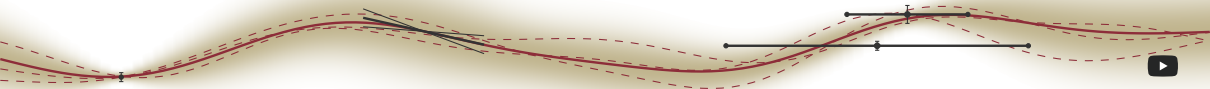
Probabilistic Machine Learning
Lecture 12
The role of linear algebra in GPs

Philipp Hennig and Marvin Pförtner

12 June 2023

Eberhard Karls
Universität
Tübingen

Faculty of Science
Department of Computer Science
Chair for the Methods of Machine Learning



Last week:

1. GPs are indeed *probability distributions on function spaces*. But the probability space is only very weakly identified by their general construction and has very little useful structure.
2. Every covariance function is a kernel, and every kernel is a covariance function.
3. Kernels have *eigenfunctions*, like matrices have eigenvectors. So we can indeed think of them as some kind of "infinite matrix" that spans a space of functions.
4. That space is the *Reproducing Kernel Hilbert Space (RKHS)*. It is identical to the space of *all possible posterior mean functions* of the GP regression method.
5. The posterior covariance function (the Bayesians' *average* square error) is a *worst-case* square error in the RKHS.
6. The sample paths of the Gaussian process do generally not lie in the RKHS of the kernel.

What about the samples?

Draws from a Gaussian process

Question from the Feedback

Why does it matter in practice that our samples can be outside of the RKHS? It seems to me that most of them are concentrated around the mean and thus inside the RKHS anyway?

What about the samples?

Draws from a Gaussian process

Question from the Feedback

Why does it matter in practice that our samples can be outside of the RKHS? It seems to me that most of them are concentrated around the mean and thus inside the RKHS anyway?

- just because two functions are close at every point, e.g. $|f(x) - g(x)| < \varepsilon$ for all $x \in \mathbb{X}$, they don't have to be close in a norm on the function space, i.e. we could have $\|f - g\| \gg \varepsilon$

What about the samples?

Draws from a Gaussian process

Question from the Feedback

Why does it matter in practice that our samples can be outside of the RKHS? It seems to me that most of them are concentrated around the mean and thus inside the RKHS anyway?

- just because two functions are close at every point, e.g. $|f(x) - g(x)| < \varepsilon$ for all $x \in \mathbb{X}$, they don't have to be close in a norm on the function space, i.e. we could have $\|f - g\| \gg \varepsilon$
- in most interesting cases, GP samples do not lie in the RKHS of the kernel k

Theorem (Driscoll's zero-one law, simplified; see Kanagawa et al., 2018, Theorem 4.9)

Let $f \sim \mathcal{GP}(m, k)$ be a Gaussian process with $m \in \mathcal{H}_k$ on the probability space $(\Omega, \mathcal{F}, P)$. If $\mathcal{H}_k$ is infinite-dimensional, then

$$P(f \in \mathcal{H}_k) = 0.$$

What about the samples?

Draws from a Gaussian process

Question from the Feedback

Why does it matter in practice that our samples can be outside of the RKHS? It seems to me that most of them are concentrated around the mean and thus inside the RKHS anyway?

- just because two functions are close at every point, e.g. $|f(x) - g(x)| < \varepsilon$ for all $x \in \mathbb{X}$, they don't have to be close in a norm on the function space, i.e. we could have $\|f - g\| \gg \varepsilon$
- in most interesting cases, GP samples do not lie in the RKHS of the kernel k

Theorem (Driscoll's zero-one law, simplified; see Kanagawa et al., 2018, Theorem 4.9)

Let $f \sim \mathcal{GP}(m, k)$ be a Gaussian process with $m \in \mathcal{H}_k$ on the probability space $(\Omega, \mathcal{F}, P)$. If $\mathcal{H}_k$ is infinite-dimensional, then

$$P(f \in \mathcal{H}_k) = 0.$$

- why should you care in practice? $\Rightarrow$ smoothness properties

What about the samples?

Sample Spaces of Gaussian Processes

for details see e.g. [Kanagawa et al, 2018] or [Pförtner et al., 2022]

- generally, it is difficult to talk about *the* sample space of a GP
- rather, one typically identifies spaces of functions which contain the samples as a subset

What about the samples?

Sample Spaces of Gaussian Processes

for details see e.g. [Kanagawa et al, 2018] or [Pförtner et al., 2022]

- generally, it is difficult to talk about *the* sample space of a GP
- rather, one typically identifies spaces of functions which contain the samples as a subset
- if we know the target function to lie in a certain space ($\Rightarrow$ prior knowledge), then we should choose the kernel of the GP such that its sample space matches that space as good as possible
  - this can accelerate learning

What about the samples?

