7th ICML Workshop on Automated Machine Learning (2020)

Task-Agnostic Amortized Inference of Gaussian Process Hyperparameters

Sulin Liu Xingyuan Sun Peter J. Ramadge Ryan P. Adams
SULINL,XS5,RAMADGE,RPA@PRINCETON.EDU
Princeton University

Abstract

Gaussian processes (GPs) are flexible priors for modeling functions. However, their success depends on the kernel accurately reflecting the properties of the data. One of the appeals of the GP framework is that the marginal likelihood of the kernel hyperparameters is often available in closed form, enabling optimization and sampling procedures to fit these hyperparameters to data. Unfortunately, point-wise evaluation of the marginal likelihood is expensive due to the need to solve a linear system; searching or sampling the space of hyperparameters thus often dominates the practical cost of using GPs. We introduce an approach to the identification of kernel hyperparameters in GP regression and related problems that sidesteps the need for costly marginal likelihoods. Our strategy is to "amortize" inference over hyperparameters by training a single neural network, which consumes a set of regression data and produces an estimate of the kernel function, useful across different tasks. To accommodate the varying dimension and cardinality of different regression problems, we use a hierarchical self-attention-based neural network that produces estimates of the hyperparameters which are invariant to the order of the input data points and data dimensions. We show that a single neural model trained on synthetic data is able to generalize directly to several different real-world GP use cases. Our experiments demonstrate that the estimated hyperparameters are comparable in quality to those from the conventional model selection procedures, while being much faster to obtain, significantly accelerating GP regression and its related applications such as Bayesian optimization and Bayesian quadrature.

1. Introduction

Gaussian processes (GPs) are powerful tools for modeling distribution over functions. They are highly flexible Bayesian nonparametric models, with the additional property that the posterior is often available in closed form. GPs are used in a variety of machine learning tasks, from regression and classification [31], to Bayesian optimization [38], to modeling of dynamics [19]. The predictive performance of a Gaussian process, however, is highly dependent on the specifics of the prior on functions, as determined by the associated positive definite kernel function. To find a good prior, one needs to first come up with a family of kernel functions that is capable of capturing the structure of the data. The kernel hyperparameters must then be determined, usually by maximizing the log marginal likelihood (MLL), i.e., empirical Bayes [23]. The MLL maximization procedure is well-known to have two major issues: costly evaluation ($\mathcal{O}(n^3)$ complexity) of the log MLL in gradient calculation and bad local maxima issue due to its non-concavity [30, 27]. To reduce the computation cost, most efforts are devoted to reducing the size of the linear system involved without significantly compromising the predictvie performance. This is usually done by intelligently selecting a subset of the data [36, 42] or constructing a low-rank approximation of the covariance matrix based on virtual "inducing" point [5, 29, 34, 37, 40, 14].

Here, however, we take an entirely different approach and focus solely on the model selection problem, without reference to linear systems at all. Instead, we *amortize* the Gaussian process model selection problem by training a neural network to consume input/output observations and emit an estimate of the hyperparameters that would otherwise arise from maximizing the log marginal likelihood. This approach is inspired by amortized variational inference approaches [18, 32, 13, 33], which similarly sidestep expensive optimization procedures in favor of directly producing estimates. In the variational inference case, the neural network produces posterior estimates of a set of unknown latent variables; here we produce point estimates of the unknown hyperparameters.

Of course, as noted above, selection of the kernel family is just as important as determination of hyperparameters, and we view this as a crucial piece of the amortized model selection puzzle. One approach to modeling the huge space of valid kernel functions would be to represent the target kernel as a composition of different base kernels [7, 22, 39]. In principle, various base kernels, composition rules, and associated hyperparameters could be modeled as latent random variables, produced by a neural network architecture. However, we have found this approach to be difficult due to the optimization challenge introduced by their complicated intercorrelated structure. Thus, we instead focus on stationary kernels in the spectral domain and directly learn the spectral density of the kernel function [43]. Turning to the spectral domain provides us with a unified and compact continuous representation of the space of stationary covariance functions.

There are two particularly salient challenges associated with training a single neural network to produce effective hyperparameters for many different regression-type tasks: both the amount of data and the dimensionality of the input can vary from problem to problem. To address this, we develop a specialized hierarchical self-attention structure that consumes datasets and produces spectral densities while being invariant to permutations of the data and the dimensions. Thus, this single "meta-model" can be applied to different problems for which a Gaussian process is applicable. The parameters of the network are trained using gradients computed via reverse-mode automatic differentiation through the log marginal likelihood of the Gaussian process, for randomly-generated synthetic data from the prior. Then we directly apply the trained neural model to real-world datasets of varying size and dimension in different GP applications. Even though the model is trained with only synthetic data, experimental evidence indicates that the estimated hyperparameters are comparable in quality to those from the conventional procedures while being $\sim$100 times faster to obtain.

2. Amortized GP Hyperparameter Inference

In this section, we introduce *amortized hyperparameters inference for GP* (AHGP), a method that replaces the MLL maximization procedure with a direct estimate of the kernel hyperparameters through a massive-parametric function approximator, i.e., a neural network. An introduction to GP hyperparameter inference can be found in Appendix A. When performing Gaussian process regression, it is common to first identify a family of Gaussian process priors. Here we focus on Gaussian process priors induced by stationary kernels which include many widely-used kernels, e.g., exponentiated quadratic, rational quadratic, and periodic.

Spectral Modeling of Stationary Kernel Functions. In lieu of a compositional approach to the kernel function, we build a flexible approach around the duality between stationary kernels and their spectral density, taking advantage of the well-known theorem by

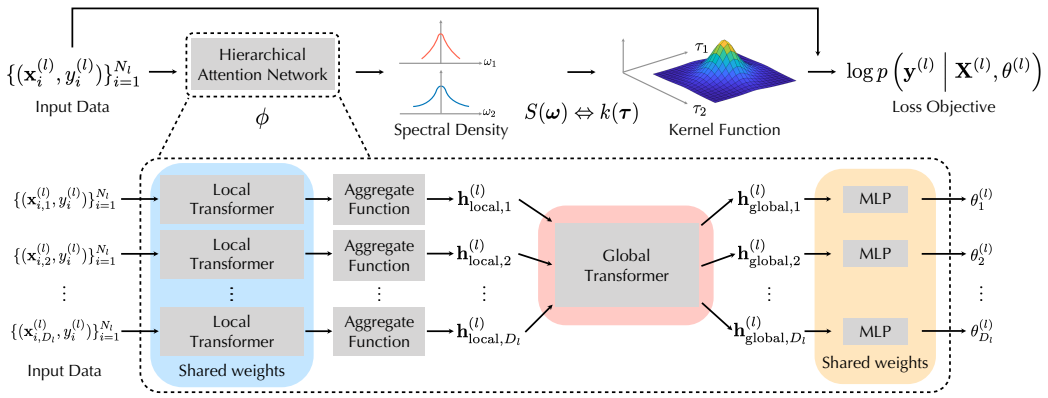


Figure 1: The top part of the figure gives an illustration of the computation graph in AHGP. The bottom part describes our hierarchical attention neural net architecture.

Bochner (details can be found in Appendix B). The theorem states that all stationary kernel functions, are uniquely described by their spectral densities in the frequency domain.

