data. *Advances in neural information processing systems*, 30:3496.

[78] Yu, B. M., Cunningham, J. P., Santhanam, G., Ryu, S. I., Shenoy, K. V., and Sahani, M. (2009a). Gaussian-process factor analysis for low-dimensional single-trial analysis of neural population activity. *Journal of Neurophysiology*, 102(1):614–635.

[79] Yu, B. M., Cunningham, J. P., Santhanam, G., Ryu, S. I., Shenoy, K. V., and Sahani, M. (2009b). Gaussian-process factor analysis for low-dimensional single-trial analysis of neural population activity. In Koller, D., Schuurmans, D., Bengio, Y., and Bottou, L., editors, *Advances in Neural Information Processing Systems 21*, pages 1881–1888. Curran Associates, Inc.

[80] Zhao, Y. and Park, I. M. (2017). Variational latent Gaussian process for recovering single-trial dynamics from population spike trains. *Neural computation*, 29(5):1293–1316.

[81] Zimnik, A. J. and Churchland, M. M. (2021). Independent generation of sequence elements by motor cortex. *Nature Neuroscience*, 24:412–424.

Appendix A

Iterative GPFA Appendix

A.1 Iterative GPFA Pseudocode

In this section I provide pseudocode for iterative GPFA (Algorithm 1) leveraging Toeplitz (gridded) structure.

Note in Line 15 that I define my vector of timepoints to be unit timepoints, so the timescales learned will be in units of timesteps. These can be scaled by the timestep size to get the timescales in terms of seconds or milliseconds.

In Line 17 I calculate $\text{vec}^{-1}(\boldsymbol{K}_{xx}\text{vec}(\boldsymbol{C}^T\boldsymbol{V}))$ which appears in Equation 2.6. In particular, I rewrite Equation 2.3 directly using column indexing in place of $\boldsymbol{e}_i\boldsymbol{e}_i^T$ and vstack in place of the summation as described in the text following Equation 2.3.

Algorithm 1: Iterative GPFA Leveraging Toeplitz Structure

input: neuron-wise mean-subtracted data $\boldsymbol{Y} \in \mathbb{R}^{N \times T}$, latent dimensionality D, # trace samples s, CG tolerance ε, learning rate γ, kernel functions $\{K_i\}_1^D$
parameters: $\theta = \{\boldsymbol{C}, \{R_{n,n}\}_1^N, \{k_d\}_1^D\}$

% solve for $\boldsymbol{b}$ in $\boldsymbol{Ab} = \boldsymbol{c}$
% A_mult is a function for computing vector products with $\boldsymbol{A}$ [30]
def *CG(A_mult,* $\boldsymbol{c}$*)*:
    $\boldsymbol{b} \leftarrow \boldsymbol{0}$
    while $\frac{\|\boldsymbol{c} - A\_mult(\boldsymbol{b})\|_2}{\|\boldsymbol{c}\|_2} > \varepsilon$ **do**
        $\boldsymbol{b} \leftarrow$ update according to CG algorithm using *A_mult* for $\boldsymbol{Av}$ computations
    return $\boldsymbol{b}$

% compute $\boldsymbol{K}_{yy}\boldsymbol{v}$ products (Section 2.2)
def *Kyy_mult(*$\boldsymbol{v}$*)*:
    % WLOG use timesteps of unit length
    $\boldsymbol{t} \leftarrow [1, 2, ..., T-1, T]$
    $\boldsymbol{V} \leftarrow \text{vec}^{-1}(\boldsymbol{v})$
    % Calculate $\text{vec}^{-1}(\boldsymbol{K}_{xx}\text{vec}(\boldsymbol{C}^T\boldsymbol{V}))$
    $\boldsymbol{X} \leftarrow \text{vstack}\left([(\text{toeplitz\_mult}(K_i(\boldsymbol{t}, 1), (\boldsymbol{V}^T\boldsymbol{C})_i)^T)]_{i=1}^D\right)$
    return $\text{vec}(\boldsymbol{CX}) + \text{vec}(\boldsymbol{RV})$

while *not converged* **do**
    % calculate K_{yy}^{-1} loss term (Section 3.1.1)
    $\mathcal{L}_{\boldsymbol{K}_{yy}^{-1}} \leftarrow \text{vec}(\boldsymbol{Y})^T CG(Kyy\_mult, \text{vec}(\boldsymbol{Y}))$
    % calculate $|\boldsymbol{K}_{yy}|$ (Section 3.1.2)
    $\boldsymbol{g} \leftarrow \boldsymbol{0}$
    for *i=1:s* **do**
        $\boldsymbol{\xi} \leftarrow$ random vector, each entry randomly and evenly sampled from $\{-1, 1\}$
        $\boldsymbol{z} \leftarrow CG(Kyy\_mult, \boldsymbol{\xi})$
        % differentiate through $\boldsymbol{z}^T Kyy\_mult(\boldsymbol{\xi})$ treating $\boldsymbol{z}$ as a constant to avoid differentiating through CG
        $\boldsymbol{g} \leftarrow \boldsymbol{g} + \nabla \boldsymbol{z}^T Kyy\_mult(\boldsymbol{\xi})$
    $\nabla \mathcal{L}_{|\boldsymbol{K}_{yy}|} \leftarrow \frac{\boldsymbol{g}}{s}$
    % update parameters based on total gradients (I used Adam in practice)
    $\theta \leftarrow \theta + \gamma(-\nabla \mathcal{L}_{|\boldsymbol{K}_{yy}|} - \nabla \mathcal{L}_{\boldsymbol{K}_{yy}^{-1}})$

A.2 GPyTorch GPFA implementation code

This section includes the code added to GPyTorch to implement GPFA. This code is also available on GitHub on my fork of GPyTorch, where relative imports work (https://github.com/syncrostone/gpytorch/tree/gpfa-newer-base).

```
import torch

from ..lazy import DiagLazyTensor, KroneckerProductLazyTensor, lazify
from .kernel import Kernel

class GPFAComponentKernel(Kernel):
    r"""
     Kernel supporting Gaussian Process Factor Analysis using
     :class:`gpytorch.kernels.GPFAComponentKernel` as a basic GPFA
    latent kernel.

    Given a base covariance module to be used for a latent, :math:`K_{XX}`, this kernel computes a latent kernel of
    specified size :math:`K_MM}` that is zeros everywhere except :math:`K_{kernel_loc,kernel_loc}` and returns
    :math:`K = K_{MM} \otimes K_{XX}`. as an :obj:`gpytorch.lazy.KroneckerProductLazyTensor`.

    :param ~gpytorch.kernels.Kernel data_covar_module: Kernel to use as the latent kernel.
    :param int num_latents: Number of latents (M)
    :param int kernel_loc: Latent number that this kernel represents.
    :param dict kwargs: Additional arguments to pass to the kernel.
    """

    def __init__(self, data_covar_module, num_latents, kernel_loc, **kwargs):
        super(GPFAComponentKernel, self).__init__(**kwargs)
        task_diag = torch.zeros(num_latents)
        task_diag[kernel_loc] = 1
        self.register_buffer("task_diag", task_diag)
        self.data_covar_module = data_covar_module
        self.num_latents = num_latents

    def forward(self, x1, x2, diag=False, last_dim_is_batch=False, **params):
        if last_dim_is_batch:
```

```
            raise RuntimeError("GPFAComponentKernel does not accept the last_dim_is_batch argument.")
        covar_i = DiagLazyTensor(self.task_diag)
        if len(x1.shape[:-2]):
            covar_i = covar_i.repeat(*x1.shape[:-2], 1, 1)
        covar_x = lazify(self.data_covar_module.forward(x1, x2, **params))
        res = KroneckerProductLazyTensor(covar_x, covar_i)
        return res.diag() if diag else res

    def num_outputs_per_input(self, x1, x2):
        """
        Given `n` data points `x1` and `m` datapoints `x2`, this
        kernel returns an `(n*num_latents) x (m*num_latents)` covariance matrix.
        """
        return self.num_latents
```

Code Example A.1 **GPyTorch GPFA component kernel.**

```
from copy import deepcopy

import torch

from ..lazy import DiagLazyTensor, KroneckerProductLazyTensor
from .gpfa_component_kernel import GPFAComponentKernel
from .kernel import AdditiveKernel, Kernel

class GPFAKernel(Kernel):
    r"""
    Kernel supporting Gaussian Process Factor Analysis using
    :class:`gpytorch.kernels.GPFAComponentKernel` as a basic GPFA latent kernel.

    Given base covariance modules to be used for the latents, :math:`k_i`, this kernel
    puts the base covariance modules in a block diagonal with :math:`M` blocks as :math:`K_{XX}`.
    This defines :math:`C \in MxN` and returns :math:`(I_T \otimes C)K_{XX}(I_T \otimes C)^T` as an
    :obj:`gpytorch.lazy.LazyEvaluatedKernelTensor`.

    :param ~gpytorch.kernels.Kernel data_covar_module: Kernel to use as the latent kernel.
    :param int num_latents: Number of latents (M).
```

```
    :param int num_obs: Number of observation dimensions (typically, the
    number of neurons, N).
    :param ~gpytorch.kernels.Kernel GPFA_component: (default
GPFAComponentKernel) Kernel to use to scale the latent
    kernels to the necessary shape.
    GPFAComponentKernel is currently the only option; if non-reversible
kernels are later added,
    there will then be another option here.
    :param dict kwargs: Additional arguments to pass to the kernel.
    """

    def __init__(
        self, data_covar_modules, num_latents, num_obs, GPFA_component=
GPFAComponentKernel, **kwargs,
    ):
        super(GPFAKernel, self).__init__(**kwargs)
        self.num_obs = num_obs
        self.num_latents = num_latents

        if not isinstance(data_covar_modules, list) or len(
data_covar_modules) == 1:
            if isinstance(data_covar_modules, list):
                data_covar_modules = data_covar_modules[0]
            data_covar_modules = [deepcopy(data_covar_modules) for i in
range(num_latents)]

        self.latent_covar_module = AdditiveKernel(
            *[GPFA_component(data_covar_modules[i], num_latents, i) for
i in range(num_latents)]
        )
        self.register_parameter(name="raw_C", parameter=torch.nn.
Parameter(torch.randn(num_obs, num_latents)))

    @property
    def C(self):
        return self.raw_C

    @C.setter
    def C(self, value):
        if not torch.is_tensor(value):
            value = torch.as_tensor(value).to(self.raw_C)

        self.initialize(raw_C=value)

