Challenges for Gaussian processes

Mark van der Wilk

Department of Computing
Imperial College London

@markvanderwilk
m.vdwilk@imperial.ac.uk

February 4, 2022

Methods for Regression in ML

Most tasks in ML are regression! Deep learning is 99% regression!

Methods for Regression in ML

Most tasks in ML are regression! Deep learning is 99% regression!

Remember where GPs excel:

- are low dimensional (e.g. tens of dimensions rather than 100s),
- have little data (or data is expensive to obtain),
- are noisy (random fluctuations that obscure the signal),
- require uncertainty estimates.

Methods for Regression in ML

Most tasks in ML are regression! Deep learning is 99% regression!

Remember where GPs excel:

- are low dimensional (e.g. tens of dimensions rather than 100s),
- have little data (or data is expensive to obtain),
- are noisy (random fluctuations that obscure the signal),
- require uncertainty estimates.

Today: Where GPs struggle, and why.

Learning objectives

Know how to perform the computations necessary for GPs

- Computing the marginal likelihood
- Computational complexity
- Low-rank approximations

Learning objectives

Know how to perform the computations necessary for GPs

- Computing the marginal likelihood
- Computational complexity
- Low-rank approximations

Understand the computational and modelling limitations of GPs

- Limitations of stationary kernels
- Limitations of local kernels in high-dimensions

Training a GP: Computations

To train, we need the marginal likelihood and its gradient:

$$\log p(\mathbf{y} \mid \theta) = \text{const} - \frac{1}{2}\log\left|\mathbf{K}_\theta + \sigma^2\mathbf{I}\right| - \frac{1}{2}\mathbf{y}^\intercal(\mathbf{K}_\theta + \sigma^2\mathbf{I})^{-1}\mathbf{y} \qquad (1)$$

$$\frac{\partial}{\partial\theta}\log p(\mathbf{y} \mid \theta) = \frac{1}{2}\,\text{Tr}\left[\left(\mathbf{K}_\theta^{-1}\mathbf{y}\mathbf{y}^\intercal\mathbf{K}_\theta - \mathbf{K}_\theta^{-1}\right)\frac{\partial\mathbf{K}_\theta}{\partial\theta}\right] \qquad (2)$$

[1] `https://nhigham.com/2020/08/04/what-is-numerical-stability/`

Training a GP: Computations

To train, we need the marginal likelihood and its gradient:

$$\log p(\mathbf{y} \,|\, \theta) = \text{const} - \frac{1}{2}\log\left|\mathbf{K}_\theta + \sigma^2\mathbf{I}\right| - \frac{1}{2}\mathbf{y}^\top(\mathbf{K}_\theta + \sigma^2\mathbf{I})^{-1}\mathbf{y} \qquad (1)$$

$$\frac{\partial}{\partial\theta}\log p(\mathbf{y} \,|\, \theta) = \frac{1}{2}\operatorname{Tr}\left[\left(\mathbf{K}_\theta^{-1}\mathbf{y}\mathbf{y}^\top\mathbf{K}_\theta - \mathbf{K}_\theta^{-1}\right)\frac{\partial\mathbf{K}_\theta}{\partial\theta}\right] \qquad (2)$$

$(\dots)^{-1}$ and $|\dots|$ are calculated from a **matrix decomposition**.

- Decompositions are expensive
- But make follow-on operations cheap
- The correct decomposition helps numerical stability[1]
- Directly computing the inverse (`np.linalg.inv()`) is a bad thing to do!

[1] `https://nhigham.com/2020/08/04/what-is-numerical-stability/`

Eigenvalue decomposition

We can compute the terms using the **eigenvalue decomposition**:

$$\mathbf{Q}\boldsymbol{\Lambda}\mathbf{Q}^\intercal = \mathbf{K}_\theta + \sigma^2\mathbf{I} \tag{3}$$

$$\log\left|\mathbf{K}_\theta + \sigma^2\mathbf{I}\right| = \log|\mathbf{Q}\boldsymbol{\Lambda}\mathbf{Q}^\intercal| = \cancelto{0}{2\log|\mathbf{Q}|} + \log|\boldsymbol{\Lambda}| = \sum_{n=1}^{N} \log \lambda_i(\mathbf{K}_\theta + \sigma^2\mathbf{I}) \tag{4}$$

$$\mathbf{y}^\intercal(\mathbf{K}_\theta + \sigma^2\mathbf{I})^{-1}\mathbf{y} = \mathbf{y}^\intercal\mathbf{Q}^\intercal\boldsymbol{\Lambda}^{-1}\mathbf{Q}\mathbf{y} \tag{5}$$

Eigenvalue decomposition is mostly applied for **theoretical analysis**.