We take advantage of this correspondence and use a neural network to predict the spectral density of the kernel function rather than the covariance function itself. Following previous work [43, 44], we model the spectral density via a Gaussian mixture, leading to interpretability and closed form evaluation of the kernel. Additionally, the fact that Gaussian mixtures are dense in the space of probability distribution functions [35, 43] makes them capable of approximating the spectral density of any stationary kernel function arbitrarily well. Here we further assume that the kernel function has a product structure over different dimensions and every dimension has its own mixture of Gaussians. The product kernel structure is a common modeling choice in Gaussian process regression, and arises naturally in many generic kernels such as the exponentiated quadratic. Additionally, it is common to compose kernels via element-wise products, with each dimension's functional properties encoded in its corresponding kernel function [31, 11, 7, 44].

Formulation. We now formalize this problem of amortized hyperparameter inference in a Gaussian process. We are interested in fitting many different regression functions of the form $f : \mathbb{R}^D \to \mathbb{R}$, with varying values of D. For the l-th regression task $\mathcal{T}^{(l)}$, a set of input/output training data are observed and are given by $\mathcal{D}^{(l)} := \{(\mathbf{x}_i^{(l)}, y_i^{(l)})\}_{i=1}^{N_l} = \{\mathbf{X}^{(l)}, \mathbf{y}^{(l)}\}$, where $y_i^{(l)} = f_l(\mathbf{x}_i^{(l)}) + \epsilon_i^{(l)}$ with $\mathbf{x}_i^{(l)} \in \mathbb{R}^{D_l}$ and $\epsilon_i^{(l)}$ being i.i.d. zero-mean Gaussian noise. Assume we are given L tasks, with each task randomly sampled i.i.d. from a distribution over tasks, i.e., $\{\mathcal{T}^{(l)}\}_{l=1}^{L} \overset{i.i.d.}{\sim} p(\mathcal{T})$. We further assume that for each task, the function values are generated by some underlying Gaussian process with its own unique kernel hyperparameters. The spectral density of each task's GP is modeled as a mixture of M Gaussians over each dimension as discussed above, with weights, means and variances denoted as $\theta_d^{(l)} = \{\{w_{d,m}^{(l)}\}_{m=1}^{M}, \{\mu_{d,m}^{(l)}\}_{m=1}^{M}, \{\sigma^{2(l)}_{d,m}\}_{m=1}^{M}\}$ for the d-th dimension in task l. For compactness, we use $\theta^{(l)} = \{\theta_d^{(l)}\}_{d=1}^{D_l}$ to denote the collective hyperparameters for task l.

The neural network, parameterized by ϕ, defines a function g_ϕ from a dataset $\mathcal{D}^{(l)}$ to an estimate of its spectral density hyperparameters $\theta^{(l)}$, i.e., $\theta^{(l)} = g_\phi(\mathcal{D}^{(l)})$. Through the duality of spectral densities and stationary kernel functions, the spectral mixture product (SMP) kernel [44] is given in closed form (see Appendix B). We can now train the neural network to produce hyperparameters using a "dataset of datasets", $\{\mathcal{D}^{(l)}\}_{l=1}^{L}$. With the closed form kernel function specified by its spectral density hyperparameters, the averaged negative log marginal likelihood (see Appendix A Eqn (3)) is used as our training objective:

$$\mathcal{L}\left(\phi, \left\{\mathcal{D}^{(l)}\right\}_{l=1}^{L}\right) = -\frac{1}{L}\sum_{l=1}^{L}\frac{1}{N_l}\log p\left(\mathbf{y}^{(l)} \,\middle|\, \mathbf{X}^{(l)}, \theta^{(l)}\right), \tag{1}$$

where $\theta^{(l)} = g_\phi(\mathcal{D}^{(l)})$ and N_l is the number of data points in the l-th dataset. Once the neural network is trained, it can be used to estimate the kernel function that would be appropriate for a new set of input/output data $\mathcal{D}^{\text{test}}$ by simply doing a forward pass of the neural model.

3. Hierarchical Attention Network for GP Hyperparameter Learning

As described in the previous section, the neural network learns a function from a dataset $\mathcal{D}^{(l)}$ to spectral density parameters $\{\theta_d^{(l)}\}_{d=1}^{D_l}$, determining the GP kernel function for task l. As in other deep learning problems, the architecture of the neural network is critical; in particular, the structure of the network must take advantage of available symmetries. In our case, for general purpose inference of GP kernel hyperparameters, we require an architecture that is versatile enough to accommodate datasets of varying input dimension and with different number of data points. Furthermore, the model should be invariant to permutation of both the data and input dimensions. In other words, for a given dataset $\mathcal{D}^{(l)}$, neither shuffling the order of the (exchangeable) data nor shuffling the order of the dimensions should change the resulting estimate of the kernel function. Importantly, the regularization and parameter sharing induced by enforcement of such invariances should enable the neural network to learn better and faster, analogously to convolutional neural networks for images.

Architecture. We draw inspiration from multi-head self-attention mechanisms and propose a hierarchical Transformer [41] type of neural network architecture for tackling the problem of learning GP hyperparameters. A general Transformer model has multiple layers and each layer consists of a multi-head self-attention sub-layer followed by a feed forward network with residual connections and layer normalizations. It serves as an autoregressive encoder that maps a set of input data to a set of output representations. In particular, the self-attention sub-layers allow each input datum to attend to the representations of other data and produce context-aware representations. Multiple layers of self-attention enable modeling of high-order non-linear interactions between input representations. For details about multi-head self-attention mechanisms, we refer readers to Vaswani et al. [41].

Briefly, our network architecture mainly consists of two hierarchically nested Transformer-like blocks. A graphic illustration of the proposed architecture is presented in Fig. 1.

Local Transformer: The first transformer block, LocalTransformer, serves as an encoder of the per-dimension local information about the observed function, e.g., length scale, smoothness, periodicity, etc. It takes in a set of input/output data specific to the d-th dimension, e.g., $\mathcal{D}_d^{(l)} = \{(\mathbf{x}_{i,d}^{(l)}, y_i^{(l)})\}_{i=1}^{N_l}$, and outputs a corresponding set of representations $\{\mathbf{h}_{i,d}^{(l)}\}_{i=1}^{N_l}$.

Aggregate Function: The outputs from LocalTransformer$\{\mathbf{h}_{i,d}^{(l)}\}_{i=1}^{N_l}$ are aggregated through an AggregateFunction that assembles a single local dimension-specific summary representation $\mathbf{h}_{\text{local},d}^{(l)}$ for the d-th dimension.

Global Transformer: After the dimension-specific local representations $\{\mathbf{h}_{\text{local},d}^{(l)}\}_{d=1}^{D_l}$ are computed, they are fed into a second dimension-level Transformer block, GlobalTransformer, where non-linear interactions between the dimensions are modeled through multiple layers of multi-head self-attention. The final per-dimensional representations $\{\mathbf{h}_{\text{global},d}^{(l)}\}_{d=1}^{D_l}$, which serve as context-aware representations at a global (dimension) level, are further passed through a MLP to produce the final spectral density hyperparameters $\{\theta_d^{(l)}\}_{d=1}^{D_l}$.