Sample Spaces of Gaussian Processes

for details see e.g. [Kanagawa et al, 2018] or [Pförtner et al., 2022]

- generally, it is difficult to talk about *the* sample space of a GP
- rather, one typically identifies spaces of functions which contain the samples as a subset
- if we know the target function to lie in a certain space ($\Rightarrow$ prior knowledge), then we should choose the kernel of the GP such that its sample space matches that space as good as possible
  - this can accelerate learning
- examples:
  - $\mathbb{R}^{\mathbb{X}}$ (too large for practical use)
  - Banach space $C(\mathbb{X})$ of continuous functions
  - Banach space $C^k(\overline{\mathbb{X}})$ of k-times continuously differentiable functions (e.g. for derivative observations in Bayesian optimization)
  - Sobolev spaces $W_2^k(\mathbb{X})$ (e.g. for inferring PDE solutions)
  - powers of RKHSs (see next slide)

GP samples are not in the RKHS!

But almost …

Theorem (Kanagawa, 2018. Restricted from Steinwart, 2017, itself generalized from Driscoll, 1973)
Let $\mathcal{H}_k$ be a RKHS and $0 < \theta \leq 1$. Consider the **θ-power of $\mathcal{H}_k$** given by

$$\mathcal{H}_k^\theta = \left\{ f(x) := \sum_{i \in I} \alpha_i \lambda_i^{\theta/2} \phi_i(x) \text{ such that } \|f\|_{\mathcal{H}_k}^2 := \sum_{i \in I} \alpha_i^2 < \infty \right\} \quad \text{with} \quad \langle f, g \rangle_{\mathcal{H}_k} := \sum_{i \in I} \alpha_i \beta_i.$$

Then,

$$\sum_{i \in I} \lambda_i^{1-\theta} < \infty \quad \Rightarrow \quad f \sim \mathcal{GP}(0, k) \in \mathcal{H}_k^\theta \text{ with prob. 1}$$

GP **samples are not in the RKHS**. They belong to a kind of "completion" of the RKHS (but that completion can be strictly larger than the RKHS).

How expensive is GP regression?
What if the dataset is very large?

The inside view on GP regression

a deeper look into the probabilistic ML stack

- **Application Layer:** We are learning a function $f : \mathbb{X} \to \mathbb{R}$ from input-output pairs $X, \boldsymbol{y} = [(x_i, y_i)]_{i=1,\dots,N}$. This is a *supervised machine learning* problem

The inside view on GP regression

a deeper look into the probabilistic ML stack

- **Application Layer:** We are learning a function $f : \mathbb{X} \to \mathbb{R}$ from input-output pairs $X, \boldsymbol{y} = [(x_i, y_i)]_{i=1,\dots,N}$. This is a *supervised machine learning* problem
- **Model Layer:** We are modeling f as a sample from a *Gaussian process* $p(f) = \mathcal{GP}(\mu, k)$ and the data $\boldsymbol{y}$ as being iid. observations of $f_X := f(X) = [f(x_1), \dots, f(x_N)]$ with Gaussian "noise", i.e.

$$p(\boldsymbol{y} \mid f_X) = \mathcal{N}(\boldsymbol{y} \mid f_X, \sigma^2 I_N),$$

and performing *Bayesian inference* to obtain the posterior $p(f \mid \boldsymbol{y})$.

The inside view on GP regression

a deeper look into the probabilistic ML stack

- **Application Layer:** We are learning a function $f : \mathbb{X} \to \mathbb{R}$ from input-output pairs $X, \boldsymbol{y} = [(x_i, y_i)]_{i=1,\dots,N}$. This is a *supervised machine learning* problem
- **Model Layer:** We are modeling f as a sample from a *Gaussian process* $p(f) = \mathcal{GP}(\mu, k)$ and the data $\boldsymbol{y}$ as being iid. observations of $f_X := f(X) = [f(x_1), \dots, f(x_N)]$ with Gaussian "noise", i.e.

$$p(\boldsymbol{y} \mid f_X) = \mathcal{N}(\boldsymbol{y} \mid f_X, \sigma^2 I_N),$$

  and performing *Bayesian inference* to obtain the posterior $p(f \mid \boldsymbol{y})$.
- **Object Layer:** We are calling

```python
def zero_mean(x):
    return jnp.zeros_like(x[:, 0])

def Matern_5(x1, x2, l=1.0):
    r = jnp.sqrt(jnp.sum((x1 - x2) ** 2, axis=-1) / l**2)
    return (1.0 + jnp.sqrt(5) * r + 5.0 / 3.0 * r**2) * jnp.exp(-jnp.sqrt(5) * r)

prior = GaussianProcess(zero_mean, Matern_5)
posterior = prior.condition(Y, X, sigma)
```