```

```
    def forward(self, x1, x2, diag=False, last_dim_is_batch=False, **
    params):
        if last_dim_is_batch:
            raise RuntimeError("GPFAKernel does not yet accept the
    last_dim_is_batch argument.")
        I_t1 = DiagLazyTensor(torch.ones_like(torch.squeeze(x1)))
        I_t2 = DiagLazyTensor(torch.ones_like(torch.squeeze(x2)))
        kron_prod_1 = KroneckerProductLazyTensor(I_t1, self.C)
        kron_prod_2 = KroneckerProductLazyTensor(I_t2, self.C)
        covar = kron_prod_1 @ self.latent_covar_module(x1, x2, **params)
     @ kron_prod_2.t()
        return covar.diag() if diag else covar

    def num_outputs_per_input(self, x1, x2):
        """`
        Given `n` data points `x1` and `m` datapoints `x2`, this
        kernel returns an `(n*num_obs) x (m*num_obs)` covariance matrix.
        """
        return self.num_obs

```

Code Example A.2 **GPyTorch GPFA kernel.**

```
import torch

import gpytorch
from gpytorch.kernels import GPFAKernel
from gpytorch.lazy import DiagLazyTensor, KroneckerProductLazyTensor,
    KroneckerProductDiagLazyTensor, AddedDiagLazyTensor

class GPFAModel(gpytorch.models.ExactGP):
    def __init__(self, train_x, train_y, likelihood,
    latent_covar_modules,  num_latents, num_obs):
        super(GPFAModel, self).__init__(train_x, train_y, likelihood)

        self.num_latents = num_latents
        self.num_obs = num_obs

        self.mean_module = gpytorch.means.MultitaskMean(
            gpytorch.means.ZeroMean(), num_tasks=num_obs
        )
        self.covar_module = GPFAKernel(
          latent_covar_modules,num_latents, num_obs)

    def forward(self, x):
```

```
        return gpytorch.distributions.MultitaskMultivariateNormal(
        self.mean_module(x), self.covar_module(x))

    def get_combined_noises(self):
      """combines the task and the global noise to give one list of noise values in R"""
        return (self.likelihood.task_noises if self.likelihood.has_task_noise
                else torch.zeros(
                self.likelihood.num_tasks)) + (
                    self.likelihood.noise
                    if self.likelihood.has_global_noise else 0)

    def latent_posterior(self, x, train_y = None, obs_dims = None, mean_only = True, preconditioner_override=None):
      """
        See Equations 1.13 and 1.14
        """
        if train_y is None:
            train_y = self.train_targets

        if obs_dims is None:
            obs_dims = range(self.num_obs)

        I_t = DiagLazyTensor(torch.ones(len(x), dtype=self.covar_module.C.dtype, device = x.device))

        combined_noise = self.get_combined_noises()[obs_dims]

        C_Kron_I = KroneckerProductLazyTensor(I_t, self.covar_module.C[obs_dims])

        Kxx = self.covar_module.latent_covar_module(x)

        # the first term of Kyy is specific to GPFA Kernel, be careful if adding nonreversible
        Kyy = AddedDiagLazyTensor(C_Kron_I @ Kxx @ C_Kron_I.t(), KroneckerProductDiagLazyTensor(
            I_t, DiagLazyTensor(combined_noise)), preconditioner_override=preconditioner_override)

        mean_rhs = (train_y - self.mean_module(x)[:,obs_dims]).view(
            *(train_y.numel(),
              )) # vertically stacks after doing the subtraction
```

```
        latent_mean = Kxx @ C_Kron_I.t() @ Kyy.inv_matmul(mean_rhs)
        latent_mean = latent_mean.view(*(len(x),
                                         int(latent_mean.shape[0] / len(
    x))))

        if mean_only:
            return latent_mean
        cov_rhs = C_Kron_I @ Kxx
        latent_cov = Kxx - Kxx @ C_Kron_I.t() @ Kyy.inv_matmul(
            cov_rhs.evaluate())
        return gpytorch.distributions.MultitaskMultivariateNormal(
            latent_mean, latent_cov)
```

Code Example A.3 **GPyTorch GPFA model.**

```
import torch
import gpytorch
from sklearn.decomposition import FactorAnalysis as FA

def gpfa_fa_init(model, Y, len_init, grid_mode = False):
    n_samples_fa, n_fa, m_fa = Y.shape

    # Fit a factor analysis model
    mod = FA(n_components=model.num_latents)
    Y_fa = Y.transpose(0, 2, 1).reshape(n_samples_fa * m_fa, n_fa)
    mudata = mod.fit_transform(Y_fa)  #m*n_samples x d

    # Apply a random unitary transform to the C matrix and initialize C
    C = torch.tensor(mod.components_.T)  # (n x d)
    Q, R = torch.qr(torch.normal(mean=torch.zeros((model.num_latents,
    model.num_latents),dtype=torch.double), std=1))
    C = C@Q
    model.covar_module.C = C

    # Ensure noise total is > .1 when initializing from FA
    noise = torch.tensor(mod.noise_variance_)
    model.likelihood.task_noises = torch.tensor([max(noise[i] - (.05+1e
    -4), (.05 + 1e-4)) for i in range(len(noise))])
    model.likelihood.noise = .05 + 1e-4

    # Initialize lengthscales with the specified len_init
    if grid_mode:
        for i in range(model.num_latents):
```

```
27            model.covar_module.latent_covar_module.kernels[i].
    data_covar_module.base_kernel.lengthscale = torch.tensor(len_init,
    dtype = torch.double)
28     else:
29         for i in range(model.num_latents):
30             model.covar_module.latent_covar_module.kernels[i].
    data_covar_module.lengthscale = torch.tensor(len_init, dtype = torch.
    double)
```

Code Example A.4 **GPyTorch GPFA initialization from Factor Analysis.**

Appendix B

BGPFA Appendix

B.1 Further analyses of preparatory dynamics in the continuous reaching task

We performed analyses as in Figure 4.3f using the raw data ($\boldsymbol{Y}$) and using factor analysis (FA) with a latent dimensionality matched to that inferred by bGPFA instead of using the bGPFA latent states. The raw data $\boldsymbol{Y}$ showed a high degree of similarity at target onset compared to movement onset, but little discernable structure as a function of reach direction at either point in time (Figure B.1a-b).

While the FA latent distances exhibited no modulation by reach direction at target onset, FA did discover weak modulation at movement onset (Figure B.1a-b). This is qualitatively consistent with our results using bGPFA but with a lower signal to noise ratio. Here and in Section 4.3.2, we defined movement onset as the first time during a reach where the cursor velocity exceeded $0.025\,\mathrm{m\,s^{-1}}$, and we observed little to no quantifiable movement before this point (Figure B.1f). We also discarded 'trials' with premature movement for all analyses here and in Section 4.3.2, which we defined as reaches with a reaction time of 75 ms or less.

To quantify and compare how neural activity was modulated by the similarity of reach directions for different analysis methods, we first computed z-scores of the similarity matrices for both the bGPFA latent states, raw activity, and the latent states from FA. z-scores were calculated as $z = (\boldsymbol{S} - mean(\boldsymbol{S}))/std(\boldsymbol{S})$ for each similarity matrix $\boldsymbol{S}$, and the diagonal elements were excluded for this analysis. We then computed the mean of the z-scored pairwise similarities as a function of difference in reach direction across all pairs of 681 reaches. We found that none of the datasets exhibited notable modulation at target onset (Figure B.1e). In contrast, the neural data exhibited modulation by reach similarity 75 ms prior to movement onset. This modulation was strongest for the bGPFA latent states followed

