Towards Scalable Gaussian Processes

Mohd Abbas Zaidi (150415), Aarsh Prakash Agarwal (150004)
Project Report
Prof. Ketan Rajawat
aarshp@iitk.ac.in, mzaidi@iitk.ac.in

Abstract

The project started with studying the first 10 lectures of *Convex Optimization by Stephen Boyd, Stanford.* It was followed by reading Gaussian Processes from lecture slides by Piyush Rai, and from the the books by Williams (1) and by Trevor Hastie (2). The task was aimed at comparing a new Parsimonious Online Gaussian Process technique with the existing algorithm. We focused primarily on Sparse Online Gaussian Process technique. We ran multiple experiments during the project to compare the two processes. In the next sections, we will explain the details of the POG and SOGP method and then the experiments run on them.

1 The problem with 'Online' Gaussian Processes

It has long been known that a single-layer fully-connected neural network with an i.i.d. prior over its parameters is equivalent to a Gaussian process (GP), in the limit of infinite network width (3). However, neural networks are much more popular as compared to Gaussian Processes. The currently used algorithms for training neural nets are all based on back-propagation approaches. Under these techniques, at each instant, a point is fed to the network and the errors are back-propagated to update the weights. This allows for a very convenient property, called 'Online' training. For example for a spam detection system, When a user marks one mail as spam, the new data fetched from this user can be easily used to tune the existing model.

However, Gaussian Processes have a strong mathematical base and have so far relied on exact updates, which are one-shot, i.e. they go through the whole of the existing data set at each instant. This makes Gaussian Processes less suitable for Online settings.

2 Parsimonious Online Gaussian Processes

The POG method tries to induce sparsity based on uniqueness or information derived from each point. It tries to reduce the complexity while maintaining the posterior consistency. The full updates in a time series format can be written as

$$\boldsymbol{\mu}_{t+1|S_t} = \boldsymbol{k}_{S_t}(\boldsymbol{x}_{t+1})[\boldsymbol{K}_t + \sigma^2\boldsymbol{I}]^{-1}\boldsymbol{y}_t$$
$$\boldsymbol{\Sigma}_{t+1|S_t} = \kappa(\boldsymbol{x}_{t+1},\boldsymbol{x}_{t+1}) - \kappa(\boldsymbol{x}_{t+1},\boldsymbol{x}_{t+1})\boldsymbol{k}_{S_t}(\boldsymbol{x}_{t+1})[\boldsymbol{K}_t + \sigma^2\boldsymbol{I}]^{-1}\boldsymbol{k}_{S_t} + \sigma^2$$

It is clear that for normal updates the size of dataset increases by 1 at each time instant, and the posterior updates at time $t+1$ use all past observations from the kernel dictionary. The inverse operation means a complexity of atleast $O(N^3)$

Under the POG method, the whole idea is to keep a subset which is representative of the whole dataset. There is obviously a trade-off between the errors in learning the unknown function and the size of this subset. POG uses Hellinger metric to throw away the least relevant point from the dictionary(also called the basis vector set). Hellinger distance is a metric to find the difference or deviation of two multivariate continuous distributions from one another. It reduces to an easily computable form in the case of multivariate Gaussian distribution. When a new point comes in, the posterior is computed by adding it to the current dictionary. The DHMP algorithm, then performs the compression of the posterior.

Under DHMP compression, each point of the dictionary(existing + new point) is removed one by one. The effect of removal is found using Hellinger distance. The least unique point is the one whose removal has the least effect on the posterior or whose Hellinger metric comes out to be the least. This point is removed from the dictionary if its Hellinger metric is below a certain threshold called the compression threshold ϵ_t. This process is repeated sequentially until the minimum Hellinger metric exceeds the threshold.

The stopping criterion of the DHMP algorithm can be suitably varied to ensure that the distribution properties of the updates remain consistent. The removal threshold will finally dictate the number of points we have in our basis vector set or dictionary.