The inside view on GP regression

a deeper look into the probabilistic ML stack

- **Application Layer:** We are learning a function $f : \mathbb{X} \to \mathbb{R}$ from input-output pairs $X, \boldsymbol{y} = [(x_i, y_i)]_{i=1,\dots,N}$. This is a *supervised machine learning* problem
- **Model Layer:** We are modeling f as a sample from a *Gaussian process* $p(f) = \mathcal{GP}(\mu, k)$ and the data $\boldsymbol{y}$ as being iid. observations of $f_X := f(X) = [f(x_1), \dots, f(x_N)]$ with Gaussian "noise", i.e.

$$p(\boldsymbol{y} \mid f_X) = \mathcal{N}(\boldsymbol{y} \mid f_X, \sigma^2 I_N),$$

  and performing *Bayesian inference* to obtain the posterior $p(f \mid \boldsymbol{y})$.
- **Object Layer:** We are calling

```python
def zero_mean(x):
    return jnp.zeros_like(x[:, 0])

def Matern_5(x1, x2, l=1.0):
    r = jnp.sqrt(jnp.sum((x1 - x2) ** 2, axis=-1) / l**2)
    return (1.0 + jnp.sqrt(5) * r + 5.0 / 3.0 * r**2) * jnp.exp(-jnp.sqrt(5) * r)

prior = GaussianProcess(zero_mean, Matern_5)
posterior = prior.condition(Y, X, sigma)
```

- But this code only builds datastructures. What happens when we call `posterior(x)` on $x \in \mathbb{X}^m$?

The inside view on GP regression

a deeper look into the probabilistic ML stack

- Math Layer:

$$\mu_y(\bullet) = \mu(\bullet) + k(\bullet, X)(k(X,X) + \sigma^2 I_N)^{-1}(\boldsymbol{y} - \mu(X))$$
$$k_y(\bullet, \circ) = k(\bullet, \circ) - k(\bullet, X)(k(X,X) + \sigma^2 I_N)^{-1} k(X, \circ)$$

```python
def _mean(self, x):
    x = jnp.asarray(x)
    return (
        self.prior.m(x)
        + self.prior.k(x[..., None, :], self.X[None, :, :])
        @ self.representer_weights
    )


@functools.partial(jnp.vectorize, signature="(d),(d)->()", excluded={0})
def _covariance(self, a, b):
    return self.prior.k(a, b) - self.prior.k(
        a, self.X
    ) @ jax.scipy.linalg.cho_solve(
        self.predictive_covariance_cho,
        self.prior.k(self.X, b),
    )
```

The inside view on GP regression

a deeper look into the probabilistic ML stack

- Numerics Layer:

```python
@functools.cached_property
def predictive_covariance(self):
    return self.prior.k(self.X[:, None, :], self.X[None, :, :]) + self.epsilon.Sigma


@functools.cached_property
def predictive_mean(self):
    return self.prior.m(self.X) + self.epsilon.mu


@functools.cached_property
def predictive_covariance_cho(self):
    # this is the core computation -- the *training* step
    return jax.scipy.linalg.cho_factor(self.predictive_covariance)


@functools.cached_property
def representer_weights(self):
    return jax.scipy.linalg.cho_solve(
        self.predictive_covariance_cho,
        self.y - self.predictive_mean,
    )
```

How do we actually *solve* linear systems of equations?

LU Decomposition: Let $A^{(1)} = A \in \mathbb{R}^{n\times n}, L^{(1)} = L \in \mathbb{R}^{n\times n}, U^{(1)} = U \in \mathbb{R}^{n\times n}$ and partition recursively

$$A^{(i)} = \left(\begin{array}{c|c} \alpha^{(i)} & (u^{(i)})^\intercal \\ \hline b^{(i)} & B^{(i)} \end{array}\right), \quad L^{(i)} = \left(\begin{array}{c|c} 1 & \\ \hline l^{(i)} & L^{(i+1)} \end{array}\right), \quad U^{(i)} = \left(\begin{array}{c|c} \alpha^{(i)} & (u^{(i)})^\intercal \\ \hline & U^{(i+1)} \end{array}\right),$$

such that always $A^{(i)} = L^{(i)}U^{(i)}$, and therefore

$$l^{(i)} = \frac{1}{\alpha^{(i)}} b^{(i)}$$
$$A^{(i+1)} := L^{(i+1)}U^{(i+1)} = B^{(i)} - l^{(i)}(u^{(i)})^\intercal$$

If all diagonal elements $\alpha^{(i)}$, so called *pivots*, are non-zero, the recursion terminates and an LU decomposition of $A \in \mathrm{GL}(n)$ exists. $\implies$ choose element with largest abs. value per column as pivot.

