Bayesian Machine Learning

Instructor: Tim G. J. Rudner

Homework 3
Due: Monday, October 31, 11:59pm via NYU Brightspace

This assignment involves experimenting with functions for the predictive distribution and marginal likelihood of a Gaussian process using the GPyTorch package. Please hand in your code. Try to make it succinct but commented.

If you use an ipython notebook, please submit it in ipynb format as well as in pdf format. If you use python scripts, please upload them to brightspace.

The installation instruction for GPyTorch can be found at here. You will need to have Python ($\geq$ 3.6 or greater) and PyTorch (v1.8.1 or greater) installed (version numbers crucial).

We suggest reading this tutorial for an intro to GPyTorch.

Gaussian Processes

Assume a dataset $\mathcal{D}$ of n input (predictor) vectors $X = \{\mathbf{x}_1, \ldots, \mathbf{x}_n\}$, each of dimension D, which index an $n \times 1$ vector of targets $\mathbf{y} = (y(\mathbf{x}_1), \ldots, y(\mathbf{x}_n))^\top$. If $f(\mathbf{x}) \sim \mathcal{GP}(\mu, k_{\boldsymbol{\gamma}})$, then any collection of function values $\mathbf{f}$ has a joint Gaussian distribution,

$$\mathbf{f} = f(X) = [f(\mathbf{x}_1), \ldots, f(\mathbf{x}_n)]^\top \sim \mathcal{N}(\boldsymbol{\mu}, K_{X,X})\,, \tag{1}$$

with mean vector and covariance matrix defined by the mean function and covariance kernel of the Gaussian process: $\boldsymbol{\mu}_i = \mu(\mathbf{x}_i)$, and $(K_{X,X})_{ij} = k_{\boldsymbol{\gamma}}(\mathbf{x}_i, \mathbf{x}_j)$, where the kernel $k_{\boldsymbol{\gamma}}$ is parametrized by $\boldsymbol{\gamma}$. Assuming additive Gaussian noise, $y(\mathbf{x})|f(\mathbf{x}) \sim \mathcal{N}(y(\mathbf{x}); f(\mathbf{x}), \sigma^2)$, the predictive distribution of the GP evaluated at the n_* test points indexed by X_*, is given by

$$\mathbf{f}_*|X_*, X, \mathbf{y}, \boldsymbol{\gamma}, \sigma^2 \sim \mathcal{N}(\mathbb{E}[\mathbf{f}_*], \mathrm{cov}(\mathbf{f}_*))\,, \tag{2}$$

$$\mathbb{E}[\mathbf{f}_*] = \boldsymbol{\mu}_{X_*} + K_{X_*,X}[K_{X,X} + \sigma^2 I]^{-1}(\mathbf{y} - \boldsymbol{\mu}_X)\,, \tag{3}$$

$$\mathrm{cov}(\mathbf{f}_*) = K_{X_*,X_*} - K_{X_*,X}^\top [K_{X,X} + \sigma^2 I]^{-1} K_{X,X_*}\,. \tag{4}$$

$K_{X_*,X}$, for example, represents the $n_* \times n$ matrix of covariances between the GP evaluated at X_* and X. $\boldsymbol{\mu}_X$ and $\boldsymbol{\mu}_{X_*}$ are mean vectors evaluated at X and X_*, and $K_{X,X}$ is the $n \times n$ covariance matrix evaluated at training inputs X. All covariance matrices implicitly depend on the kernel hyperparameters $\boldsymbol{\gamma}$. In practice, we are typically interested in the diagonal of Eq. (4) to characterize predictive uncertainty.

The marginal likelihood of the Gaussian process is given by

$$\log p(\mathbf{y}|\boldsymbol{\gamma}, X) \propto -(\mathbf{y} - \boldsymbol{\mu}_X)^\top (K_{\boldsymbol{\gamma}} + \sigma^2 I)^{-1}(\mathbf{y} - \boldsymbol{\mu}_X) - \log|K_{\boldsymbol{\gamma}} + \sigma^2 I|\,, \tag{5}$$

where we have used $K_{\boldsymbol{\gamma}}$ as shorthand for $K_{X,X}$ given $\boldsymbol{\gamma}$.

In the following questions, we will assume a mean function $\mu = 0$.

