MOBO Documentation

COLA-lab

October 11, 2021

1 Introduction

MOBO (MOdular Bayesian Optimization) is Bayesian Optimization and Gaussian Process modelling library built from the Google JAX library for auto-gradient computation. Even though the library is very young in its development and subject to changes, this document outlines how to interact with the library for implementing Bayesian Optimization algorithms and how to implement your own kernels.

2 Gaussian Process Modelling

Currently, the library only supports the use of a single-task Gaussian Process but multi-task Gaussian Process are planned to be developed in future.

2.1 Example

The outlined example can be found in *MOBO/Deployable/PWv1.py*. If you wish to bulid your own deployable BO algorithms, please create a folder to hold these files, i.e. *MOBO/Deployable/Name/file.py*.
To implement a MOBO Gaussian Regression model you will require the Kernel, GPregression, Likelihood and model optimizer classes. These can be accessed as outlined below:

```
from Kernels.JAX.rbf import RBF
from Likelihoods.JAX.chol import Likelihood
from Models.JAX.GPregression import GPregression
from ModelOptimizers.lbfgsb import lbfgsb
```

Note: Analytical models can be accessed by *.Analytical but are under developed
A standard Gaussian Process with RBF Kernel can be created using the following code:

```
x, y = Dataset
kernel = RBF(x.shape, ARD=True) # ARD can be false also
model = GPregression(kernel, x, y)
likelihood = Likelihood(model)
```

To save computation time for prediction we store training parameters such as the co-variance matrix and it's Cholesky decomposition to avoid re-computation. If you wish to predict the value of some unseen inputs without training the model **you must call .evaluate() function on the Likelihood function**, for example:

```
likelihood = Likelihood(model)
likelihood.evaluate()
mean, variance = model.predict(Xtest, full_cov=False)
# for full covariance matrix set full_cov=True
```

If you wish to fit the model first we call a model optimizer, this can be done as follows:

```
model_optimizer = lbfgsb(model)
model_optimizer.opt()
mean, variance = model.predict(Xtest, full_cov=False)
```

3 Acquisition Optimization

3.1 Example

By optimizing the expected improvement acquisition function with the lbfgs optimizer we select the next point to be sampled in optimization process. For this the following libraries our required:

```
from Acquisitions.JAX.EI import AcquisitionEI # Expected Improvement
from AcquisitionOptimizers.lbfgsb import AOlbfgs #lbfgsb optimizer
```

We optimize the acquisition function by:

```
acquisition = AcquisitionEI(model, min(y))
acquisition_optimizer = AOlbfgs(acquisition, self.lb, self.ub)
sample, ei_val = acquisition_optimizer.opt(restarts=5) # 5 restarts
```

The sample minimizing the **negative expected improvement function** is returned with its function value.

4 Implementing your own kernel

JAX is a useful tool because we can implement our kernel without writting its gradients explicitly. In this subsection we outline how to implement your own kernel references the implemented RBF kernel in the MOBO library.
For a JAX based kernel to work within the MOBO library and the implemented likelihood, it only requires 3 functions and a dictionary! Details of the attributes are as follows:

- Function: **function(X, params)**: this function takes the training data X and a dictionary of the kernels parameters *params*. Given these inpurs the function should return the covariance matrix of data X.
- Function: **cov(X, X2)**: this function returns the covariance matrix between two datasets X and X2.
- Function: $\mathbf{set}_parameters(params) : this function takes the dictionary \boldsymbol{params} and sets the parameters of kernel.$
- Attribute: **parameters**: this variable is a dictionary that holds the names and values of the kernels parameters, for example the RBF kernel:

```
if ARD:
    self.parameters = {"variance": jnp.ones((1,)) * variance * 1.,
                       "lengthscale": jnp.ones((dim,)) * lengthscale * 1.}
else:
    self.parameters = {"variance": jnp.ones((1,)) * variance * 1.,
                       "lengthscale": jnp.ones((1,)) * lengthscale * 1.}
```

5 Getting JAXY

JAX is very useful but as it is advertised, it is an ongoing research project and so it also has bugs we have navigate around. In this section we outline some food for thought when developing your own kernels and why the current RBF kernel is implemented in such a way.

1. Matrix operations are generally quicker than vectorization. When computing the matrix of squared euclidean distances of a dataset X we found that the diagonals are not **0**. Even the diagonals are close to **0** if we take a very small length-scale then the diagonal will grow in size, **the diagonal needs to always be 0**. To handle this we multiply the covariance we a zero-diagonal matrix, the function is shown below:

```
def ieuclidean_distance_s(X, diagonals):
    Xsq = jnp.sum(jnp.square(X), 1)
    r2 = -2. * jnp.dot(X, X.T) + (Xsq[:, None] + Xsq[None, :])
    r2 = r2 * diagonals
    return r2
```

But why not just get the value of the diagonals to zero?: Using auto-gradient means the JAX library computes the gradients by back tracking through all operations, if we have data set of N entries, the diagonal will be of size N, the larger N get the longer it takes JAX to compute the gradient. **Note; Try and avoid setting entries when using JAX**

2. jit keyword: You will see this keyword frequently during implementation, it means Just in-time compilations, and JAX uses this to improve the speed computing the function and its gradients by compiling the function just before execution. Adding **@jit** compiled the function and improves its speed, in particular when used with JAX gpu.

3. partial keyword: Some function may take inputs that aren't jit applicable, such as objects, in this case we partially compile the function stating other arguments remain static. An example of this is as follows:

```
@partial(jit, static_argnums=(0,))
def ARDf(self, X, params):
    r = ieuclidean_distance_s(X / params["lengthscale"], self.diagonal)
    return params["variance"] * jnp.exp(-0.5 * r)
```

static_argnums details the index of the arguments that are static.

6 Rough Edges

As previously mentioned JAX is still being developed with bug fixes being done continuously. During the development of this library one particular issue occurred when calculating the gradients of an acquisition function and the log-likelihood.
For some reason JAX does not like squaring the euclidean distance within the rbf kernel, which produces *nan's* for the gradient. Overcoming this we simply do not take the square-root as the euclidean distance does, for example:

```
@jit
def euclidean_distance_s(X, X2):
    X1sq = jnp.sum(jnp.square(X), 1)
    X2sq = jnp.sum(jnp.square(X2), 1)
    r2 = -2. * jnp.dot(X, X2.T) + (X1sq[:, None] + X2sq[None, :])
    r2 = jnp.clip(r2, 0, jnp.inf)
    return r2
```

7 Finally

Finally, this library is not the finished article and will further developed in the future! If you have any question please send my an email at pw384@exeter.ac.uk.