Cholesky decomposition

Or alternatively the **Cholesky decomposition**:

$$\mathbf{L}\mathbf{L}^\intercal = \mathbf{K}_\theta + \sigma^2\mathbf{I}\,, \qquad \text{where } \mathbf{L} \text{ is lower triangular.} \tag{6}$$

$$\log\left|\mathbf{K}_\theta + \sigma^2\mathbf{I}\right| = \log|\mathbf{L}\mathbf{L}^\intercal| = 2\sum_{n=1}^{N}\log[\mathbf{L}]_{nn} \tag{7}$$

$$\mathbf{y}^\intercal(\mathbf{K}_\theta + \sigma^2\mathbf{I})^{-1}\mathbf{y} = \mathbf{y}^\intercal\mathbf{L}^{\intercal -1}\mathbf{L}^{-1}\mathbf{y} \tag{8}$$

Cholesky decomposition is used for **practical implementation**.

Cholesky decomposition

Or alternatively the **Cholesky decomposition**:

$$\mathbf{L}\mathbf{L}^\intercal = \mathbf{K}_\theta + \sigma^2\mathbf{I}\,, \qquad \text{where } \mathbf{L} \text{ is lower triangular.} \tag{6}$$

$$\log\left|\mathbf{K}_\theta + \sigma^2\mathbf{I}\right| = \log|\mathbf{L}\mathbf{L}^\intercal| = 2\sum_{n=1}^{N}\log[\mathbf{L}]_{nn} \tag{7}$$

$$\mathbf{y}^\intercal(\mathbf{K}_\theta + \sigma^2\mathbf{I})^{-1}\mathbf{y} = \mathbf{y}^\intercal\mathbf{L}^{\intercal-1}\mathbf{L}^{-1}\mathbf{y} \tag{8}$$

Cholesky decomposition is used for **practical implementation**.
(For the coursework, all implementations are fine.)

Computational complexity

1. Computing kernel matrix $O(N^2)$ time and $O(N^2)$ space
2. Eigendecomp and Cholesky are both often quoted to be $O(N^3)$ time.[2]
3. Logdet or inverse given the decomposition are fast
   - Logdet for both are $O(N)$
   - Inverse is $O(N^2)$

[2] Algorithms do exist with better asymptotic complexity, but they are definitely slower than $O(N^2)$.

Computational complexity

1. Computing kernel matrix $O(N^2)$ time and $O(N^2)$ space
2. Eigendecomp and Cholesky are both often quoted to be $O(N^3)$ time.[2]
3. Logdet or inverse given the decomposition are fast
    - Logdet for both are $O(N)$
    - Inverse is $O(N^2)$

Can we take advantage of structure in the kernel matrix to do better?

[2] Algorithms do exist with better asymptotic complexity, but they are definitely slower than $O(N^2)$.

Low-rank kernels

For kernel matrices that are low-rank (have some zero eigenvalues), i.e. with $M \ll N$

$$\mathbf{K} = \mathbf{P}\mathbf{P}^\intercal\,, \qquad \text{where } \mathbf{K} \in \mathbb{R}^{N\times N}, \mathbf{P} \in \mathbb{R}^{N\times M} \tag{9}$$

[3] Can be proved by applying the Woodbury Matrix Identity, and the similar Matrix Determinant Lemma.

Low-rank kernels

For kernel matrices that are low-rank (have some zero eigenvalues), i.e. with $M \ll N$

$$\mathbf{K} = \mathbf{P}\mathbf{P}^\top, \qquad \text{where } \mathbf{K} \in \mathbb{R}^{N\times N}, \mathbf{P} \in \mathbb{R}^{N\times M} \tag{9}$$

we can compute the marginal likelihood more cheaply:[3]

$$\log\left|\mathbf{K} + \sigma^2\mathbf{I}_N\right| = \log\left|\sigma^2\mathbf{I}_N\right| + \log\left|\mathbf{I}_M + \sigma^{-2}\mathbf{P}^\top\mathbf{P}\right| \tag{10}$$

$$\mathbf{y}^\top(\mathbf{K} + \sigma^2\mathbf{I}_N)^{-1}\mathbf{y} = \mathbf{y}^\top\left(\sigma^{-2}\mathbf{I}_N - \sigma^{-2}\mathbf{P}(\sigma^2\mathbf{I}_M + \mathbf{P}^\top\mathbf{P})^{-1}\mathbf{P}^\top\right)\mathbf{y} \tag{11}$$

[3] Can be proved by applying the Woodbury Matrix Identity, and the similar Matrix Determinant Lemma.

