Probabilistic Programming with Gaussian Process Memoization

Anonymous Author(s)
Affiliation
Address
email

Abstract

This paper describes the *Gaussian process memoizer*, a probabilistic programming technique that uses Gaussian processes to provide a statistical alternative to memorization. Memoizing a target procedure results in a self-caching wrapper that remembers previously computed values. Gaussian process memoization additionally produces a statistical emulator based on a Gaussian process whose predictions automatically improve whenever a new value of the target procedure becomes available. This paper also introduces an efficient implementation, named `gpmem`, that can use kernels given by a broad class of probabilistic programs. The flexibility of `gpmem` is illustrated via three applications: (i) GP regression with hierarchical hyper-parameter learning, (ii) Bayesian structure learning via compositional kernels generated by a probabilistic grammar, and (iii) a bandit formulation of Bayesian optimization with automatic inference and action selection. All applications share a single 50-line Python library and require fewer than 20 lines of probabilistic code each.

1 Introduction

Probabilistic programming could be revolutionary for machine intelligence due to universal inference engines and the rapid prototyping for novel models (Ghahramani, 2015). Probabilistic programming languages aim to provide a formal language to specify probabilistic models in the style of computer programming and can represent any computable probability distribution as a program. We can write such programs in Venture (Mansinghka et al., 2014), the first probabilistic programming language suitable for general purpose use. Venture comes with scalable performance on hard problems and with a general purpose inference engine.

A family of statistical distributions rarely treated in probabilistic programming languages are Gaussian Process (GP). GPs are gaining increasing attention for representing unknown functions by posterior probability distributions in various fields such as machine learning (Rasmussen and Williams, 2006), signal processing (Clifton et al., 2013), computer vision (Kemmler et al., 2013) and biomedical data analysis (Shepherd and Owenius, 2012). When incorporated in probabilistic programming, GPs allow us to treat a wide range of problems naturally within the same unified semantic framework. Examples for hard problems are hierarchical prior construction (Neal, 1997), systems for inductive learning of symbolic expressions such as the one introduced in the Automated Statistician project (Duvenaud et al., 2013; Lloyd et al., 2014) and Bayesian Optimization (Snoek et al., 2012).

In the following, we will present GP memoization (`gpmem`) as a novel probabilistic programming technique that solves such hard problems and provides the necessary semantic framework. `gpmem` introduces a statistical alternative to standard memoization. Our contribution is threefold. Firstly, we introduce an efficient implementation of `gpmem` in form of a self-caching wrapper that remembers

previously computed values. Secondly,we illustrate the statistical emulator that gpmem produces and how it improves with every data point that becomes available. Finally, we show how one can solve hard problems of state-of-the-art machine learning related to GP using gpmem. In the following, we will introduce GP memoization. We will start by introducing gpmem. We then elaborate on its use at hand of the problems mentioned above. To evaluate the performance we use synthetic and real-world data.

2 Gaussian Processes Memoization

Let the data be pairs of real-valued scalars $\{(x_i, y_i)\}_{i=1}^n$ (complete data will be denoted by column vectors $\mathbf{x}$, $\mathbf{y}$). GPs present a non-parametric way to express prior knowledge on the space of possible functions f modeling a regression relationship. Formally, a GP is an infinite-dimensional extension of the multivariate Gaussian distribution. Often one assumes the values $\mathbf{y}$ are noisily measured, that is, one only sees the values of $\mathbf{y}_{\text{noisy}} = \mathbf{y} + \mathbf{w}$ where $\mathbf{w}$ is Gaussian white noise with variance σ^2_{noise}. In that case, the log-likelihood of a GP is

$$\log p\left(\mathbf{y}_{\text{noisy}} \mid \mathbf{x}\right) = -\frac{1}{2}\mathbf{y}^\top(\mathbf{\Sigma} + \sigma^2_{\text{noise}}\mathbf{I})^{-1}\mathbf{y} - \frac{1}{2}\log\left|\mathbf{\Sigma} + \sigma^2_{\text{noise}}\mathbf{I}\right| - \frac{n}{2}\log 2\pi \tag{1}$$

where n is the number of data points. Both log-likelihood and predictive posterior can be computed efficiently in a Venture SP with an algorithm that resorts to Cholesky factorization(Rasmussen and Williams, 2006, chap. 2) resulting in a computational complexity of $\mathcal{O}(n^3)$ in the number of data points.

The covariance function (or kernel) of a GP governs high-level properties of the observed data such as linearity, periodicity and smoothness. It comes with few free parameters that we call hyper-parameters. Adjusting these results in minor changes, for example with regards to when two data points are treated similar. More drastically different covariance functions are achieved by changing the structure of the covariance function itself. Note that covariance function structures are compositional: adding or multiplying two valid covariance functions results in another valid covariance function.

Venture includes the primitive make_gp, which takes as arguments a unary function mean and a binary (symmetric, positive-semidefinite) function cov and produces a function g distributed as a Gaussian process with the supplied mean and covariance. For example, a function $\mathrm{g} \sim \mathcal{GP}(0, \mathrm{SE})$, where SE is a squared-exponential covariance

$$\mathrm{SE}(x, x') = \sigma^2 \exp\left(\frac{(x - x')^2}{2\ell}\right)$$

with $\sigma = 1$ and $\ell = 1$, can be instantiated as follows:

```
assume zero = make_const_func( 0.0)
assume se = make_squaredexp( 1.0, 1.0)
assume g  = make_gp( zero, se)
```

There are two ways two view g as a "random function." In the first view, the assume directive that instantiates g does not use any randomness—only the subsequent calls to g do—and coherence constraints are upheld by the interpreter by keeping track of which evaluations of g exist in the current execution trace. Namely, if the current trace contains evaluations of g at the points $x_1, \ldots, x_N$ with return values $y_1, \ldots, y_N$, then the next evaluation of g (say, jointly at the points $x_{N+1}, \ldots, x_{N+n}$) will be distributed according to the joint conditional distribution

$$P\left((\mathrm{g}\ x_{N+1}), \ldots, (\mathrm{g}\ x_{N+n}) \;\middle|\; (\mathrm{g}\ x_i) = y_i \text{ for } i = 1, \ldots, N\right).$$

In the second view, g is a randomly chosen deterministic function, chosen from the space of all deterministic real-valued functions; in this view, the assume directive contains *all* the randomness, and subsequent invocations of g are deterministic. The first view is procedural and is faithful to the computation that occurs behind the scenes in Venture. The second view is declarative and is faithful to notations like "$g \sim P(g)$" which are often used in mathematical treatments. Because a model program could make arbitrarily many calls to g, and the joint distribution on the return values of the

calls could have arbitrarily high entropy, it is not computationally possible in finite time to choose the entire function g all at once as in the second view. Thus, it stands to reason that any computationally implementable notion of "nonparametric random functions" must involve incremental random choices in one way or another, and Gaussian processes in Venture are no exception.

