Stochastic Linear Algebra for Scalable Gaussian Processes

David Bindel
09 Oct 2018
Collaborators

• Kun Dong (Cornell CAM)
• David Eriksson (Cornell CAM)
• Jake Gardner (Cornell CS)
• Eric Lee (Cornell CS)
• Hannes Nickisch (Phillips Research)
• Geoff Pleiss (Cornell CS)
• Kilian Weinberger (Cornell CS)
• Andrew Wilson (Cornell ORIE)
Kernel and GP basics
Let $u = (u_1, u_2)$. Given $u_1$, what is $u_2$?

We need an assumption! Two different standard takes.
Let $u = (u_1, u_2)$ s.t. $\|u\|_{K^{-1}}^2 \leq 1$. Given $u_1$, what is $u_2$?

Optimal recovery: $\|u_2 - w\|_{S^{-1}}^2 \leq 1 - \|u_1\|_{(K_{11})^{-1}}^2$

$$w = K_{21}K_{11}^{-1}u_1$$

$$S = K_{22} - K_{21}K_{11}^{-1}K_{12}$$
Let $U = (U_1, U_2) \sim N(0, K)$. Given $U_1 = u_1$, what is $U_2$?

Posterior distribution: $(U_2|U_1 = u_1) \sim N(w, S)$ where

$$w = K_{21}K_{11}^{-1}u_1$$
$$S = K_{22} - K_{21}K_{11}^{-1}K_{12}$$
Kernel functions four ways

Generalizations where $K$ matrix turns into kernel $k(x, y)$:

- Feature maps: $\phi : \mathbb{R}^d \rightarrow \mathcal{H}$, $k(x, y) = \langle \phi(x), \phi(y) \rangle_{\mathcal{H}}$. Approx scheme: $f(x) \approx w^* \phi(x)$ with $\|w\|_{\mathcal{H}}$ minimal.
- Defining a RKHS norm (equiv to $\|w\|_{\mathcal{H}}$).
- Shape functions: $f(x) \approx \sum_j c_j k(x, x_j)$. Equivalent to feature map picture (“kernel trick”).
- Covariance for Gaussian process.


Today I will use GP language.
Gaussian processes (GPs) are

- Key building block for ML and spatio-temporal statistics
- Tightly connected to integral equations, kernel regression
- Straightforward to reason about (just linear algebra)
- But hard to scale to big data (because of dense LA)

Goal today: Make these methods scale!
Basic ingredient: Gaussian Processes (GPs)
Basic ingredient: Gaussian Processes (GPs)

Our favorite continuous distributions over

\[ \mathbb{R} : \quad \text{Normal}(\mu, \sigma^2), \quad \mu, \sigma^2 \in \mathbb{R} \]
\[ \mathbb{R}^n : \quad \text{Normal}(\mu, C), \quad \mu \in \mathbb{R}^n, C \in \mathbb{R}^{n \times n} \]
\[ \mathbb{R}^d \to \mathbb{R} : \quad \text{GP}(\mu, k), \quad \mu : \mathbb{R}^d \to \mathbb{R}, k : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R} \]

More technically, define GPs by looking at finite sets of points:

\[ \forall X = (x_1, \ldots, x_n), x_i \in \mathbb{R}^d, \]
\[ \text{have } f_X \sim N(\mu_X, K_{XX}), \text{ where } \]
\[ f_X \in \mathbb{R}^n, \quad (f_X)_i \equiv f(x_i) \]
\[ \mu_X \in \mathbb{R}^n, \quad (\mu_X)_i \equiv \mu(x_i) \]
\[ K_{XX} \in \mathbb{R}^{n \times n}, \quad (K_{XX})_{ij} \equiv k(x_i, x_j) \]

When \( X \) is unambiguous, we will sometimes just write \( K \).
Call the \textit{kernel} (or \textit{covariance}) function $k$. Required property:

- **Pos def**: $K_{xx}$ is always positive definite

Often desirable:

- **Stationary**: $k(x, y)$ depends only on $x - y$
- **Isotropic**: $k(x, y)$ depends only on $x$ and $||x - y||$

Often want both (sloppy notation: $k = k(r)$).