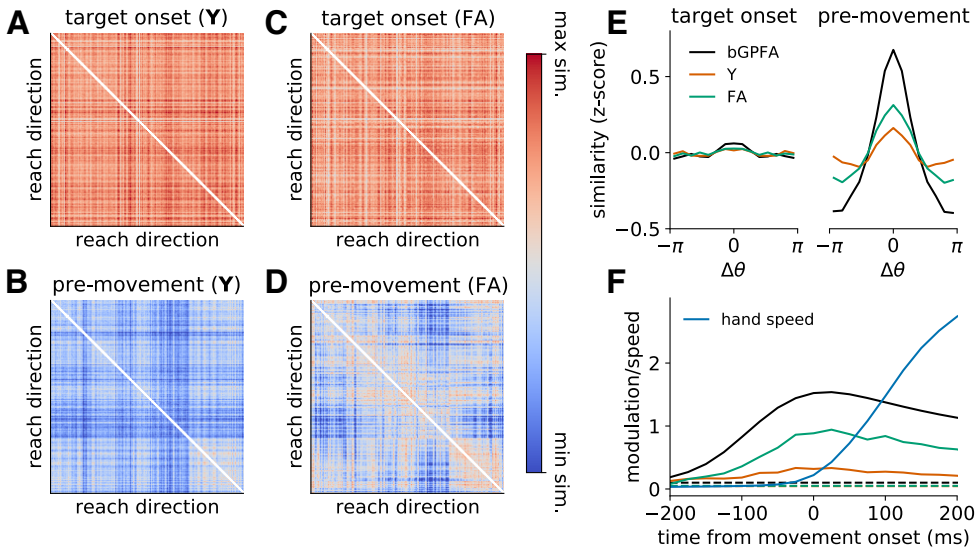


Fig. B.1 **Further analyses of M1 preparatory dynamics. (A-D)** Similarity matrix of raw neural activity $\boldsymbol{Y}$ (A & B) and latent states found by FA (C & D) at target onset (A & C) and 75 ms prior to movement onset (B & D), with analyses performed as in Figure 4.3F. **(E)** z-scored similarity as a function of difference in reach direction; here, the mean similarity across pairs of reaches is shown at target onset (left) and 75 ms prior to movement onset (right). The bGPFA latent states show much stronger modulation than either raw neural activity ($\boldsymbol{Y}$) or latent states from FA. **(F)** Modulation of similarity by reach direction as a function of time from movement onset. Modulation was defined as the difference between maximum and minimum z-scored similarity as a function of difference in reach direction (peak-to-trough in panel E). Blue solid line indicates the z-scored hand speed, confirming the absence of premature movement relative to our definition of movement onset. bGPFA latent similarity increases well before hand speed and starts decreasing substantially before the hand speed peaks. Dashed lines indicate modulation at target onset for each method.

by the FA latents, and the modulation by reach similarity was very weak for the raw neural activity (Figure B.1e). To see how this modulation by reach direction varied as a function of time from movement onset, we computed the difference between the maximum and minimum of the modulation curves and repeated this analysis for various delays. We found that the modulation in neural activity space increased much before any detectable movement, with bGPFA showing the strongest signal followed by factor analysis and then the raw activity (Figure B.1f). Indeed, the bGPFA latent modulation was maximized at movement onset while the reach speed did not peak until several hundred milliseconds after movement onset where bGPFA latent trajectories have started to diverge again. Taken together, these results confirm that our analyses of bGPFA preparatory states do not reflect premature movement onset, and that they are not artifacts of the temporal correlations introduced by our GP prior since noisier but qualitatively similar results arise from the use of factor analysis.

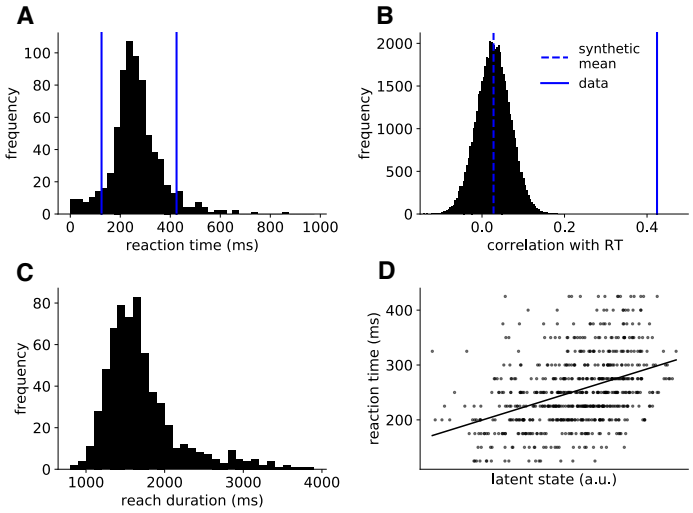


Fig. B.2 **Further reaction time analyses. (A)** Histogram of reaction time across all successful reaches. For our correlation analyses, we only considered reaches with a reaction time between 125 ms and 425 ms (blue vertical lines). **(B)** Pearson correlations between distance to prep state and reaction time in synthetic data. Histogram corresponds to correlations between the true reaction times and 50,000 draws from the learned generative model. Blue dashed line indicates mean across all synthetic datasets (0.028) which is much smaller than the observed correlation in the experimental data of 0.424 (blue solid line). **(C)** Histogram of reach durations for all reaches with a reaction time between 125 ms and 425 ms. **(D)** Plot of reaction time against the value of the latent dimension with the longest timescale ($\tau = 2.1$ s) at target onset.

B.2 Further reaction time analyses

For analyses of correlations between latent distances and reaction times, we only considered reaches with a reaction time of at least 125 ms and at most 425 ms which retained 638 of 681 reaches (Figure B.2a). This is because very long reaction times may reflect the monkey not being fully engaged with the task during those reaches, and very short reaction times may reflect spurious movement. To confirm that our finding of a strong correlation between latent distance and reaction time in Figure 4.3g is not an artifact of the temporal correlations introduced by the bGPFA generative model, we generated a synthetic control. Here we drew 50,000 synthetic latent trajectories from our learned generative model with trajectory durations matched to those observed experimentally on each trial. We then computed mean preparatory states and latent distances to preparatory states as in the experimental data (Section 4.3.2). We found a mean correlation of 0.028 and a range of -0.14 to 0.18 in the synthetic data, suggesting that our generative model may introduce weak correlations between latent distances and reaction times. However, the experimentally observed correlation of 0.424 was much larger than what could be expected by chance, verifying that the distance

from the latent state at target onset to the corresponding preparatory state has behavioral relevance with better initial states leading to shorter reaction times.

Although we already find a fairly strong relationship between these latent states and reaction times, it is worth noting that several additional considerations may further improve such predictions. Notably, our naïve measure of Euclidean latent distances could be improved by instead defining a metric based on the probabilistic model itself [67]. Additionally, while we divide reaches by reach direction, reaches in the same direction can still have different start and end points on the grid (Figure 4.3a), leading to different posture and muscle activations which is likely to significantly affect neural activity. Our analysis by reach direction therefore only represents a coarse categorization of the rich behavioral space, and it remains to be seen how neural activity and latent trajectories are affected by e.g. posture during the task.

Finally we considered whether any long-timescale latent dimensions could be predictive of reaction time across trials by reflecting e.g. motivation or engagement with the task. Here we found that two dimensions had timescales longer than the duration of most reaches with latent timescales of $\tau = 2.1$ s and $\tau = 2.0$ s while the majority of reaches had durations between 1 and 2 seconds (Figure B.2c). Intriguingly, the latent state in these dimensions at target onset was predictive of reaction time with correlations of 0.38 and 0.34 respectively (Figure B.2d). While the information about reaction time contained in these two dimensions was largely redundant, it was orthogonal to that encoded by the distance to preparatory state in the fast dimensions. In particular, a linear model had 18.0% variance explained from the distance to prep in fast dimensions, 14.7% variance explained from the slowest latent dimension, and 28.2% when combining these two features which corresponds to 86.5% of the additive value.

B.3 Task engagement

The experimental recordings were characterized by a period of approximately five minutes towards the end of the recording session during which the monkey did not participate actively in the task and the cursor velocity was near-constant at zero (Figure B.3a). For the decoding analyses, we excluded data from this period since there was little to no behavior to predict. This period also did not contain any successful reaches, and so was excluded from the analyses of individual reaches and reaction times in Section 4.3.2, Appendix B.1, and Appendix B.2.

When analysing neural activity across the periods with and without task participation, we found that neural dynamics moved to a largely orthogonal subspace as the monkey stopped engaging with the task (Figure B.3b). Importantly, we were able to simultaneously capture

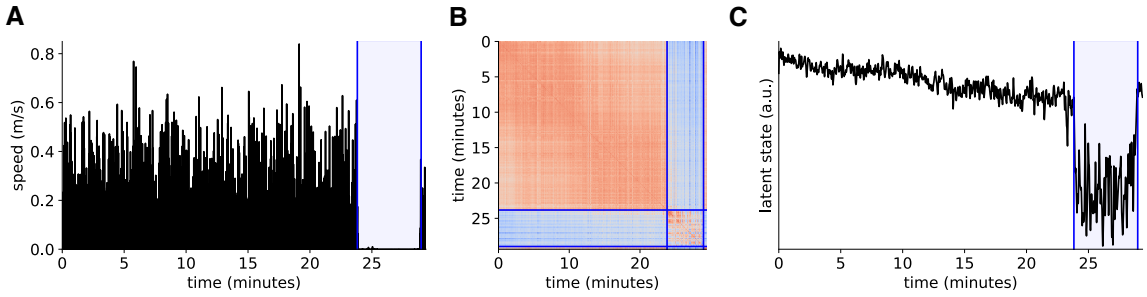


Fig. B.3 **Analyses of a period without task participation. (A)** Cursor speed over the course of the recording session. Blue horizontal lines indicate the last succesful trial before and first succesful trial after a period with no active task participation (blue shading). **(B)** Latent similarity matrix as a function of time during the task. The latent dynamics during task participation occur in a largely orthogonal subspace to the dynamics during the period with no active task participation. **(C)** Plot of latent state over time for the latent dimension with the longest timescale ($\tau = 2.1$ s).

these context-dependent changes as well as movement-specific and preparatory dynamics (Section 4.3.2) by fitting a single model to the full 30-minute dataset, illustrating the utility of bGPFA for analyses of neural data during unconstrained behaviors. Indeed when fitting bGPFA only to the neural data recorded before the monkey stopped participating in the task, our reach-specific analyses gave qualitatively similar results compared to the models fitted to the full dataset. This suggests that bGPFA can capture behaviorally relevant dynamics within individual contexts even when trained on richer datasets with changing contexts.

