Batched Large-scale Bayesian Optimization in High-dimensional Spaces (Appendix)

Zi Wang
MIT CSAIL

Clement Gehring
MIT CSAIL

Pushmeet Kohli
DeepMind

Stefanie Jegelka
MIT CSAIL

A An Illustration of EBO

We give an illustration of the proposed EBO algorithm on a 2D function shown in Fig. 1. This function is a sample from a 2D TileGP, where the decomposition parameter is $z = [0, 1]$, the cut parameter is (inverse bandwidth) $k = [10, 10]$, and the noise parameter is $\sigma = 0.01$.

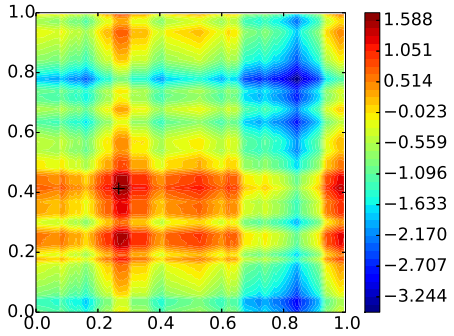

Figure 1: The 2D additive function we optimized in Fig. 2. The global maximum is marked with "+".

The global maximum of this function is at $(0.27, 0.41)$. In this example, EBO is configured to have at least 20 data points on each partition, at most 50 Mondrian partitions, and 100 layers of tiles to approximate the Laplace kernel. We run EBO for 10 iterations with 20 queries each batch. The results are shown in Fig. 2. In the first iteration, EBO has no information about the function; hence it spreads the 10 queries (blue dots) "evenly" in the input domain to collect information. In the 2nd iteration, based on the evaluations on the selected points (yellow dots), EBO chooses to query batch points (blue dots) that have high acquisition values, which appear to be around the global optimum and some other high valued regions. As the number of evaluations exceeds 20, the minimum number of data points on each partition, EBO partitions the input space with a Mondrian process in the following iterations. Notice that each iteration draws a different partition (shown as the black lines) from the Mondrian process so that the results will not "over-fit" to one partition setting and the computation can remain efficient. In each partition, EBO runs the Gibbs sampling inference algorithm to fit a local TileGP and uses batched BO select a few candidates. Then EBO uses a filter to decide the final batch of candidate queries (blue dots) among all the recommended ones from each partition as described in Sec. C.

B Partitioning the input space via a Mondrian process

Alg. 1 shows the full 'Mondrian partitioning" algorithm, i.e., the input space partitioning strategy mentioned in Section 3.1.

Algorithm 1 Mondrian Partitioning

function MONDRIANPARTITIONING (V, N_p, S)
  while $|V| < N_p$ **do**
    $p_j \leftarrow length(v_j) \cdot max(0, |\mathcal{D}^j| - S), \forall v_j \in V$
    if $p_j = 0, \forall j$ **then**
      break
    end if
    Sample $v_j \sim \frac{p_j}{\sum_j p_j}, v_j \in V$
    Sample a dimension $d \sim \frac{h_d^j - l_d^j}{\sum_d h_d^j - l_d^j}, d \in [D]$
    Sample cut location $u_d^j \sim U[l_d^j, h_d^j]$
    $v_{j(left)} \leftarrow [l_1^j, h_1^j] \times \cdots \times [l_d^j, u_d^j] \times \cdots \times \times [l_D^j, h_D^j]$
    $v_{j(right)} \leftarrow [l_1^j, h_1^j] \times \cdots \times [u_d^j, h_d^j] \times \cdots \times \times [l_D^j, h_D^j]$
    $V \leftarrow V \cup \{v_{j(left)}, v_{j(right)}\} \setminus v_j$
  end while
  return V
end function