3 Sparse Online Gaussian Processes(4)

As the name suggests, the algorithm aims to reduce the time complexity of Gaussian Process in an online setting by approximating it with a sparse function. It derives its major Gaussian updates from (5) which approximates the posterior using KL divergence metric. The function likelihood and kernel are respectively given by:

$$< f_x >_t = \sum_{i=1}^{t} K_o(x, x_i)\alpha_t(i) = \boldsymbol{\alpha}_t^T \boldsymbol{k_x}$$

$$K_t(x, x') = K_o(x, x') + \boldsymbol{k_x}^T \boldsymbol{C_t} \boldsymbol{k_{x'}}$$

The final updates are given by:

$$\boldsymbol{\alpha_{t+1}} = \boldsymbol{T_{t+1}}(\boldsymbol{\alpha_t}) + q^{(t+1)} \boldsymbol{s_{t+1}}$$

$$\boldsymbol{C_{t+1}} = \boldsymbol{U_{t+1}}(\boldsymbol{C_{t+1}}) + r^{(t+1)} \boldsymbol{s_{t+1}} \boldsymbol{s_{t+1}}^T$$

$$\boldsymbol{s_{t+1}} = \boldsymbol{T_{t+1}}(\boldsymbol{C_t} \boldsymbol{k_{t+1}}) + \boldsymbol{e_{t+1}}$$

The terms $\boldsymbol{T_{t+1}}$ and $\boldsymbol{U_{t+1}}$ represent the increase in the dimension of vector and matrix by adding a zero element(for vector), and row column(for matrix). These updates are called *full updates* as they result in increasing the size of the Gaussian defining co-variance matrix. Performing these updates over a certain point essentially translates to major contribution from that point, and that point is considered part of dictionary called *basis vectors*. The major takeaway from these steps is that unlike *standard Gaussian* updates (which were used in POG), these updates at no point require inversion of the co-variance matrix, which makes these steps cheaper in time complexity than *full updates* of POG regression and, they can be performed in an *additive* fashion.

The kernels can be approximated using sparse representation. The error between the actual kernel and approximated sparse kernel is calculated and minimized and the sparse vector is found to be a function of existing Gram kernel matrix at any given time. With the known sparse vector *cheap update* is performed instead of full update as follows:

$$\boldsymbol{\hat{e}_t} = \boldsymbol{K_t}^{-1} \boldsymbol{k_{t+1}}$$

$$\boldsymbol{s_{t+1}} = \boldsymbol{C_t} \boldsymbol{k_{t+1}} + \boldsymbol{\hat{e}_t}$$

As we can see, update does not involve the expansion of basis vector, or the covariance matrix and is therefore computationally cheap. The error calculation involving the calculation of $\boldsymbol{K_t}^{-1}$ can also be made computationally cheap by clever book-keeping. This is because of fact that the sparse vectors are orthogonal and Gaussian updates are additive.

In order to decide whether the current point is to be included in the basis vector set, we calculate the error in the likelihood function (whose absolute value turns out to be the minimum error we calculated earlier multiplied by a constant) and if the error is above a threshold, we include that point in the *basis vector set*, otherwise the *cheap updates* are performed for that point.

The algorithm also has a provision of removing a point from the basis vector. This is performed when maximum size limit of basis vector is reached. It basically involves the addition of a point in the basis vector followed by the error score calculation (error in the likelihood function) for each point in the basis vector. Finally, the point with the minimum error contribution is deleted and basis vector is re-arranged.

4 Experimental Setting

For comparing the algorithms we are learning the sinc function. We have 160 training data points which are sampled from sinc function with added white Gaussian noise (with variance = 1). For testing we have 20 data points sampled from sinc function.

5 Analysis: Comparing the two methods

Before running the experiments, one needs to be sure to equalize any variable which may later confound the findings to one side. Other than keeping the data-set size, data-set values, noise, Kernel type and Model order same for both methods, the following points were observed during the course of analysis.

5.1 Bias towards picking outliers

Initially, as from the Figure 1 and Figure 2, we can see that SOGP outperformed the POG, even though model order of POG(51) was significantly greater than that of SOGP(17). We tried to figure out the reasons why SOGP had a better performance over POG. We realised that while removing points from dictionary POG will keep the points with higher Hellinger metric and remove the one with lower value. This would mean that any outlier point which arises due to random noise is less likely to be removed since they will change the learnt Gaussian significantly.