Finally, we wondered how the neural activity patterns during periods with and without task participation compared to our previous analyses of latent dimensions predictive of task engagement (Appendix B.2). Here we found that a long-timescale latent dimension predictive of reaction times for successful reaches (Figure B.2) also exhibited a prominent change to a different state as the monkey stopped participating in the task (Figure B.3). This is consistent with our hypothesis that this latent dimension does indeed capture a feature related to task engagement which slowly deteriorated during the first 20 minutes of the task followed by a discrete switch to a state with no engagement in the task. During the period of active task participation, this latent dimension was also strongly correlated with time within the session. Indeed, reach number and latent state were both predictive of reaction times, but with the long-timescale latent trajectory exhibiting a slightly stronger correlation (Pearson $\rho = 0.383$ vs. $\rho = 0.353$ respectively). It is perhaps unsurprising that motivation or task engagement decreases with time, and it is difficult in this case to tease apart exactly how motivation vs. time is represented in such latent dimensions. However, based on the strong and abrupt modulation by task participation, this long timescale latent dimension does appear

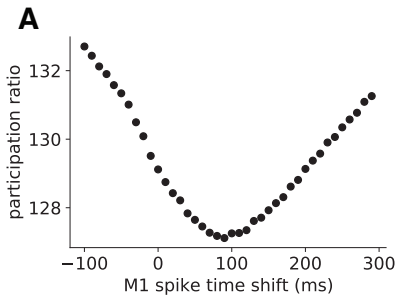


Fig. B.4 **Neural dimensionality. (A)** Participation ratio (Equation B.1) as a function of temporal offset added to M1 spike times in the primate dataset.

to represent some aspect of engagement with the task beyond being a simple measure of time.

B.4 Latent dimensionality

In this section we estimate the dimensionality of the primate data as a function of the offset between M1 and S1 spike times using participation ratios computed on the basis of PCA. The participation ratio is defined as

$$PR = \left(\sum_i \lambda_i\right)^2 / \sum_i \lambda_i^2, \tag{B.1}$$

where λ_i is the i^{th} eigenvalue of the covariance matrix $\boldsymbol{YY}^T$. Here we find that the dimensionality of the data is minimized for a spike time shift of 75-100 ms (Figure B.4). This suggests that the neural recordings can be explained more concisely when taking into account the offset in decoding between M1 and S1 which is consistent with the increased log likelihood after shifting the M1 spikes (Section 4.3.2). We observe a similar trend when considering the number of dimensions retained by bGPFA (21.9 ± 0.30 vs 22.3 ± 0.38 across 10 model fits with and without a 100 ms shift of M1 spiketimes), although the difference is small enough to not be statistically significant in this case. However, it is worth noting that bGPFA explains the data with only a handful of latent dimensions, and this is much lower than the dimensionality of 127-129 estimated by the participation ratio which tends to increase for noisier datasets.

B.5 Parameterizations of approximate GP posterior

In this section, we compare different forms of the variational posterior $q(\boldsymbol{X})$ discussed in Section 4.2.2. For factorizing likelihoods, the optimal posterior takes the form

$$q(\boldsymbol{x}_d) \propto p(\boldsymbol{x}) \prod_t \mathcal{N}(x_t | g_t, v_t), \tag{B.2}$$

where g_t and v_t are variational parameters [51]. Equation B.2 might therefore seem to be an appropriate form of the variational distribution $q(\boldsymbol{X})$. However, this formulation is computationally expensive and the likelihood $p(\boldsymbol{Y}|\boldsymbol{X})$ does not factorize across time in bGPFA.

Instead, we therefore consider approximate parameterizations of the form

$$q(\boldsymbol{x}_d) = \mathcal{N}(\boldsymbol{\mu}_d, \boldsymbol{\Sigma}_d) \tag{B.3}$$

$$\boldsymbol{\mu}_d = \boldsymbol{K}_d^{\frac{1}{2}} \boldsymbol{\nu}_d \tag{B.4}$$

$$\boldsymbol{\Sigma}_d = \boldsymbol{K}_d^{\frac{1}{2}} \boldsymbol{\Lambda}_d \boldsymbol{\Lambda}_d^T \boldsymbol{K}_d^{\frac{1}{2}}, \tag{B.5}$$

where $\boldsymbol{K}_d^{\frac{1}{2}}$ is a matrix square root of the prior covariance matrix $\boldsymbol{K}_d$ and $\boldsymbol{\nu}_d \in \mathbb{R}^T$ is a vector of variational parameters. This formulation simplifies the KL divergence term for each latent dimension in Equation 4.6 from

$$\mathrm{KL}[q(\boldsymbol{x}_d)||p(\boldsymbol{x}_d|\boldsymbol{t})] = \frac{1}{2}\left(\mathrm{Tr}(\boldsymbol{K}_d^{-1}\boldsymbol{\Sigma}_d) + \log|\boldsymbol{K}_d| - \log|\boldsymbol{\Sigma}_d| + \boldsymbol{\mu}_d^T \boldsymbol{K}_d^{-1} \boldsymbol{\mu}_d - T\right) \tag{B.6}$$

to

$$\mathrm{KL}[q(\boldsymbol{x}_d)||p(\boldsymbol{x}_d|\boldsymbol{t})] = \frac{1}{2}\left(||\boldsymbol{\Lambda}_d||_{\mathrm{F}}^2 - 2\log|\boldsymbol{\Lambda}_d| + ||\boldsymbol{\nu}_d||^2 - T\right). \tag{B.7}$$

In the following, we drop the $\cdot_d$ subscript to remove clutter, and we use the notation $\boldsymbol{\Psi} = \mathrm{diag}(\psi_1, ..., \psi_T)$ with positive elements $\psi_t > 0$, to denote a positive definite diagonal matrix.

B.5.1 Square root of the prior covariance

For a stationary prior covariance $\boldsymbol{K}$, we can directly parameterize $\boldsymbol{K}^{\frac{1}{2}}$ by taking the square root of $k(\cdot,\cdot)$ in the Fourier domain and computing the inverse Fourier transform. For the

RBF kernel used in this work we get

$$k(t_i,t_j) = \exp\left(-\frac{(t_i-t_j)^2}{2\tau^2}\right) \tag{B.8}$$

$$k^{\frac{1}{2}}(t_i,t_j) = \left(\frac{2}{\pi}\right)^{\frac{1}{4}}\left(\frac{\delta t}{\tau}\right)^{\frac{1}{2}}\exp\left(-\frac{(t_i-t_j)^2}{\tau^2}\right). \tag{B.9}$$

In this expression, δt is the time difference between consecutive data points, we have assumed a signal variance of 1 in the prior kernel, and we note that our parameterization only gives rise to the exact matrix square root of the RBF kernel in the limit where $T \gg \tau$. Note that this is the case in the present work since $T \approx 30$ minutes is much larger than the longest timescales learned by bGPFA ($\tau \approx 2$ s). For most experiments in neuroscience, observations are binned such that time is on a regularly spaced grid and our parameterization can be applied directly. In other cases, kernel interpolation should first be used to construct a covariance matrix with Toeplitz structure [76, 75].

B.5.2 Parameterization of the posterior covariance

We now proceed to describe the various parameterizations of $\boldsymbol{\Lambda}$ whose performance is compared in Figure B.5. Other parameterizations are explored in [10].

Diagonal $\boldsymbol{\Lambda}$ We parameterize each latent dimension with $\boldsymbol{\Lambda} = \boldsymbol{\Psi}$. This gives rise to a KL term:

$$2\mathrm{KL}[q(\boldsymbol{x})||p(\boldsymbol{x})] = \sum_t \psi_t^2 + ||\boldsymbol{\nu}||^2 - T - 2\sum_t \log\psi_t. \tag{B.10}$$

We can compute $\boldsymbol{\Lambda}\boldsymbol{\nu}$ in linear time since $\boldsymbol{\Lambda}$ is diagonal which allows for cheap (differentiable) sampling:

$$\boldsymbol{\eta} \sim \mathcal{N}(0,\boldsymbol{I}) \tag{B.11}$$

$$\text{sample} = \boldsymbol{K}^{\frac{1}{2}}(\boldsymbol{\Lambda}\boldsymbol{\eta} + \boldsymbol{\nu}), \tag{B.12}$$

where the multiplication by $\boldsymbol{K}^{\frac{1}{2}}$ is done in $\mathcal{O}(T\log T)$ time in the Fourier domain.

Circulant $\boldsymbol{\Lambda}$ We parameterize each latent dimension with $\boldsymbol{\Lambda} = \boldsymbol{\Psi}\boldsymbol{C}$. Here, $\boldsymbol{C} \in \mathbb{R}^{T\times T}$ is a positive definite circulant matrix with $1 + \frac{T}{2}$ (integer division) free parameters, which we parameterize directly in the Fourier domain as $\hat{\boldsymbol{c}} = \mathrm{rfft}(\boldsymbol{c}) \in \mathbb{R}^{1+T/2}$ where $\boldsymbol{c}$ is the first

column of $\boldsymbol{C}$ with $\hat{\boldsymbol{c}} \geq 0$ elementwise. We compute the KL as

$$2\mathrm{KL}[q(\boldsymbol{x})||p(\boldsymbol{x})] = \left(\sum_t c_t^2\right)\left(\sum_t \psi_t^2\right) + ||\boldsymbol{\nu}||^2 - T - 2\sum_t \log \psi_t - 2\log|\boldsymbol{C}| \tag{B.13}$$

$$\log|\boldsymbol{C}| = \log \hat{c}_1 + \log \hat{c}_{\frac{T}{2}+1} + 2\sum_{i=2}^{\frac{T}{2}} \log \hat{c}_i \qquad \text{(even T)} \tag{B.14}$$

$$\log|\boldsymbol{C}| = \log \hat{c}_1 + 2\sum_{i=2}^{\frac{T+1}{2}} \log \hat{c}_i \qquad \text{(odd T)}, \tag{B.15}$$

where $\boldsymbol{c} = \mathrm{irfft}(\hat{\boldsymbol{c}})$. We can sample differentiably in $\mathcal{O}(T \log T)$ time by computing

$$\boldsymbol{\eta} \sim \mathcal{N}(0, \boldsymbol{I}) \tag{B.16}$$

$$\boldsymbol{C}\boldsymbol{\eta} = \mathrm{irfft}(\hat{\boldsymbol{c}} \odot \mathrm{rfft}(\boldsymbol{\eta})) \tag{B.17}$$

$$\text{sample} = \boldsymbol{K}^{\frac{1}{2}}(\boldsymbol{\Psi}\boldsymbol{C}\boldsymbol{\eta} + \boldsymbol{\nu}), \tag{B.18}$$

where $\odot$ denotes the complex element-wise product.