In particular, we denote the maximum number of Mondrian partitions by N_p (usually the worker pool size in the experiments) and the minimum number of data points in each partition to be S. The set of partitions computed by the Mondrian tree (a.k.a. the leaves of the tree), V, is initialized to be the function domain $V = \{[0, R]^D\}$, the root of the tree. For each $v_j \in V$ described by a hyperrectangle $[l_1^j, h_1^j] \times \cdots \times [l_D^j, h_D^j]$, the length of v_j is computed to be $length(v_j) = \sum_{d=1}^{D} (h_d^j - l_d^j)$. The observations associated with v_j is $\mathcal{D}^j$. Here, for all $(x, y) \in \mathcal{D}^j$, we have $x \in [l_1^j - \epsilon, h_1^j + \epsilon] \times \cdots \times [l_D^j - \epsilon, h_D^j + \epsilon]$, where ϵ controls

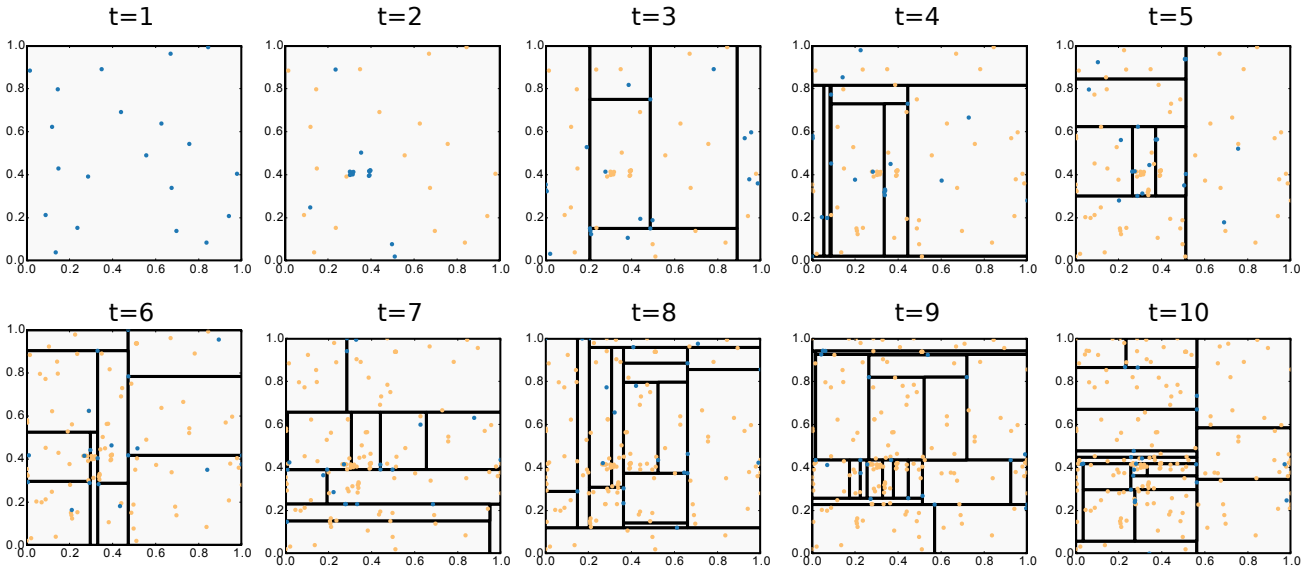

Figure 2: An example of 10 iterations of EBO on a 2D toy example plotted in Fig. 1. The selections in each iteration are blue and the existing observations orange. EBO quickly locates the region of the global optimum while still allocating budget to explore regions that appear promising (e.g. around the local optimum $(1.0, 0.4)$).

the how many neighboring data points to consider for the partition v_j. In our experiments, ϵ is set to be 0. Alg. 1 is different from Algorithm 1 and 2 of Lakshminarayanan et al. (2016) in the stop criterion. Lakshminarayanan et al. (2016) uses an exponential clock to count down the time of splitting the leaves of the tree, while we split the leaves until the number of Mondrian partitions reaches N_p or there is no partition that have more than S data points. We designed our stop criterion this way to balance the efficiency of EBO and the quality of selected points. Usually EBO is faster with larger number of partitions N_p (i.e., more parallel computing resources) and the quality of the selections are better with larger size of observations on each partition (S).