Common examples (e.g. Matérn, SE) also depend on \textit{hyper-parameters} $\theta$ — suppressed in notation unless needed.
Matérn and SE kernels

![Graph showing various Matérn kernels with different parameters.](image-url)
Observations on kernel matrices

Kernel is *chosen by modeler*

- Choose Matérn / SE for regularity and simplicity
- Rarely have the intuition to pick the “right” kernel
- Common choices are *universal* — can recover anything
  - … with less data for “good” choice (inductive bias)
- Can combine with DNNs ("deep kernel learning")

Properties of kernel matrices:

- Positive definite by design, but not well conditioned!
- Weyl: $k(r) \in C'' \implies |\lambda_n| = o(n^{-\nu-1/2})$
- SE case: eigenvalues decay (super)exponentially
- Adding $\sigma^2 I$ “wipes out” small eigenvalues
Learning parameters and hyperparameters
Being Bayesian

Now consider prior of \( f \sim \text{GP}(\mu, k) \), noisy measurements

\[ f_x \sim y + \epsilon, \quad \epsilon \sim N(0, W), \quad \text{typically } W = \sigma^2 I \]

Posterior is \( f \sim \text{GP}(\mu', k') \) with

\[
\mu'(x) = \mu(x) + K_{xx}c \\
k'(x, x') = K_{xx'} - K_{xx} \tilde{K}^{-1} K_{xx'}
\]

\[
\tilde{K} = K_{xx} + W \\
c = \tilde{K}^{-1} (y - \mu_x)
\]

The expensive bit: solves with \( \tilde{K} \).

NB: Other (linear) measurements also allowed – e.g. derivatives.
Kernel hyper-parameters

How to estimate hyper-parameters $\theta$?

- Bayesian approach? Expensive...
- Usually just do maximum likelihood estimation (MLE)

Likelihood function is same as for a multivariate normal:

$$
\ell(\theta|y) = \frac{1}{\sqrt{\det(2\pi \tilde{K})}} \exp \left( -\frac{1}{2} (y - \mu_x)^T \tilde{K}^{-1} (y - \mu_x) \right).
$$

Of course, we usually work with log-likelihood and derivatives.
Kernel hyper-parameters

How to estimate hyper-parameters $\theta$?

- Bayesian approach? Expensive...
- Usually just do maximum likelihood estimation (MLE)

Log-likelihood function for kernel hypers $\theta$

$$
\mathcal{L}(\theta|y) = \mathcal{L}_y + \mathcal{L}_{|K|} - \frac{n}{2} \log(2\pi)
$$

where (again with $c = \tilde{K}^{-1}(y - \mu_X)$)

$$
\mathcal{L}_y = -\frac{1}{2}(y - \mu_X)^T c,
\mathcal{L}_{|K|} = -\frac{1}{2} \log \det \tilde{K},
\frac{\partial \mathcal{L}_y}{\partial \theta_i} = \frac{1}{2} c^T \left( \frac{\partial \tilde{K}}{\partial \theta_i} \right) c,
\frac{\partial \mathcal{L}_{|K|}}{\partial \theta_i} = -\frac{1}{2} \text{tr} \left( \tilde{K}^{-1} \frac{\partial \tilde{K}}{\partial \theta_i} \right)
$$
In an optimization loop:

1. \[ L = \text{chol}(\tilde{K}(\theta)) \];
2. \[ c = L' \backslash (L \backslash (y - \mu)) \];
3. \[ \text{liky} = -0.5 \ast ((y - \mu)' \ast c) \];
4. \[ \text{likK} = -\sum(\log(\text{diag}(L))) \];
5. \[ \text{for} \ i = 1 : \text{len}(\theta) \]
   6. \[ \text{dliky} = 0.5 \ast (c' \ast dK(\theta, i) \ast c) \];
   7. \[ \text{dlikK} = -0.5 \ast \text{trace}(L' \backslash (L \backslash dk(\theta, i))) \];
6. \[ \text{end} \]
Consider $n$ data points

