Bayesian Optimization with a Neural Network Kernel

Akshara Rai
Robotics Institute
arai@andrew.cmu.edu

Ruta Desai
Robotics Institute
rutad@andrew.cmu.edu

Siddharth Goyal
Computer Science Department
sgoyal1@andrew.cmu.edu

1 Introduction

Many real-world problems involve optimizing expensive to evaluate cost functions. Bayesian Optimization (BO) is a promising tool for sample-efficient optimization, but its performance degrades in higher dimensions. To counter this, informed kernels or similarity metrics between points are usually employed to increase the sample-efficiency of BO. Neural networks are powerful tools for automatically discovering such underlying patterns in data and hence can prove to be effective kernels for high dimensional optimization problems. The goal of our project is to explore sample-efficient optimization for high-dimensional problems, by learning a similarity metric using neural networks and using this as a kernel for BO.

As a test bed, we will use layout optimization for arbitrarily shaped 3D components within a constrained space. Such optimization problems often arise in industrial production, VLSI design, city and floor planning as well as in packaging. This is a high-dimensional optimization problem and evaluating the cost function is computationally expensive. For such high dimensional problems, performance of sample-efficient methods such as BO, typically degrades. An informed kernel in the underlying Gaussian Process (GP) can help with this.

In recent years, there has been a lot of work on learning kernels for GP from data. This has several advantages: it automatically extracts useful information from the data to generate a more informed kernel [6],[2], and it can be trained in an unsupervised manner with unlabeled points [6]. [7] replace the GP completely with a NN in a BO framework to make it scalable in terms of number of data points.

Calandra et al [2] model convex and non-differentiable functions using GPs by proposing a new supervised method, Manifold Gaussian Processes (MGP). The MGP approach finds an alternate representation of the input data using a NN, and applies GP regression on the transformed input. This was shown to be able to learn discontinuities in the target function, which was otherwise difficult with a standard squared-exponential kernel. Building on this work, [11] optimized a deep NN kernel and the hyper-parameters of the GP together, which performed better than first learning the NN kernel followed by optimizing the GP hyper-parameters (as in [2]).

For our experiments, we use Bayesian Optimization which is a framework for sequential global search to find a vector $\boldsymbol{x}^*$ that minimizes a cost function $f(\boldsymbol{x})$, while evaluating f as few times as possible ([9] gives an overview).

$$\boldsymbol{x}^* = \arg\min_{\boldsymbol{x}} f(\boldsymbol{x})$$

The optimization starts with a prior over the value of $f(\boldsymbol{x})$ for each $\boldsymbol{x}$ in the domain. An auxiliary function u, called an acquisition function, is used to sequentially select the next parameter vector to test, $\boldsymbol{x}_t$. $f(\boldsymbol{x}_t)$ is then evaluated and used to update our estimate of f. The steps are outlined in Algorithm 1.

A common way to model the prior and posterior for f is by using a Gaussian Process $f(\boldsymbol{x}) \sim \mathcal{GP}(\mu(\boldsymbol{x}), k(\boldsymbol{x}_i, \boldsymbol{x}_j))$, with mean function μ and kernel k. The kernel $k(\boldsymbol{x}_i, \boldsymbol{x}_j)$ encodes how similar f is expected to be for two inputs $\boldsymbol{x}_i, \boldsymbol{x}_j$: points close together are expected to influence each other strongly, while points far apart would have almost no influence.

30th Conference on Neural Information Processing Systems (NIPS 2016), Barcelona, Spain.