Computational complexity: $\#\text{flops} \simeq \sum_{i=1}^{n} \underbrace{2(n-i)^2}_{\text{Cost of } A^{(i+1)}} = 2\sum_{i=1}^{n-1} i^2 \simeq \frac{2}{3}n^3$

Solution of Linear Systems via LU Decomposition

Amortizing computational cost by reusing a known LU decomposition.

To solve a linear system given an LU decomposition, decompose into two systems.

$$\boldsymbol{L}\boldsymbol{U}\boldsymbol{x} = \boldsymbol{b} \iff \boldsymbol{L}(\boldsymbol{U}\boldsymbol{x}) = \boldsymbol{b} \iff \boldsymbol{L}\boldsymbol{y} = \boldsymbol{b} \wedge \boldsymbol{U}\boldsymbol{x} = \boldsymbol{y}$$

Then solve $\boldsymbol{L}\boldsymbol{y} = \boldsymbol{b}$ by forward substitution and $\boldsymbol{U}\boldsymbol{x} = \boldsymbol{y}$ by backward substitution.

Forward Substitution: Let $\boldsymbol{L}^{(n)} = \boldsymbol{L} \in \mathbb{R}^{n \times n}$ lower triangular, $\boldsymbol{y}^{(n)} = \boldsymbol{y} \in \mathbb{R}^n$, $\boldsymbol{b}^{(n)} = \boldsymbol{b} \in \mathbb{R}^n$ and partition recursively

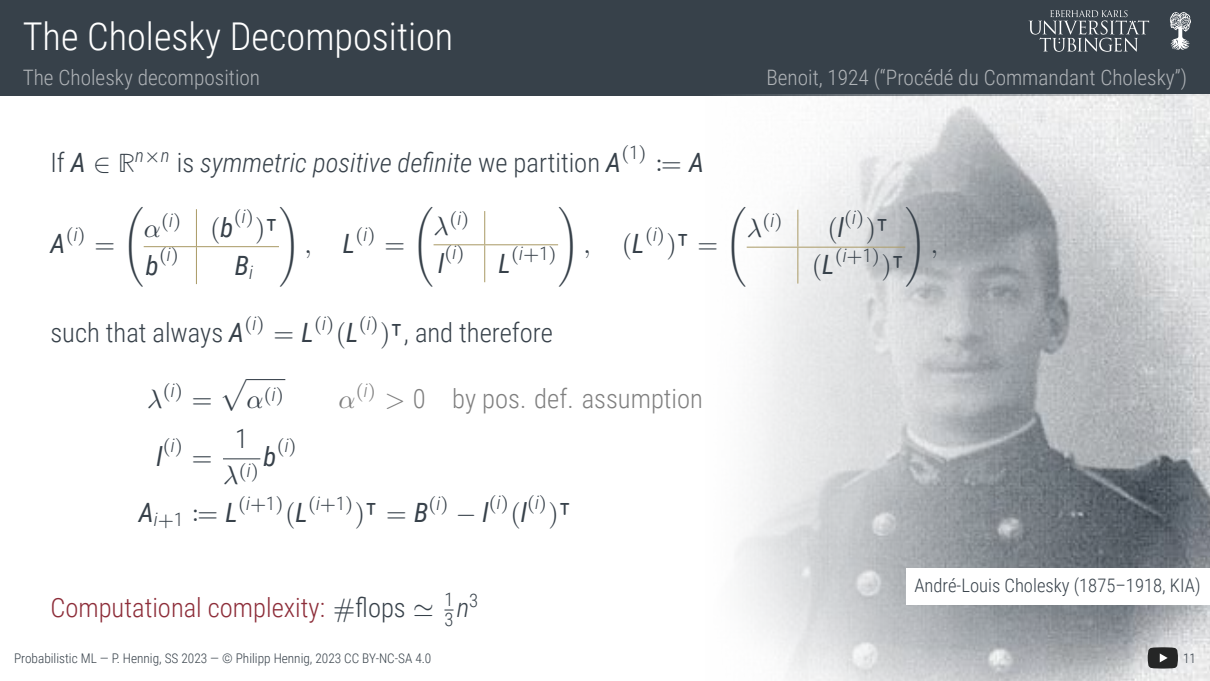
$$\boldsymbol{L}^{(i)} = \left(\begin{array}{c|c} \boldsymbol{L}^{(i-1)} & \\ \hline (\boldsymbol{l}^{(i-1)})^\intercal & \lambda^{(i)} \end{array}\right), \quad \boldsymbol{y}^{(i)} = \begin{pmatrix} \boldsymbol{y}^{(i-1)} \\ \hline \gamma^{(i)} \end{pmatrix}, \quad \boldsymbol{b}^{(i)} = \begin{pmatrix} \boldsymbol{b}^{(i-1)} \\ \hline \beta^{(i)} \end{pmatrix}.$$