2.1 Memoization

Memoization is the practice of storing previously computed values of a function so that future calls with the same inputs can be evaluated by lookup rather than recomputation. Although memoization does not change the semantics of a deterministic program, it does change that of a stochastic program (Goodman et al., 2008). The authors provide an intuitive example: let f be a function that flips a coin and return "head" or "tails". The probability that two calls of f are equivalent is 0.5. However, if the function call is memoized, it is 1. In fact, there is an infinite range of possible caching policies (specifications of when to use a stored value and when to recompute), each potentially having a different semantics. Any particular caching policy can be understood by random world semantics (Poole, 1993; Sato, 1995) over the stochastic program: each possible world corresponds to a mapping from function input sequence to function output sequence (McAllester et al., 2008). In Venture, these possible worlds are first-class objects, and correspond to the *probabilistic execution traces traces* (Mansinghka et al., 2014).

To transfer this idea to probabilistic programming, we now introduce a language construct called a *statistical memoizer*. Suppose we have a function `f` which can be evaluated but we wish to learn about the behavior of `f` using as few evaluations as possible. The statistical memoizer, which here we give the name `gpmem`, was motivated by this purpose. It produces two outputs:

$$\mathtt{f} \xrightarrow{\mathtt{gpmem}} (\mathtt{f}_{\text{probe}}, \mathtt{f}_{\text{emu}}).$$

The function $\mathtt{f}_{\text{probe}}$ calls `f` and stores the output in a memo table, just as traditional memoization does. The function $\mathtt{f}_{\text{emu}}$ is an online statistical emulator which uses the memo table as its training data. A fully Bayesian emulator, modelling the true function `f` as a random function $f \sim P(f)$, would satisfy

$$(\mathtt{f}_{\text{emu}}\ \mathtt{x}_1\ \ldots\ \mathtt{x}_k) \sim P\left(f(\mathtt{x}_1), \ldots, f(\mathtt{x}_k) \mid f(\mathtt{x}) = (\mathtt{f}\ \mathtt{x}) \text{ for each } \mathtt{x} \text{ in memo table}\right).$$

Different implementations of the statistical memoizer can have different prior distributions $P(f)$; in this paper, we deploy a Gaussian process prior (implemented as `gpmem` below). Note that we require the ability to sample $\mathtt{f}_{\text{emu}}$ jointly at multiple inputs because the values of $f(\mathtt{x}_1), \ldots, f(\mathtt{x}_k)$ will in general be dependent. To illustrate the linguistic power of `gpmem` consider a simple model program that uses `gpmem` procedure abstraction, and a loop:

```
define choose_next_point = proc(em) { gridsearch_argmax( em) }

 // Here stats( em) is the memo table {(x_i, y_i)}.
 // Take the (x,y) pair with the largest y.
define extract_answer = proc(em) { first(max( stats( em, second)))}

 // Hyper-parameter
assume theta =  tag( quote( params), 0, gamma(1,1))
assume (f_probe f_emu) =  gpmem(f, make_squaredexp(1.0, theta))

for t...T:
  f_probe( choose_next_point( f_emu))
extract_answer( f_emu)
```

An equivalent model in typical statistics notation, written in a way that attempts to be as linguistically faithful as possible, is

$$\begin{aligned} f^{(0)} &\sim P(f) \\ \mathtt{x}^{(t)} &= \arg\max_x f^{(t)}(x) \\ f^{(t+1)} &\sim P\left(f^{(t)} \;\middle|\; f^{(t)}(\mathtt{x}^{(t)}) = (\mathtt{f}\ \mathtt{x}^{(t)})\right). \end{aligned}$$

The linguistic constructs for abstraction are not present in statistics notation. The equation $\mathsf{x}^{(t)} = \arg\max_x f^{(t)}(x)$ has to be inlined in statistics notation: while something like

$$\mathsf{x}^{(t)} \sim F(P^{(t)})$$

(where $P^{(t)}$ is a probability measure, and F here plays the role of `choose_next_point`) is mathematically coherent, it is not what statisticians would ordinarily write. This lack of abstraction then makes modularity, or the easy digestion of large but finely decomposable models, more difficult in statistics notation.

Adding a single line to the program, such as

```
infer mh( quote( params), one, 50)
```

after line 12 to infer the parameters of the emulator, would require a significant refactoring and elaboration of the statistics notation. One basic reason is that probabilistic programs are written procedurally, whereas statistical notation is declarative; reasoning declaratively about the dynamics of a fundamentally procedural inference algorithm is often unwieldy due to the absence of programming constructs such as loops and mutable state.

We implement `gpmem` by memoizing a target procedure in a wrapper that remembers previously computed values. This comes with interesting implications: from the standpoint of computation, a data set of the form $\{(x_i, y_i)\}$ can be thought of as a function $y = f_{\text{restr}}(x)$, where f_{restr} is restricted to only allow evaluation at a specific set of inputs x (Alg. 1). Modelling the data set with a GP then

Algorithm 1 Restricted Function

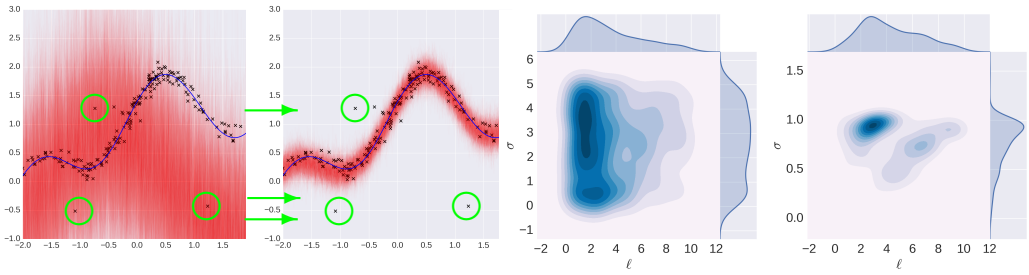
1: **function** $f_{\text{RESTR}}(x)$
2: **if** $x \in \mathcal{D}$ **then**
3: **return** $\mathcal{D}[x]$
4: **else**
5: raise Exception('Illegal input')
6: **end if**
7: **end function**

amounts to trying to learn a smooth function f_{emu} ("emu" stands for "emulator") which extends f to its full domain. Indeed, if f_{restr} is a foreign procedure made available as a black-box to Venture, whose secret underlying pseudo code is in Alg. 1, then the `observe` code can be rewritten using `gpmem` as in Listing 1 (where here the data set `D` has keys `x[1]`,...,`x[n]`):

Listing 1: Observation with `gpmem`

```
1 assume (f_probe f_emu) =  gpmem( f_restr)
2 for i=1 to n:
3   predict f_compute( x[i])
4   infer mh(quote(hyper-parameters), one, 100)
5 sample (f_emu( array( 1, 2, 3))
```

This rewriting has at least two benefits: (i) readability (in some cases), and (ii) amenability to active learning. As to (i), the statistical code of creating a Gaussian process is replaced with a memoization-like idiom, which will be more familiar to programmers. As to (ii), when using `gpmem`, it is quite easy to decide incrementally which data point to sample next: for example, the loop from `x[1]` to `x[n]` could be replaced by a loop in which the next index `i` is chosen by a supplied decision rule. In this way, we could use `gpmem` to perform online learning using only a subset of the available data.