Algorithm 1: Bayesian Optimization Algorithm (BOA)

Let $D = \{\}, n = 0$
while *no convergence* **do**
  select the next point to evaluate: $\boldsymbol{x}_{n+1} = \arg\min_{\boldsymbol{x}} u(\boldsymbol{x}|D_n)$;
  update $D_{n+1} = D_n \cup (\boldsymbol{x}_{n+1}, f(\boldsymbol{x}_{n+1}))$

A commonly used GP kernel is Squared Exponential (SE) kernel of the form $k_{SE}(\boldsymbol{x}_i, \boldsymbol{x}_j) = \exp\Big(-\frac{1}{2}\|\boldsymbol{x}_i - \boldsymbol{x}_j\|^2\Big)$. However, this is often not sample-efficient for high dimensional optimization problems. We want to learn the kernel for the GP from data using a neural network, and then use this GP-NN framework as part of a BO problem. In the following sections we describe the framework for training a NN inside a GP kernel and our experiments to test our method. We formulate a series of increasingly complex layout optimization problems, and compare our kernel to a squared exponential (SE) kernel, as well as other optimization techniques like gradient descent (GD) and Covariance Matrix Adaptation (CMA-ES).

2 Methods

2.1 A neural network kernel for Gaussian processes

Kernels in GPs encode similarity or distances between points. In high dimensional problems, projecting the input space to a lower dimensional manifold with a kernel transformation can make the search very efficient [1]. However, typically, these transformations are hand-designed. Since NNs can automatically discover patterns in datasets, we learn such a transformation for a GP with a NN from data.

Let the kernel k be squared exponential (SE), and the neural network be represented by g. Using the NN transform, new kernel distances become

$$k(x_i, x_j|\lambda) = k(g(x_i, w), g(x_j, w)|\lambda) \tag{1}$$

where $g(x_i, w)$ is the output of a network with weights w for input x_i and λ are the SE hyperparameters. For SE kernels, this becomes

$$k(g_i, g_j) = \exp(-\frac{1}{2\lambda}||g_i - g_j||^2) \tag{2}$$

where g_i is $g(x_i, \mathbf{w})$ and g_j is $g(x_j, \mathbf{w})$. We will jointly learn the parameters of the network and the GP, $\gamma = w, \theta$, where $\theta = \lambda, \sigma$ (σ is the noise standard deviation)

Let $\mathcal{L}$ be the log marginal likelihood of the GP. From [11],

$$\log p(\mathbf{y}|\mathbf{x}, \gamma) = \mathcal{L} \tag{3}$$

$$\frac{\partial \mathcal{L}}{\partial \theta} = \frac{\partial \mathcal{L}}{\partial K_\gamma}\frac{\partial K_\gamma}{\partial \theta} \tag{4}$$

$$K_\gamma = K_{X,X} + \sigma^2 I \tag{5}$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{w}} = \frac{\partial \mathcal{L}}{\partial K_\gamma}\frac{\partial K_\gamma}{\partial g(\mathbf{x},\mathbf{w})}\frac{\partial g(\mathbf{x},\mathbf{w})}{\partial \mathbf{w}} \tag{6}$$

$$\frac{\partial \mathcal{L}}{\partial K_\gamma} = \frac{1}{2}(K_\gamma^{-1}\mathbf{y}\,\mathbf{y}^T K_\gamma^{-1} - K_\gamma^{-1}) \tag{7}$$

Now for the derivative with respect to the network weights, we derive a gradient of the GP likelihood with respect to the network parameters, and back-propogate it through the network.

$$\frac{\partial K_{\gamma,i,j}}{\partial g(\mathbf{x},\mathbf{w})} = \frac{\partial K_{\gamma,i,j}}{\partial g_i} + \frac{\partial K_{\gamma,i,j}}{\partial g_j} \tag{8}$$

$$\frac{\partial K_{\gamma,i,j}}{\partial g_i} = \frac{\partial}{\partial g_i} \exp(-\frac{1}{2\lambda}(g_i - g_j)^2) \tag{9}$$

$$\frac{\partial K_{\gamma,i,j}}{\partial g_i} = -\frac{1}{2\lambda}(g_i - g_j)K_{\gamma,ij} \tag{10}$$

$$\frac{\partial K_{\gamma,i,j}}{\partial g(\mathbf{x},\mathbf{w})} = -\frac{1}{\lambda}(g_i - g_j)K_{\gamma,i,j} \tag{11}$$

This gives an expression for the derivative of the kernel with respect to the network outputs g. $\frac{\partial g}{\partial w}$ is the back-propogated gradient of the network. We can use this to update the network weights in the kernel of a GP when trying to approximate a function.

