Machine Learning in the Physical World

Gaussian Processes

Carl Henrik Ek

October 12, 2023

Abstract

In this work-sheet we will look at Gaussian processes or GPs. These are non-parametric models that allows us to reason not about parameters of the function that we want to infer but in terms of more interpretable components as the characteristics of the function. GPs are used exentsively for many different applications, for reinforcement learning Deisenroth et al., 2015 for modelling complex dynamical systems Kaiser et al., 2018 and for black-box optimisation Snoek et al., 2012. As an example of the latter Gaussian processes played a central role in how AlphaGo managed to beat the world GO champion Chen et al., 2018. In this unit the last use case is going to be one focus for us where we will use GPs as a surrogate model for an underlying physical system.

Last week we looked at linear regression as an example of how to think about probabilistic models. However, our hypothesis space was rather limited and we could only work with linear functions. This week we will look at the same model again, but now use a much richer set of hypothesis, Gaussian processes. The most natural way to think of Gaussian processes are as distributions over the space of functions. Importantly they have this wonderful characteristic that they put non-zero probability over every continous function. This seems like a strange concept at first as we naturally then think about *parametrisations* of functions. What would be a sufficiently rich parametrisation that includes all continuous functions? This is where the concept of non-parametrics comes in, we will not think about functions as parametric objects but as non-parametric objects where we specifically describe the characteristic of the *output* of the function. This is the weird and beautiful world of non-parametrics and getting some clarity about this is the aim of this lab.

1 Model

We are going to work with GPs in exactly the same setting as in the case of linear regression. This means we are interested in the task of observing a data-set $\mathcal{D} = \{x_i, y_i\}_{i=1}^{N}$ where we assume the following relationship between the variates,

$$y_i = f(x_i). \tag{1}$$

Our task is to infer the function $f(\cdot)$ from $\mathcal{D}$. We will make the same assumptions as in the linear regression case which leads us to the following likelihood function,

$$p(\mathbf{y}|f(\mathbf{x})) = \prod_{i=1}^{N} p(y_i|f(x_i)), \tag{2}$$

where $f(x_i)$ is the output of the function at location x_i. As you can see the generative model here is slightly different as we have not made any assumptions on the structure of the function. But lets for a minute assume that we had access to a distribution that encoded our belief in a general function $p(f)$ we could then formulate the joint distribution such as,

$$p(\mathbf{y}, f|\mathbf{x}) = p(\mathbf{y}|f, \mathbf{x})p(f). \tag{3}$$

We will now proceed to define $p(f)$.

2 Prior

Our task here is to define a prior distribution over the space of functions, this means a distribution $p(f)$ such that you can evaluate the probability of any function. This is a very strange concept so we are going to re-write this in a different and non-parametric way. Lets think of what a function actually is, or rather, how we can observe a function. What we can see is the *values* of a function, so how about we actually parametrise the function directly by its output values? To do so let us alter our representation slightly, lets write down the output of the function as a variable,

$$f_i = f(x_i). \tag{4}$$

Now because we do not know what f_i is we are going to treat this as a random variable. In specific we are going to make the assumption that this variable is Gaussian such as,

$$f_i \sim \mathcal{N}(\mu(x_i), k(x_i)), \tag{5}$$

where $\mu(x_i)$ and $k(x_i)$ are functions that evaluates the mean and the variance of the random variable as a function of the input value. Now a function can be evaluate in more than one place, so what we will say is that not only is each of the function values normally distributed, *they are jointly normally distributed.* What this means is that if we now have a set of input locations $\mathbf{x} = [x_1, \ldots, x_N]$ they induce the following set of random variable,

$$\begin{bmatrix} f_1 \\ f_2 \\ \vdots \\ f_N \end{bmatrix} \sim \mathcal{N}\left(\begin{bmatrix} \mu(x_1) \\ \mu(x_2) \\ \vdots \\ \mu(x_N) \end{bmatrix}, \begin{bmatrix} k(x_1,x_1) & k(x_1,x_2) & \cdots & k(x_1,x_N) \\ k(x_2,x_1) & k(x_2,x_2) & \cdots & k(x_2,x_N) \\ \vdots & \vdots & \cdots & \vdots \\ k(x_N,x_1) & k(x_N,x_2) & \cdots & k(x_N,x_N) \end{bmatrix} \right), \tag{6}$$

where $k(\cdot,\cdot)$ is a function that computes the covariance between of the function values at two locations[1] As it is possible to evaluate the function at infinitely many locations the cardinality of the set is actually the *uncountable infinity.* Now we will use one very important result from the Gaussian identities namely the *marginal property* of the Gaussian distribution. If we want the marginal distribution of a sub-set of variables where the whole set is jointly Gaussian we only need to "pick" the corresondping parts of the mean and the co-variance matrix. What this means is that somewhere there exists an infinite Gaussian distribution, where each dimension of the distribution corresponds to a random varible but we can project this to *any* finite subspace by just evaluating or picking the relevant part of the mean and co-variance. This infinite dimensional object is called a *Gaussian process.* This marginal property is incredibly powerful as it implies that whatever we do in the finite case can be considered a projection from the infinite, therefore we can think of Eq. 6 as a distribution over the function evaluated at a finite set of locations $\mathbf{x}$ which are consistent with the whole infinite input space.