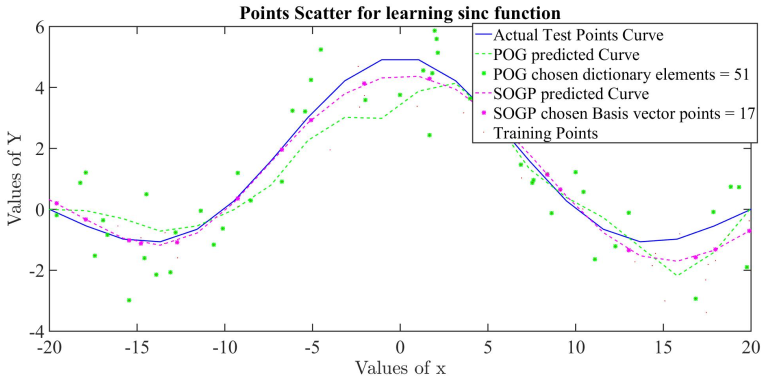

Figure 1: First Comparison of SOGP and POG

However, any sparseness inducing method based on uniqueness of points will suffer from this bias. Why is the effect on SOGP lesser than that on POG? We will discuss that in the next point.

5.2 SOGP: Additivity and Soft Removal

Another advantage for SOGP seemed to be the fact that they did not throw away points completely. Rather, they had cheap updates which they used to update their weight vectors even if they did not include the points in the dictionary.

Taking into consideration the last point, we discussed that the POG method is more biased towards outliers. If a new point comes in which does not cause any change in the distribution, it should ideally increase our confidence in the existing distribution. However, in the case of removal of the point, it is same as if we never saw the point. We end up removing those points from the dataset which are affected less by noise addition. In the case of SOGP, these points are not completely thrown away since they still contribute.

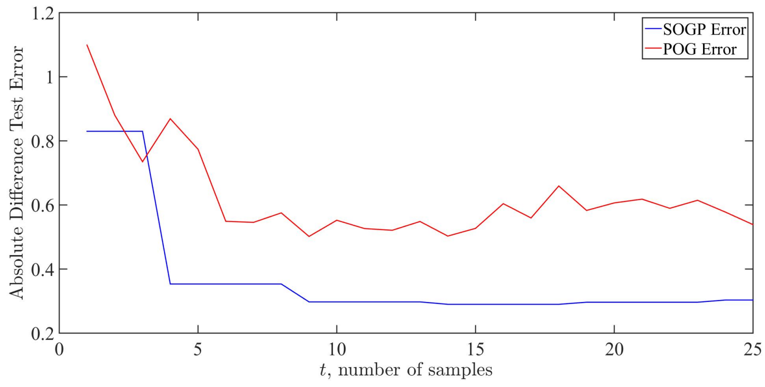

Figure 2: SOGP seemed to outperforme POG at first

These cheap updates could not be replicated in the case of POG since the learning in POG was one shot after having learnt the dictionary elements, and depends solely on basis vector set.

5.3 Without the Cheap Updates

We next tried to compare the POG with SOGP after having removed the cheap update conditions from SOGP. As evident from Figure 3 and Figure 4 SOGP still seemed to perform better. Quite unexpected results were obtained when we saw that SOGP outperforms even the full model learnt by POG. We tried to debug each step of both the algorithms to find out which steps were missing from POG. We next discuss our findings after the debugging process.

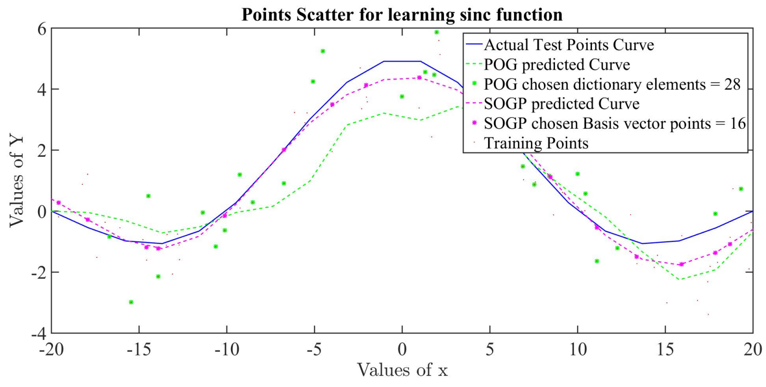

Figure 3: POG and SOGP comparison after removing *cheap updates*

5.4 Epochs in an Online Setting

As mentioned earlier, in order to keep the solution tractable the SOGP algorithm tries to fit the posterior by an approximate projected Gaussian. Due to this reason, they can reduce the subsequent update step into seemingly additive steps, where the array or vector sizes are incremented when a new element comes in. Due to this reason, it is clear that the SOGP method enjoys the liberty of having a pre-trained model to start with over which they can perform their subsequent updates. SOGP gave superior performance due to multiple data sweeps. This however should not be allowed in an Online setting for two reasons. Firstly, you encounter each point only once. Secondly, we do not have access to all the data points after the first run, there is no concept of having an epoch in an online setting since the data points continuously flow in.