2.2 Bayesian Optimization with Neural Networks

For all our experiments described later, we have a 4 layer network with 100-50-20-2 hidden units. The last hidden layer is sent as input to the GP (Figure 1). Its important to have a lower number of hidden units (2-10) in the last layer, as this determines the input dimension for the GP. Higher dimensions typically make the GP hyper-parameter optimization as well as posterior estimation difficult and slow.

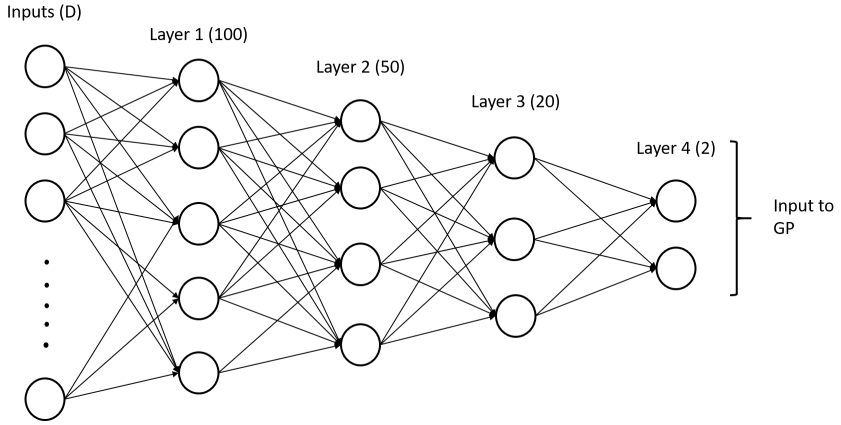

Figure 1: Neural network structure used for the GP kernel.

We have a two-step training procedure for training the NN before introducing into the kernel. We train our network in caffe [8] to predict the cost, with a 1 unit output by minimizing the L-2 norm of error from a dataset. This is followed by a second stage where we train a GP to predict this output with the NN kernel. Since our data-sets are huge, we use a sparse scalable GP formulation from [10]. This is the "fine-tuning" stage, where the GP hyperparameters are optimized with the NN, and we run it for about 10 epochs. Now, we use the trained NN and tuned GP hyperparameters in a BO framework.

2.3 2-D layout optimization

For a simplified 2D layout optimization, we define fixed sized rectangular objects bounded within a 2D space (defined by a bounding box between $(-x_{bound}, -y_{bound})$ and (x_{bound}, y_{bound}). An object o_i is defined by its 2D position and a rotation (x_i, y_i, θ_i). We are trying to minimize the overlap between the objects in the bounding box. For a set of n objects, the layout cost becomes:

$$\underset{p}{\text{minimize}} \quad J(o_1, o_2, \cdots, o_n) \tag{12}$$

$$J(o_1, o_2, \cdots, o_n) = \sum_{i=1}^{n} \sum_{j=i+1}^{n} overlap(o_i, o_j) + I(i) \cdot \sum_{i=1}^{n} (x_i - x_{bound})^2 + (y_i - y_{bound})^2 \tag{13}$$

where $p = [(x_1, y_1, \theta_1), \cdots, (x_n, y_n, \theta_n)]$ is the parameter vector consisting of positions and orientations of all objects and $J(o_1, o_2, \cdots, o_n)$ is the layout cost function. $I(i)$ is the indicator function which is 1 if o_i is inside the bounding box and 0 otherwise. The first term in J corresponds to overlap area between a pair of objects(see fig. 2)), and the second term is a quadratic cost that penalizes objects that are out of the bounding box. We use Sutherland-Hodgman polygon clipping algorithm [4] for finding the overlap area. We consider two cases for experimentation: *Non-rotated objects case*, and *Rotated objects case*. For the non-rotated case, we set $\theta_i = 0$.