C Budget allocation and batched BO

In the EBO algorithm, we first use a batch of workers to learn the local GPs and recommend potential good candidate points from the local information. Then we aggregate the information of all the workers, and use a filter to select the points to evaluate from the set of points recommended by all the workers based on the aggregated information on the function.

There are two important details we did not have space to discuss in the main paper: (1) how many points to recommend from each local worker (budget allocation); and (2) how to select a batch of points from the Mondrian partition on each worker. Usually in the beginning of the iterations, we do not have a lot of Mondrian partitions (since we stop splitting a partition once it reaches a minimum number of data points). Hence, it is very likely that the number of partitions J is smaller than the size of the batch. Hence we need to allocate the budget of recommendations from each worker properly and use batched BO for each Mondrian partition.

Budget allocation In our current version of EBO, we did the budget allocation using a heuristic, where we would like to generate at least $2B$ recommendations from all the workers, and each worker gets the budget proportional to a score, the sum of the Mondrian partition volume (volume of the domain of the partition) and the best function value of the partition.

Batched BO For batched BO, we also use a heuristic where the points achieving the top n acquisition function values are always included and the other ones come from random points selected in that partition. For the optimization of the acquisition function over each block of dimensions, we sample 1000 points in the low dimensional space associated with the additive component and minimize the acquisition function via L-BFGS-B starting from the point that gives the best acquisition value. We add the optimized $\arg\min$ to the 1000 points and sort them according to their acquisition values, and then select the top n random ones, and combine with the sorted selections from other additive components. Other batched BO methods can also be used and can potentially improve upon our results.

D Relations to Mondrian Kernels and Random Binning

TileGP can use Mondrian grids or (our version of) tile coding to achieve efficient parameter inference for the decom-

position z and the number of cuts k (inverse of kernel bandwidth). Mondrian grids and tile coding are closely related to Mondrian kernels and random binning, but there are some subtle differences. We illustrate the differences between one

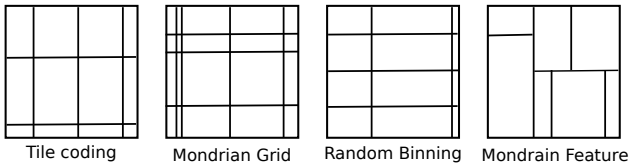

Figure 3: Illustrations of (our version of) tile coding, Mondrian Grid, random binning and Mondrian feature.

layer of the features constructed by tile coding, Mondrian grid, Mondrian feature and random binning in Fig. 3. For each layer of (our version of) tile coding, we sample a positive integer k (number of cuts) from a Poisson distribution parameterized by λR, and then set the offset to be a constant uniformly randomly sampled from $[0, \frac{R}{k}]$. For each layer of the Mondrian grid, the number of cuts k is sampled tile in coding, but instead of using an offset and uniform cuts, we put the cuts at locations independently uniformly randomly from $[0, R]$. Random binning does not sample k cuts but samples the distance δ between neighboring cuts by drawing $\delta \sim \text{GAMMA}(2, \lambda R)$. Then, it samples the offset from $[0, \delta]$ and finally places the cuts. All of the above-mentioned three types of random features can work individually for each dimension and then combine the cuts from all dimensions. The Mondrian feature (Mondrian forest features to be exact), contrast, partitions the space jointly for all dimensions. More details of Mondrian features can be found in Lakshminarayanan et al. (2016); Balog et al. (2016). For all of these four types of random features and for each layer of the total L layers, the kernel is $\kappa_L(\boldsymbol{x}, \boldsymbol{x}') = \frac{1}{L}\sum_{l=1}^{L} \chi_l(\boldsymbol{x}, \boldsymbol{x}')$ where

$$\chi_l(\boldsymbol{x}, \boldsymbol{x}') = \begin{cases} 1 & \boldsymbol{x} \text{ and } \boldsymbol{x}' \text{ are in the same cell on the layer } l \\ 0 & \text{otherwise} \end{cases} \tag{D.1}$$

For the case where the kernel has M additive components, we simply use the tiling for each decomposition and normalize by LM instead of L. More precisely, we have $\kappa_L(\boldsymbol{x}, \boldsymbol{x}') = \frac{1}{LM}\sum_{m=1}^{M}\sum_{l=1}^{M} \chi_l(\boldsymbol{x}^{A_m}, \boldsymbol{x}'^{A_m})$.