Low-rank kernels

For kernel matrices that are low-rank (have some zero eigenvalues), i.e. with $M \ll N$

$$\mathbf{K} = \mathbf{P}\mathbf{P}^\top, \qquad \text{where } \mathbf{K} \in \mathbb{R}^{N\times N}, \mathbf{P} \in \mathbb{R}^{N\times M} \tag{9}$$

we can compute the marginal likelihood more cheaply:[3]

$$\log\left|\mathbf{K} + \sigma^2\mathbf{I}_N\right| = \log\left|\sigma^2\mathbf{I}_N\right| + \log\left|\mathbf{I}_M + \sigma^{-2}\mathbf{P}^\top\mathbf{P}\right| \tag{10}$$

$$\mathbf{y}^\top(\mathbf{K} + \sigma^2\mathbf{I}_N)^{-1}\mathbf{y} = \mathbf{y}^\top\left(\sigma^{-2}\mathbf{I}_N - \sigma^{-2}\mathbf{P}(\sigma^2\mathbf{I}_M + \mathbf{P}^\top\mathbf{P})^{-1}\mathbf{P}^\top\right)\mathbf{y} \tag{11}$$

Can be computed in $O(NM^2 + M^3)$.

[3] Can be proved by applying the Woodbury Matrix Identity, and the similar Matrix Determinant Lemma.

Decaying eigenvalues

For example, the squared exponential kernel

$$k(x, x') = \sigma_f^2 \exp\left(\frac{|x - x'|^2}{2\ell^2}\right), \qquad \theta = \{\sigma_f, \ell\} \tag{12}$$

with 1D inputs i.i.d. from[4] $\mathcal{N}\left(0, \sigma_D^2\right)$, we have that the eigenvalues of **K** decay exponentially

$$\lambda_m = \sqrt{\frac{2a}{A}} B^m \qquad \text{with } B < 1 \tag{13}$$

$$a^{-1} = 4\sigma_D^2\,, \;\; b^{-1} = 2\ell^2\,, \;\; c = \sqrt{a^2 + 2ab}\,, \;\; A = a + b + c\,, \;\; B = b/A$$

as the number of data $N \to \infty$.

[4] Conditions can be much weaker, but this is easier.

Low-rank approximations

So we know that for large N our $\mathbf{K}$ will be **approximately low-rank**. Can we approximate it with a low-rank matrix?

$$\mathbf{K} = \mathbf{Q}\mathbf{\Lambda}\mathbf{Q}^\top = \begin{bmatrix}\mathbf{Q}_1 & \mathbf{Q}_2\end{bmatrix}\begin{bmatrix}\mathbf{\Lambda}_1 & \\ & \mathbf{\Lambda}_2\end{bmatrix}\begin{bmatrix}\mathbf{Q}_1^\top \\ \mathbf{Q}_2^\top\end{bmatrix} = \mathbf{Q}_1\mathbf{\Lambda}_1\mathbf{Q}_1^\top + \mathbf{Q}_2\mathbf{\Lambda}_2\mathbf{Q}_2^\top \quad (14)$$

Low-rank approximations

So we know that for large N our $\mathbf{K}$ will be **approximately low-rank**. Can we approximate it with a low-rank matrix?