We will primarily consider six kernel functions:

$$k_{\text{BM}}(x, x') = \min(x, x') \tag{6}$$

$$k_{\text{RBF}}(x, x') = a^2 \exp(-\frac{||x - x'||^2}{2\ell^2}) \tag{7}$$

$$k_{\text{ARD}}(x, x') = a^2 \exp(-(x - x')^\top A^{-1}(x - x')) \, , A_{ij} = \delta_{ij}\ell_i^2 \tag{8}$$

$$k_{\text{OU}}(x, x') = a^2 \exp(-\frac{||x - x'||}{\ell}) \tag{9}$$

$$k_{\text{PER}}(x, x') = a^2 \exp(-\frac{2\sin^2(\frac{x-x'}{2})}{\ell^2}) \tag{10}$$

$$k_{\text{IQ}}(x, x') = (c + \frac{||x - x'||^2}{\ell^2})^{-1/2} \tag{11}$$

1. Prior Sampling
(10 marks)

   (a) (4 marks): For $\ell = 20$, $a^2 = 5$, draw a sample prior function from a Gaussian process evaluated at $X = \{1, 2, \ldots, 100\}$, $\mathbf{f} = f(X)$, with each of the above kernels, and a zero mean function. Create a plot and label each of the functions. Note that in this problem the inputs are one dimensional, $x \in \mathbb{R}^1$, and so k_{RBF} and k_{ARD} are equivalent.

   For this task we recomend using the class `ExactGPModel` defined in this tutorial.

   We have reproduced the `ExactGPModel` class here:

```
class ExactGPModel(gpytorch.models.ExactGP):
        def __init__(self, train_x, train_y, likelihood):
                super(ExactGPModel, self).__init__(train_x, train_y,
                                        likelihood)
                self.mean_module = gpytorch.means.ConstantMean()
                self.covar_module = gpytorch.kernels.ScaleKernel(
                                                gpytorch.kernels.RBFKernel())

        def forward(self, x):
                mean_x = self.mean_module(x)
                covar_x = self.covar_module(x)
                return gpytorch.distributions.MultivariateNormal(mean_x,
                                                covar_x)
```

   You can sample from the prior distribution by setting the training data for `train_x`, `train_y` parameters of `ExactGPModel`:

```
model = ExactGPModel(train_x, None, likelihood)
samples = model(test_data).sample(sample_shape=torch.Size(1000,))
```

   You can manually set a kernel parameter `<parameter_name>` by setting the `_<parameter_name>` attribute of the `gpytorch.Kernel` instance. For example you can create an RBF kernel with output scale 5 and length-scale 20 as follows:

```
kernel = gpytorch.kernels.ScaleKernel(gpytorch.kernels.RBFKernel())
kernel.outputscale = 5.
kernel.base_kernel.lengthscale = 20.
```

We recommend checking the hyper-parameter setting tutorial at this tutorial if you have any further issues. For your own kernels, you may instead wish to set parameters like

```
kernel.base_kernel.alpha.data = 2.
```

You can find a list and descriptions of the kernels implemented in GPyTorch here. Note that k_{BM}, k_{OU} and k_{IQ} are not implemented in the library. You will need to implement them yourself. See the source code for e.g. k_{poly} (here) and k_{PER} (here) for reference.

Here is an example implementation of RBF kernels:

```
class RBFKernel(gpytorch.kernels.Kernel):
    def __init__(self, **kwargs):
        super(RBFKernel, self).__init__(has_lengthscale=True, **kwargs)

    def postprocess_rbf(self, dist_mat):
        return dist_mat.div_(-2).exp_()

        def forward(self, x1, x2, diag=False, **params):
                x1_ = x1.div(self.lengthscale)
                x2_ = x2.div(self.lengthscale)
                return self.covar_dist(x1_, x2_, square_dist=True, diag=diag,
                      dist_postprocess_func=self.postprocess_rbf,
                 postprocess=True, **params)
```

Note that automatic relevance determination (ARD) is built within the covar distance function and can be specified with `ard_num_dims`.

(b) (6 marks): Comment on the behaviour of the distribution over prior functions as you vary the hyperparameters a, ℓ, and α, with both analytic and empirical support (plots of sample prior functions) for your reasoning.

2. Posterior Sampling
(12 marks)

Load the dataset $\mathcal{D}$ from `datatest.mat` . These data $\mathcal{D} = \{X, \mathbf{y}\}$ were generated from the model:

$$y(x) = f(x) + \epsilon \tag{12}$$
$$f(x) = x + x^2 - 14x^3 + x^4 + 1000\cos(x) \tag{13}$$
$$\epsilon \sim \mathcal{N}(0, 900^2) \tag{14}$$
$$\tag{15}$$

evaluated at $X = \{1, 2, \ldots, 15\}$ to form $\mathbf{y} = y(X)$.