Therefore, a faithful comparison of the two methods should run both of them for a single data sweep.

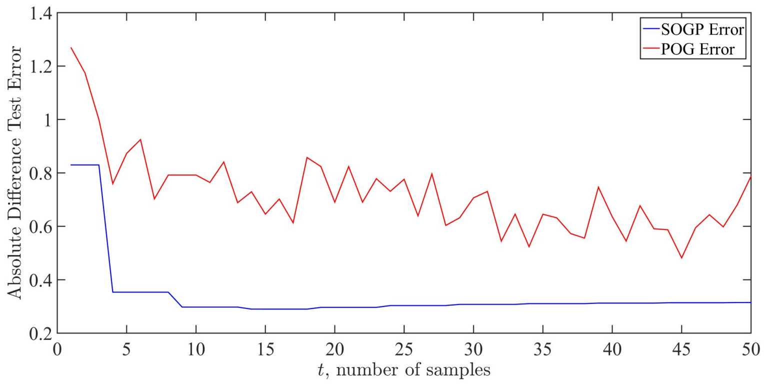

Figure 4: SOGP performed better even after removing cheap updates

5.5 Hyper-Parameter Optimization

After deciding the control parameters, both the algorithms were run side by side for randomly chosen hyper-parameters. As expected the POG algorithm gave much better results (Figure 5). Note that in this comparison, only the epoch was set to 1 and *cheap update* step was restored in SOGP. The metric for comparison was chosen as the test error at same Model Order.

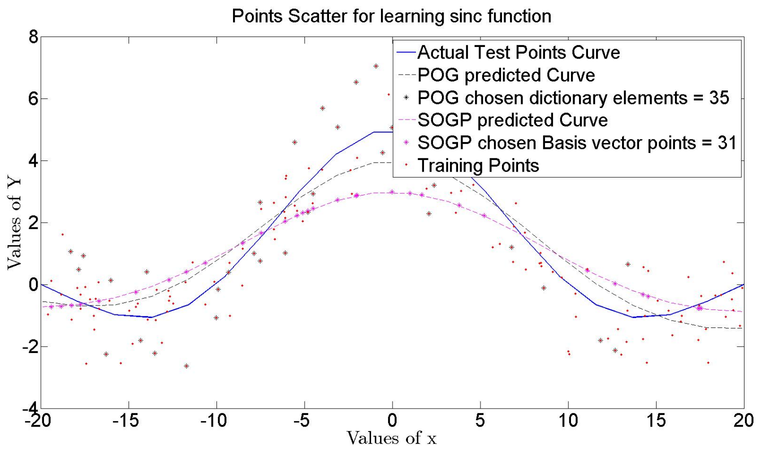

Figure 5: POG and SOGP comparison for one epoch

However, after running the algorithm, SOGP optimized the kernel parameters (hyper parameters) by maximizing the likelihood of the basis vector set via Stochastic Gradient descent algorithm. This tremendously improved their performance and they outperformed POG at lower Model Orders. We tried to run POG on the optimized kernel parameters obtained from SOGP. The performance of POG improved from the initial POG, but it did not beat the SOGP at lower Model Orders.

However, at higher model orders(dictionary/basis vector size) POG did much better as compared to SOGP.