$$\mathbf{K} = \mathbf{Q}\boldsymbol{\Lambda}\mathbf{Q}^\intercal = \begin{bmatrix}\mathbf{Q}_1 & \mathbf{Q}_2\end{bmatrix}\begin{bmatrix}\boldsymbol{\Lambda}_1 & \\ & \boldsymbol{\Lambda}_2\end{bmatrix}\begin{bmatrix}\mathbf{Q}_1^\intercal \\ \mathbf{Q}_2^\intercal\end{bmatrix} = \mathbf{Q}_1\boldsymbol{\Lambda}_1\mathbf{Q}_1^\intercal + \mathbf{Q}_2\boldsymbol{\Lambda}_2\mathbf{Q}_2^\intercal \quad (14)$$

$$\hat{\mathbf{K}} = \mathbf{Q}_1\boldsymbol{\Lambda}_1\mathbf{Q}_1^\intercal + \cancelto{\approx 0}{\mathbf{Q}_2\boldsymbol{\Lambda}_2\mathbf{Q}_2^\intercal} = \mathbf{P}\mathbf{P}^\intercal, \qquad \text{with } \mathbf{P} = \mathbf{Q}_1\boldsymbol{\Lambda}^{\frac{1}{2}}. \quad (15)$$

Low-rank approximations

So we know that for large N our $\mathbf{K}$ will be **approximately low-rank**. Can we approximate it with a low-rank matrix?

$$\mathbf{K} = \mathbf{Q}\mathbf{\Lambda}\mathbf{Q}^\top = \begin{bmatrix}\mathbf{Q}_1 & \mathbf{Q}_2\end{bmatrix}\begin{bmatrix}\mathbf{\Lambda}_1 & \\ & \mathbf{\Lambda}_2\end{bmatrix}\begin{bmatrix}\mathbf{Q}_1^\top \\ \mathbf{Q}_2^\top\end{bmatrix} = \mathbf{Q}_1\mathbf{\Lambda}_1\mathbf{Q}_1^\top + \mathbf{Q}_2\mathbf{\Lambda}_2\mathbf{Q}_2^\top \tag{14}$$

$$\hat{\mathbf{K}} = \mathbf{Q}_1\mathbf{\Lambda}_1\mathbf{Q}_1^\top + \cancelto{\approx 0}{\mathbf{Q}_2\mathbf{\Lambda}_2\mathbf{Q}_2^\top} = \mathbf{P}\mathbf{P}^\top\,, \qquad \text{with } \mathbf{P} = \mathbf{Q}_1\mathbf{\Lambda}^{\frac{1}{2}}\,. \tag{15}$$

This gives the approximation with the minimal Frobenius norm:

$$\left\|\mathbf{K} - \hat{\mathbf{K}}\right\|_{\mathrm{F}} = \sqrt{\sum_{i=M+1}^{N} \lambda_i} \tag{16}$$

Low-rank approximation: Data fit

Starting with the data-fit term:[5]

$$\begin{aligned}\mathbf{y}^\intercal\left(\mathbf{K}+\sigma^2\mathbf{I}\right)^{-1}\mathbf{y} &= \mathbf{y}^\intercal(\hat{\mathbf{K}}+\sigma^2\mathbf{I}+\mathbf{K}-\hat{\mathbf{K}})^{-1}\mathbf{y} \\ &= \mathbf{y}^\intercal(\hat{\mathbf{K}}+\sigma^2\mathbf{I})^{-1}\mathbf{y} - \underbrace{\mathbf{y}^\intercal\left(\hat{\mathbf{K}}+\sigma^2\mathbf{I}\right)^{-1}(\mathbf{K}-\hat{\mathbf{K}})(\mathbf{K}+\sigma^2\mathbf{I})^{-1}\mathbf{y}}_{\to 0 \text{ as } \left\|\mathbf{K}-\hat{\mathbf{K}}\right\|_F \to 0} \qquad (17)\end{aligned}$$

[5] Applying the identity $(A+B)^{-1} = A^{-1} - A^{-1}B(A+B)^{-1}$.

Low-rank approximation: Data fit

Starting with the data-fit term:[5]

$$\begin{aligned}\mathbf{y}^\intercal\left(\mathbf{K}+\sigma^2\mathbf{I}\right)^{-1}\mathbf{y} &= \mathbf{y}^\intercal(\hat{\mathbf{K}}+\sigma^2\mathbf{I}+\mathbf{K}-\hat{\mathbf{K}})^{-1}\mathbf{y} \\ &= \mathbf{y}^\intercal(\hat{\mathbf{K}}+\sigma^2\mathbf{I})^{-1}\mathbf{y} - \underbrace{\mathbf{y}^\intercal\left(\hat{\mathbf{K}}+\sigma^2\mathbf{I}\right)^{-1}(\mathbf{K}-\hat{\mathbf{K}})(\mathbf{K}+\sigma^2\mathbf{I})^{-1}\mathbf{y}}_{\to 0 \text{ as } \left\|\mathbf{K}-\hat{\mathbf{K}}\right\|_F \to 0} \qquad (17)\end{aligned}$$

You can place bounds on the error using knowledge of λ_M (the final included eigenvalue).