- Straightforward regression: factor $\tilde{K}$ at $O(n^3)$ cost
- Kernel hyper MLE requires multiple $O(n^3)$ ops
  - To compute $\log \det \tilde{K}$ is $O(n^3)$ per step
  - To compute $\text{tr} \left( \tilde{K}^{-1} \frac{\partial \tilde{K}}{\partial \theta_i} \right)$ is $O(n^3)$ per hyper per step

Two possible work-arounds

- Data-sparse factorization methods
- Methods that avoid factorization (e.g. iterative solvers)
  - Q: how to handle determinants and traces?
Scaling GPs: Factorization approach
Simplest data-sparse approach

For $K$ (nearly) low rank: *partial pivoted Cholesky*

Diagonal element: $1.00e+00$
Simplest data-sparse approach

For $K$ (nearly) low rank: *partial pivoted Cholesky*

Diagonal element: $6.77 \times 10^{-2}$
For $K$ (nearly) low rank: *partial pivoted Cholesky*
Simplest data-sparse approach

For $K$ (nearly) low rank: *partial pivoted Cholesky*

Diagonal element: $5.11e-04$
Simplest data-sparse approach

For $K$ (nearly) low rank: *partial pivoted Cholesky*

Diagonal element: $1.19e-04$
For $K$ (nearly) low rank: *partial pivoted Cholesky*

Diagonal element: $4.18e-05$
Simplest data-sparse approach

For $K$ (nearly) low rank: *partial pivoted Cholesky*

Diagonal element: $8.54e^{-7}$
Simplest data-sparse approach

For $K$ (nearly) low rank: *partial pivoted Cholesky*

Diagonal element: $3.58e-07$
Simplest data-sparse approach

For $K$ (nearly) low rank: *partial pivoted Cholesky*

Diagonal element: $1.92e-07$
Simplest data-sparse approach

Smooth kernel and long length scales: $K$ nearly rank $m \ll n$

- $P(LL^T)P^T$ = partial pivoted Cholesky (select $m$ inducing points); does not require forming $K_{xx}$
- Solve $P(LL^T + \sigma^2 I)P^T c = f_x$ stably by reformulating $c$ as a scaled regularized least squares residual:

$$\text{minimize } \|f_x - Lw\|^2 + \sigma^2 \|w\|^2, \quad c = \sigma^{-2} (f_x - Lw)$$

- Compute $\log \det \tilde{K} = \log \det (L^T L + \sigma^2 I) + 2(n - m) \log \sigma$; similar cheap rearrangement for derivatives.
- Prediction and predictive variance are also cheap.

If $K$ is not low rank, can still use rank-structured factorization.
Scaling GPs: Black box approach
Basic ingredients for “black box” approach

- Fast MVMs with kernel matrices
- Krylov methods for linear solves and matrix functions
- Stochastic estimators: trace, diagonal, and other
Kernel approximations

- Low-rank approximation (via *inducing variables*)
  - Non-smooth kernels, small length scales $\rightarrow$ large rank
  - Only semi-definite
- Sparse approximation
  - OK with SE kernels and short length scales
  - Less good with heavy tails or long length scales
  - May again lose definiteness
- More sophisticated: fast multipole, Fourier transforms
  - Same picture as in integral eq world (FMM, PFFT)
  - Main restriction: low dimensional spaces (2-3D)
- Kernel a model choice — how does approx affect results?
Example: Structured Kernel Interpolation (SKI)

Write \( K_{XX} \approx W^T K_{UU} W \) where

- \( U \) is a uniform mesh of \( m \) points
- \( K_{UU} \) has Toeplitz or block Toeplitz structure
- Sparse \( W \) interpolates values from \( X \) to \( U \)