(a) Observed (b) Inferred (c) Before (d) After

Figure 1: (a) and (b) show gpmem on Neal's example. Outliers are marked with green circles. After gpmem observes some data-points, an arbitrary smooth trend with a high level of noise is sampled (a). After running inference on the hierarchical system of hyper-parameters we see that the posterior reflects the actual curve well (b). We see that the outliers are rejected. This is partly due to the shift in the hyper-parameters for lengthscale ℓ and noise σ. From a uniform/gamma prior (c) we compute a bimodal posterior (d)

3 Example Applications

gpmem is an elegant linguistic framework for function learning-related tasks. The technique allows language constructs from programming to help express models which would be cumbersome to express in statistics notation. We will now illustrate this with three example applications.

3.1 Hyperparameter estimation with structured hyper-prior

The probability of the hyper-parameters of a GP as defined above and given covariance function structure $\mathbf{K}$ is:

$$P(\boldsymbol{\theta} \mid \mathbf{D}, \mathbf{K}) = \frac{P(\mathbf{D} \mid \boldsymbol{\theta}, \mathbf{K}) P(\boldsymbol{\theta} \mid \mathbf{K})}{P(\mathbf{D} \mid \mathbf{K})}. \tag{2}$$

Let the $\mathbf{K}$ be the sum of a smoothing and a white noise (WN) kernel. For this case, Neal (1997) suggested the problem of outliers in data as a use-case for a hierarchical Bayesian treatment of Gaussian processes[1]. The work suggests a hierarchical system of hyper-parameterization. Here, we draw hyper-parameters from a Γ distributions:

$$\ell^{(t)} \sim \Gamma(\alpha_1, \beta_1),\ \sigma^{(t)} \sim \Gamma(\alpha_2, \beta_2) \tag{3}$$

and in turn sample the α and β from Γ distributions as well:

$$\alpha_1^{(t)} \sim \Gamma(\alpha_\alpha^1, \beta_\alpha^1),\ \alpha_2^{(t)} \sim \Gamma(\alpha_\alpha^2, \beta_\alpha^2), \cdots \tag{4}$$

One can represent this kind of model using gpmem (Listing 2). Neal provides a custom inference algorithm setting and evaluates it using the following synthetic data problem. Let f be the underlying function that generates the data:

$$f(x) = 0.3 + 0.4x + 0.5\sin(2.7x) + \frac{1.1}{(1+x^2)} + \eta \quad with\ \ \eta \sim \mathcal{N}(0, \sigma) \tag{5}$$

We synthetically generate outliers by setting $\sigma = 0.1$ in 95% of the cases and to $\sigma = 1$ in the remaining cases. gpmem can capture the true underlying function within only 100 MH steps on the hyper-parameters to get a good approximation for their posterior (see Fig. 1). Note that Neal devises an additional noise model and performs a large number of Hybrid-Monte Carlo and Gibbs steps. We illustrate the hyper-parameters by showing the shift of the distribution on the noise parameter σ (Fig. 1). We see that gpmem learns the posterior distribution well, the posterior even exhibits a bimodal histogram when sampling σ 100 times reflecting the two modes of data generation, that is normal noise and outliers.

[1] In (Neal, 1997) the sum of an SE plus a constant kernel is used. We keep the WN kernel for illustrative purposes.

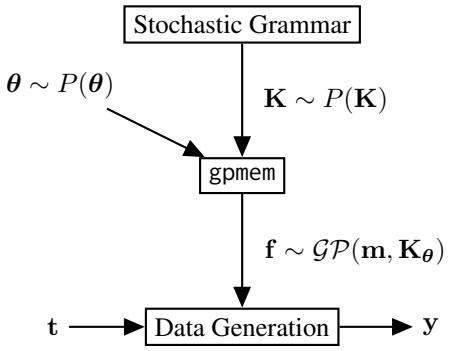


Figure 2: Graphical description of Bayesian GP structure learning.

Listing 2: Hyper-Prior Learning

```
/// SETTING UP THE MODEL
assume alpha_sf = tag(quote(hyperhyper), 0, gamma(7, 1))
assume beta_sf = tag(quote(hyperhyper), 1, gamma(7, 1))
assume alpha_l = tag(quote(hyperhyper), 2, gamma(7, 1))
assume beta_l = tag(quote(hyperhyper), 3, gamma(7, 1))

// Parameters of the covariance function
assume sf = tag(quote(hyper), 0, gamma(alpha_sf, beta_sf)))
assume l = tag(quote(hyper), 1, gamma(alpha_l, beta_l)))
assume sigma = tag(quote(hyper), 2, uniform_continuous(0, 2))

// The covariance function
assume se = make_squaredexp(sf, l)
assume wn = make_whitenoise(sigma)
assume composite_covariance = add_funcs(se, wn)

/// PERFORMING INFERENCE
// Create a prober and emulator using gpmem
assume f_restr = get_neal_blackbox()
assume (f_compute, f_emu) = gpmem(f_restr, composite_covariance)

// Probe all data points
for n ... N
    predict f_compute(get_neal_data_xs(n))

// Infer hypers and hyperhypers
infer repeat(100, do(
    mh(quote(hyperhyper), one, 2),
    mh(quote(hyper), one, 1)))
```

3.2 Discovering symbolic models for time series

Inductive learning of symbolic expression for continuous-valued time series data is a hard task which has recently been tackled using a greedy search over the approximate posterior of the possible kernel compostions for GPs (Duvenaud et al., 2013; Lloyd et al., 2014)[2].

With `gpmem` we can provide a fully Bayesian treatment of this, previously unavaible, using a stochastic grammar (see Fig. 2).

[2] `http://www.automaticstatistician.com/`

We deploy a probabilistic context free grammar for our prior on structures. An input of non-composite kernels (base kernels) is supplied to generate a posterior distributions of composite structure to express local and global aspects of the data.

We approximate the following intractable integrals of the expectation for the prediction:

$$\mathbb{E}[y^* \mid x^*, \mathbf{D}, \mathbf{K}] = \iint f(x^*, \boldsymbol{\theta}, \mathbf{K})\, P(\boldsymbol{\theta} \mid \mathbf{D}, \mathbf{K})\, P(\mathbf{K}|\boldsymbol{\Omega}, s, n)\, \mathbf{d}\boldsymbol{\theta}\mathbf{d}\mathbf{K}. \tag{6}$$

This is done by sampling from the posterior probability distribution of the hyper-parameters and the possible kernel:

$$y^* \approx \frac{1}{T} \sum_{t=1}^{T} f(x^*|\boldsymbol{\theta}^{(t)}, \mathbf{K}^{(t)}). \tag{7}$$

In order to provide the sampling of the kernel, we introduce a stochastic process that simulates the grammar for algebraic expressions of covariance function algebra:

$$\mathbf{K}^{(t)} \sim P(\mathbf{K} \mid \boldsymbol{\Omega}, s, n) \tag{8}$$

Here, we start with the set of given base kernels and draw a random subset. For this subset of size n, we sample a set of possible operators $\boldsymbol{\Omega}$ combining base kernels. The marginal probability of a composite structure

$$P(\mathbf{K} \mid \boldsymbol{\Omega}, s, n) = P(\boldsymbol{\Omega} \mid s, n) \times P(s \mid n) \times P(n), \tag{9}$$

is characterized by the prior $P(n)$ on the number of base kernels used, the probability of a uniformly chosen subset of the set of n possible covariance functions

$$P(s \mid n) = \frac{n!}{\mid s \mid!}, \tag{10}$$

and the probability of sampling a global or a local structure, which is given by a binomial distribution:

$$P(\boldsymbol{\Omega} \mid s, n) = \binom{n}{r} p_{+\times}^{k} (1 - p_{+\times})^{n-k}. \tag{11}$$