Low-rank $\boldsymbol{\Lambda}$ We let $\boldsymbol{Q} \in \mathbb{Q}^{T \times r}$ with $\boldsymbol{Q}^T\boldsymbol{Q} = \boldsymbol{I}_r$ and write

$$\boldsymbol{\Lambda} = \boldsymbol{I}_T - \boldsymbol{Q}\boldsymbol{\Psi}\boldsymbol{Q}^T, \tag{B.19}$$

where we now constrain $0 < \psi_i < 1$ to maintain the positive definiteness of $\boldsymbol{\Lambda}$. Technically, keeping $\boldsymbol{Q}$ on the Stiefel manifold (i.e. $\boldsymbol{Q}^T\boldsymbol{Q} = \boldsymbol{I}_r$) is done by (differentiably) computing the QR decomposition of a $T \times r$ matrix of free parameters.

Circulant inverse $\boldsymbol{\Lambda}$ We let $\boldsymbol{C}$ be a circulant positive definite matrix as above and parameterize

$$\boldsymbol{\Lambda} = (\boldsymbol{I} + \boldsymbol{\Psi}\boldsymbol{C}\boldsymbol{\Psi})^{-1}. \tag{B.20}$$

Computing $\boldsymbol{\Lambda}\boldsymbol{\nu}$ products is done using the conjugate gradients algorithm, taking advantage of fast products with $\boldsymbol{\Psi}$ and $\boldsymbol{C}$; the same algorithm is also used to stochastically estimate $\log|\boldsymbol{\Lambda}|$ and its gradient (see the appendix of 60).

Toeplitz inverse $\boldsymbol{\Lambda}$ This proceeds just as for the circulant inverse form, with the circulant matrix $\boldsymbol{C}$ replaced by an arbitrary Toeplitz matrix (also exploiting fast $\boldsymbol{T}\boldsymbol{\nu}$ products):

$$\boldsymbol{\Lambda} = (\boldsymbol{I} + \boldsymbol{\Psi}\boldsymbol{T}\boldsymbol{\Psi})^{-1}. \tag{B.21}$$

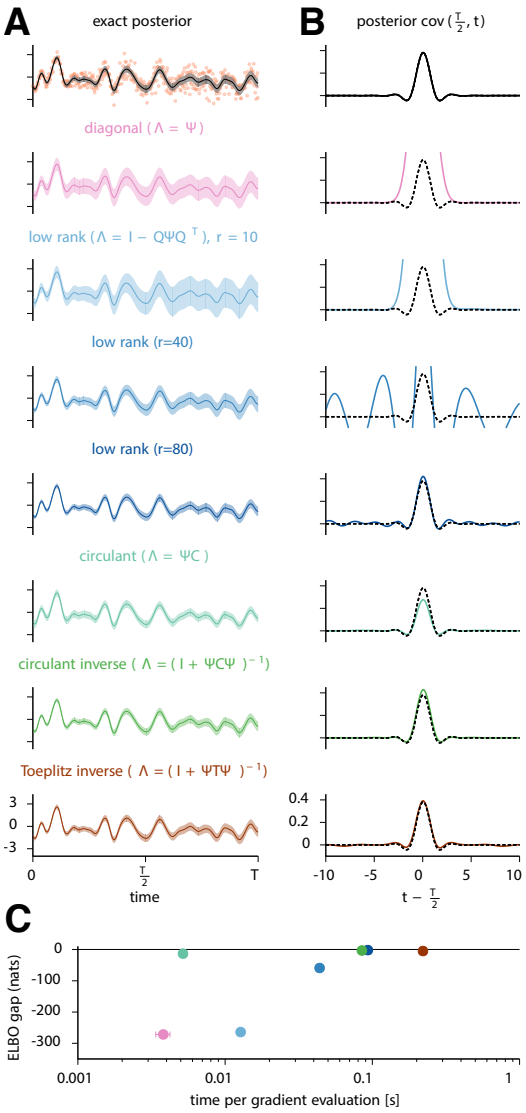


Fig. B.5 **Comparisons of different forms of the approximate posterior $q(\boldsymbol{x})$. (A)** Synthetic data (orange dots) plotted together with the exact posterior (black) as well as the variational posteriors inferred by each whitened parameterization. The solid lines denote the (approximate) posterior means, and shaded areas indicate ± 1 posterior standard deviations. **(B)** Slice through the posterior covariance ($\mathrm{Cov}_{x\sim q(x)}\left[x_{T/2}, x_t\right]$) for the true posterior (top and black dotted lines) and the approximate methods. Each method has different characteristics and the circulant parameterization again provides a good qualitative fit at very low computational cost. **(C)** We defined the 'ELBO gap' of each method as ELBO $-$ LL where LL is the true data log likelihood. We plotted this against the time per gradient evaluation and found that the circulant parameterization achieved high accuracy with cheap gradients.

B.5.3 Numerical comparisons between different parameterizations

To compare these parameterizations, we generated a synthetic dataset (Figure B.5a, orange dots) over $T = 1000$ time bins by drawing samples $\{y_1, \ldots, y_T\}$ as $y_t = x_t + \sigma_t \xi_t$ where $\xi(t) \sim \mathcal{N}(0,1)$ with non-stationary σ_t growing linearly from 0.1 to 0.5 over the whole range $0 \leq t < T$, and $x_i \sim \mathcal{N}(0, \boldsymbol{K}^{1/2}\boldsymbol{K}^{1/2})$ with $\boldsymbol{K}^{1/2}$ given by Equation B.9. We fixed these generative parameters to their ground truth and optimized the ELBO w.r.t. the variational parameters in this simple regression setting. We found that all of the parameterizations accurately recapitulated the GP posterior mean (Figure B.5a). However, the degree to which they captured the non-stationary posterior covariance and data log likelihood varied between methods (Figure B.5b-c). To quantify this, we computed the difference between the asymptotic ELBO of each method and the exact log marginal likelihood. This ELBO

gap was small for the circulant parameterization, the inverse methods, and the low rank parameterization with sufficiently high r. Although the circulant parameterization did not fully capture the non-stationary aspect of the posterior variance, this did not affect the ELBO gap substantially; importantly, however, the circulant parameterization was more than an order of magnitude faster per gradient evaluation than the other methods with comparable accuracy (Figure B.5c). For these reasons as well as the good performance in a latent variable setting (Section 4.3.1, Section 4.3.2), we used the circulant parameterization for all experiments.

B.6 Relation between variational posterior over F and true posterior

Here we show that our parameterization of $q(\boldsymbol{f}_n)$ includes the exact posterior in the case of Gaussian noise.

When the noise model is Gaussian (i.e., $p(\boldsymbol{y}_n|\boldsymbol{f}_n) = \mathcal{N}(\boldsymbol{y}|\boldsymbol{f}_n, \sigma_n^2 I)$), we can compute the posterior over $\boldsymbol{f}_n^* = f_n(\boldsymbol{X}^\star)$ at locations $\boldsymbol{X}^\star$ in closed form:

$$\boldsymbol{f}_n^*|\boldsymbol{X}^\star, \boldsymbol{X}, \boldsymbol{y}_n \sim \mathcal{N}(\boldsymbol{X}^{\star T}\boldsymbol{S}^2\boldsymbol{X}\hat{\boldsymbol{K}}^{-1}\boldsymbol{y}_n, \boldsymbol{X}^{\star T}\boldsymbol{S}(\boldsymbol{I} - \boldsymbol{X}\hat{\boldsymbol{K}}^{-1}\boldsymbol{X}^T)\boldsymbol{S}\boldsymbol{X}^\star) \tag{B.22}$$

where $\hat{\boldsymbol{K}} = \boldsymbol{X}^T\boldsymbol{S}^2\boldsymbol{X} + \sigma_n^2\boldsymbol{I}$. Note that the posterior is low-rank as the rank of $\boldsymbol{I} - \boldsymbol{X}\hat{\boldsymbol{K}}^{-1}\boldsymbol{X}^T$ is at most D. This means that when we do variational inference, we can parameterize our approximate posterior as:

$$q(\boldsymbol{f}_n^*) = \mathcal{N}(\boldsymbol{f}|\boldsymbol{X}^{\star T}\boldsymbol{S}\boldsymbol{v}_n, \boldsymbol{X}^{\star T}\boldsymbol{S}\boldsymbol{L}_n\boldsymbol{L}_n^T\boldsymbol{S}\boldsymbol{X}^\star) \tag{B.23}$$

where $\boldsymbol{v}_n \in \mathbb{R}^D$ and $\boldsymbol{L}_n \in \mathbb{R}^{D\times D}$ are the parameters of the approximate posterior (Section 4.2.2). We see that this parameterization is exact when:

$$\boldsymbol{v}_n = \boldsymbol{S}\boldsymbol{X}\hat{\boldsymbol{K}}^{-1}\boldsymbol{y}_n \tag{B.24}$$

$$\boldsymbol{L}_n\boldsymbol{L}_n^T = \boldsymbol{I} - \boldsymbol{X}\hat{\boldsymbol{K}}^{-1}\boldsymbol{X}^T. \tag{B.25}$$

Note that the right-hand side of Equation B.25 is guaranteed to be positive definite because the true posterior must be positive definite. Importantly, for this parameterization, the KL term in Equation 4.11 simplifies to

$$\mathrm{KL}(q(\boldsymbol{f}_n|\boldsymbol{X})||p(\boldsymbol{f}_n|\boldsymbol{X})) = \mathrm{KL}(\mathcal{N}(\boldsymbol{v}_n, \boldsymbol{L}_n\boldsymbol{L}_n^T)||\mathcal{N}(0, \boldsymbol{I})), \tag{B.26}$$

which is independent of $\boldsymbol{X}$ and allows us to do efficient inference due to the low dimensionality of $\boldsymbol{\nu}_n$ and $\boldsymbol{L}_n$.