[5] Applying the identity $(A+B)^{-1} = A^{-1} - A^{-1}B(A+B)^{-1}$.

Low-rank approximation: Complexity penalty

And finishing with the complexity penalty:

$$
\begin{aligned}
\log\left|\mathbf{K} + \sigma^2\mathbf{I}\right| &= \sum_{n=1}^{N} \log(\lambda_n(\mathbf{K}) + \sigma^2) \\
&= \sum_{m=1}^{M} \log(\lambda_m(\mathbf{K}) + \sigma^2) + \sum_{r=M+1}^{N} \log(\underbrace{\lambda_r(\mathbf{K}) + \sigma^2}_{\approx\sigma^2}) && (18) \\
&\approx \sum_{m=1}^{M} \log(\lambda_m(\mathbf{K}) + \sigma^2) + \sum_{r=M+1}^{N} \log(\sigma^2) && (19) \\
&= \log\left|\hat{\mathbf{K}} + \sigma^2\mathbf{I}\right| && (20)
\end{aligned}
$$

Low-rank approximations overview

- Learning with exact matrix decompositions is expensive — $O(N^3)$ time, $O(N^2)$ memory
- Low-rank kernels improve things when $N \gg M$ — $O(NM^2)$ time, $O(NM)$ memory

Low-rank approximations overview

- Learning with exact matrix decompositions is expensive — $O(N^3)$ time, $O(N^2)$ memory
- Low-rank kernels improve things when $N \gg M$ — $O(NM^2)$ time, $O(NM)$ memory
- Low-rank approximations do exist!

Low-rank approximations overview

- Learning with exact matrix decompositions is expensive — $O(N^3)$ time, $O(N^2)$ memory
- Low-rank kernels improve things when $N \gg M$ — $O(NM^2)$ time, $O(NM)$ memory
- Low-rank approximations do exist!
- We used an impractical method (eigendecomposition) to find **which** low-rank columns to use, with a cost of $O(N^2M)$ time.

Low-rank approximations overview

- Learning with exact matrix decompositions is expensive — $O(N^3)$ time, $O(N^2)$ memory
- Low-rank kernels improve things when $N \gg M$ — $O(NM^2)$ time, $O(NM)$ memory
- Low-rank approximations do exist!
- We used an impractical method (eigendecomposition) to find **which** low-rank columns to use, with a cost of $O(N^2M)$ time.

For certain approximations, we have 1) fast methods for finding $\mathbf{P}$, and 2) proofs for how large M needs to be.
E.g. Burt et al [2019] show that for regression and the Squared Exponential kernel, we can get arbitrarily good approximations in

$$O(N(\log N)^{2D}(\log\log N)^2) = O(N^{1+\epsilon})\,, \quad \forall \epsilon > 0 \tag{21}$$

Practical constraints and implementation

GPs are slow, but often not even because of asymptotic complexity!

Practical constraints and implementation

GPs are slow, but often not even because of asymptotic complexity!

- Memory is often the main bottleneck (storing large matrices).

Practical constraints and implementation

GPs are slow, but often not even because of asymptotic complexity!

- Memory is often the main bottleneck (storing large matrices). This is poorly suited to modern-day hardware

Practical constraints and implementation

GPs are slow, but often not even because of asymptotic complexity!

- Memory is often the main bottleneck (storing large matrices). This is poorly suited to modern-day hardware (i.e. GPUs)

Practical constraints and implementation

GPs are slow, but often not even because of asymptotic complexity!

- Memory is often the main bottleneck (storing large matrices). This is poorly suited to modern-day hardware (i.e. GPUs)
- Decompositions require serial computations.

Practical constraints and implementation

GPs are slow, but often not even because of asymptotic complexity!

- Memory is often the main bottleneck (storing large matrices).
  This is poorly suited to modern-day hardware (i.e. GPUs)
- Decompositions require serial computations.
  This is poorly suited to modern-day hardware

Practical constraints and implementation

GPs are slow, but often not even because of asymptotic complexity!

- Memory is often the main bottleneck (storing large matrices).
  This is poorly suited to modern-day hardware (i.e. GPUs)
- Decompositions require serial computations.
  This is poorly suited to modern-day hardware (i.e. GPUs)

Practical constraints and implementation

GPs are slow, but often not even because of asymptotic complexity!