Many equivalent covariance structures can be sampled due to covariance function algebra and equivalent representations with different parameterization (Lloyd et al., 2014). To inspect the posterior of these equivalent structures we convert each kernel expression into a sum of products and subsequently simplify. All base kernels can be found in Appendix A, rules for this simplification can be found in appendix B. The code for learning of kernel structure is as follows:

```
// GRAMMAR FOR KERNEL STRUCTURE
assume kernels = list(se, wn, lin, per, rq) // defined as above

// prior on the number of kernels
assume p_number_k = uniform_structure(n)
assume sub_s = tag(quote(grammar), 0,
                           subset(kernels, p_number_k))

assume grammar = proc(l) {
  // kernel composition
  if (size(l) <= 1)
       { first(l) }
  else { if (bernoulli())
                  { add_funcs(first(l), grammar(rest(l))) }
              else { mult_funcs(first(l), grammar(rest(l))) }
       }
  }

assume K = tag(quote(grammar), 1, grammar(sub_s))

assume (f_compute f_emu) = gpmem(f_restr, K)

// Probe all data points
for n ... N
    predict f_compute(get_data_xs(n))

// PERFORMING INFERENCE
infer  repeat(2000, do(
                        mh(quote(grammar), one, 1),
                        for kernel ∈ K
                            mh(quote(hyper_kernel), one, 1)))
```

We defined the space of covariance structures in a way that allows us to produce results coherent with work presented in Automatic Statistician. For example, for the airline data set describing monthly totals of international airline passengers (Box et al., 1997, according to Duvenaud et al., 2013). Our most frequent sample is identical with the highest scoring result reported in previous work using a search-and-score method (Duvenaud et al., 2013) for the CO_2 data set (see Rasmussen and Williams, 2006 for a description) and the predictive capability is comparable. However, the components factor in a different way due to different parameterization of the individual base kernels. We see that the most probable alternatives for a structural description both recover the data dynamics (Fig. 7 for the airline data set).

Confident about our results, we can now query the data for certain structures being present. We illustrate this using the Mauna Loa data used in previous work on automated kernel discovery (Duvenaud et al., 2013). We assume a relatively simple hypothesis space consisting of only four kernels, a linear, a smoothing, a periodic and a white noise kernel. In this experiment, we resort to the white noise kernel instead RQ (similar to (Lloyd et al., 2014)). We can now run the algorithm, compute a posterior of structures. We can also query this posterior distribution for the marginal of certain simple structures to occur. We demonstrate this in Fig. 3

3.3 Bayesian optimization via Thompson sampling

We introduce Thompson sampling, the basic solution strategy underlying the Bayesian optimization with `gpmem`. Thompson sampling Thompson (1933) is a widely used Bayesian framework for addressing the trade-off between exploration and exploitation in multi-armed (or continuum-armed) bandit problems. We cast the multi-armed bandit problem as a one-state Markov decision process, and describe how Thompson sampling can be used to choose actions for that Markov Decision Processes (MDP).

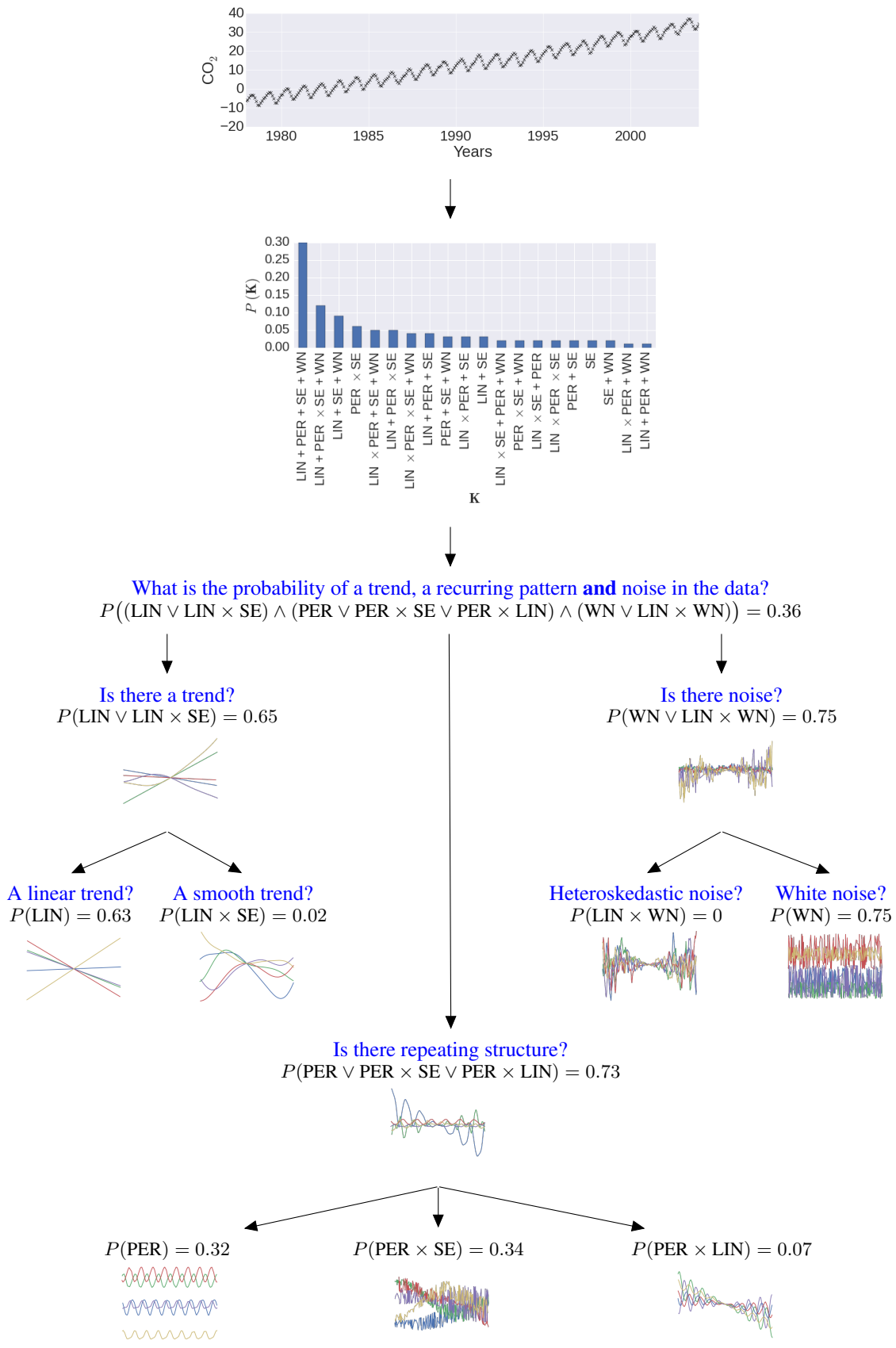


Figure 3: We can query the data if some logical statements are probable to be true, for example, is it true that there is a trend?