B.7 Relation between variational posterior over F and SVGP

For general non-Gaussian noise models, the parameterization in Appendix B.6 will no longer be exact. However, here we show that it is in this case equivalent to a stochastic variational Gaussian process (SVGP; 27). In SVGP, we choose a variational distribution:

$$q(\boldsymbol{u}) = \mathcal{N}(\boldsymbol{u}|\boldsymbol{Z}^T\boldsymbol{S}\boldsymbol{\mu}, \boldsymbol{Z}^T\boldsymbol{S}\boldsymbol{M}\boldsymbol{M}^T\boldsymbol{S}\boldsymbol{Z}) \tag{B.27}$$

at inducing points $\boldsymbol{Z} \in \mathbb{R}^{D\times m}$, where $\boldsymbol{\mu}$ and $\boldsymbol{M}$ are the "whitened" parameters [29]. This gives an approximate posterior:

$$\begin{aligned} q(\boldsymbol{f}^*) &= \mathbb{E}_{q(\boldsymbol{u})}\left[p(\boldsymbol{f}|\boldsymbol{u})\right] && \text{(B.28)}\\ &= \mathcal{N}(\boldsymbol{f}|\boldsymbol{X}^{\star T}\boldsymbol{S}\boldsymbol{\Pi}_z\boldsymbol{\mu}; \boldsymbol{X}^{\star T}\boldsymbol{S}\boldsymbol{\Pi}_z(\boldsymbol{M}\boldsymbol{M}^T - \boldsymbol{I})\boldsymbol{\Pi}_z\boldsymbol{S}\boldsymbol{X}^\star) && \text{(B.29)} \end{aligned}$$

where $\boldsymbol{\Pi}_{\boldsymbol{z}} = \boldsymbol{S}\boldsymbol{Z}(\boldsymbol{Z}^T\boldsymbol{S}^2\boldsymbol{Z})^{-1}\boldsymbol{Z}^T\boldsymbol{S}$. If we choose $m = D$ inducing points such that $\boldsymbol{Z} \in \mathbb{R}^{D\times D}$ and make sure $\boldsymbol{Z}$ has full rank, then $\boldsymbol{\Pi}_z = \boldsymbol{I}$ and thus

$$q(\boldsymbol{f}^*) = \mathcal{N}(\boldsymbol{f}|\boldsymbol{X}^{\star T}\boldsymbol{S}\boldsymbol{\mu}, \boldsymbol{X}^{\star T}\boldsymbol{S}(\boldsymbol{M}\boldsymbol{M}^T - \boldsymbol{I})\boldsymbol{S}\boldsymbol{X}^\star). \tag{B.30}$$

We recover the parameterization in Section 4.2.2 when

$$\boldsymbol{\mu} = \boldsymbol{\nu} \quad \text{and} \quad \boldsymbol{M}\boldsymbol{M}^T - \boldsymbol{I} = \boldsymbol{L}\boldsymbol{L}^T. \tag{B.31}$$

For these more general noise models, the whitened parameterization of $q(\boldsymbol{f})$ still gives rise to a computationally cheap KL divergence that is independent of $\boldsymbol{X}$ as in Equation B.26:

$$\mathrm{KL}(q(\boldsymbol{f}_n|\boldsymbol{X})||p(\boldsymbol{f}_n|\boldsymbol{X})) = \mathrm{KL}(\mathcal{N}(\boldsymbol{\nu}_n, \boldsymbol{L}_n\boldsymbol{L}_n^T)||\mathcal{N}(0,\boldsymbol{I})). \tag{B.32}$$

In summary, we have shown that (i) our parameterization of $q(\boldsymbol{f}_n)$ has sufficient flexibility to learn the true posterior when the noise model is Gaussian (Appendix B.6), and (ii) it is equivalent to performing SVGP where the locations of the inducing points do not matter provided that their rank is at least as high as the number of latent dimensions.

B.8 Automatic relevance determination

Here we briefly consider why introducing a prior over the factor matrix enables automatic relevance determination. These ideas reflect results by Bishop [7] and in Section 4.3.1.

For simplicity, we will first consider the case of factor analysis where $p(\boldsymbol{X}) = \prod_{d,t} \mathcal{N}(x_{dt}; 0, 1)$. This gives rise to a marginal likelihood (with Gaussian noise) equal to

$$\log p(\boldsymbol{Y}) = \sum_t \log \mathcal{N}(\boldsymbol{y}_t; 0, \boldsymbol{C}\boldsymbol{C}^T + \boldsymbol{\Sigma}), \tag{B.33}$$

where $\boldsymbol{\Sigma} = \text{diag}(\sigma_1^2, ..., \sigma_N^2)$ is a diagonal matrix of noise parameters. It is in this case quite clear that the optimal marginal likelihood is a monotonically increasing function of the latent dimensionality, since any marginal likelihood reachable with a certain rank D is also reachable with a larger rank $D' > D$; increasing D can only increase model flexibility. We could in this case threshold the magnitude of the columns of $\boldsymbol{C}$ to subselect more 'informative' dimensions, but this is not inherently different from putting an arbitrary cut-off on the variance explained in PCA, and there is no Bayesian "Occam's razor" built into the method [45].

Consider now the case where we put a unit Gaussian prior on c_{nd}. In this case $\{c_{nd}\}$ are no longer parameters of the model, but rather latent variables to be inferred which intuitively should reduce the risk of overfitting. To expand on this intuition, consider the ELBO (c.f. Section 4.2.1) that results from introducing such a prior over c_{nd}:

$$\log p(\boldsymbol{Y}) \geq \mathbb{E}_{q(\boldsymbol{X})} \left[\log p(\boldsymbol{Y}|\boldsymbol{X})\right] - \sum_{d,t} \text{KL}\left[q(x_{dt}) || \mathcal{N}(0,1)\right] \tag{B.34}$$

$$\log p(\boldsymbol{Y}|\boldsymbol{X}) = \sum_n \log \mathcal{N}(\boldsymbol{y}_n; 0, \boldsymbol{X}^T\boldsymbol{X} + \sigma_n \boldsymbol{I}). \tag{B.35}$$

Here we see that if a dimension d is truly uninformative, it should have $x_{dt} = 0 \, \forall_t$ to avoid contributing noise to the likelihood term via $\boldsymbol{X}^T\boldsymbol{X}$. However, reducing this noise will increase the prior KL term, driving it to infinity in the limit of zero noise since the variational posterior over the d^{th} latent at time t, $q(x_{dt})$, is in this case a delta function at zero. Optimizing the ELBO therefore involves a balance between mitigating the noise induced by $\boldsymbol{X}^T\boldsymbol{X}$ and reducing the KL penalty, with both of these terms contributing to a decreased ELBO compared to the model without uninformative dimensions. Thus the prior over c_{nd} counteracts the overfitting that would normally occur when increasing the latent dimensionality in classical factor analysis, and this Bayesian treatment will lead to a decrease in the ELBO with increasing dimensionality beyond the optimal $D^\star$ that is needed to adequately explain the data.

Finally let us consider the case where we learn the prior scale of the factor matrix such that $c_{nd} \sim \mathcal{N}(0, s_d^2)$ with s_d optimized w.r.t. the ELBO. Critically, the likelihood term now becomes:

$$\log p(\boldsymbol{Y}|\boldsymbol{X}) = \sum_n \log \mathcal{N}(\boldsymbol{y}_n; 0, \boldsymbol{X}^T \boldsymbol{S}^2 \boldsymbol{X} + \sigma_n \boldsymbol{I}). \tag{B.36}$$

with $\boldsymbol{S} = \text{diag}(s_1, \ldots, s_D)$. In this case, adding uninformative dimensions beyond the optimal $D^\star$ still cannot increase the ELBO (in the limit of large N). However, letting $s_d \to 0$ for these superfluous dimensions will prevent them from contributing to $p(\boldsymbol{Y}|\boldsymbol{X})$, thus allowing $q(x_{dt}) \to \mathcal{N}(0,1)$ to drive the prior KL term to zero for these dimensions. In this limit, we recover both the ELBO and the posteriors associated with the $D^\star$- dimensional model. We thus have a built-in Occam's razor which will shave off any uninformative latent dimensions, and these will be identifiable as dimensions for which $s_d \approx 0$ and $q(x_{dt}) \approx \mathcal{N}(0,1)$.