Apply \( K_{UU} \) via FFTs in \( O(m \log m) \) time.
The power of fast MVMs

With MVMs alone, natural to explore nested Krylov subspaces:

$$\mathcal{K}_{d+1}(\tilde{K}, b) = \text{span}\{b, \tilde{K}b, \tilde{K}^2b, \ldots, \tilde{K}^d b\} = \{p(\tilde{K})b : p \in \mathcal{P}_k\}$$

Lanczos process: expansion + Gram-Schmidt

$$\beta_j q_{j+1} = \tilde{K}q_j - \alpha_j q_j - \beta_{j-1} q_{j-1}$$

Lanczos factorization: \(\tilde{K}Q_k = Q_k \bar{T}_k\) where

\[
Q_k = \begin{bmatrix}
q_1 & q_2 & \ldots & q_k
\end{bmatrix},
\]

\[
\bar{T}_k = \begin{bmatrix}
\alpha_1 & \beta_1 & & \\
\beta_1 & \alpha_2 & \beta_2 & \\
& \beta_2 & \alpha_3 & \beta_3 & \\
& & \ddots & \ddots & \ddots \\
& & & \beta_{k-1} & \alpha_k \\
& & & & \beta_k
\end{bmatrix} = \begin{bmatrix}
T_k \\
\beta_k e_k^T
\end{bmatrix}
\]
The power of fast MVMs

Fast MVM with symmetric $\tilde{K} \implies$ try Lanczos!

- Incrementally computes $\tilde{K}Q = QT$ where
  - $Q$ has orthonormal columns
  - Leading $k$ columns span $k$-dim Krylov space
  - $T$ is tridiagonal

- Building block for
  - Solving linear systems (CG)
  - Approximating eigenvalues
  - Approximating matrix functions: $f(\tilde{K})b$
  - Quadrature vs spectral measure for $\tilde{K}$

- Fast (three-term recurrence) and elegant...
- ... but not forward stable in finite precision
Function application via Lanczos

A computational kernel: $f(\tilde{K})b$

- Run Lanczos from starting vector $b/\|b\|$.
- At $n$ steps in exact arithmetic,
  \[ f(\tilde{K})b = Qf(T)Q^Tb = \|b\|Qf(T)e_1 \]
- Truncate at $k \ll n$ steps, use
  \[ f(\tilde{K})b \approx \|b\|Q_kf(T_k)e_1 \]
- Error analysis hinges on quality of poly approx
  \[ \min_{f \in P_k} \max_{\lambda \in \Lambda(\tilde{K})} |f(\lambda) - \hat{f}(\lambda)| \]
- Compare: Chebyshev methods just use $[\lambda_{\text{min}}, \lambda_{\text{max}}]$.

CG is a special case corresponding to $f(z) = z^{-1}$.  

Tractable traces

CG solves systems with $\tilde{K}$; problem terms are

$$L_{|K|} = -\frac{1}{2} \text{tr} \left( \log \tilde{K} \right)$$

$$\frac{\partial L_{|K|}}{\partial \theta_i} = -\frac{1}{2} \text{tr} \left( \tilde{K}^{-1} \frac{\partial \tilde{K}}{\partial \theta_i} \right)$$

Q: How do we parley fast MVMs into trace computations?
Stochastic trace estimation trick:

• $z \in \mathbb{R}^n$ has independent random entries
• $\mathbb{E}[z_i] = 0$ and $\mathbb{E}[z_i^2] = 1$

Then

$$\mathbb{E}[z^T Az] = \sum_{i,j} a_{ij} \mathbb{E}[z_i z_j] = \text{tr}(A).$$

NB: $\mathbb{E}[z \odot Az] = \text{diag}(A)$.