Setting $\boldsymbol{L}^{(i)}\boldsymbol{y}^{(i)} = \boldsymbol{b}^{(i)}$ we obtain

$$(\boldsymbol{l}^{(i-1)})^\intercal \boldsymbol{y}^{(i-1)} + \lambda^{(i)}\gamma^{(i)} = \beta^{(i)} \iff \gamma^{(i)} = \frac{\beta^{(i)} - (\boldsymbol{l}^{(i-1)})^\intercal \boldsymbol{y}^{(i-1)}}{\lambda_i}, \qquad \gamma_1 = \frac{\beta_1}{\lambda_1}$$

Computational complexity: $\#\text{flops} \simeq \sum_{i=1}^{n} \underbrace{2i}_{\text{Cost of } (\boldsymbol{l}^{(i-1)})^\intercal \boldsymbol{y}^{(i-1)}} \simeq n^2$

Computational complexity of k solves with the same system matrix $\boldsymbol{A}$: $\#\text{flops} \simeq \frac{2}{3}n^3 + 2kn^2$



Benoit, 1924 ("Procédé du Commandant Cholesky")

If $A \in \mathbb{R}^{n \times n}$ is *symmetric positive definite* we partition $A^{(1)} := A$

$$A^{(i)} = \left(\begin{array}{c|c} \alpha^{(i)} & (b^{(i)})^\intercal \\ \hline b^{(i)} & B_i \end{array}\right), \quad L^{(i)} = \left(\begin{array}{c|c} \lambda^{(i)} & \\ \hline l^{(i)} & L^{(i+1)} \end{array}\right), \quad (L^{(i)})^\intercal = \left(\begin{array}{c|c} \lambda^{(i)} & (l^{(i)})^\intercal \\ \hline & (L^{(i+1)})^\intercal \end{array}\right),$$

such that always $A^{(i)} = L^{(i)}(L^{(i)})^\intercal$, and therefore

$$\begin{aligned} \lambda^{(i)} &= \sqrt{\alpha^{(i)}} \qquad \alpha^{(i)} > 0 \quad \text{by pos. def. assumption} \\ l^{(i)} &= \frac{1}{\lambda^{(i)}} b^{(i)} \\ A_{i+1} &:= L^{(i+1)}(L^{(i+1)})^\intercal = B^{(i)} - l^{(i)}(l^{(i)})^\intercal \end{aligned}$$

Computational complexity: #flops $\simeq \frac{1}{3}n^3$

André-Louis Cholesky (1875–1918, KIA)

The Cholesky Decomposition

Pseudocode

Algorithm 1 Cholesky Decomposition

Input: spd matrix A

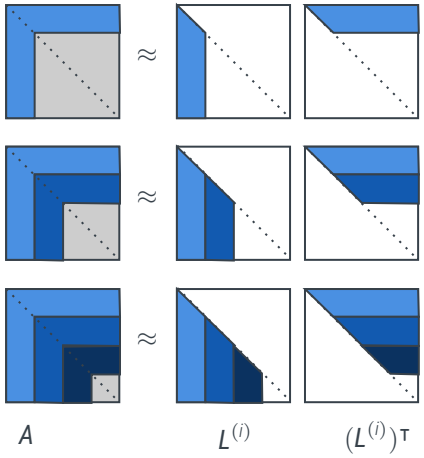
Output: lower triangular L, s.t. $LL^\intercal = A$

```
1 procedure CHOLESKY(A)
2     A' ← A
3     for i ∈ {1, ..., n} do
4         l^(i) ← A'_:i / √A'_ii = A'(e_i / ||e_i||_A')
5         A' ← A' − l^(i)(l^(i))^⊺ = A − L^(i)(L^(i))^⊺
6         L^(i) ← ( L^(i−1)   l^(i) )
7     end for
8     return L^(i)
9 end procedure
```

$i = 1$

$i = 2$

$i = 3$



Note that iteration i of this process is $\mathcal{O}((N-i)^2)$. The first step has to "touch" all N data points.

- Gaussian process regression – training a learning machine – reduces, computationally to decomposing a matrix
- The (pivoted) Cholesky decomposition is the computationally stable, efficient way to do this (numerically) exactly
- Both the posterior mean and posterior covariance can be computed efficiently once the decomposition is available

Cholesky decompositions are $\mathcal{O}(N^3)$. This isn't good enough for big data.
But it's not "being Bayesian" (computing the posterior covariance) that causes this cost!
The point estimate has the same complexity.