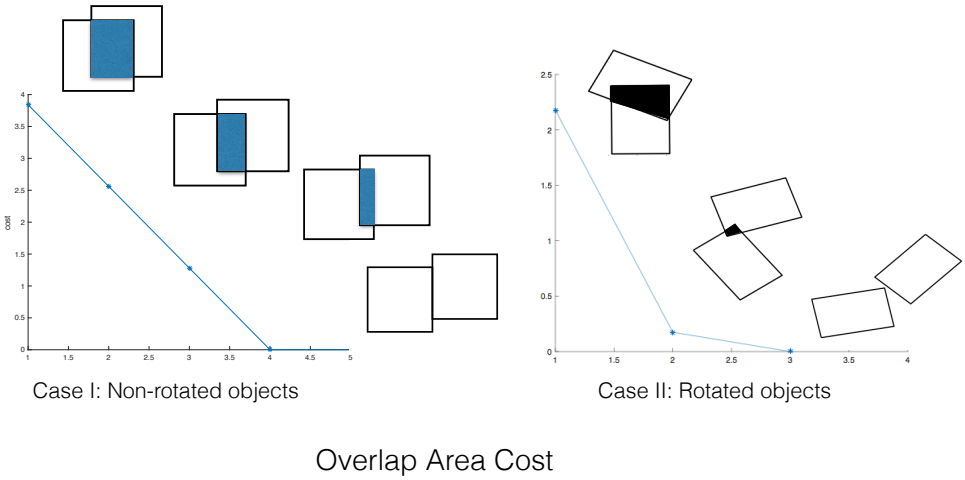

Figure 2: Overlap area cost between pairs of objects is shown for the two cases (non-rotated and rotated objects) in multiple scenarios. The total cost is the sum of overlap area between all pairs of objects in the scene as well as a bounding cost.

2.4 3-D layout optimization

We use a realistic 3-D physical layout simulation system developed for a computational design application for layout optimization in 3-D. Our test set-up consists of n arbitrary shaped objects such as spheres and cuboids (fig. 3 (right)). The cost is calculated by computing collisions between these shapes using Bullet Physics [3].

3 Experiments and Results

In order to test the performance of BO with NN kernel, we first experiment on a standard optimization test function (Ackley function, described below). We then experiment with layout optimization in our simplified 2-D set-up and then in our realistic 3-D system. We run 10 runs of all experiments, each consisting of 100 function evaluations (except 2-D non-rotated case, where we run each experiment for 120 evaluations).

3.1 Standard test function optimization

We tested BO with NN kernel on 30-D Ackley function (a 2-D visualization of this function is shown in fig. 4(left)). Since the optimum of this function is known apriori, it allowed us to fine-tune and test our framework for NN, GP training as well as for integration withing BO. In particular, we

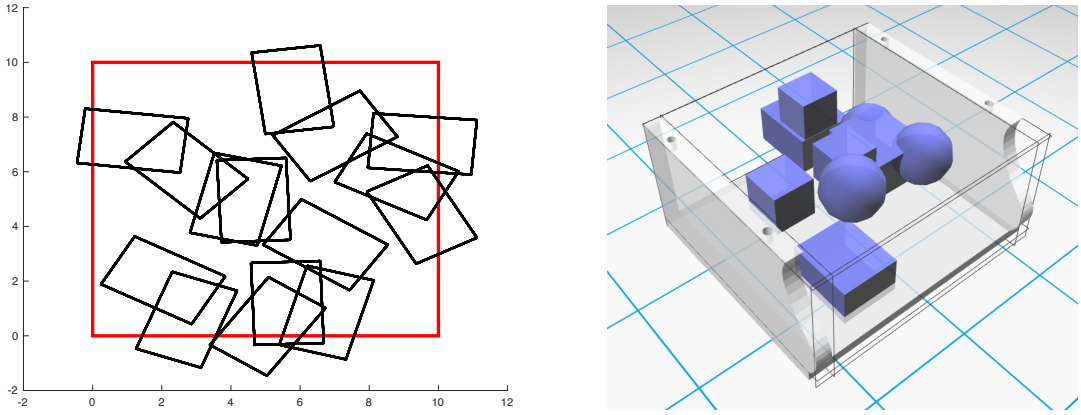

Figure 3: An example layout of objects is shown in 2-D (left) and 3-D (right). The red box denoted the bounding box in 2-D while the black outline represents the bounding box in 3-D.

trained two NN kernels, one with 100,000 points, and other with 500,000 points. The data was collected by random sampling. When the NN was trained with more data points and input-data normalization, the BO performed much better (Fig. 4(right)). BO found a global optimum in 4 iterations in all 10 experiment runs. However, when the NN was trained with fewer points and no input data normalization, the BO performance was poor.