Versatility and permutation invariance. Self-attention enables the model to consume a set of input/output data with arbitrary data cardinality and dimensionality, making it possible to train a single neural model to predict GP kernel hyperparameters of different tasks with varying data cardinality and dimensionality, as long as the inputs are real-valued. The proposed model also possesses the following permutation equivariance/invariance properties.

Proposition 1 *If* AggregateFunction *is permutation invariant, and weights of* LocalTransformer *and* MLP *are shared across dimensions, then the proposed neural network is permutation equivariant with respect to data dimensions and invariant with respect to data points.*

4. Experimental Results

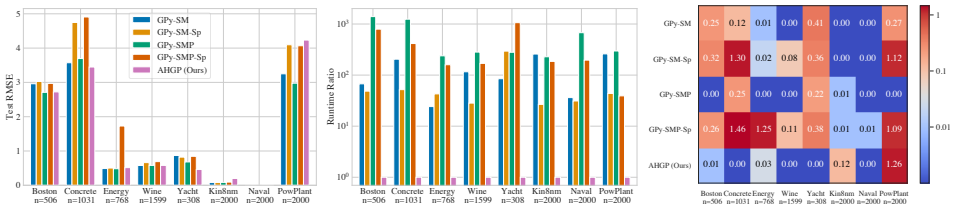
Baselines. We compare our method to the standard approach of maximizing the log marginal likelihood with respect to hyperparameters. We also compare with the sparse variational Gaussian processes method (SGPR) [40, 14], which uses inducing points to approximate the full GP. The focus of the comparisons will be on the quality of the selected kernel hyperparameters and the run time of the hyperparameter selection procedure.

The baselines are implemented with two popular GP packages: GPy [12] (implemented for CPU) and GPyTorch [10] (implemented for GPU). The spectral mixture (SM) kernel and spectral mixture product (SMP) kernel are used as the kernel functions. These give rise to eight different baselines: GPy-SM, GPy-SM-Sp, GPy-SMP, GPy-SMP-Sp, GPT-SM, GPT-SM-Sp, GPT-SMP, GPT-SMP-Sp, where "GPT" denotes "GPyTorch" and "Sp" denotes "SGPR". The GPyTorch baselines make use of batched conjugate gradient to invert the kernel matrix for efficient approximate inference. We additionally implement a full GP baseline with SMP kernel that uses Cholesky decomposition in PyTorch [28], and its MLL is optimized via reverse-mode automatic differentiation. We will refer to this baseline as PyT-AD-SMP.

Training Setup. In our experiments, the training data are constructed by sampling multiple sets of synthetic input/output data from a GP prior with a stationary kernel. Dimensions vary from 2 to 15. More details about data generation are provided in Appendix E. A single neural model is trained on the synthetic data using Adam [17] with a fixed learning rate, and the same trained model is then used across all evaluations.

Regression benchmarks. We evaluate our method and the baselines on regression benchmarks from the UCI collection [1] used in Hernández-Lobato and Adams [15] and Sun et al. [39] following the same setup: the data are randomly split to 90% for training and 10% for testing. This splitting process is repeated 10 times and the average test performance is reported. Comparisons with CPU-based baselines are presented in Fig. 2 and Table 1 (Appendix E). We observe that AHGP has consistently lower run times than the baselines, averaging $\sim$100 times faster. Nevertheless, the predictive performance of AHGP is comparable to (and sometimes better than) the strongest baselines, which perform MLL optimization without approximation. Notably, AHGP seems to perform slightly better on datasets with fewer data points, such as Yacht. We believe this demonstrates the robustness of AHGP when there is not enough data for MLL-opt based approaches to form reasonable point estimates. The sparse variational GP methods are faster than the full GP methods in general, but with lower performance on both test RMSE and test log-likelihood. Comparisons with GPU-based methods are in Appendix E, where similar findings are obtained.

Bayesian optimization. Bayesian optimization [24] (BO) uses a GP as a surrogate model when the objective function is expensive to evaluate. The method involves fitting the GP



(a) Test RMSE **(b)** Runtime ratio over AHGP **(c)** Gap with best RMSE

Figure 2: Comparison of AHGP against the CPU baselines on regression benchmarks. In (c), the numbers are the differences of the corresponding method's test RMSE with the best RMSE on the respective dataset. Note that for Naval, the RMSEs are all very close to 0. (Only average test performance is shown here. Refer to Appendix E for complete results with error bars.)

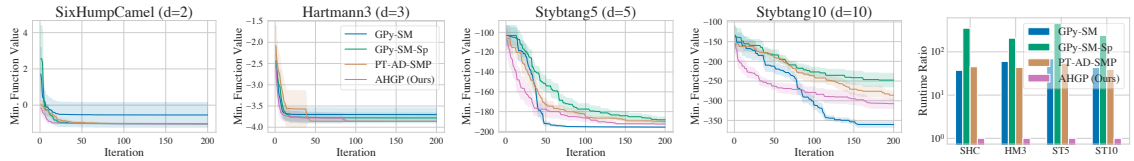


Figure 3: BO performance comparisons. *Left*: Minimum function values found v.s. number of BO iterations. Shaded region represents 0.5 standard deviation over 10 runs. *Right*: Runtime ratio over AHGP. (Only average is plotted here, refer to Appendix E for mean and standard deviation.)

kernel hyperparameters and maximizing an acquisition function to select a candidate point that is highly promising to achieve the function minima (maxima) under the model. Since the method is iterative, MLL optimization needs to be conducted at every BO iteration to update the GP. An amortized approach would greatly reduce the computation involved. We pick the best performing baselines on the regression benchmarks (GPy-SM, GPy-SM-Sp, PyT-AD-SMP) and compare them with AHGP. We use standard test functions for global optimization [6] as the target functions for Bayesian optimization.

At the start of every BO iteration, the hyperparameters are randomly re-initialized for all baselines. A sample of the experimental results is shown in Fig. 3. (Full results can be found in Appendix E.) Again, AHGP is a substantial improvement in run-time. In terms of minimum values found, AHGP is on par with the baselines on some functions and slightly worse on functions with higher dimensionality. Of particular note—consistent with what was seen on the regression benchmarks—AHGP has the greatest improvement in the beginning when there are few observations available.

To ensure fairness to baseline fitting procedures, we also conducted experiments where the hyperparameter selection in the BO inner loop was initialized using the best from the previous iteration. As expected, this warm-starting results in decreased run times for the baselines, although still slower than AHGP (in Appendix E). This warm-starting, however, seems to compromise the hyperparameter selection—presumably due to local minima—and damage the overall outer loop optimization.

5. Conclusions

We introduced *amortized hyperparameters inference for Gaussian processes* (AHGP). Our proposed neural model is not only versatile to accommodate tasks of different size and dimensionality but also permutation invariant of both data and dimensions. We experimentally show that a single amortized inference model trained on synthetic data is able to directly generalize to different real-world GP use cases. Our model is capable of producing hyperparameters that are comparable in quality to those from the conventional MLL maximization approaches, while being on average $\sim$100 times faster.

References

[1] Arthur Asuncion and David Newman. UCI machine learning repository, 2007.

[2] James O. Berger. *Statistical decision theory and Bayesian analysis*. Springer Science & Business Media, 2013.

[3] Salomon Bochner. *Lectures on Fourier integrals*, volume 42. Princeton University Press, 1959.

[4] Rewon Child, Scott Gray, Alec Radford, and Ilya Sutskever. Generating long sequences with sparse transformers. *arXiv preprint arXiv:1904.10509*, 2019.