Deep learning is $\mathcal{O}(N)$, maybe even $\mathcal{O}(1)$. GPs and kernel machines are $\mathcal{O}(N^3)$. Why?

Grass is greener on the deep side

Is Deep Learning more scalable than shallow learning?

Training a deep network is (potentially regularized) *empirical risk minimization* (ERM) to find the weights $\boldsymbol{w}$ that minimize

$$\boldsymbol{w}_* = \arg\min_{\boldsymbol{w}\in\mathbb{R}^D} \mathcal{L}(\boldsymbol{w}) = \arg\min_{\boldsymbol{w}\in\mathbb{R}^D} \sum_{i=1}^{N} \ell(y_i, f(\boldsymbol{w}, \boldsymbol{x}_i)) + r(\boldsymbol{w})$$

This is commonly done using a *stochastic, first-order method* like SGD, Adam, etc., which involves repeatedly calling the *mini-batch* gradient $g(\boldsymbol{w})$ constructed from

$$\nabla\mathcal{L}(\boldsymbol{w}) = \frac{1}{N}\sum_{i=1}^{N} \nabla\ell(y_i, f(\boldsymbol{w}, \boldsymbol{x}_i)) + \nabla r(\boldsymbol{w}) \approx \frac{1}{B}\sum_{j=1}^{B} \nabla\ell(y_{i(j)}, f(\boldsymbol{w}, \boldsymbol{x}_{i(j)})) + \nabla r(\boldsymbol{w}) =: g(\boldsymbol{w})$$

Due to random sampling of the mini-batch, the mini-batch gradient $g(\boldsymbol{w})$ is a random variable. Since evaluating $g(\boldsymbol{w})$ is $\mathcal{O}(B)$, the cost of each iteration is ostensibly independent of N.

Can GPs be trained with SGD?

Step 1: Re-phrasing inference as an optimization problem

- The posterior mean μ_y evaluated at the training data X is the *mode* of $f(X) \mid y$:

$$\begin{aligned}\mu_y(X) &= \arg\max_{f_X \in \mathbb{R}^N} \log p(f_X \mid \boldsymbol{y}) = \arg\max_{f_X \in \mathbb{R}^N} \log p(\boldsymbol{y} \mid f_X) + \log p(f_X) \\ &= \arg\min_{f_X \in \mathbb{R}^N} -\log p(\boldsymbol{y} \mid f_X) - \log p(f_X) \\ &= \arg\min_{f_X \in \mathbb{R}^N} \frac{1}{2\sigma^2} \sum_{i=1}^{N} \underbrace{\|y_i - (f_X)_i\|^2}_{\ell(f_X, y_i, x_i)} + \underbrace{\frac{1}{2}(f_X - \mu_X)^\intercal k_{XX}^{-1}(f_X - \mu_X)}_{r(f_X)} + \text{const.}\end{aligned}$$

- it suffices to find $\mu_y(X)$, due to conditional independence $f(x_*) \perp\!\!\!\perp \boldsymbol{y} \mid f(X)$:

$$p(f_\bullet \mid \boldsymbol{y}) = \int p(f_\bullet \mid f_X) p(f_X \mid \boldsymbol{y}) \, \mathrm{d}f_X = \int \mathcal{N}(f_\bullet; \mu_{f_\bullet} + k_{\bullet X} k_{XX}^{-1}(f_X - \mu_X), k_{\bullet\bullet} - k_{\bullet X} k_{XX}^{-1} k_{X\bullet}) \, \mathrm{d}p(f_X \mid \boldsymbol{y})$$

- So we can train a GP by minimizing a least-squares loss! **First problem:** The model is nonparametric, so the "weights" are the datapoints themselves. The gradient itself will thus contain N terms, and we will always be at least $\mathcal{O}(N)$.

The cost of being nonparametric

but not of being probabilistic

$$\mu_y(X) = \arg\min_{f_X \in \mathbb{R}^N} \frac{1}{2\sigma^2} \sum_{i=1}^{N} \underbrace{\|y_i - (f_X)_i\|^2}_{\ell(f_X, y_i, x_i)} + \underbrace{\frac{1}{2}(f_X - \mu_X)^\intercal k_{XX}^{-1}(f_X - \mu_X)}_{r(f_X)}$$

- Nonparametric models ("infinitely-wide neural networks") have no finite-dimensional representation of the model. The weight space grows with the number of data points (see representer theorems in SML)
- This has nothing to do with being probabilistic / Bayesian.
- Possible solutions:
  - returning to a parametric representation $f(x) = \phi(x)\boldsymbol{w}$.
  - approximating the representers in some other, finite representation that is somehow adapted to the problem ($\rightarrow$ sparse GP regression, inducing point methods, spectral methods, …)
- but let's keep going for now and accept that we have to deal with N parameters.