While, bigger data-set increased the probability of having samples around the optimum and thereby resulted in a better approximation of the function, input normalization allowed for a better training. Together, this allowed for the network to learn the underlying structure better, and could quickly guide BO to the optimum. This highlights the need for better training the NN kernel as well as a data-set which captures the problem structure well.

3.2 2D layout optimization: Non-rotated case

We next examine the 2D layout optimization problem with the assumption that rotations of rectangles are not allowed ($\theta_i = 0$). In order to compare the performance of BO (using SE and NN kernel) with other standard optimization methods, we consider 10 rectangles (in $\mathcal{R}^2$, each of size 2×1) in a bounded region $[-2.5, 2.5]^2$. Thus, the dimensionality of the parameter space for BO is 20. In addition to BO with SE and NN kernels, we investigate the performance of CMA-ES algorithm [5] and gradient descent using numerical gradients.

For training the NN kernel, we sample 100,000 points in the bounded region, and the corresponding costs. The performance of the methods averaged over 10 runs (considering only 120 function evaluations) is shown in the LHS of Figure 5. Gradient descent (GD) is piece-wise constant because computing the numerical gradient at a D-dimensional point takes $2 \times D$ function evaluations. BO with either of the 2 kernels has similar performance to the GD and CMA-ES performs slightly better than the other methods. However, all the four methods have comparable performance, as this is a fairly simple optimization problem and most methods perform competitively.

However, we observe that none of the methods find the global optimum point in the given number of function evaluations. If we allow for more number of evaluations, then it is expected that all methods will eventually find the optimum. The right side of Figure 5 shows gradient descent converging to an optimum with 600 function evaluations.

3.3 2D layout optimization: Rotated case

We next relax the no-rotation assumption, and consider the general 2D layout optimization problem. We consider 15 rectangles, each of dimensions 3×2, and an appropriate bounding region ($[2, 8]^2$), and evaluate the performance of BO using the 2 kernels. The total dimensionality of the problem

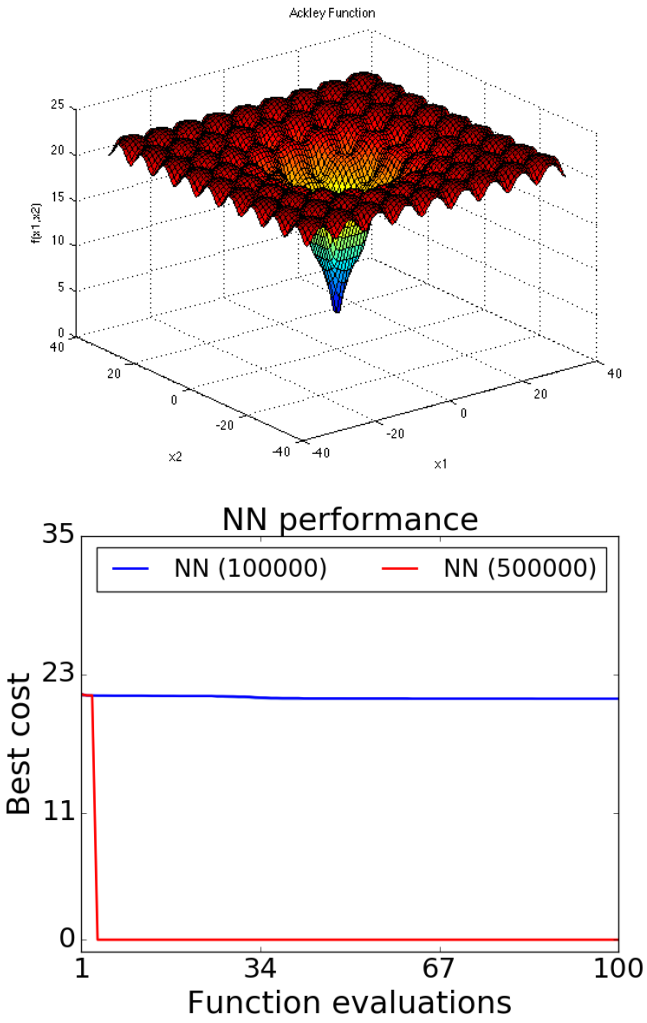

Figure 4: Ackley function in 2-D (left) and BO optimization using NN kernels over a 30-D Ackley function (right).