Here, an agent is to take a sequence of actions $a_1, a_2, \ldots$ from a (possibly infinite) set of possible actions $\mathcal{A}$. After each action, a reward $r \in \mathbb{R}$ is received, according to an unknown conditional distribution $P_{\text{true}}(r \mid a)$. The agent's goal is to maximize the total reward received for all actions in an online manner. In Thompson sampling, the agent accomplishes this by placing a prior distribution $P(\vartheta)$ on the possible "contexts" $\vartheta \in \Theta$. Here a context is a believed model of the conditional distributions $\{P(r \mid a)\}_{a \in \mathcal{A}}$, or at least, a believed statistic of these conditional distributions which is sufficient for deciding an action a. If actions are chosen so as to maximize expected reward, then one such sufficient statistic is the believed conditional mean $V(a \mid \vartheta) = \mathbb{E}[r \mid a; \vartheta]$, which can be viewed as a believed value function. For consistency with what follows, we will assume our context ϑ takes the form $(\theta, \mathbf{a}_{\text{past}}, \mathbf{r}_{\text{past}})$ where $\mathbf{a}_{\text{past}}$ is the vector of past actions, $\mathbf{r}_{\text{past}}$ is the vector of their rewards, and θ (the "semicontext") contains any other information that is included in the context.

In this setup, Thompson sampling has the following steps:

Algorithm 2 Thompson sampling.

Repeat as long as desired:

1. **Sample.** Sample a semicontext $\theta \sim P(\theta)$.
2. **Search (and act).** Choose an action $a \in \mathcal{A}$ which (approximately) maximizes $V(a \mid \vartheta) = \mathbb{E}[r \mid a; \vartheta] = \mathbb{E}[r \mid a;\, \theta, \mathbf{a}_{\text{past}}, \mathbf{r}_{\text{past}}]$.
3. **Update.** Let r_{true} be the reward received for action a. Update the believed distribution on θ, i.e., $P(\theta) \leftarrow P_{\text{new}}(\theta)$ where $P_{\text{new}}(\theta) = P(\theta \mid a \mapsto r_{\text{true}})$.

Note that when $\mathbb{E}[r|a; \vartheta]$ (under the sampled value of θ for some points a) is far from the true value $\mathbb{E}_{P_{\text{true}}}[r \mid a]$, the chosen action a may be far from optimal, but the information gained by probing action a will improve the belief ϑ. This amounts to "exploration." When $\mathbb{E}[r \mid a; \vartheta]$ is close to the true value except at points a for which $\mathbb{E}[r \mid a; \vartheta]$ is low, exploration will be less likely to occur, but the chosen actions a will tend to receive high rewards. This amounts to "exploitation." The trade-off between exploration and exploitation is illustrated in Figure 4. Roughly speaking, exploration will happen until the context ϑ is reasonably sure that the unexplored actions are probably not optimal, at which time the Thompson sampler will exploit by choosing actions in regions it knows to have high value.

Typically, when Thompson sampling is implemented, the search over contexts $\vartheta \in \Theta$ is limited by the choice of representation. In traditional programming environments, θ often consists of a few numerical parameters for a family of distributions of a fixed functional form. With work, a mixture of a few functional forms is possible; but without probabilistic programming machinery, implementing a rich context space Θ would be an unworkably large technical burden. In a probabilstic programing language, however, the representation of heterogeneously structured or infinite-dimensional context spaces is quite natural. Any computable model of the conditional distributions $\{P(r \mid a)\}_{a \in \mathcal{A}}$ can be represented as a stochastic procedure $(\lambda(a) \ldots)$. Thus, for computational Thompson sampling, the most general context space $\widehat{\Theta}$ is the space of program texts. Any other context space Θ has a natural embedding as a subset of $\widehat{\Theta}$.

A Mathematical Specification

Our mathematical specification assert the following properties:

- The regression function has a Gaussian process prior.
- The actions $a_1, a_2, \ldots \in \mathcal{A}$ are chosen by a Metropolis-like search strategy with Gaussian drift proposals.
- The hyperparameters of the Gaussian process are inferred using Metropolis–Hastings sampling after each action.

In this version of Thompson sampling, the contexts ϑ are Gaussian processes over the action space $\mathcal{A} = [-20, 20] \subseteq \mathbb{R}$. That is,