These ideas generalize to GPFA where the posterior over latents will instead approach the GP prior $q(\boldsymbol{x}_d) \approx \mathcal{N}(0, \boldsymbol{K})$ for uninformative dimensions. This corresponds to the limit of $\boldsymbol{\nu} \to 0$ and $\boldsymbol{\Psi} \to \boldsymbol{I}$ in our circulant parameterization in Section 4.2.2 and Appendix B.5. In all of our simulations, we found a clear clustering of dimensions after training with some clustered near zero s_d, and others clustered with much larger s_d (Figure 4.2c and Figure 4.3b). Note that in practice we do not actively truncate the model by discarding dimensions with $s_d \approx 0$ but merely use the terminology to indicate that these dimensions have negligible contributions to the posterior predictive $q(\boldsymbol{y}_n)$, as well as to the latent posteriors $q(\boldsymbol{x}_d)$ for the dimensions with large s_d.

B.9 Most informative dimensions

In this work, we refer to the latent dimensions with the highest values of s_d as the 'most informative dimensions'. We do this because (i) observing the value of the corresponding latent x_d decreases the variance of the expected distribution of neural activity more as s_d increases, and (ii) the Fisher information of x_d increases as s_d increases.

To show this, we consider how the distribution over f_n (the activity of neuron n) given $\boldsymbol{c}_n$ (the n^{th} row of $\boldsymbol{C}$) changes when x_d (the value of the d^{th} latent) is known, and how this varies with s_d. In the following, we omit the $\cdot_n$ subscript for notational simplicity, and we note that f, x_d and c_d are all scalar values. With unknown x_d, f is Gaussian with zero mean and variance $\mathbb{E}_{p(\boldsymbol{x})}\left[\boldsymbol{c}^T \boldsymbol{x}\boldsymbol{x}^T \boldsymbol{c}\right] = \boldsymbol{c}^T \boldsymbol{c}$. Thus,

$$p(f|\boldsymbol{c}) = \mathcal{N}(f; 0, \boldsymbol{c}^T \boldsymbol{c}) \tag{B.37}$$

In contrast, for known x_d, we have

$$p(f|\boldsymbol{c},x_d) = \mathcal{N}(f; c_d x_d, \boldsymbol{c}_{-d}^T \boldsymbol{c}_{-d}), \tag{B.38}$$

where $\boldsymbol{c}_{-d}$ is $\boldsymbol{c}$ with the d^{th} element removed. We thus see that the decrease in variance of f from observing x_d is c_d^2. Finally we can approximate the process of averaging this quantity over neurons by noting that $c_d \sim \mathcal{N}(0, s_d^2)$ and marginalising out $\boldsymbol{c}$:

$$\mathbb{E}_{p(\boldsymbol{c})}[\sigma^2_{f|\boldsymbol{c}} - \sigma^2_{f|\boldsymbol{c},x_d}] = \mathbb{E}_{p(\boldsymbol{c})}[c_d^2] = s_d^2, \tag{B.39}$$

where $\sigma^2_{f|\boldsymbol{c}}$ is the variance of $p(f|\boldsymbol{c})$. Thus, s_d^2 can be interpreted as the expected decrease in the variance of the denoised neural activity f when learning the value of the d^{th} latent.

This can also be understood in information-theoretic terms by considering the Fisher information of the d^{th} latent dimension which is given by

$$\mathcal{I}(x_d|\boldsymbol{c}) = -\mathbb{E}_{p(f|x_d,\boldsymbol{c})}\left[\frac{\partial^2}{\partial x_d^2} \log p(f|x_d,\boldsymbol{c})\right] \tag{B.40}$$

$$= \left[\sum_{d' \neq d} c_{d'}^2\right]^{-1}. \tag{B.41}$$

To relate this quantity to our prior scale parameters $\{s_d\}$, we consider the expectation of the inverse Fisher information:

$$\mathbb{E}_{p(\boldsymbol{c})}[\mathcal{I}(x_d|\boldsymbol{c})^{-1}] = \sum_{d' \neq d} s_{d'}^2. \tag{B.42}$$

For a given set of latent dimensions $[1,D]$ with corresponding $\{s_d\}_1^D$, we thus see that the expected *inverse* Fisher information is *minimized* for the dimension with the highest value of s_d. In Figure 4.2 and Figure 4.3 we use s_d together with the posterior latent mean parameters $\boldsymbol{v}_d$ to identify 'discarded' dimensions.

B.10 Noise models and evaluation of their expectations

Gaussian The Gaussian noise model is given by

$$\log p(y_{nt}|f_{nt}) = -\frac{1}{2}\log(2\pi) - \frac{1}{2}(y_{nt} - f_{nt})^2/\sigma_n^2, \tag{B.43}$$

where σ_n is a learnable parameter. In this case we can easily compute the expected log-density under the approximate posterior analytically:

$$\mathbb{E}_{q(f_{nt}|\boldsymbol{X})}\left[\log p(y_{nt}|f_{nt})\right] = -\frac{1}{2}\left(\log(2\pi) + \frac{(y_{nt}-\mu_{nt})^2 + \Sigma_{ntt}}{\sigma_n^2}\right), \tag{B.44}$$

where $q(\boldsymbol{f}_n|\boldsymbol{X}) = \mathcal{N}(\boldsymbol{f}_n; \boldsymbol{\mu}_n, \boldsymbol{\Sigma}_n)$ and Σ_{ntt} is the approximate posterior variance of neuron n at time t (i.e., the t^{th} diagonal element of $\boldsymbol{\Sigma}_n$).

Poisson The Poisson noise model is given by

$$\log p(y_{nt}|f_{nt}) = y_{nt}\log g(f_{nt}) - g(f_{nt}) - \log(y_{nt}!), \tag{B.45}$$

where g is a link function. If we choose an exponential link function (i.e., $g(x) = \exp(x)$), we can compute in closed-form the expected log-density of the approximate posterior as:

$$\begin{aligned}\mathbb{E}_{q(f_{nt}|\boldsymbol{X})}\left[\log p(y_{nt}|f_{nt})\right] &= \mathbb{E}_{q(f_{nt}|\boldsymbol{X})}\left[y_{nt}f_{nt} - \exp(f_{nt}) - \log(y_{nt}!)\right] &\text{(B.46)}\\ &= y_{nt}\mu_{nt} - \exp\left(\mu_{nt} + \frac{1}{2}\Sigma_{ntt}\right) - \log(y_{nt}!). &\text{(B.47)}\end{aligned}$$

For the analyses shown in Figure 4.2c-d, we use the exponential link function.

For general link functions g, we may not be able to evaluate the expected log-density in closed-form. In this case, we approximate it with Gauss-Hermite quadrature:

$$\mathbb{E}_{q(f_{nt}|\boldsymbol{X})}\left[\log p(y_{nt}|f_{nt})\right] \approx \frac{1}{\sqrt{\pi}}\sum_{i=1}^{k_{\text{GH}}} \omega_i \log p(y_{nt}|f_{nt}^{(i)}) \tag{B.48}$$

where

$$\omega_i = \frac{2^{k_{\text{GH}}-1}k_{\text{GH}}!\sqrt{\pi}}{{k_{\text{GH}}}^2[H_{k_{\text{GH}}-1}(r_i)]^2}, \tag{B.49}$$

$$f_{nt}^{(i)} = \left(\sqrt{2\Sigma_{ntt}}\right)r_i + \mu_{nt}, \tag{B.50}$$

$H_k(r)$ are the physicist's Hermite polynomials, and r_i with $i = 1,\ldots,k$ are roots of $H_k(r)$. For a given order of approximation k_{GH}, we can evaluate both ω_i and r_i using standard numerical software packages such as Numpy. In practice, we find that $k_{\text{GH}} = 20$ gives an accurate approximation to the expected log-density under the approximate posterior. Note that we could also estimate the expectation over $q(f_{nt})$ for general link functions g using a

Monte Carlo estimate, but we use Gauss-Hermite quadrature in this work since it has a lower computational cost and is not stochastic.

Negative binomial The negative binomial noise model is given by

$$\log p(y_{nt}|f_{nt}) = \log \binom{y_{nt}+\kappa_n-1}{y_{nt}} + \kappa_n \log(1-g(f_{nt})) + y \log(g(f_{nt})), \tag{B.51}$$

where $g(f_{nt})$ denotes the probability of success in a Bernoulli trial. Here, each success corresponds to the emission of one spike in bin t, and thus $p(y_{nt}|f_{nt})$ is the distribution over the number of successful trials (spikes) before reaching κ_n failed trials. The link function $g(x) : \mathbb{R} \to [0,1)$ maps f_{nt} to a real number between 0 and 1. In practice we use a sigmoid link-function $g(x) = 1/(1+\exp(-x))$.

In this model, κ_n is a learnable parameter which effectively modulates the overdispersion of the distribution since the mean and variance of $p(y_{nt}|f_{nt})$ are given by:

$$\mu_{NB} = \frac{g(f_{nt})\kappa_n}{1-g(f_{nt})} \tag{B.52}$$

$$\sigma^2_{NB} = \mu_{NB}\left(1+\frac{\mu_{NB}}{\kappa_n}\right). \tag{B.53}$$

This is the parameter which we compare between the ground truth and trained models in Figure 4.2, and we see that the Poisson model is recovered for neuron n as $\kappa_n \to \infty$.

For the negative binomial noise model we cannot compute the expected log-density in closed-form. We instead approximate this expectation using Gauss-Hermite quadrature as described above.

B.11 Implementation

In this section we provide pseudocode for bGPFA (Algorithm 2) with the circulant parameterization for $q(\boldsymbol{X})$ and discuss other implementation details.