[5] Lehel Csató and Manfred Opper. Sparse on-line Gaussian processes. *Neural computation*, 14(3):641–668, 2002.

[6] Derek Bingham. Global optimization test problems. `http://www-optima.amp.i.kyoto-u.ac.jp/member/student/hedar/Hedar_files/TestGO.htm.`, June 2013.

[7] David Duvenaud, James Robert Lloyd, Roger Grosse, Joshua B. Tenenbaum, and Zoubin Ghahramani. Structure discovery in nonparametric regression through compositional kernel search. *arXiv preprint arXiv:1302.4922*, 2013.

[8] Maurizio Filippone and Mark Girolami. Pseudo-marginal Bayesian inference for Gaussian processes. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 36(11): 2214–2226, 2014.

[9] Reeves Fletcher and Colin M. Reeves. Function minimization by conjugate gradients. *The Computer Journal*, 7(2):149–154, 1964.

[10] Jacob Gardner, Geoff Pleiss, Kilian Q. Weinberger, David Bindel, and Andrew G. Wilson. GPyTorch: Blackbox matrix-matrix Gaussian process inference with GPU acceleration. In *Advances in Neural Information Processing Systems*, pages 7576–7586, 2018.

[11] Mehmet Gönen and Ethem Alpaydin. Multiple kernel learning algorithms. *Journal of machine learning research*, 12(64):2211–2268, 2011.

[12] GPy. GPy: A Gaussian process framework in python. `http://github.com/SheffieldML/GPy`, 2012.

[13] Karol Gregor, Ivo Danihelka, Andriy Mnih, Charles Blundell, and Daan Wierstra. Deep autoregressive networks. In *International Conference on Machine Learning*, pages 1242–1250, 2014.

[14] James Hensman, Nicolo Fusi, and Neil D. Lawrence. Gaussian processes for big data. In *Uncertainty in Artificial Intelligence*, page 282, 2013.

[15] José Miguel Hernández-Lobato and Ryan P. Adams. Probabilistic backpropagation for scalable learning of Bayesian neural networks. In *International Conference on Machine Learning*, pages 1861–1869, 2015.

[16] Magnus R. Hestenes et al. Methods of conjugate gradients for solving linear systems. *Journal of Research of the National Bureau of Standards*, 49(6):409–436, 1952.

[17] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[18] Diederik P. Kingma and Max Welling. Auto-encoding variational Bayes. *arXiv preprint arXiv:1312.6114*, 2013.

[19] Malte Kuss and Carl E. Rasmussen. Gaussian processes in reinforcement learning. In *Advances in Neural Information Processing Systems*, pages 751–758, 2004.

[20] Juho Lee, Yoonho Lee, Jungtaek Kim, Adam Kosiorek, Seungjin Choi, and Yee Whye Teh. Set Transformer: A framework for attention-based permutation-invariant neural networks. In *International Conference on Machine Learning*, pages 3744–3753, 2019.

[21] Dong C. Liu and Jorge Nocedal. On the limited memory BFGS method for large scale optimization. *Mathematical programming*, 45(1-3):503–528, 1989.

[22] James Robert Lloyd, David Duvenaud, Roger Grosse, Joshua Tenenbaum, and Zoubin Ghahramani. Automatic construction and natural-language description of nonparametric regression models. In *AAAI conference on artificial intelligence*, 2014.

[23] David J.C. MacKay. The evidence framework applied to classification networks. *Neural computation*, 4(5):720–736, 1992.

[24] Jonas Močkus. On Bayesian methods for seeking the extremum. In *Optimization Techniques IFIP Technical Conference*, pages 400–404. Springer, 1975.

[25] Iain Murray and Ryan P. Adams. Slice sampling covariance hyperparameters of latent Gaussian models. In *Advances in Neural Information Processing Systems*, pages 1732–1740, 2010.

[26] Iain Murray and Matthew Graham. Pseudo-marginal slice sampling. In *Artificial Intelligence and Statistics*, pages 911–919, 2016.

[27] Radford M. Neal. Regression and classification using Gaussian process priors. *Bayesian statistics 6*, pages 475–501, 1999.

[28] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. PyTorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems*, pages 8024–8035, 2019.

[29] Joaquin Quiñonero-Candela and Carl Edward Rasmussen. A unifying view of sparse approximate Gaussian process regression. *Journal of Machine Learning Research*, 6: 1939–1959, 2005.

[30] Carl Edward Rasmussen. *Evaluation of Gaussian processes and other methods for non-linear regression*. PhD thesis, University of Toronto Toronto, Canada, 1997.

[31] Carl Edward Rasmussen and Christopher K. I. Williams. *Gaussian processes for machine learning*. Adaptive Computation and Machine Learning. MIT Press, 2006. ISBN 026218253X.

[32] Danilo Jimenez Rezende, Shakir Mohamed, and Daan Wierstra. Stochastic backpropagation and approximate inference in deep generative models. In *International Conference on Machine Learning*, pages 1278–1286, 2014.

[33] Daniel Ritchie, Paul Horsfall, and Noah D Goodman. Deep amortized inference for probabilistic programs. *arXiv preprint arXiv:1610.05735*, 2016.

[34] Matthias W. Seeger, Christopher K. I. Williams, and Neil D. Lawrence. Fast forward selection to speed up sparse Gaussian process regression. In *Artificial Intelligence and Statistics*, 2003.

[35] Bernard W. Silverman. *Density estimation for statistics and data analysis*, volume 26. CRC press, 1986.

[36] Alex J. Smola and Peter L. Bartlett. Sparse greedy Gaussian process regression. In *Advances in Neural Information Processing Systems*, pages 619–625, 2001.

[37] Edward Snelson and Zoubin Ghahramani. Sparse Gaussian processes using pseudo-inputs. In *Advances in Neural Information Processing Systems*, pages 1257–1264, 2006.

[38] Jasper Snoek, Hugo Larochelle, and Ryan P. Adams. Practical Bayesian optimization of machine learning algorithms. In *Advances in Neural Information Processing Systems*, pages 2951–2959, 2012.

[39] Shengyang Sun, Guodong Zhang, Chaoqi Wang, Wenyuan Zeng, Jiaman Li, and Roger Grosse. Differentiable compositional kernel learning for Gaussian processes. In *International Conference on Machine Learning*, pages 4828–4837, 2018.

[40] Michalis Titsias. Variational learning of inducing variables in sparse Gaussian processes. In *Artificial Intelligence and Statistics*, pages 567–574, 2009.

[41] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems*, pages 5998–6008, 2017.

[42] Christopher K.I. Williams and Matthias Seeger. Using the Nyström method to speed up kernel machines. In *Advances in Neural Information Processing Systems*, pages 682–688, 2001.

[43] Andrew G. Wilson and Ryan P. Adams. Gaussian process kernels for pattern discovery and extrapolation. In *International Conference on Machine Learning*, pages 1067–1075, 2013.

[44] Andrew G. Wilson, Elad Gilboa, Arye Nehorai, and John P Cunningham. Fast kernel learning for multidimensional pattern extrapolation. In *Advances in Neural Information Processing Systems*, pages 3626–3634, 2014.