$$V \sim \mathcal{GP}(\mu, K),$$

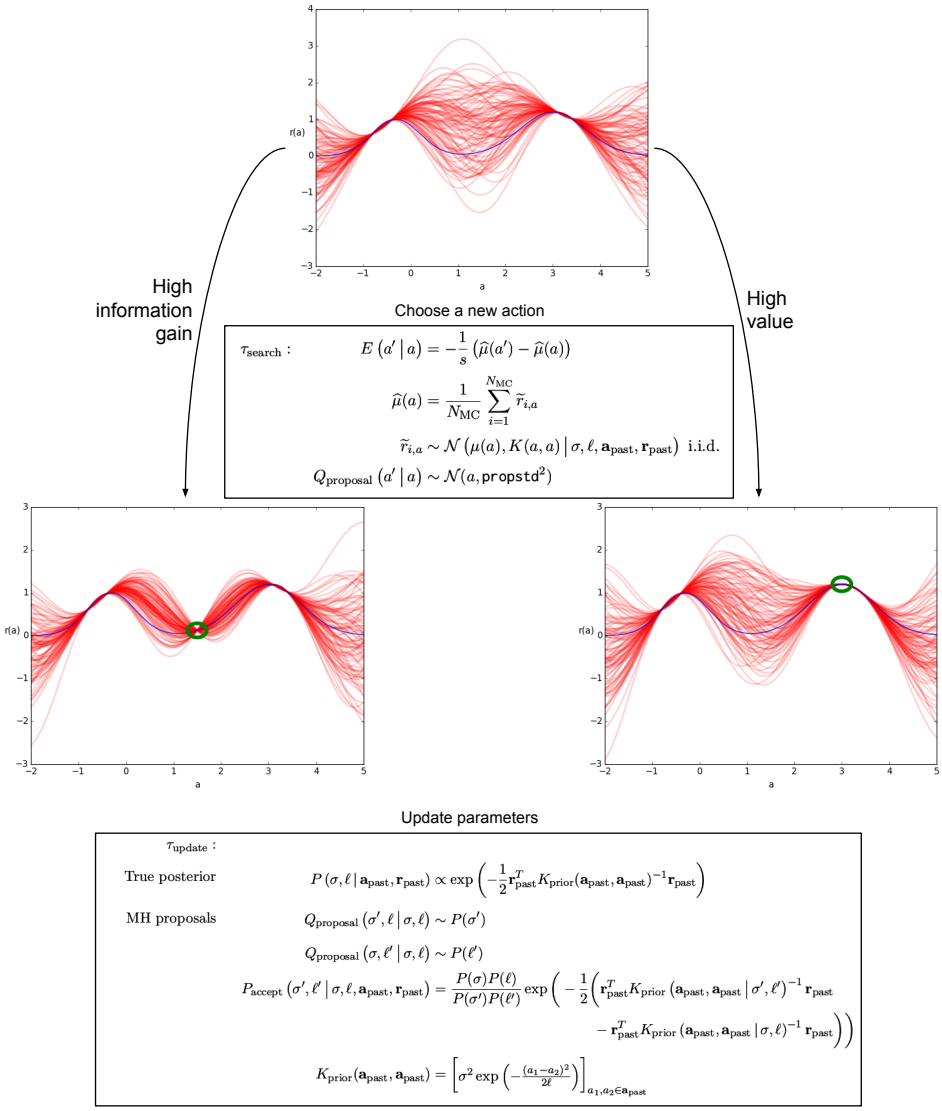


Figure 4: Two possible actions (in green) for an iteration of Thompson sampling. The believed distribution on the value function V is depicted in red. In this example, the true reward function is deterministic, and is drawn in blue. The action on the right receives a high reward, while the action on the left receives a low reward but greatly improves the accuracy of the believed distribution on V. The transition operators τ_{search} and τ_{update} are described in Section 3.3.

where the mean μ is a computable function $\mathcal{A} \to \mathbb{R}$ and the covariance K is a computable (symmetric, positive-semidefinite) function $\mathcal{A} \times \mathcal{A} \to \mathbb{R}$. This represents a Gaussian process $\{R_a\}_{a \in \mathcal{A}}$, where R_a represents the reward for action a. Computationally, we represent a context as a data structure

$$\vartheta = (\theta, \mathbf{a}_{\text{past}}, \mathbf{r}_{\text{past}}) = (\mu_{\text{prior}}, K_{\text{prior}}, \eta, \mathbf{a}_{\text{past}}, \mathbf{r}_{\text{past}}),$$

where μ_{prior} is a procedure to be used as the prior mean function. w.l.o.g. we set $\mu_{\text{prior}} \equiv 0$. K_{prior} is a procedure to be used as the prior covariance function, parameterized by η.

The posterior mean and covariance for such a context ϑ are gotten by the usual conditioning formulas (assuming, for ease of exposition as above, that the prior mean is zero):[3]

$$\begin{aligned}
\mu(\mathbf{a}) &= \mu\left(\mathbf{a} \mid \mathbf{a}_{\text{past}}, \mathbf{r}_{\text{past}}\right) \\
&= K_{\text{prior}}(\mathbf{a}, \mathbf{a}_{\text{past}})\, K_{\text{prior}}(\mathbf{a}_{\text{past}}, \mathbf{a}_{\text{past}})^{-1}\, \mathbf{r}_{\text{past}} \\
K(\mathbf{a}, \mathbf{a}) &= K\left(\mathbf{a}, \mathbf{a} \mid \mathbf{a}_{\text{past}}, \mathbf{r}_{\text{past}}\right) \\
&= K_{\text{prior}}(\mathbf{a}, \mathbf{a}) - K_{\text{prior}}(\mathbf{a}, \mathbf{a}_{\text{past}})\, K_{\text{prior}}(\mathbf{a}_{\text{past}}, \mathbf{a}_{\text{past}})^{-1}\, K_{\text{prior}}(\mathbf{a}_{\text{past}}, \mathbf{a}).
\end{aligned}$$

Note that the context space Θ is not a finite-dimensional parametric family, since the vectors $\mathbf{a}_{\text{past}}$ and $\mathbf{r}_{\text{past}}$ grow as more samples are taken. Θ is, however, representable as a computational procedure together with parameters and past samples, as we do in the representation $\vartheta = (\mu_{\text{prior}}, K_{\text{prior}}, \eta, \mathbf{a}_{\text{past}}, \mathbf{r}_{\text{past}})$.

We combine the Update and Sample steps of Algorithm 2 by running a Metropolis–Hastings (MH) sampler whose stationary distribution is the posterior $P\left(\theta \mid \mathbf{a}_{\text{past}}, \mathbf{r}_{\text{past}}\right)$. The functional forms of μ_{prior} and K_{prior} are fixed in our case, so inference is only done over the parameters $\eta = \{\sigma, \ell\}$; hence we equivalently write $P\left(\sigma, \ell \mid \mathbf{a}_{\text{past}}, \mathbf{r}_{\text{past}}\right)$ for the stationary distribution. We make MH proposals to one variable at a time, using the prior as proposal distribution:

$$Q_{\text{proposal}}\left(\sigma', \ell \mid \sigma, \ell\right) = P(\sigma')$$

and

$$Q_{\text{proposal}}\left(\sigma, \ell' \mid \sigma, \ell\right) = P(\ell').$$

The MH acceptance probability for such a proposal is

$$P_{\text{accept}}\left(\sigma', \ell' \mid \sigma, \ell\right) = \min\left\{1,\ \frac{Q_{\text{proposal}}\left(\sigma, \ell \mid \sigma', \ell'\right)}{Q_{\text{proposal}}\left(\sigma', \ell' \mid \sigma, \ell\right)} \cdot \frac{P\left(\mathbf{a}_{\text{past}}, \mathbf{r}_{\text{past}} \mid \sigma', \ell'\right)}{P\left(\mathbf{a}_{\text{past}}, \mathbf{r}_{\text{past}} \mid \sigma, \ell\right)}\right\}$$

Because the priors on σ and ℓ are uniform in our case, the term involving Q_{proposal} equals 1 and we have simply

$$\begin{aligned}
P_{\text{accept}}\left(\sigma', \ell' \mid \sigma, \ell\right) &= \min\left\{1,\ \frac{P\left(\mathbf{a}_{\text{past}}, \mathbf{r}_{\text{past}} \mid \sigma', \ell'\right)}{P\left(\mathbf{a}_{\text{past}}, \mathbf{r}_{\text{past}} \mid \sigma, \ell\right)}\right\} \\
&= \min\left\{1,\ \exp\left(-\frac{1}{2}\left(\mathbf{r}_{\text{past}}^T K_{\text{prior}}\left(\mathbf{a}_{\text{past}}, \mathbf{a}_{\text{past}} \mid \sigma', \ell'\right)^{-1} \mathbf{r}_{\text{past}}\right.\right.\right. \\
&\qquad \left.\left.\left. - \mathbf{r}_{\text{past}}^T K_{\text{prior}}\left(\mathbf{a}_{\text{past}}, \mathbf{a}_{\text{past}} \mid \sigma, \ell\right)^{-1} \mathbf{r}_{\text{past}}\right)\right)\right\}.
\end{aligned}$$

The proposal and acceptance/rejection process described above define a transition operator τ_{update} which is iterated a specified number of times; the resulting state of the MH Markov chain is taken as the sampled semicontext θ in Step 1 of Algorithm 2.