The next logical step would be to write a separate optimizer for POG discussed in the next point.
Details of the experiment are summarized in table below:
HO - Hyper-parameter Optimization(of Kernel parameters)

Experiments	Test-Error SOGP	Test Error POG
No HO in both SOGP and POG	0.8424	0.6216
HO only in SOGP	0.2250	0.6216
HO in both SOGP and POG(Lower Model Order)	0.2250	0.39
HO in both SOGP and POG(Higher Model Order)	0.2231	0.1559

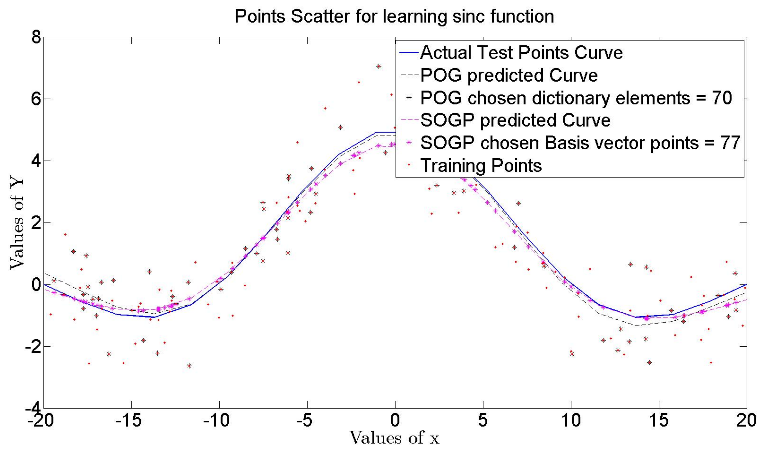

Figure 6: POG and SOGP comparison on optimized parameters

5.5.1 Optimizing the Kernel Parameters

The kernel optimization techniques try to maximize the confidence(likelihood) of the basis vector set or the dictionary elements which is selected. For this, we need to have the expressions for log-likelihood of the data-set and its derivatives with respect to the parameters for POG. As of now, hyper parameter optimization seems to worsen the performance of POG instead of improving it. We plan to work in this direction in future.

5.6 When and Why does POG outperform SOGP?

We observed that SOGP outperforms POG at lower model. However, the rate of decay of error with Model Order is faster in the case of POG. Hence, POG outpeforms SOGP at sligthly higher Model Orders. Possible explanations for this can be:

5.6.1 Full vs Cheap Updates for SOGP

Since SOGP also learns from the points not included in the dictionary, therefore choosing the right dictionary elements(even though they maybe less) helps to learn the distribution satisfactorily as a large number of points(which are not included) also contribute towards the finally learnt parameters.

As the number of points in dictionary increases the number of full updates increases. However, this is countered by a reduction in number of cheap updates. As a net result the performance may not drastically improve. Therefore, a small number of points if chosen correctly perform exceptionally well and we don't see much improvement upon increasing the number of points.

5.6.2 Proved improvement in case of POG

Theorem 1 in POG paper formulates the tradeoff between the compression budget ϵ_t and the accuracy. It is therefore proven that as ϵ_t reduces the performance in case of POG will definitely improve.

6 Future Work

6.1 Optimizing the Hyper-parameters for POG using other methods

We would want to use other methods to optimize the Kernel parameters using the other possible methods. POG may outperform SOGP even at lower model orders if hyperparameters are optimised correctly.

6.2 Comparing POG with other papers in the field

After SOGP we will try to compare the POG algorithms with other papers in the field. We hope to get better results over there since the POG method has proven posterior convergence(theoretically).

6.3 Hyperparamter Optimization with each new Dictionary Element

We may sync the process of addition of elements to dictionary and updation of hyperparameter. This will maximise the likelihood at each step and may improve the performance even further.

Acknowledgments

We are thankful to Prof. Ketan Rajawat and PhD student Amrit Singh Bedi for the opportunities given and the constant help provided throughout the semester.

References

[1] C. E. Rasmussen and C. K. I. Williams, *Gaussian Processes for Machine Learning (Adaptive Computation and Machine Learning)*. The MIT Press, 2005.

[2] G. James, D. Witten, T. Hastie, and R. Tibshirani, *An Introduction to Statistical Learning: With Applications in R*. Springer Publishing Company, Incorporated, 2014.

[3] J. Lee, Y. Bahri, R. Novak, S. S. Schoenholz, J. Pennington, and J. Sohl-Dickstein, "Deep neural networks as gaussian processes," *arXiv preprint arXiv:1711.00165*, 2017.

[4] L. Csató and M. Opper, "Sparse on-line gaussian processes," *Neural computation*, vol. 14, no. 3, pp. 641–668, 2002.

[5] C. K. Williams, "Prediction with gaussian processes: From linear regression to linear prediction and beyond," in *Learning in graphical models*, pp. 599–621, Springer, 1998.