[45] Andrew G. Wilson, Zhiting Hu, Ruslan Salakhutdinov, and Eric P Xing. Deep kernel learning. In *Artificial Intelligence and Statistics*, pages 370–378, 2016.

[46] Manzil Zaheer, Satwik Kottur, Siamak Ravanbakhsh, Barnabas Poczos, Russ R. Salakhutdinov, and Alexander J. Smola. Deep sets. In *Advances in Neural Information Processing Systems*, pages 3391–3401, 2017.

Appendix A. Gaussian Processes

In this section, we establish background concepts and notations necessary for the discussion of the amortized hyperparameters inference approach described in Section 2.

A Gaussian process defines a distribution over functions $f : \mathcal{X} \to \mathbb{R}$, and is specified by its mean function $\mu(\mathbf{x})$ and positive-definite covariance function $k(\mathbf{x}, \mathbf{x}')$, where $\mathbf{x}, \mathbf{x}' \in \mathcal{X}$:

$$f(\mathbf{x}) \sim \mathcal{GP}\left(\mu(\cdot), k\left(\cdot,\cdot\right)\right), \qquad \mu(\mathbf{x}) = \mathbb{E}[f(\mathbf{x})], \qquad k\left(\mathbf{x}, \mathbf{x}'\right) = \operatorname{cov}\left(f(\mathbf{x}), f(\mathbf{x}')\right) \tag{2}$$

For any finite set of points in $\mathcal{X}$, $\mathbf{X} := \{\mathbf{x}_1, \ldots, \mathbf{x}_N\}$, the corresponding function values $\mathbf{f} := (f(\mathbf{x}_1), \cdots, f(\mathbf{x}_N))$ follow a multivariate Gaussian distribution: $\mathbf{f} \sim \mathcal{N}(\boldsymbol{\mu}, \mathbf{K}_{\mathbf{XX}})$, where $\boldsymbol{\mu}_i = \mu(\mathbf{x}_i)$ and $(\mathbf{K}_{\mathbf{XX}})_{ij} = k(\mathbf{x}_i, \mathbf{x}_j)$. For a training dataset $\mathcal{D} = \{(\mathbf{x}_i, y_i)\}_{i=1}^N$, each y_i is commonly assumed to be generated by adding an i.i.d. zero-mean Gaussian noise to $f(\mathbf{x}_i)$, i.e., $y_i = f(\mathbf{x}_i) + \epsilon_i$, where $\epsilon_i \sim \mathcal{N}(0, \sigma_\epsilon^2)$. Denote $\mathbf{y} := [y_1, \cdots, y_N]^\top \in \mathbb{R}^{N\times 1}$. For new data input $\widetilde{\mathbf{X}} := \{\widetilde{\mathbf{x}}_1, \ldots, \widetilde{\mathbf{x}}_{N'}\}$ of size N', the Gaussianity of the prior and likelihoods make it possible to compute the predictive distribution in closed form:

$$\widetilde{\mathbf{f}}|\widetilde{\mathbf{X}}, \mathcal{D} \sim \mathcal{N}\left(\widetilde{\boldsymbol{\mu}}, \mathbf{K}_{\widetilde{\mathbf{f}}}\right), \quad \widetilde{\boldsymbol{\mu}} = \mathbf{K}_{\widetilde{\mathbf{X}}\mathbf{X}}\big(\mathbf{K}_{\mathbf{XX}} + \sigma_\epsilon^2\mathbf{I}\big)^{-1}\mathbf{y}, \quad \mathbf{K}_{\widetilde{\mathbf{f}}} = \mathbf{K}_{\widetilde{\mathbf{X}}\widetilde{\mathbf{X}}} - \mathbf{K}_{\widetilde{\mathbf{X}}\mathbf{X}}\big(\mathbf{K}_{\mathbf{XX}} + \sigma_\epsilon^2\mathbf{I}\big)^{-1}\mathbf{K}_{\mathbf{X}\widetilde{\mathbf{X}}},$$

where $\mathbf{K}_{\mathbf{X}\widetilde{\mathbf{X}}} \in \mathbb{R}^{N\times N'}$ with $\left(\mathbf{K}_{\mathbf{X}\widetilde{\mathbf{X}}}\right)_{ij} = k(\mathbf{x}_i, \widetilde{\mathbf{x}}_j)$.

A.1 Choice of kernel function

The choice of kernel function is crucial to Gaussian process generalization, as different kernel functions impose various model assumptions, e.g., smoothness, periodicity, etc. (See Chapter 4 of Rasmussen and Williams [31] for an extensive discussion.) If the problem has a known structure, one can sometimes choose a kernel to capture it. Otherwise, kernel learning must be performed by defining an expressive space of kernel functions and selecting the best one through optimization [43, 45, 39] or search [7, 22].

A.2 Hyperparameter inference.

Beyond the particular choice of kernel, it is also common for the covariance function to have so-called *hyperparameters* θ that govern its specific structure, and the parameterized kernel function is written as $k_\theta(\cdot,\cdot)$. Although a fully-Bayesian treatment is possible [27, 25, 8, 26], the most common approach to determining hyperparameters is to use empirical Bayes and maximize the log marginal likelihood (evidence) with respect to the hyperparameter θ, i.e., perform type II maximum likelihood [2, 23]. The log MLL for observed data $\{\mathbf{X}, \mathbf{y}\}$ is given by:

$$\log p(\mathbf{y}|\mathbf{X}, \theta) = -\frac{1}{2}\mathbf{y}^\top\left(\mathbf{K}_{\mathbf{XX}}(\theta) + \sigma_\epsilon^2\mathbf{I}\right)^{-1}\mathbf{y} - \frac{1}{2}\log\left|\mathbf{K}_{\mathbf{XX}}(\theta) + \sigma_\epsilon^2\mathbf{I}\right| - \frac{N}{2}\log 2\pi\,, \tag{3}$$

where we write $\mathbf{K_{XX}}(\theta)$ to indicate the dependence of the Gram matrix on the hyperparameters.

To solve the above optimization problem, quasi-Newton methods such as L-BFGS [21] or nonlinear conjugate gradient [16, 9] are usually used. These iterative optimization methods involve taking the gradient of the objective several times for each optimization step. As the gradient of Eqn 3 scales as $\mathcal{O}(N^3)$, this optimization becomes prohibitively expensive on large-scale problems, dominating the computational cost of using GP. Moreover, the non-concavity of the objective in Eqn 3 makes it difficult to ensure convergence to a good maximum.

To address the scaling issue, a low-rank approximation to the kernel matrix is often used either by subsampling the data or via virtual "inducing" points [36, 42, 5, 29, 34, 37, 40, 14]. These methods require inversion of a smaller matrix and reduce the computational complexity to $\mathcal{O}(NM^2)$ (M is the number of subsampled data or "inducing" points), at the cost of a larger and often more challenging optimization problem alongside the potential loss of important information from the dataset.

Appendix B. Spectral Density of Kernel Functions

We start by introducing Bochner's Theorem stated below.

Theorem 2 *(Bochner [3]) A complex-valued function k on $\mathbb{R}^d$ is the covariance function of a weakly stationary mean square continuous complex valued random process on $\mathbb{R}^d$ if and only if it can be represented as*

$$k(\tau) = \int_{\mathbb{R}^d} e^{2\pi i \omega^\top \tau} d\mu(\omega), \tag{4}$$

where μ is a positive finite measure, often known as the spectral measure of the random process.