For Step 2 (Search) of Thompson sampling, we explore the action space using an MH-like transition operator τ_{search}. As in MH, each iteration of τ_{search} produces a proposal which is either accepted or rejected, and the state of this Markov chain after a specified number of steps is the new action a. The Markov chain's initial state is the most recent action, and the proposal distribution is Gaussian drift:

$$Q_{\text{proposal}}\left(a' \mid a\right) \sim \mathcal{N}(a, \texttt{propstd}^2),$$

[3] Here, for vectors $\mathbf{a} = (a_i)_{i=1}^n$ and $\mathbf{a}' = (a'_i)_{i=1}^{n'}$, $\mu(\mathbf{a})$ denotes the vector $(\mu(a_i))_{i=1}^n$ and $K(\mathbf{a}, \mathbf{a}')$ denotes the matrix $\left[K(a_i, a'_j)\right]_{1 \le i \le n, 1 \le j \le n'}$.

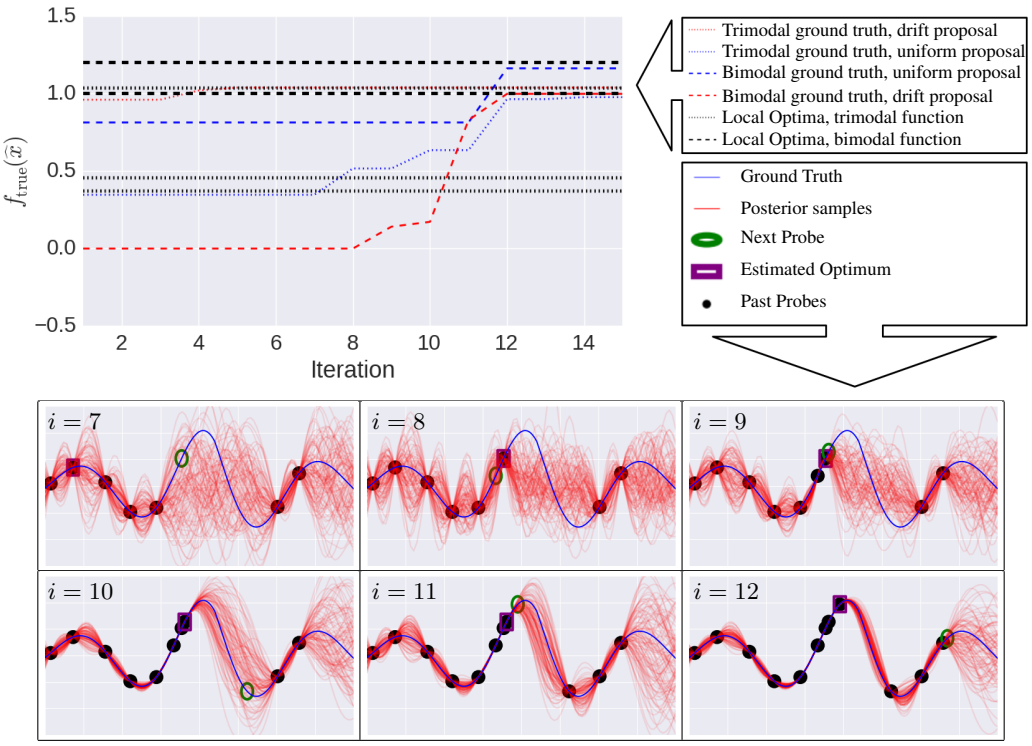


Figure 5: Top: the estimated optimum over time. Blue and Red represent optimization with uniform and Gaussian drift proposals. Black lines indicate the local optima of the true functions. Bottom: a sequence of actions. Depicted are iterations 7-12 with uniform proposals.

where the drift width `propstd` is specified ahead of time. The acceptance probability of such a proposal is

$$P_{\text{accept}}\left(a' \mid a\right) = \min\left\{1,\ \exp\left(-E\left(a' \mid a\right)\right)\right\},$$

where the energy function $E\left(\bullet \mid a\right)$ is given by a Monte Carlo estimate of the difference in value from the current action:

$$E\left(a' \mid a\right) = -\frac{1}{s}\left(\widehat{\mu}(a') - \widehat{\mu}(a)\right)$$

where

$$\widehat{\mu}(a) = \frac{1}{N_{\text{avg}}} \sum_{i=1}^{N_{\text{avg}}} \widetilde{r}_{i,a}$$

and

$$\widetilde{r}_{i,a} \sim \mathcal{N}(\mu(a), K(a,a))$$

and $\{\widetilde{r}_{i,a}\}_{i=1}^{N_{\text{avg}}}$ are i.i.d. for a fixed a. Here the temperature parameter $s \geq 0$ and the population size N_{avg} are specified ahead of time. Proposals of estimated value higher than that of the current action are always accepted, while proposals of estimated value lower than that of the current action are accepted with a probability that decays exponentially with respect to the difference in value. The rate of the decay is determined by the temperature parameter s, where high temperature corresponds to generous acceptance probabilities. For $s = 0$, all proposals of lower value are rejected; for $s = \infty$, all proposals are accepted. For points a at which the posterior mean $\mu(a)$ is low but the posterior variance $K(a,a)$ is high, it is possible (especially when N_{avg} is small) to draw a "wild" value of $\widehat{\mu}(a)$, resulting in a favorable acceptance probability.

Indeed, taking an action a with low estimated value but high uncertainty serves the useful function of improving the accuracy of the estimated value function at points near a (see Figure 4).[4,5] We see a comlete probabilistic program with gpmem implementing Bayesian optimization with Thompson Sampling below (Listing 3).

Listing 3: Bayesian optimization using gpmem

```
assume sf = tag(quote(hyper), 0, uniform_continuous(0, 10))
assume l = tag(quote(hyper), 1, uniform_continuous(0, 10))
assume se = make_squaredexp(sf, l)
assume blackbox_f = get_bayesopt_blackbox()
assume (f_compute, f_emulate) = gpmem(blackbox_f, se)

// A naive estimate of the argmax of the given function
define mc_argmax = proc(func) {
  candidate_xs = mapv(proc(i) {uniform_continuous(-20, 20)},
                      arange(20));
  candidate_ys = mapv(func, candidate_xs);
  lookup(candidate_xs, argmax_of_array(candidate_ys))
};

// Shortcut to sample the emulator at a single point without packing
// and unpacking arrays
define emulate_pointwise = proc(x) {
    run(sample(lookup(f_emulate(array(unquote(x))), 0)))
};

// Main inference loop
infer repeat(15, do(pass,
    // Probe V at the point mc_argmax(emulate_pointwise)
    predict(f_compute(unquote(mc_argmax(emulate_pointwise)))),
    // Infer hyperparameters
    mh(quote(hyper), one, 50)));
```

4 Discussion

We provided gpmem, an elegant linguistic framework for function learning-related tasks such as Bayesian optimization and GP kernel structure learning. We highlighted how gpmem overcomes shortcomings of the notations currently used in statistics, and how language constructs from programming allow the expression of models which would be cumbersome (prohibitively so, in some cases) to express in statistics notation. We evaluated our contribution on a range of hard problems for state-of-the-art Bayesian nonparametrics.

[4] At least, this is true when we use a smoothing prior covariance function such as the squared exponential.

[5] For this reason, we consider the sensitivity of $\widehat{\mu}$ to uncertainty to be a desirable property; indeed, this is why we use $\widehat{\mu}$ rather than the exact posterior mean μ.