We next prove the lemma mentioned in Section 3.5.

Lemma 3.1. Let the random variable $k_{di} \sim \text{POISSON}(\lambda_d R)$ be the number of cuts in the Mondrian grids of TileGP for dimension $d \in [D]$ and layer $i \in [L]$. The kernel of TileGP κ_L satisfies $\lim_{L\to\infty} \kappa_L(\boldsymbol{x}, \boldsymbol{x}') = \frac{1}{M}\sum_{m=1}^{M} e^{\lambda_d R|\boldsymbol{x}^{A_m} - \boldsymbol{x}'^{A_m}|}$, where $\{A_m\}_{m=1}^{M}$ is the additive decomposition.

Proof. When constructing the Mondrian grid for each layer and each dimension, one can think of the process of getting another cut as a Poisson point process on the interval $[0, R]$, where the time between two consecutive cuts is modeled as an exponential random variable. Similar to Proposition 1 in Balog et al. (2016), we have $\lim_{L\to\infty} \kappa_L^{(m)}(\boldsymbol{x}^{A_m}, \boldsymbol{x}'^{A_m}) = \mathbb{E}[\text{no cut between } \boldsymbol{x}_d \text{ and } \boldsymbol{x}'_d, \forall d \in A_m] = e^{-\lambda_d R|\boldsymbol{x}^{A_m} - \boldsymbol{x}'^{A_m}|}$. By the additivity of the kernel, we have $\lim_{L\to\infty} \kappa_L(\boldsymbol{x}, \boldsymbol{x}') = \frac{1}{M}\sum_{m=1}^{M} e^{\lambda_d R|\boldsymbol{x}^{A_m} - \boldsymbol{x}'^{A_m}|}$. □

E Experiments

Verifying the acquisition function As introduced in Section 3.3, we used a different acquisition function optimization technique from (Kandasamy et al., 2015; Wang and Jegelka, 2017). In (Kandasamy et al., 2015; Wang and Jegelka, 2017), the authors used the fact that each additive component is by itself a GP. Hence, they did posterior inference on each additive component and Bayesian optimization independently from other additive components. In this work, we use the full GP with the additive kernel to derive its acquisition function and optimize it with a block coordinate optimization procedure, where the blocks are selected according to the decomposition of the input dimensions. One reason we did this instead of following (Kandasamy et al., 2015; Wang and Jegelka, 2017) is that we observed the over-estimation of variance for each additive component if inferred independently from others. We conjecture that this over-estimation could result in an invalid regret bound for Add-GP-UCB (Kandasamy et al., 2015). Nevertheless, we found that using the block coordinate optimization for the acquisition function on the full GP is actually very helpful. In Figure. 4, we compare the acquisition function we described in Section 3.3 (denoted as BlockOpt) with Add-GP-UCB (Kandasamy et al., 2015), Add-MES-R and Add-MES-G (Wang and Jegelka, 2017) on the same experiment described in the first experiment of Section 6.5 of (Wang and Jegelka, 2017), averaging over 20 functions. Notice that we used the maximum value of the function as part of our acquisition function in our approach (BlockOpt). Add-GP-UCB, ADD-MES-R and ADD-MES-G cannot use this max-value information even if they have access to it, because then they don't have a strategy to deal with "credit assignment", which assigns the maximum value to each additive component. We found that BlockOpt is able to find a solution as well as or even better than the best of the three competing approaches.