When μ is absolutely continuous with respect to the Lebesgue measure, its Radon–Nikodym derivative (density) $S(\omega)$ is called the *spectral density* of the random process. The kernel function $k(\cdot)$ and the spectral density $S(\cdot)$ are *Fourier transform* pairs. In other words, Bochner's Theorem establishes the correspondence between any stationary kernel and its spectral density.

By resorting to the duality of spectral density and kernel function given by Bochner's Theorem, for each dimension d, if we assume its spectral density is described by a M mixtures of Gaussians with means $\{\mu_{(d),m}\}_{m=1}^M$ and variances $\{\sigma^2_{(d),m}\}_{m=1}^M$, we have its kernel function in closed form [43, 44] given by,

$$k_{\mathrm{SMP}_{\theta_{(d)}}}(\tau_{(d)}) = \sum_{m=1}^{M} w_{(d),m} \exp\left\{-2\pi^2 \tau^2_{(d)} \sigma^2_{(d),m}\right\} \cos\left(2\pi \tau_d \mu_{(d),m}\right). \tag{5}$$

where $\boldsymbol{\tau}_{(d)}$ is the d-th component of $\boldsymbol{\tau} = \mathbf{x} - \mathbf{x}' \in \mathbb{R}^D$, i.e. the difference of two data points. Taking the product of kernels for different dimensions, the spectral mixture product (SMP) kernel [44] is given by:

$$k_{\mathrm{SMP}_\theta}(\boldsymbol{\tau}) = \prod_{d=1}^{D} k_{\mathrm{SMP}_{\theta_{(d)}}}\left(\boldsymbol{\tau}_{(d)}\right), \tag{6}$$

Appendix C. Proof of Proposition 1

We start by defining permutation equivariant and permutation invariant functions.

Definition 3 *Let S_n be the set of all permutations of indices $\{1, 2, \cdots, n\}$. A function $f : \mathcal{X}^n \to \mathcal{Y}^n$ is **permutation equivariant** if and only if for any permutation $\pi \in S_n$, $f(\pi x) = \pi f(x)$.*

Definition 4 *Let S_n be the set of all permutations of indices $\{1, 2, \cdots, n\}$. A function $f : \mathcal{X}^n \to \mathcal{Y}$ is **permutation invariant** if and only if for any permutation $\pi \in S_n$, $f(\pi x) = f(x)$.*

Next, we start the proof.

Proof We assume our neural network takes in the l-th dataset $\mathcal{D}^{(l)}$ as input. By definition of the multi-head self-attention mechanism, it is obvious that a single multi-head self-attention sublock is permutation equivariant. From [46] we know stacks of permutation equivariant layers are still permutation equivariant. Therefore, for every dimension, the input/output data $\{(\mathbf{x}_{i,d}^{(l)}, y_i^{(l)})\}_{i=1}^{N_l}$ are always mapped to the same corresponding $\{\mathbf{h}_{i,d}^{(l)}\}_{i=1}^{N_l}$ regardless of their order. Further if AggregateFunction is permutation invariant, we have $\mathbf{h}_{\text{local},d}^{(l)}$ invariant of the order of data points for each dimension.

As the LocalTransformer is sharing the same weights for all dimensions,$\{\mathbf{h}_{\text{local},d}^{(l)}\}_{d=1}^{D_l}$ will remain equivariant with regards to permutation of dimensions. Next $\{\mathbf{h}_{\text{local},d}^{(l)}\}_{d=1}^{D_l}$ are passed through stacks of multi-head self-attention subblocks which is permutation equivariant, therefore the final $\{\mathbf{h}_{\text{global},d}^{(l)}\}_{d=1}^{D_l}$ are permutation equivariant. Again the weight sharing of the final MLP ensures that the final predicted spectral density hyperparameters $\{\theta_d^{(l)}\}_{d=1}^{D_l}$ are permutation equivariant with regards to data dimensions. They are also permutation invariant with regards to data points, as $\{\mathbf{h}_{\text{local},d}^{(l)}\}_{d=1}^{D_l}$ are invariant of the order of input/output data. ■

Appendix D. Complexity analysis

For each multi-head self-attention layer, the computational complexity is $\mathcal{O}(h \cdot n^2)$, where h is the representation dimension and n is the size of the input set. Assuming that LocalTransformer has l_1 layers with representation dimension h_1 and GlobalTransformer has l_2 layers with representation dimension h_2, the complexity of our model is $\mathcal{O}(l_1 \cdot h_1 \cdot N^2 + l_2 \cdot h_2 \cdot D^2)$, where N is the number of data points and D is the dimensionality. In comparison, exact marginal likelihood optimization scales as $\mathcal{O}(r \cdot N^3)$, where r denotes the number of gradient evaluations during optimization. It is possible to further reduce the complexity of our model to $\mathcal{O}(l_1 \cdot h_1 \cdot m \cdot N + l_2 \cdot h_2 \cdot m \cdot D)$ if we restrict the number of attentions to m by either introducing sparse attentions [4] or inducing points [20], which we leave for future work.

Appendix E. Experimental Details and Results

E.1 Synthetic Training Dataset Generation

In our experiments, our training dataset is constructed by sampling multiple sets of input/output data from synthetically generated GPs with stationary kernel functions. For each GP, its data dimensionality is sampled uniform randomly from 2∼15. To sample flexible kernel functions, we randomly generate mixtures of Gaussians to represent its spectral density. The weights of the Gaussian mixtures are drawn from Dirichlet distribution and the lengthscales (i.e., $1/\sqrt{2}\pi\sigma_{(d),m}$'s) are sampled from a log-uniform distribution. The number of mixtures is set to 10 and the concentration parameter of the Dirichlet distribution are set to 1. The input of the data points $\{\mathbf{x}_i^{(l)}\}_{i=1}^{N_l}$ are generated from a Poisson point process within the hypercube $[-1,1]^{D_l}$ with average density equals to 30. The data output values $\{y_i^{(l)}\}_{i=1}^{N_l}$ are generated from priors of the GP with its specified kernel function. The observation noise is i.i.d. randomly sampled from $\mathcal{N}(0, 0.01)$. We generate 10000 sets of input/output data to be used as the whole dataset, of which we do a split with 50% used for training and the rest for validation.

E.2 Training Details

Our model is trained with PyTorch using the Adam [17] optimizer with a fixed learning rate and a batch size of 64. A description of our model architecture is provided in Table 4. We also apply 0.1 dropout to self-attention and the MLPs. The number of Gaussian mixtures in the spectral density prediction is fixed at 10. To validate the effectiveness of our neural network model, we minimize the efforts of hyperparameter tuning during training. The only hyperparameters we tuned are learning rate ($\{10^{-3}, 10^{-4}, 10^{-5}, 10^{-6}\}$) and number of layers in LocalTransformer and GlobalTransformer (2∼8). The hyperparameters are tuned based on performance on the validation set.