Computing Gradients

For the least-squares loss, the gradient is "just" a matrix-vector product

$$\mu_y(X) = \arg\min_{f_X \in \mathbb{R}^N} \frac{1}{2\sigma^2} \sum_{i=1}^{N} \underbrace{\|y_i - (f_X)_i\|^2}_{\ell(f_X, y_i, x_i)} + \underbrace{\frac{1}{2}(f_X - \mu_X)^\intercal k_{XX}^{-1}(f_X - \mu_X)}_{r(f_X)}$$

- We could try to implement this loss directly, but that requires k_{XX}^{-1}, which is $\mathcal{O}(N^3)$.
- Instead, note that we already know the solution *on paper*:

$$\mu_y(\bullet) = \mu_\bullet + k_{\bullet X}(k_{XX} + \sigma^2 I_N)^{-1}(y - \mu_X) = \mu_\bullet + k_{\bullet X}\boldsymbol{\alpha}$$

where $\boldsymbol{\alpha}$ is the vector that minimizes the quadratic function (with $\tilde{\boldsymbol{y}} = y - \mu_X$)

$$\begin{aligned}\mathcal{L}(\boldsymbol{\alpha}) &= \frac{1}{2}((k_{XX} + \sigma^2 I_N)^{-1}\tilde{\boldsymbol{y}} - \boldsymbol{\alpha})^\intercal (k_{XX} + \sigma^2 I_N)((k_{XX} + \sigma^2 I_N)^{-1}\tilde{\boldsymbol{y}} - \boldsymbol{\alpha}) \\ &= \frac{1}{2}\boldsymbol{\alpha}^\intercal (k_{XX} + \sigma^2 I_N)\boldsymbol{\alpha} - \tilde{\boldsymbol{y}}^\intercal \boldsymbol{\alpha} + \text{const.}\end{aligned}$$

with gradient $\nabla_{\boldsymbol{\alpha}} \mathcal{L}(\boldsymbol{\alpha}) = (k_{XX} + \sigma^2 I_N)\boldsymbol{\alpha} - \tilde{\boldsymbol{y}}$

CODE

What about the uncertainty?

getting the posterior covariance from auto-diff

First approach: Laplace approximation (exact in this case)

$$
\begin{aligned}
\log p(f_X \mid \boldsymbol{y}) &= \log p(\boldsymbol{y} \mid f_X) + \log p(f_X) + \text{const.} \\
&= \sum_{i=1}^{N} -\frac{1}{2\sigma^2}\|y_i - (f_X)_i\|^2 - \frac{1}{2}(f_X - \mu_X)^\intercal k_{XX}^{-1}(f_X - \mu_X) + \text{const.} \\
\nabla \log p(f_X \mid \boldsymbol{y}) &= \sigma^{-2}(y - f_X) - k_{XX}^{-1}(f_X - \mu_X) \\
\nabla\nabla^\intercal \log p(f_X \mid \boldsymbol{y}) &= -(\sigma^{-2} I + k_{XX}^{-1}) \qquad \text{(note this is independent of } f_X\text{)}
\end{aligned}
$$

For Gaussians, $\nabla\nabla^\intercal \log \mathcal{N}(x; m, V) = -V^{-1}$. Thus, the posterior covariance on f_X is

$$
\operatorname{cov}(f_X, f_X) = (\sigma^{-2} I + k_{XX}^{-1})^{-1} = k_{XX} - k_{XX}(\sigma^2 I + k_{XX})^{-1} k_{XX}
$$

We still need to compute the inverse of $k_{XX} + \sigma^2 I$ (at cost in $\mathcal{O}(N^3)$) if we want to be uncertain…

What about the uncertainty?

getting the posterior covariance from auto-diff

Second approach: Jacobian/Sensitivity of minimizer

$$\begin{aligned}\boldsymbol{\alpha}^\star &= \arg\min_{\alpha\in\mathbb{R}^N} \frac{1}{2}\boldsymbol{\alpha}^\intercal(k_{XX}+\sigma^2 I_N)\boldsymbol{\alpha} - (\underbrace{\boldsymbol{y}-\mu_X}_{=\tilde{\boldsymbol{y}}})^\intercal\boldsymbol{\alpha} \\ &= (k_{XX}+\sigma^2 I_N)^{-1}(\boldsymbol{y}-\mu_X) \\ \Rightarrow \qquad \frac{\mathrm{d}\boldsymbol{\alpha}^\star}{\mathrm{d}\boldsymbol{y}} &= (k_{XX}+\sigma^2 I_N)^{-1}\end{aligned}$$