Scalability of EBO For EBO, the maximum number of Mondrian partitions is set to be 1000 and the minimum number of data points in each Mondrian partition is 100. The function that we used to test was generated from a fully partitioned 20 dimensional GP with an additive Laplace kernel ($|A_m| = 1, \forall m$).

Effectiveness of EBO In this experiment, we sampled 4 functions from a 50-dimensional GP with additive kernel.

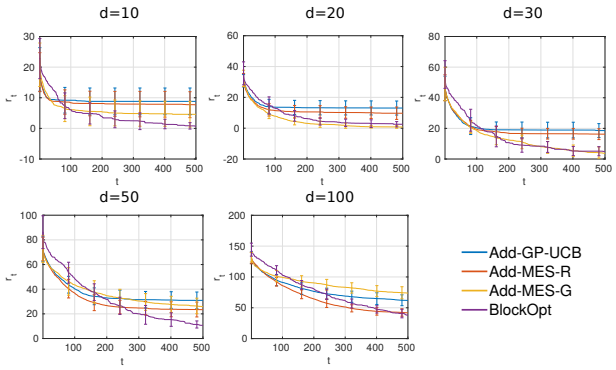

Figure 4: Comparing different acquisition functions for BO with an additive GP. Our strategy, BlockOpt, achieves comparable or better results than other methods.

Each component of the additive kernel is a Laplace kernel, whose lengthscale parameter is set to be 0.1, variance scale to be 1 and active dimensions are around 1 to 4. Namely, the kernel we used is $\kappa(x, x') = \sum_{i=1}^{M} \kappa^{(m)}(x^{A_m}, x'^{A_m})$ where $\kappa^{(m)}(x^{A_m}, x'^{A_m}) = e^{\frac{|x^{A_m} - x'^{A_m}|}{0.1}}, \forall m$. The domain of the function is $[0, 1]^{50}$. We implemented the BO-SVI and BO-Add-SVI using the same acquisition function and batch selection strategy as EBO but with SVI-GP (Hensman et al., 2013) and SVI-GP with additive kernels instead of TileGPs. We used the SVI-GP implemented in GPy (since 2012) and defined the additive Laplace kernel according to the priors of the tested functions. For both BO-SVI and BO-Add-SVI, we used 100 batchsize, 200 inducing points and the parameters were optimized for 100 iterations. For EBO, we set the minimum size of data points on each Mondrian partition to be 100. We set the maximum number of Mondrian partitions to be 1000 for both EBO and PBO. The evaluations of the test functions are negligible, so the timing results in Figure 5 reflect the actual runtime of each method.

Optimizing control parameters for robot pushing We implemented the simulation of pushing two objects with two robot hands in the Box2D physics engine Catto (2011). The 14 parameters specifies the location and rotation of the robot hands, pushing speed, moving direction and pushing time. The lower limit of these parameters is $[-5, -5, -10, -10, 2, 0, -5, -5, -10, -10, 2, 0, -5, -5]$ and the upper limit is $[5, 5, 10, 10, 30, 2\pi, 5, 5, 10, 10, 30, 2\pi, 5, 5]$. Let the initial positions of the objects be s_{i0}, s_{i1} and the ending positions be s_{e0}, s_{e1}. We use s_{g0} and s_{g1} to denote the goal locations for the two objects. The reward is defined to be $r = \|s_{g0} - s_{i0}\| + \|s_{g1} - s_{i1}\| - \|s_{g0} - s_{e0}\| - \|s_{g1} - s_{e1}\|$, namely, the progress made towards pushing the objects to the goal.