Embed each $(\mathbf{x}_{i,d}^{(l)}, y_i^{(l)})$ to 256 dim
Transformer sublayer (4 heads, representation dim 256, feedforward dim 512)
Transformer sublayer (4 heads, representation dim 256, feedforward dim 512)
Transformer sublayer (4 heads, representation dim 256, feedforward dim 512)
Transformer sublayer (4 heads, representation dim 256, feedforward dim 512)
Transformer sublayer (4 heads, representation dim 256, feedforward dim 512)
Transformer sublayer (4 heads, representation dim 256, feedforward dim 512)
Transformer sublayer (4 heads, representation dim 256, feedforward dim 512)
Transformer sublayer (4 heads, representation dim 256, feedforward dim 512)
AggregateFunction: Average pooling

(a) LocalTransformer architecture and AggregateFunction

Transformer sublayer (4 heads, representation dim 256, feedforward dim 512)
Transformer sublayer (4 heads, representation dim 256, feedforward dim 512)
Transformer sublayer (4 heads, representation dim 256, feedforward dim 512)
Transformer sublayer (4 heads, representation dim 256, feedforward dim 512)
Transformer sublayer (4 heads, representation dim 256, feedforward dim 512)
Transformer sublayer (4 heads, representation dim 256, feedforward dim 512)
Transformer sublayer (4 heads, representation dim 256, feedforward dim 512)
Transformer sublayer (4 heads, representation dim 256, feedforward dim 512)
MLP for predicting weights, means and variances (each with hidden dim [256, 128])

(b) GlobalTransformer architecture and the final MLP

Figure 4: Details of the neural network architecture.

Since spectral mixture kernel assume the variance of the kernel function is normalized, we standardize the function values $\{y_i^{(l)}\}_{i=1}^{N_l}$. We also standardize the data input $\{\mathbf{x}_i^{(l)}\}_{i=1}^{N_l}$ as a standard procedure. During training, we find the validation performance is most sensitive to learning rate and increasing layers of the Transformers slightly helps. The final model is trained for 200 epochs with batch size 64, learning rate 10^{-5} and 8 layers in both Local and Global Transformer. The training is done on an NVIDIA GTX 1080 Ti GPU. All the

evaluation experiments are run using one core of an Intel(R) Core(TM) i7-6850K CPU for CPU runtime comparisons and an NVIDIA GTX 1080 Ti GPU for GPU runtime comparisons.

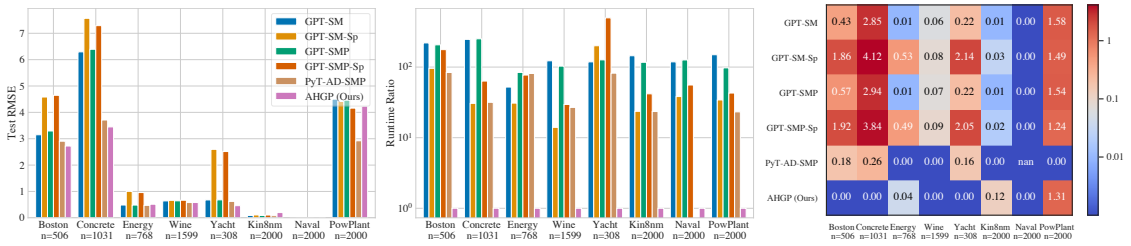
E.3 Regression Benchmarks

For SGPR methods, the number of inducing points is set to 10% of the number of training data. The comparisons with the CPU-based baselines in terms of test log marginal likelihood are presented in Table 1.

Dataset	GPy-SM	GPy-SM-Sp	GPy-SMP	GPy-SMP-Sp	**AHGP (Ours)**
Boston	-2.649±0.364	-3.527±1.086	-2.538±0.281	-2.939±0.642	**-2.367±0.115**
Concrete	**-2.656±0.637**	-3.435±0.238	-2.690±0.574	-3.757±0.579	-3.460±1.334
Energy	-1.059±0.019	-1.103±0.046	-1.079±0.028	-2.534±0.557	**-0.837±0.215**
Wine	-0.427±0.058	-4.331±1.328	-0.410±0.053	-1.053±0.051	**-0.321±0.075**
Yacht	-1.573±0.254	-1.526±0.108	-1.500±0.077	-1.979±0.742	**-0.997±0.092**
Kin8nm	**1.119±0.044**	0.655±0.177	1.020±0.054	0.748±0.198	0.192±0.039
Naval	6.157±0.267	5.433±1.063	**6.211±0.011**	3.643±0.164	5.393±0.102
PowPlant	-2.591±0.115	-3.831±0.330	**-2.475±0.060**	-2.936±0.047	-3.112±0.151

Table 1: Test log-likelihood on regression benchmarks: CPU-based methods

The comparisons of AHGP with the GPU-based baselines are presented in Fig 5 and Table 2. Results are similar to comparisons with the CPU-based baselines in the main section. Note that GPyTorch baselines perform slightly worse than GPy baselines, since GPyTorch uses conjugate gradient method to approximately solve for matrix inverse instead of doing the Cholesky decomposition. In comparison, PyT-AD-SMP uses Cholesky decomposition and generally achieves better performance than GPT-SMP.



(a) Test RMSE **(b)** GPU runtime ratio over AHGP **(c)** Gap with best RMSE

Figure 5: Comparison of AHGP against the GPU-based baselines on regression benchmarks. In (c), the numbers are the differences of the corresponding method's test RMSE with the best RMSE on the respective dataset. Note that for Naval, the RMSEs are all very close to 0 except PyT-AD-SMP which runs out of GPU memory.

E.4 Bayesian Optimization

We pick the best performing baselines on the regression benchmarks (GPy-SM, GPy-SM-Sp, PyT-AD-SMP) and compare them with our method. The number of inducing points in GPy-SM-Sp is set to 20. Standard test functions for global optimization [6] are used as the target functions for Bayesian optimization. Five initial input points are randomly sampled and function values at the points are evaluated; those input points with their function values serve as the initial input/output data of GP. At the beginning of each BO iteration,

Dataset	GPT-SM	GPT-SM-Sp	PyT-AD-SMP	GPT-SMP	GPT-SMP-Sp	**AHGP (Ours)**
Boston	-2.692±0.442	-12.440±5.930	-2.757±0.416	-2.687±0.439	-11.390±5.270	**-2.367±0.115**
Concrete	**-3.030±0.640**	-3.347±0.107	-3.132±1.262	-3.095±0.720	-3.285±0.078	-3.460±1.334
Energy	-1.073±0.020	-1.269±0.064	-1.496±0.400	-1.068±0.023	-1.266±0.067	**-0.837±0.215**
Wine	-0.517±0.079	-1.067±0.129	**-0.306±0.064**	-0.527±0.076	-1.028±0.115	-0.321±0.075
Yacht	-1.492±0.105	-2.416±0.634	-1.030±0.770	-1.492±0.106	-2.477±1.071	**-0.997±0.092**
Kin8nm	0.921±0.086	0.819±0.042	**1.108±0.047**	0.917±0.090	0.854±0.045	0.192±0.039
Naval	5.892±0.028	5.253±0.215	Out of memory	**5.933±0.017**	5.372±0.113	5.393±0.102
PowPlant	-2.923±0.120	-2.903±0.135	**-2.540±0.153**	-2.920±0.150	-2.834±0.078	-3.112±0.151

Table 2: Test log-likelihood on regression benchmarks: GPU-based methods

hyperparameters are randomly reinitialized for all the baselines and then optimized through MLL optimization. The full results are shown in Fig. 6, 7a.

For comparison, we also implement another warmstart initialization strategy which sets the initialization as the best hyperparameters from the previous BO iteration. The full results are shown in Fig. 8, 7b.