becomes $15 \times 3 = 45$. For training the NN kernel, we sample 500,000 points in the bounding region and the corresponding costs. The left side of Figure 6 shows the performance of BO with NN and SE kernels for this cost. BO with NN kernel quickly finds an optimum in less than 30 evaluations, unlike SE, which does not always find it in 100 evaluations.

3.4 3D layout optimization

We collected 200,000 data points from our physical layout simulation for 9 objects (consisting of spheres and cuboids). We then ran BO by setting up a grid using this dataset for both NN and SE kernels. During the optimization, the cost was evaluated by doing a look-up in the data-set, and the next sampled point was chosen from the grid. NN kernel out performs SE kernel by a small margin (fig. 6 (right). However, none of them finds the optimum in the data-set in any of the runs.

4 Discussion

We present a way of using NN kernels for optimizing difficult to evaluate cost functions in high dimensions. We use the NN to learn a lower dimensional representation of our input data, and then feed this as a kernel through a GP. Next, we use this GP in a BO framework to optimize cost functions.

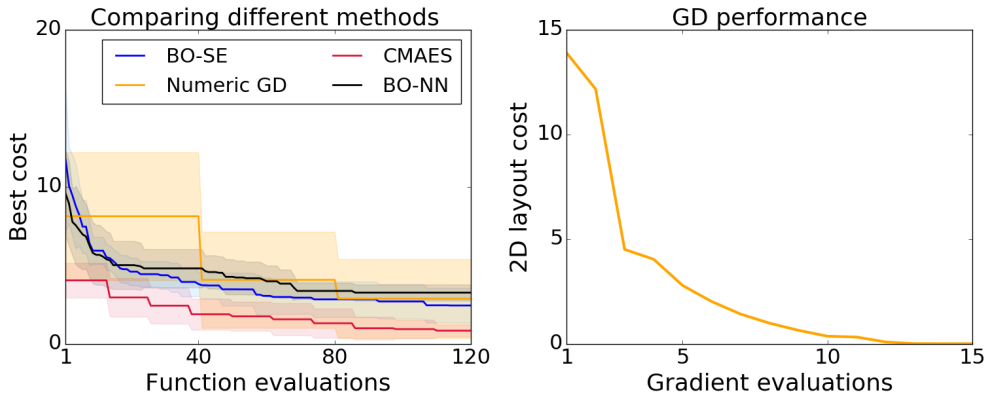

Figure 5: Performance of different methods on optimizing the non-rotated 2D layout problem. (left) CMA-ES, GD, and BO with SE, NN kernels are quite close in performance on this simple cost. (right) When allowed to run for longer, GD converges to an optimum better than the solution found in 100 evaluations.