- Memory is often the main bottleneck (storing large matrices).
  This is poorly suited to modern-day hardware (i.e. GPUs)
- Decompositions require serial computations.
  This is poorly suited to modern-day hardware (i.e. GPUs)
- Decompositions need high floating point precision.

Practical constraints and implementation

GPs are slow, but often not even because of asymptotic complexity!

- Memory is often the main bottleneck (storing large matrices).
  This is poorly suited to modern-day hardware (i.e. GPUs)
- Decompositions require serial computations.
  This is poorly suited to modern-day hardware (i.e. GPUs)
- Decompositions need high floating point precision.
  This is poorly suited to modern-day hardware

Practical constraints and implementation

GPs are slow, but often not even because of asymptotic complexity!

- Memory is often the main bottleneck (storing large matrices).
  This is poorly suited to modern-day hardware (i.e. GPUs)
- Decompositions require serial computations.
  This is poorly suited to modern-day hardware (i.e. GPUs)
- Decompositions need high floating point precision.
  This is poorly suited to modern-day hardware (i.e. GPUs)

There is hope

Approximations can make the computations look more similar to a neural network by

- Reduce the size of the matrix decomposition [Titsias, 2009]
- Improve data parallelism [Hensman et al, 2013]
- Increase reliance on fast matrix-matrix multiplications [Gardner et al, 2018; Wang et al, 2019]
- Remove matrix inverses altogether [van der Wilk et al, 2019]

There is hope

Approximations can make the computations look more similar to a neural network by

- Reduce the size of the matrix decomposition [Titsias, 2009]
- Improve data parallelism [Hensman et al, 2013]
- Increase reliance on fast matrix-matrix multiplications [Gardner et al, 2018; Wang et al, 2019]
- Remove matrix inverses altogether [van der Wilk et al, 2019]

This slide for reference only, not examinable.

Implementation matters!

We have good asymptotic bounds on the computational complexity!

Constants cause slowness $\implies$ implementation matters

Implementation matters!

We have good asymptotic bounds on the computational complexity!

Constants cause slowness $\implies$ implementation matters

E.g. *Exact Gaussian Processes on a Million Data Points*, Wang et al [2019] tailors a very good approximation to GPU hardware, with impressive results.

Conclusions on Computational Issues

- Gaussian processes are computationally expensive to train
- Low-rank approximations to the kernel matrix $\mathbf{K}$ do exist
- Low-rank approximations improve the asymptotic complexity
- Still need work to optimise algorithms and tailor to modern hardware

Modelling limitations

We know that the generalisation capability of a model depends strongly on the properties of its prior.

Do we really have good priors?

Modelling limitations

We know that the generalisation capability of a model depends strongly on the properties of its prior.

Do we really have good priors?

Some criticisms...

Stationary kernels

We have seen many examples of **stationary** kernels:

$$k(\mathbf{x}, \mathbf{x}') = k(\mathbf{x} - \mathbf{x}') \tag{22}$$

e.g. the squared exponential.

- Same generalisation characteristic is applied through the entire input space.
- See `stationary-kernel-failure.ipynb` in the `inference-plots` GitHub repo.

Local kernels in high dimensions

Many kernels depend on some form of the Euclidean distance:

$$k(\mathbf{x}, \mathbf{x}') = \sigma_f^2 \exp\left[-\frac{||\mathbf{x} - \mathbf{x}'||^2}{2\ell^2}\right] \tag{23}$$

This behaves badly in high dimensions (even MNIST, $D = 28^2 = 784$):

- Small differences in each dimension add up to large distances $\rightarrow$ low correlation
- Function is allowed to vary independently along each input dimension
- Prior with low lengthscale is too flexible, large lengthscale is too inflexible
- No middle ground

Conclusions

Gaussian processes have limitations for both

- low-dimensional inputs (e.g. stationarity)
- high-dimensional inputs (e.g. locality)

We need to either make much better priors, or incorporate GPs into more complex models.

References I

[1] J. Gardner, G. Pleiss, K. Q. Weinberger, D. Bindel, and A. G. Wilson. Gpytorch: Blackbox matrix-matrix gaussian process inference with gpu acceleration. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems 31*, pages 7576–7586. Curran Associates, Inc., 2018.