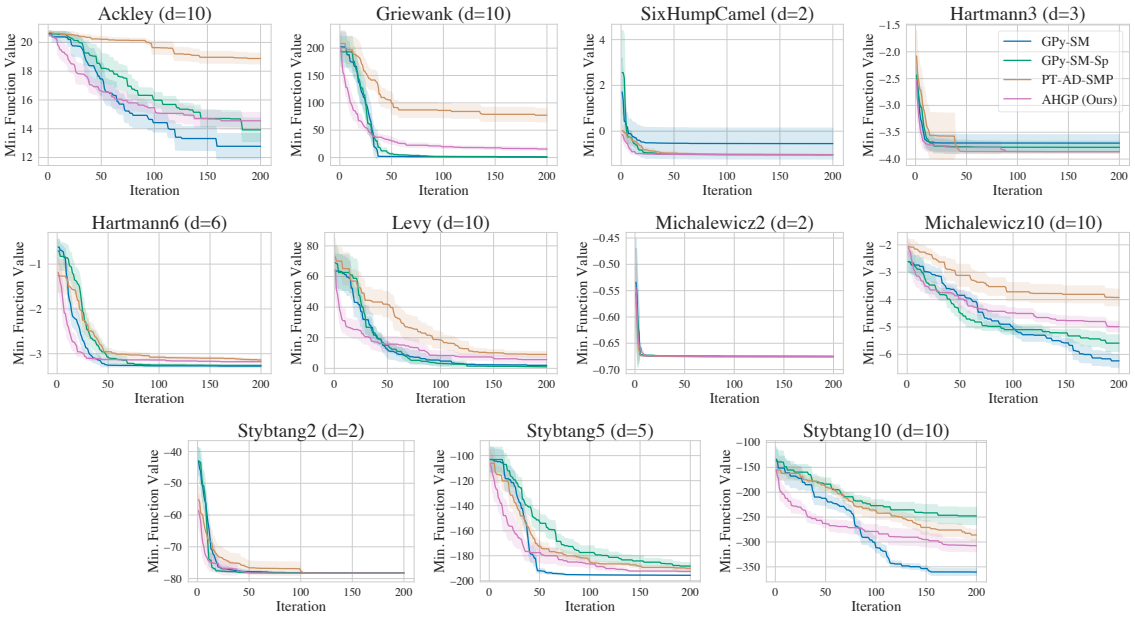
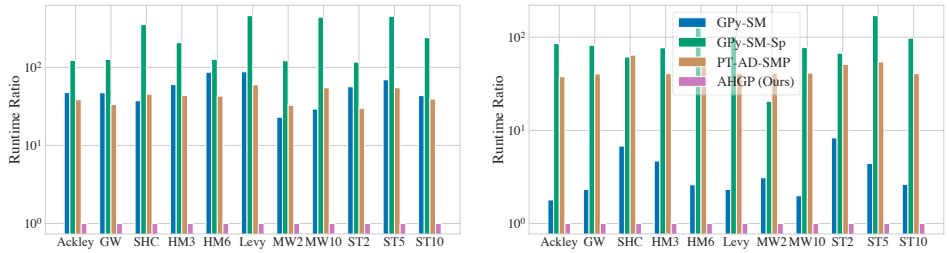


Figure 6: Bayesian optimization performance comparison: random initialization strategy. Shaded region indicates 0.5 standard deviations over 10 runs.

To evaluate our method on real-world Bayesian optimization problems, we additionally applied our method to tuning learning and model hyperparameters of logistic regression via BO. The goal is to find the training hyperparameters that achieves the highest test accuracy when a fixed amount of time is used for training. We experiment on the task of training logistic regression on MNIST data with stochastic gradient descent. The training involves four hyperparameters: learning rate, ℓ_2 regularization parameter, ℓ_1 regularization parameter and mini-batch size. We present the results in Fig 9. AHGP achieves comparable test accuracy with the strongest baselines (GPy-SM, PyT-AD-SMP) while being much faster.



(a) Random initialization strategy. **(b)** Warmstart initialization strategy.

Figure 7: Runtime on Bayesian optimization tasks.

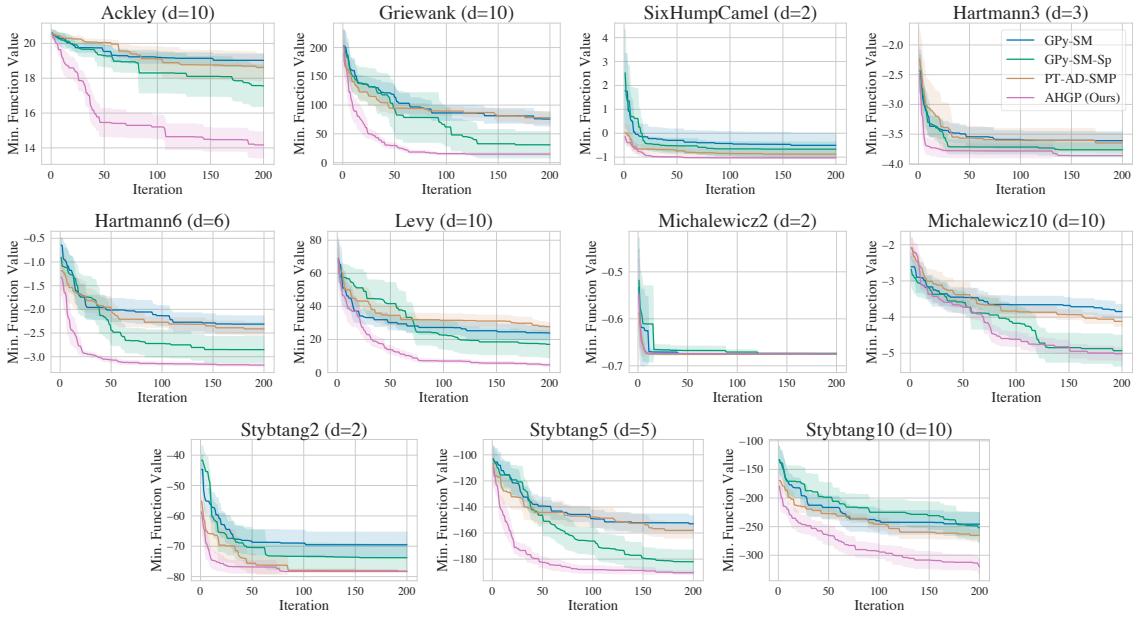
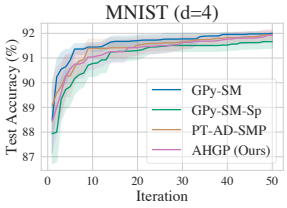


Figure 8: Bayesian optimization performance comparison: warmstart initialization strategy. Shaded region indicates 0.5 standard deviations over 10 runs.

The SGPR baseline (GPy-SM-Sp), however, has a lower test accuracy and much longer runtime.

Method	Runtime(s)
GPy-SM	195.90 ± 99.81
GPy-SM-Sp	1067.70±41.40
PyT-AD-SMP	23.85±0.45
AHGP(Ours)	**1.23±0.06**

(a) Runtime



(b) Test accuracy performance

Figure 9: Performance and runtime of BO for training logistic regression on MNIST.