Standard choices for the probe vector $z$:

• Hutchinson: $z_i = \pm 1$ with probability 0.5
• Gaussian: $z_i \sim N(0,1)$

See Avron and Toledo review, JACM 2011.
Putting it together

For each probe vector $z$ until error bars small enough:

- Run Lanczos from $z/||z||$
- Use Lanczos to estimate $\tilde{K}^{-1}z$ and $\log(\tilde{K})z$
- Dot products yield estimators:

\[
\mathcal{L}_{|K|} = -\frac{1}{2} \mathbb{E} \left[ z^T \log(\tilde{K})z \right]
\]
\[
\frac{\partial \mathcal{L}_{|K|}}{\partial \theta_i} = -\frac{1}{2} \mathbb{E} \left[ (\tilde{K}^{-1}z)^T \left( \frac{\partial \tilde{K}}{\partial \theta_i} z \right) \right]
\]

Cost per probe:

- One Lanczos process
- One matvec per parameter with derivative

Quite effective in practice! And amenable to preconditioning.
Let \( M = P(LL^T + \sigma^t I)P^T \approx \tilde{K} \) with \( L \in \mathbb{R}^{n \times m}, \ m \ll n \):

- **Preconditioned CG**: works (implicitly) with \( M^{-1}\tilde{K} \)
- Note \( \log \det \tilde{K} = \log \det M + \log \det M^{-1}\tilde{K} \)
- Know how to do fast direct solves and \( \log \det \) with \( M \)
- All boils down to generalized Lanczos with \( (\tilde{K}, M) \)
- Smooth kernels (e.g. SE) and long length scales \( \implies \) convergence in few steps
Generalized Lanczos per probe vector involves

- One matvec with $\tilde{K}$ per step
- One solve with $M$ per step
- Barrier between steps
- Low arithmetic intensity (flops / memory access)
- Limited opportunities for parallelism

Idea: Lanczos for several probes in parallel

- Multiply $\tilde{K}$ or $M^{-1}$ by panel of vectors / step
- Improves cache use and parallelism
The whole package

So we have

- Stochastic estimators + Krylov iterations
- Preconditioning to reduce steps to convergence
- Blocking to reduce time per step
- GPU acceleration speeds things up further

For all the tricks together: https://gpytorch.ai
Examples
Example: Rainfall
Example: Rainfall

Map generated by NOAA's National Climatic Data Center, 2007
**Example: Rainfall**

<table>
<thead>
<tr>
<th>Method</th>
<th>( n )</th>
<th>( m )</th>
<th>MSE</th>
<th>Time [min]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lanczos</td>
<td>528k</td>
<td>3M</td>
<td>0.613</td>
<td>14.3</td>
</tr>
<tr>
<td>Scaled eigenvalues</td>
<td>528k</td>
<td>3M</td>
<td>0.621</td>
<td>15.9</td>
</tr>
<tr>
<td>Exact</td>
<td>12k</td>
<td>-</td>
<td>0.903</td>
<td>11.8</td>
</tr>
</tbody>
</table>

- Data: Hourly precipitation data at 5500 weather stations
- Aggregate into daily precipitation
- Total data: 628K entries
- Train on 100K data points, test on remainder
- Use SKI with 100 points per spatial dim, 300 in time
- Comparison: scaled eigenvalues approx, exact solve

NB: This is with an older MATLAB code (GPML), not GPyTorch
Can build other stochastic processes via GPs

- Example: Log-Gaussian Cox process model
  - Models count data (e.g. events in spatial bins)
  - Poisson conditional on intensity function
  - Log intensity drawn from a GP
- Laplace approximation for posterior
- Data set is point pattern of 703 hickory trees in Michigan
Example: Hickory data

Figure 1: Prediction by different methods on the Hickory dataset.

| Method      | $S_f$ | $\ell_1$ | $\ell_2$ | $-\log p(y|\theta)$ | Time [s] |
|-------------|-------|----------|----------|-----------------------|----------|
| Exact       | 0.696 | 0.063    | 0.085    | 1827.56               | 465.9    |
| Lanczos     | 0.693 | 0.066    | 0.096    | 1828.07               | 21.4     |
| Scaled eigs | 0.543 | 0.237    | 0.112    | 1851.69               | 2.5      |

Table 1: Hyper-parameters recovered by different methods
Recovering the Stanford bunny model from 25K noisy normals.


GPyTorch: https://gpytorch.ai