Note that we need to sample the full trajectory $\boldsymbol{x}_d$ before subsampling for each batch due to the correlations introduced by $\boldsymbol{K}$. In practice, we run the optimization for 2500 passes over the full data which we found empirically lead to convergence of the ELBO and parameters. We used $M = 20$ Monte Carlo samples for each update step when fitting the synthetic data in Figure 4.2 and $M = 10$ for the primate data. For all models, $q(\boldsymbol{X})$ was initialized at the prior $p(\boldsymbol{X})$. The prior scale parameters were initialized as $s_d = \rho||\boldsymbol{c}_d||_2^2$ where $\boldsymbol{c}_d$ is the d^{th} row of the factor matrix $\boldsymbol{C}$ found by factor analysis [56] and $\rho = 3$ was found empirically to give

Algorithm 2: Bayesian GPFA with automatic relevance determination

input: data $\boldsymbol{Y} \in \mathbb{R}^{N \times T}$, maximum latent dimensionality D, # of Monte Carlo samples M, learning rate γ

parameters: $\theta = \{\{s_d\}_1^D, \{\tau_d\}_1^D, \{\boldsymbol{\nu}_d\}_1^D, \{\tilde{\boldsymbol{c}}_d\}_1^D, \{\boldsymbol{\Psi}_d\}_1^D, \{\boldsymbol{L}_n\}_1^N, \{\hat{\boldsymbol{\nu}}_n\}_1^N, \{\hat{\sigma}_n \text{ or } \kappa_n\}_1^N\}$

while *not converged* **do**

$\nabla\mathcal{L} \leftarrow 0$

for *batch in batches* **do**

%For each of M Monte Carlo samples

for $m = 1 : M$ **do**

% sample from approximate posterior $q(\boldsymbol{X})$

for $d = 1 : D$ **do**

$\boldsymbol{\eta}_d^{(m)} \sim \mathcal{N}(\boldsymbol{0}, \boldsymbol{I}_T)$

$\boldsymbol{k}_d^{\frac{1}{2}} = \sigma_{\frac{1}{2},d} \exp\left(-\frac{(\boldsymbol{t}-t_0)^2}{2\tau_{\frac{1}{2},d}^2}\right)$ `// single column of` $\boldsymbol{K}$

$\boldsymbol{x}_d^{(m)} = \text{Toeplitz\_mult}(\boldsymbol{k}_d^{\frac{1}{2}}, \boldsymbol{\nu}_d + \boldsymbol{C}\boldsymbol{\eta}_d^{(m)})$ `// Appendix B.5`

$\boldsymbol{X}_m = [\boldsymbol{x}_1^{(m)}; \ldots; \boldsymbol{x}_d^{(m)}]$

% compute $q(\boldsymbol{F})$ and $\mathbb{E}_{q(\boldsymbol{F})}[p(\boldsymbol{Y}|\boldsymbol{F})]$

$\hat{\boldsymbol{\mu}}_n = \boldsymbol{X}_m^\top \hat{\boldsymbol{\nu}}_n$ `// variational mean`

$\hat{\boldsymbol{\sigma}}_n^2 = \text{diag}\left(\boldsymbol{X}_m^T \boldsymbol{S}\boldsymbol{L}_n\boldsymbol{L}_n^\top \boldsymbol{S}\boldsymbol{X}_m\right)$

$\log p_{YF}^{(m)} = \sum_{n,t \in batch} \mathbb{E}_{\mathcal{N}(f_{nt}; \hat{\mu}_{nt}, \hat{\sigma}_{nt}^2)}[\log p(y_{nt}|f_{nt})]$ `// Appendix B.10`

% compute KL terms

$\text{KL}_x = \frac{\text{size}(batch)}{\text{size}(data)} \sum_d \text{KL}[q(\boldsymbol{x}_d)||p(\boldsymbol{x}_d)]$ `// Appendix B.5`

$\text{KL}_f = \frac{\text{size}(batch)}{\text{size}(data)} \sum_n \text{KL}[q(\boldsymbol{f}_n)||p(\boldsymbol{f}_n)]$ `// Appendix B.6`

% update gradient with batch gradient

$\tilde{\mathcal{L}} = \frac{1}{M} \sum_m \log p_{YF}^{(m)} - \text{KL}_x - \text{KL}_f$

$\nabla\mathcal{L} \leftarrow \nabla\mathcal{L} + \nabla\tilde{\mathcal{L}}$

% update parameters based on total gradients (we use Adam in practice)

$\theta \leftarrow \theta + \gamma\nabla\mathcal{L}$

good convergence on the primate data. When using a Gaussian noise model, noise variances were initialized as the σ_n^2 found by factor analysis. For negative binomial noise models, we initialized $\kappa_n = \frac{1}{T}\sum_t y_{nt}$ which matches the mean of the distribution to the data for $f = 0$. Length scales τ were initialized at 200 ms for all latent dimensions for the primate data and at $\approx 80\%$ of the ground truth value for the synthetic data. Synthetic data was fitted on a single GPU with 8GB RAM. Primate data was fitted on a single GPU with 12GB RAM and took approximately 30 hours for a single model fit to the full dataset at 25 ms resolution. We also note that when fitting data with a Gaussian noise model, we mean-subtracted the original data, whereas we include explicit mean parameters in the Poisson and negative binomial noise models since they are non-linear (c.f. Appendix B.10).

B.12 Cross-validation and kinematic decoding

In this section we describe the procedure for computing cross-validated errors in Figure 4.2 and performing kinematic decoding analyses in Figure 4.3. In these analyses, expectations over $\boldsymbol{X}$ were computed using the posterior mean of $q(\boldsymbol{X})$ and expectations over $\boldsymbol{F}$ were computed using Monte Carlo samples from $q(\boldsymbol{F})$.

Prediction errors To compute cross-validated errors we divide the time points into a training and a test set, $\mathcal{T}_{train} = \{t_1, t_2, ..., t_{T_{train}}\}$ and $\mathcal{T}_{test} = \{t_{T_{train}+1}, ..., T\}$, and similarly for the neurons $\mathcal{N}_{train}$ and $\mathcal{N}_{test}$. We also define $\mathcal{T}_{tot} = \mathcal{T}_{train} \bigcup \mathcal{T}_{test}$ and $\mathcal{N}_{tot} = \mathcal{N}_{train} \bigcup \mathcal{N}_{test}$. We first fit the generative parameters θ_{gen} of each model to data from all the neurons at the training time points using variational inference:

$$\theta_{gen} = \text{argmax}_{\theta_{gen}} \left[p(\boldsymbol{Y}_{\mathcal{N}_{tot}, \mathcal{T}_{train}} | \theta_{gen}) \right]. \tag{B.54}$$

We then fix the generative parameters and infer a distribution over latents from the training neurons recorded at all time points using a second pass of variational inference:

$$q(\boldsymbol{X}_{1:D, \mathcal{T}_{tot}} | \boldsymbol{Y}_{\mathcal{N}_{train}, \mathcal{T}_{tot}}, \theta_{gen}) \approx p(\boldsymbol{X}_{1:D, \mathcal{T}_{tot}} | \boldsymbol{Y}_{\mathcal{N}_{train}, \mathcal{T}_{tot}}, \theta_{gen}). \tag{B.55}$$

Finally we use the inferred latent states and generative parameters to predict the activity of the test neurons at the test time points

$$\hat{\boldsymbol{Y}}_{\mathcal{N}_{test}, \mathcal{T}_{test}} = \int \boldsymbol{Y} p(\boldsymbol{Y}_{\mathcal{N}_{test}, \mathcal{T}_{test}} | \boldsymbol{X}_{1:D, \mathcal{T}_{test}}, \theta_{gen}) q(\boldsymbol{X}_{1:D, \mathcal{T}_{test}} | \boldsymbol{Y}_{\mathcal{N}_{train}, \mathcal{T}_{tot}}, \theta_{gen}) d\boldsymbol{X}_{1:D, \mathcal{T}_{test}} \tag{B.56}$$

This allows us to compute a cross-validated predictive mean squared error as

$$\varepsilon = \frac{1}{|\mathcal{N}_{test}|\,|\mathcal{T}_{test}|}||\hat{\boldsymbol{Y}}_{\mathcal{N}_{test},\mathcal{T}_{test}} - \boldsymbol{Y}_{\mathcal{N}_{test},\mathcal{T}_{test}}||_2^2. \tag{B.57}$$

Kinematic decoding For kinematic decoding analyses, we only considered the latents and behavior prior to a period of approximately 5 minutes where the monkey disengaged from the task (the first 1430 seconds; Appendix B.3). Cursor positions in the x and y directions were first fitted with cubic splines and velocities extracted as the first derivative of these splines. To evaluate kinematic decoding performance, we followed Keshtkaran et al. [38] and computed the expected activity of all neurons at all time points under our model:

$$\hat{\boldsymbol{Y}} = \int \boldsymbol{Y} p(\boldsymbol{Y}|\boldsymbol{F}) q(\boldsymbol{F}|\boldsymbol{X}) q(\boldsymbol{X}|\boldsymbol{t}) d\boldsymbol{X} d\boldsymbol{F}. \tag{B.58}$$

This can be viewed as the first non-linear step of a decoding model from the latent states $\boldsymbol{X}$. We then performed 10-fold cross-validation where 90% of the data was used to fit a ridge regression model which was tested on the held-out 10% of the data. The regularization strength was determined using 10-fold cross-validation on the 90% training data. The predictive performance was computed as the mean across the 10 folds. Models were fitted and evaluated independently for the hand x and y velocities, and the final performance was computed as the mean variance accounted for across these two dimensions. Results in Section 4.3.2 are reported as mean $\pm$ std across 10 different splits of the data into folds used for cross-validation.