We compare EBO, BO-SVI, BO-Add-SVI and CEM Szita and Lörincz (2006) with the same 10^4 random observations and repeat each experiment 10 times. All the methods choose a batch of 100 parameters to evaluate at each iteration. CEM uses the top 30% of the 10^4 initial observations to fit its initial Gaussian distribution. At the end of each iteration in CEM, 30% of the new observations with top values were used to fit the new distribution. For all the BO based methods, we use the maximum value of the reward function in the acquisition function. The standard deviation of the observation noise in the GP models is set to be 0.1. We set EBO to have Modrian partitions with fewer than 150 data points and constrain EBO to have no more than 200 Mondrian partitions. In EBO, we set the hyper parameters $\alpha = 1.0$, $\beta = [5.0, 5.0]$, and the Mondrian observation offset $\epsilon = 0.05$. In BO-SVI, we used 100 batchsize in SVI, 200 inducing points and 500 iterations to optimize the data likelihood with 0.1 step rate and 0.9 momentum. BO-Add-SVI used the same parameters as BO-SVI, except that BO-Add-SVI uses 3 outer loops to randomly select the decomposition parameter z and in each loop, it uses an inner loop of 50 iterations to maximize the data likelihood over the kernel parameters. The batch BO strategy used in BO-SVI and BO-Add-SVI is identical to the one used in each Mondrian partition of EBO.

We run all the methods for 200 iterations, where each iteration has a batch size of 100. In total, each method obtains 2×10^4 data points in addition to the 10^4 initializations.

Optimizing rover trajectories We illustrate the problem in Fig. 5 with an example trajectory found by EBO. We set the trajectory cost to be -20.0 for any collision, λ to be -10.0 and the constant $b = 5.0$. This reward function is non smooth, discontinuous, and concave over the first two and last two dimensions of the input. These 4 dimensions represent the start and goal position of the trajectory. We maximize the reward function f over the points on the trajectory. All the methods choose a batch of 500 trajectories to evaluate. Each method is initialized with 10^4 trajectories randomly uniformly selected from $[0, 1]^{60}$ and their reward function values. We again compare EBO with BO-SVI, BO-Add-SVI and CEM (Szita and Lörincz, 2006). All the methods choose a batch of 500 trajectories to evaluate. Each method is initialized with 10^4 trajectories randomly uniformly selected from $[0, 1]^{60}$ and their reward function values. The initializations are the same for each method, and we repeat the experiments 5 times. CEM uses the top 30% of the 10^4 initial observations to fit its initial Gaussian distribution. At the end of each iteration in CEM, 30% of the new observations with top values were used to fit the new distribution. For all the BO based methods, we use the maximum value of the reward function, 5.0, in the acquisition function. The standard deviation of the observation noise in the GP models is set to be 0.01. We set EBO to attempt to have Modrian partitions with fewer than 100 data points, with a hard constraint of no more than 1000 Mondrian partitions. In EBO, we set the hyper parameters $\alpha = 1.0$, $\beta = [2.0, 5.0]$, and the Mondrian observation off-

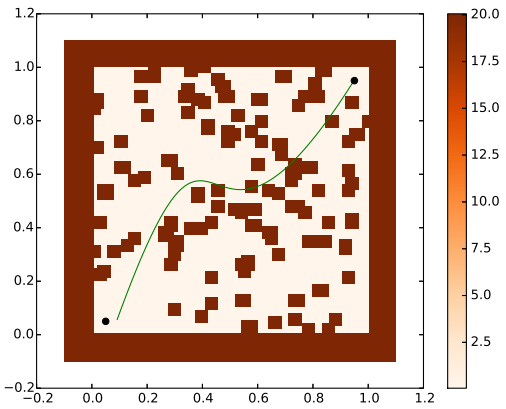

Figure 5: An example trajectory found by EBO.

set $\epsilon = 0.01$. In BO-SVI, we used 100 batchsize in SVI, 200 inducing points and 500 iterations to optimize the data likelihood with 0.1 step rate and 0.9 momentum. BO-Add-SVI used the same parameters as BO-SVI, except that BO-Add-SVI uses 3 outer loops to randomly select the decomposition parameter z and in each loop, it uses an inner loop of 50 iterations to maximize the data likelihood over the kernel parameters. The batch BO strategy used in BO-SVI and BO-Add-SVI is identical to the one used in each Mondrian partition of EBO.