The observed data are stored at the key `y`, and the data are indexed by inputs X stored at the key `x`; the test data are indexed by inputs X_* storred at the key `xstar`. You can load the data using `scipy.io.loadmat`:

```
D = loadmat("datatest.mat")
x = D["x"]
y = D["y"]
x_star = D["xstar"]
x = torch.tensor(x)
...
```

Our goal is to recover the true noise-free latent $f(x)$ at a set of locations X_* given by the variable `xstar`. Although this seems like an almost intractable problem – there is non-trivial variation in the true function and a relatively large amount of noise – and we would have *no idea* of what to guess as a parametric form for this function, we can almost exactly recover the true function using a Gaussian process with an RBF kernel even with 15 training points!

Model the data as being generated by a Gaussian process noise free function $g(x)$ plus additive Gaussian noise with noise variance 900^2. Use GPs with OU and RBF kernel functions, with length-scale $\ell = 5$ and $a = \text{stdev}[\mathbf{y}]$, showing the mean predictions, and 95% of the posterior predictive mass, evaluated at the test points. Create a plot that contains (1) the data points; (2) the predictive mean; and (3) 95% of the predictive probability mass $(2 \times \text{stdev}[f(x_*)], x_* \in X_*)$ for GPs with each of these two kernels. Before fitting the data, be sure to subtract the empirical mean from the data, to fit your modelling assumptions. When you make predictions with the GP, add back on the empirical mean. Also, on a separate plot, show two sample functions from the posterior function of the Gaussian process for each kernel (4 sample functions in total), evaluated at `xstar`, alongside the observed data points.

(Optional - 1 bonus mark): See what happens as you vary the length-scale ℓ and signal variance a^2 parameters. Generate datasets from this fourth degree polynomial with more or less noise. Try different functional forms for the noise free functions. See how well the Gaussian process with various kernels can reconstruct functions with discontinuities, sudden changes, etc. What happens as you get more data? Less data?

3. Learning the Kernel Hyperparameters
(20 marks)

   (a) Use the marginal likelihood gpytorch.mlls.ExactMarginalLogLikelihood with ExactGP-Model to learn the kernel hyperparameters (including noise variance) on the problem in question 2. Plot the resulting posterior predictive mean functions for GPs with the RBF kernel and OU kernel using the optimized hyperparameters versus previously used hyperparameters ($\ell = 5$ and $a = \text{stdev}[\mathbf{y}]$) in question 2. Explain why or why not there may be any significant differences in this particular case.

   (b) Load `2dfunc.mat`. This data has two input dimensions and a scalar output. Visualise the data, for example using `matplotlib`'s `plot_surface`. Fit the data using a GP using the ARD kernel. Comment on the fit. How much noise is in the data? Instead fit the data without ARD and comment on the difference. What is the relative probability of a GP with the `ard` and `rbf` kernels (note in the RBF kernel the length-scale is the same for each dimension)?

   (c) (Optional - 1 bonus mark): Generate data from a GP with a given kernel and known ground truth hyperparameters. See how accurately you can recover these ground truth parameters through marginal likelihood optimization, as we vary the number of data-points and the noise in the data. Try comparing with any other approach to learning the kernel hyperparameters (e.g. looking at the empirical autocorrelations). See how sensitive good performance is to the settings of hyperparameters (by manually adjusting these parameters) under different settings.

Hint - applies to both 2) and 3): You may get better results out of rescaling the inputs to $[0, 1]$ and the outputs to have zero mean and standard deviation one. It's also possible to do

all optimization on the original scale. But, if you do use rescaling, please remember to plot everything on the original scale.

4. Learning the Kernel
(20 marks)

   Load `airline.mat`. Create a composition of kernels (any discussed in class – except the spectral mixture kernel) to model these data. Justify your choices. Show an extrapolation 20 years outside of the training data, with both the predictive mean and 95% of the predictive posterior mass. Compare your predictions at testing locations `xtest` to the withheld data `ytest`.

   Hint, you can combine kernels like

```
covar_module = ScaleKernel(RBFKernel()) + ScaleKernel(MaternKernel())
```