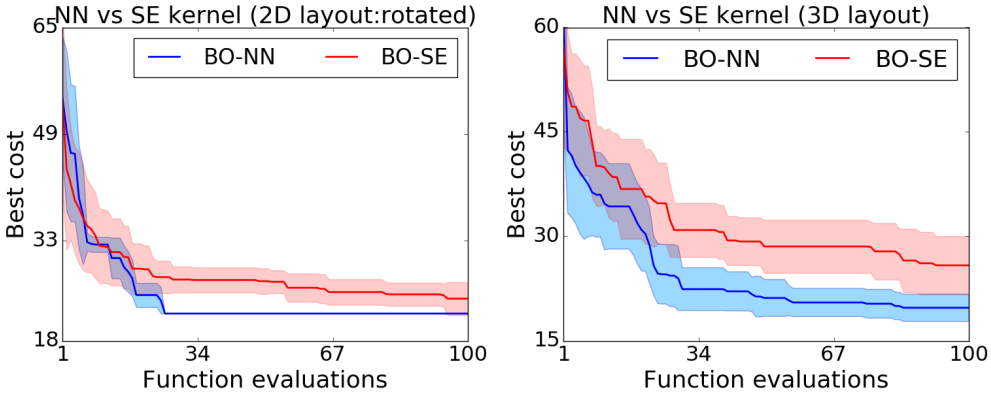

Figure 6: Performance of BO with SE and NN kernels on (left) 2D layout and (right) 3D layout problems. While SE and NN are close, NN performs slightly better in both cases.

A neural network (like ours) needs a lot of data points to train. For doing this, currently we use previously sampled points from our cost, but this defeats the 'sample-efficient' argument for BO. One motivation for this approach is that in some systems, like walking robots, collecting data while the robot is supported on a winch is easy as compared to when learning a controller. Another approach could be to train the network using an approximation of the real cost, for example from simulation, and run the final experiments on the real system. Due to shortage of time, we did not explore these approaches.

The performance of BO is quite sensitive to the training of the NN. If poorly trained, it can skew the space in a way that makes it very difficult for BO to sample potentially good points. However, if trained properly, it can learn a much lower dimensional representation of the original space, leading to much better sample-efficiency. During our experiments with Ackley function (see section 3.1), we found that with a well trained NN kernel (trained with bigger dataset and proper normalization of the data), BO could sample the optimum in less than 10 evaluations. However, if trained poorly, BO doesn't sample the optimum in even 100 evaluations.

In the layout cost, we find that a SE kernel performs very close to the NN kernel. This could be because the problem of layout is very well suited for euclidean distances between objects. Farther objects are indeed farther away in the kernel, and closer objects are closer. However, this is not true in general. In many problems, SE kernels are in fact a poor representation of function behavior. In such cases, our approach might be more useful and the NN kernel would perform better.

References

[1] Rika Antonova, Akshara Rai, and Christopher G Atkeson. Sample efficient optimization for learning controllers for bipedal locomotion. *arXiv preprint arXiv:1610.04795*, 2016.

[2] Roberto Calandra, Jan Peters, Carl Edward Rasmussen, and Marc Peter Deisenroth. Manifold gaussian processes for regression. *arXiv preprint arXiv:1402.5876*, 2014.

[3] Erwin Coumans et al. Bullet physics library. *Open source: bulletphysics. org*, 15, 2013.

[4] James D Foley, Andries Van Dam, Steven K Feiner, John F Hughes, and Richard L Phillips. *Introduction to computer graphics*, volume 55. Addison-Wesley Reading, 1994.

[5] Nikolaus Hansen. The cma evolution strategy: a comparing review. In *Towards a new evolutionary computation*, pages 75–102. Springer, 2006.

[6] Geoffrey E Hinton and Ruslan R Salakhutdinov. Using deep belief nets to learn covariance kernels for gaussian processes. In *Advances in neural information processing systems*, pages 1249–1256, 2008.

[7] Wenbing Huang, Deli Zhao, Fuchun Sun, Huaping Liu, and Edward Chang. Scalable gaussian process regression using deep neural networks. In *Proceedings of the 24th International Conference on Artificial Intelligence*, pages 3576–3582. AAAI Press, 2015.

[8] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*, 2014.

[9] Bobak Shahriari, Kevin Swersky, Ziyu Wang, Ryan P Adams, and Nando de Freitas. Taking the human out of the loop: A review of bayesian optimization. *Proceedings of the IEEE*, 104(1):148–175, 2016.

[10] Andrew Gordon Wilson and Ryan Prescott Adams. Gaussian process kernels for pattern discovery and extrapolation. In *ICML (3)*, pages 1067–1075, 2013.

[11] Andrew Gordon Wilson, Zhiting Hu, Ruslan Salakhutdinov, and Eric P Xing. Deep kernel learning. *arXiv preprint arXiv:1511.02222*, 2015.