F Discussion

F.1 Failure modes of EBO

EBO is a general framework for running large scale batched BO in high-dimensional spaces. Admittedly, we made some compromises in our design and implementation to scale up BO to a degree that conventional BO approaches cannot deal with. In the following, we list some limitations and aspects that we can improve in EBO in our future work.

- EBO partitions the space into smaller regions $\{[l_j, h_j]\}_{j=1}^J$ and only uses the observations within $[l_j - \epsilon, h_j + \epsilon]$ to do inference and Bayesian optimization. It is hard to determine the value of ϵ. If ϵ is large, we may have high computational cost for the operations within each region. But if ϵ is very small, we found that some selected BO points are on the boundaries of the regions, partially because of the large uncertainty on the boundaries. We used $\epsilon = 0$ in our experiments, but the results can be improved with a more appropriate ϵ.

- Because of the additive structure, we need to optimize the acquisition function for each additive component. As a result, EBO has increased computational cost when there are more than 50 additive components, and it becomes harder for EBO to optimize functions more than a few hundred dimensions. One solution is to combine the additive structure with a low dimensional projection approach (Wang et al., 2016). We can also simply run block coordinate descent on the acquisition function, but it is harder to ensure that the acquisition function is fully optimized.

F.2 Importance of avoiding variance starvation

Neural networks have been applied in many applications and received success for tasks including regression and classification. While researchers are still working on the theoretical understanding, one hyoothesis is that neural networks "overfit" Zhang et al. (2017). Due to the similarity between the test and training set in the reported experiments in, for example, the computer vision community, overfitting may seem to be less of a problem. However, in active learning (e.g. Bayesian optimization), we do not have a "test set". We require the model to generalize well across the search space, and using the classic neural network may be detrimental to the data selection process, because of variance starvation (see Section 2). Gaussian processes, on the contrary, are good at estimating confidence bounds and avoid overfitting. However, the scaling of Gaussian processes is hard in general. We would like to reinforce the awareness about the importance of estimating confidence of the model predictions on new queries, i.e., avoiding variance starvation.

F.3 Future directions

Possible future directions include analyzing theoretically what should be the best input space partition strategy, batch worker budget distribution strategy, better ways of predicting variance in a principled way (not necessarily GP), better ways of doing small scale BO and how to adapt it to large scale BO. Moreover, add-GP is only one way of reducing the function space, and there could be others suitable ones too.

References

Matej Balog, Balaji Lakshminarayanan, Zoubin Ghahramani, Daniel M Roy, and Yee Whye Teh. The Mondrian kernel. In *Uncertainty in Artificial Intelligence (UAI)*, 2016.

Erin Catto. Box2D, a 2D physics engine for games. `http://box2d.org`, 2011.

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

James Hensman, Nicolo Fusi, and Neil D Lawrence. Gaussian processes for big data. In *Uncertainty in Artificial Intelligence (UAI)*, 2013.

Kirthevasan Kandasamy, Jeff Schneider, and Barnabas Poczos. High dimensional Bayesian optimisation and bandits via additive models. In *International Conference on Machine Learning (ICML)*, 2015.

Balaji Lakshminarayanan, Daniel M Roy, and Yee Whye Teh. Mondrian forests for large-scale regression when uncertainty matters. In *International Conference on Artificial Intelligence and Statistics (AISTATS)*, 2016.

István Szita and András Lörincz. Learning tetris using the noisy cross-entropy method. *Learning*, 18(12), 2006.

Zi Wang and Stefanie Jegelka. Max-value entropy search for efficient Bayesian optimization. In *International Conference on Machine Learning (ICML)*, 2017.

Ziyu Wang, Frank Hutter, Masrour Zoghi, David Matheson, and Nando de Feitas. Bayesian optimization in a billion dimensions via random embeddings. *Journal of Artificial Intelligence Research*, 55:361–387, 2016.

Chiyuan Zhang, Samy Bengio, Moritz Hardt, Benjamin Recht, and Oriol Vinyals. Understanding deep learning requires rethinking generalization. In *International Conference on Learning Representations (ICLR)*, 2017.