Appendix

A Covariance Functions

$$\text{SE} = \sigma^2 \exp(-\frac{(x-x')^2}{2\ell^2}) \tag{12}$$

$$\text{LIN} = \sigma^2(xx') \tag{13}$$

$$\text{C} = \sigma^2 \tag{14}$$

$$\text{WN} = \sigma^2\delta_{x,x'} \tag{15}$$

$$\text{RQ} = \sigma^2\left(1 + \frac{(x-x')^2}{2\alpha\ell^2}\right)^{-\alpha} \tag{16}$$

$$\text{PER} = \sigma^2 \exp\left(\frac{2\sin^2(\pi(x-x')/p}{\ell^2}\right). \tag{17}$$

B Covariance Simplification

`SE` $\times$ `SE`	$\rightarrow$	`SE`
`{SE,PER,C,WN}` $\times$ `WN`	$\rightarrow$	`WN`
`LIN` + `LIN`	$\rightarrow$	`LIN`
`{SE,PER,C,WN,LIN}` $\times$ `C`	$\rightarrow$	`{SE,PER,C,WN,LIN}`

Rule 1 is derived as follows:

$$\begin{aligned}
\sigma_c^2 \exp(-\frac{(x-x')^2}{2\ell_c^2}) &= \sigma_a^2 \exp(-\frac{(x-x')^2}{2\ell_a^2}) \times \sigma_b^2 \exp(-\frac{(x-x')^2}{2\ell_b^2}) \\
&= \sigma_c^2 \exp(-\frac{(x-x')^2}{2\ell_a^2}) \times \exp(-\frac{(x-x')^2}{2\ell_b^2}) \\
&= \sigma_c^2 \exp\left(-\frac{(x-x')^2}{2\ell_a^2} - \frac{(x-x')^2}{2\ell_b^2}\right) \\
&= \sigma_c^2 \exp\left(-\frac{(x-x')^2}{2\ell_c^2}\right)
\end{aligned} \tag{18}$$

For stationary kernels that only depend on the lag vector between x and x' it holds that multiplying such a kernel with a WN kernel we get another WN kernel (Rule 2). Take for example the SE kernel:

$$\sigma_a^2 \exp\left(-\frac{(x-x')^2}{2\ell_c^2}\right) \times \sigma_b\delta_{x,x'} = \sigma_a\sigma_b\delta_{x,x'} \tag{19}$$

Rule 3 is derived as follows:

$$\theta_c(x \times x') = \theta_a(x \times x') + \theta_b(x \times x') \tag{20}$$

Multiplying any kernel with a constant obviously changes only the scale parameter of a kernel (Rule 4).

C Additional Structure Learning Results

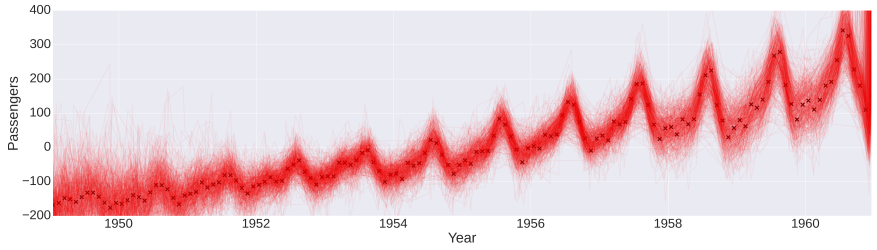


Figure 6: Data (black x) and posterior samples (red) for the airline data set.

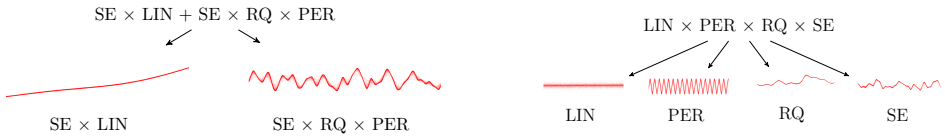


Figure 7: We see two possible hypotheses for the composite structure of $\mathbf{K}$. (a) Most frequent sample drawn from the posterior on structure. We have found two global components. First, a smooth trend (LIN $\times$ SE) with a non-linear increasing slope. Second, a periodic component with increasing variation and noise. (b) Second most frequent sample drawn from the posterior on structure. We found one global component. It is comprised of local changes that are periodic and with changing variation.

References

Box, G. E. P., Jenkins, G. M., and Reinsel, G. C. (1997). *Time series analysis: forecasting and control*.

Clifton, L., Clifton, D. A., Pimentel, M. A. F., Watkinson, P. J., and Tarassenko, L. (2013). Gaussian processes for personalized e-health monitoring with wearable sensors. *Biomedical Engineering, IEEE Transactions on*, 60(1):193–197.

Duvenaud, D., Lloyd, J. R., Grosse, R., Tenenbaum, J., and Ghahramani, Z. (2013). Structure discovery in nonparametric regression through compositional kernel search. In *Proceedings of the International Conference on Machine Learning (ICML)*, pages 1166–1174.

Ghahramani, Z. (2015). Probabilistic machine learning and artificial intelligence. *Nature*, 521(7553):452–459.

Goodman, N. D .and Mansinghka, V. K., Roy, D., Bonawitz, K., and Tenenbaum, J. (2008). Church: A language for generative models. In *Proceedings of the Conference on Uncertainty in Artificial Intelligence (UAI)*, pages 220–229.

Kemmler, M., Rodner, E., Wacker, E., and Denzler, J. (2013). One-class classification with gaussian processes. *Pattern Recognition*, 46(12):3507–3518.

Lloyd, J. R., Duvenaud, D., Grosse, R., Tenenbaum, J., and Ghahramani, Z. (2014). Automatic construction and natural-language description of nonparametric regression models. In *Proceedings of the Conference on Artificial Intelligence (AAAI)*.

Mansinghka, V. K., Selsam, D., and Perov, Y. (2014). Venture: a higher-order probabilistic programming platform with programmable inference. *arXiv preprint arXiv:1404.0099*.

McAllester, D., Milch, B., and Goodman, N. D. (2008). Random-world semantics and syntactic independence for expressive languages. Technical report.

Neal, R. M. (1997). Monte carlo implementation of gaussian process models for bayesian regression and classification. *arXiv preprint physics/9701026*.

Poole, D. (1993). Probabilistic horn abduction and bayesian networks. *Artificial Intelligence*, 64(1):81–129.

Rasmussen, C. E. and Williams, C. K. I. (2006). *Gaussian Processes for Machine Learning (Adaptive Computation and Machine Learning)*. The MIT Press.

Sato, T. (1995). A statistical learning method for logic programs with distribution semantics. In *In Proceedings of the International Conference on Logic Programming*. Citeseer.

Shepherd, T. and Owenius, R. (2012). Gaussian process models of dynamic pet for functional volume definition in radiation oncology. *Medical Imaging, IEEE Transactions on*, 31(8):1542–1556.

Snoek, J., Larochelle, H., and Adams, R. P. (2012). Practical bayesian optimization of machine learning algorithms. In *Advances in Neural Information Processing Systems (NIPS)*, pages 2951–2959.

Thompson, W. R. (1933). On the likelihood that one unknown probability exceeds another in view of the evidence of two samples. *Biometrika*, pages 285–294.