This means that the posterior covariance is given by

$$\begin{aligned}k_{\boldsymbol{y}}(\bullet,\circ) &= k_{\bullet\circ} - k_{\bullet X}(k_{XX}+\sigma^2 I_N)^{-1}k_{X\circ} \\ &= k_{\bullet\circ} - k_{\bullet X}\left(\frac{\mathrm{d}\boldsymbol{\alpha}^\star}{\mathrm{d}\boldsymbol{y}}\right)k_{X\circ}\end{aligned}$$

What about the uncertainty?

getting the posterior covariance from auto-diff

Second approach: Jacobian/Sensitivity of minimizer

$$\boldsymbol{\alpha}^\star = \arg\min_{\alpha \in \mathbb{R}^N} \frac{1}{2}\boldsymbol{\alpha}^\intercal (k_{XX} + \sigma^2 I_N)\boldsymbol{\alpha} - (\underbrace{\boldsymbol{y} - \mu_X}_{=\tilde{\boldsymbol{y}}})^\intercal \boldsymbol{\alpha}$$

$$= (k_{XX} + \sigma^2 I_N)^{-1}(\boldsymbol{y} - \mu_X)$$

$$\Rightarrow \qquad \frac{\mathrm{d}\boldsymbol{\alpha}^\star}{\mathrm{d}\boldsymbol{y}} = (k_{XX} + \sigma^2 I_N)^{-1}$$

This means that the posterior covariance is given by

$$k_{\boldsymbol{y}}(\bullet, \circ) = k_{\bullet\circ} - k_{\bullet X}(k_{XX} + \sigma^2 I_N)^{-1} k_{X\circ}$$

$$= k_{\bullet\circ} - k_{\bullet X}\left(\frac{\mathrm{d}\boldsymbol{\alpha}^\star}{\mathrm{d}\boldsymbol{y}}\right) k_{X\circ}$$

Still at least $\Omega(N^3)$ time complexity to compute the posterior covariance exactly…

What about the uncertainty?

getting the posterior covariance from auto-diff

Second approach: Jacobian/Sensitivity of minimizer

$$\begin{aligned}\boldsymbol{\alpha}^\star &= \arg\min_{\alpha \in \mathbb{R}^N} \frac{1}{2}\boldsymbol{\alpha}^\intercal(k_{XX} + \sigma^2 I_N)\boldsymbol{\alpha} - (\underbrace{\boldsymbol{y} - \mu_X}_{=\tilde{y}})^\intercal \boldsymbol{\alpha} \\ &= (k_{XX} + \sigma^2 I_N)^{-1}(\boldsymbol{y} - \mu_X) \\ \Rightarrow \qquad \frac{\mathrm{d}\boldsymbol{\alpha}^\star}{\mathrm{d}\boldsymbol{y}} &= (k_{XX} + \sigma^2 I_N)^{-1}\end{aligned}$$

This means that the posterior covariance is given by

$$\begin{aligned}k_{\boldsymbol{y}}(\bullet, \circ) &= k_{\bullet\circ} - k_{\bullet X}(k_{XX} + \sigma^2 I_N)^{-1}(k_{XX} + \sigma^2 I_N)(k_{XX} + \sigma^2 I_N)^{-1} k_{X\circ} \\ &= k_{\bullet\circ} - k_{\bullet X}\left(\frac{\mathrm{d}\boldsymbol{\alpha}^\star}{\mathrm{d}\boldsymbol{y}}\right)(k_{XX} + \sigma^2 I_N)\left(\frac{\mathrm{d}\boldsymbol{\alpha}^\star}{\mathrm{d}\boldsymbol{y}}\right)^\intercal k_{X\circ}\end{aligned}$$

Still at least $\Omega(N^3)$ time complexity to compute the posterior covariance exactly…

1. Instantiating a Gaussian process model is essentially free
2. "Training" to find the posterior mean is an optimization problem, and can be solved with SGD
3. Making point predictions ("inference") at test time is $\mathcal{O}(N)$ when the model is trained
4. However, so far we have not found a linear-time solution for computing the posterior covariance

Please cite this course, as

```
@techreport{Tuebingen_ProbML23,
    title =
    {Probabilistic Machine Learning},
    author = {Hennig, Philipp},
    series = {Lecture Notes
        in Machine Learning},
    year = {2023},
    institution = {Tübingen AI Center}}
```

Next lecture: A closer look at Cholesky, data loading, estimating uncertainty