Using a Gaussian process as a prior over the function implies that we need to specify a mean function $\mu(\cdot)$ and a co-variance function $k(\cdot,\cdot)$. For now let us use the constant zero function as mean and focus on the co-variance function. The co-variance function describes how the function outputs varies together as a function of their inputs. Its not free to be any function as the matrix evaluate on any subset of the input space still has to be a valid covariance matrix. The class of functions is the same as the class of valid kernel functions, i.e. it need to define an inner-product space if you are interested in generalisations of Euclidean spaces you can read more about this here. Without going into too much depth on what consistutes a valid covariance function the main things that you need to think about is that the matrix needs to be positive-definite, i.e. all its eigenvalues needs to be positive[2]. One simple function that is guaranteed to do this is what is called the exponentiated quadratic, the squared exponential or the radial-basis kernel,

$$k(x,x') = \sigma^2 e^{\frac{-(x-x')^{\mathrm{T}}(x-x')}{\ell^2}} \tag{7}$$

where σ^2 is referred to as the variance and ℓ as the length-scale of the function.

[1] through the definition of the variance it therefore needs to compute the variance of the random varialbe if both arguments are the same.

[2] you can show this by using the spectral theorem.

2.1 Implementation

In order to work with a GP it makes sense to write a bit more sensible structured code and break up a few things into functions. The first thing we want to implement is a function that allows us to compute the covariance matrix. It could be nice to try and modularise this so that you can easily use your code with several different covariance functions.

Code

```
from scipy.spatial.distance import cdist

   def rbf_kernel(x1, x2, varSigma, lengthscale):
       if x2 is None:
           d = cdist(x1, x1)
       else:
           d = cdist(x1, x2)

       K = varSigma*np.exp(-np.power(d, 2)/lengthscale)

       return K
```

Now when we have written the our covariance function we are ready to sample from our prior. Now let us try and sample from a GP defined with a zero mean and the exponentiated quadratic defined above. We will first look at functions with one-dimensional inputs so the infinite process will be indexed by $\mathbb{R}$. The first thing we will do is to decide which marginal of the real-line that we will look at. We will refer to this as the index set. Lets now compute this marginal of the process, which being finite is just a Gaussian distribution, and plot a couple of samples from this.

Code

```
# choose index set for the marginal
   x = np.linspace(-6, 6, 200).reshape(-1, 1)
   # compute covariance matrix
   K = rbf_kernel(x, None, 1.0, 2.0)
   # create mean vector
   mu = np.zeros(x.shape)

   # draw samples 20 from Gaussian distribution
   f = np.random.multivariate_normal(mu.flatten(), K, 20)

   fig = plt.figure()
   ax = fig.add_subplot(111)
   ax.plot(x, f.T)
```

Run the code above and test a few different parameters for the covariance function, how does the behaviour of the samples change? The important thing to understand is that even if the process is infinite, as are the number of values the function can be evaluated across, due to the consistency of a GP we can decide to only look at a finite subset of the process. This is what we define using `x`. Test to increase and decrease the cardinality of the index set.

Sampling from the prior is very important as it allows us to see what assumptions we can encode with the parameters. Importantly, the GP places non-zero probability on every function so if you sample for long enough everything will appear, with a few samples you are only seeing the most likely things.

Now lets try to change the covariance function and see how the samples change, lets implement three more covariance functions, a white-noise, a linear and a periodic covariance.

Code

```
def lin_kernel(x1, x2, varSigma):
    if x2 is None:
        return varSigma*x1.dot(x1.T)
    else:
        return varSigma*x1.dot(x2.T)

def white_kernel(x1, x2, varSigma):
    if x2 is None:
        return varSigma*np.eye(x1.shape[0])
    else:
        return np.zeros(x1.shape[0], x2.shape[0])

def periodic_kernel(x1, x2, varSigma, period, lenthScale):
    if x2 is None:
        d = cdist(x1, x1)
    else:
        d = cdist(x1, x2)

    return varSigma*np.exp(-(2*np.sin((np.pi/period)*d)**2)/lengthScale**2)
```

Generate samples from GPs defined by the different kernels and see what type of behaviour the prior now encodes. There are several operations you are allowed to do to a kernel while in Bishop, 2006 there is a list of operations. What you can try is to add different kernels together, you can also multiply them together. Try this out and look at the samples. The key thing that I want you to see is how much larger the class of beliefs that you can specify with GPs compared to the lines that we did in the previous lab[3].

3 Posterior

Now when we are happy about our prior it is time to move to the posterior distribution. As the GP is a non-parametric model where we dirctly model the output rather than a specific set of parameters the posterior takes a form much more similar to the predictive posterior in the linear regression case. Deriving the posterior is a very simple proceedure making use of the definition of the GP. In specific a GP is defined as a infinite collection of random variables which are all *jointly* Gaussian distributed. So lets make use of this. Lets assume that we have observed data $\mathcal{D} = \{y_i, x_i\}_{i=1}^{N}$ and now we want to predict what the output of the function is at locations $\mathbf{x}_* = \{x_{*1}, \dots, x_{*M}\}$. Now the union of the input locations specifies the index set of the infinite process meaning that we can write up the joint distribution as,

$$\begin{aligned} &p(f_1, \dots, f_N, f_{*1} \dots, f_{*M} | \mathbf{x}, \mathbf{x}_*, \boldsymbol{\theta}) \\ &= \mathcal{N}\left(\begin{bmatrix} \mu(x_1) \\ \vdots \\ \mu(x_N) \\ \mu(x_{*1}) \\ \vdots \\ \mu(x_{*M}) \end{bmatrix}, \begin{bmatrix} k(x_1, x_1) & \cdots & k(x_1, x_N) & k(x_1, x_{*1}) & \cdots & k(x_1, x_{*M}) \\ \vdots & \ddots & \vdots & \vdots & \ldots & \vdots \\ k(x_N, x_1) & \cdots & k(x_N, x_N) & k(x_N, x_{*1}) & \cdots & k(x_N, x_{*M}) \\ k(x_{*1}, x_1) & \cdots & k(x_{*1}, x_N) & k(x_{*1}, x_{*1}) & \cdots & k(x_{*1}, x_{*M}) \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ k(x_{*M}, x_1) & \cdots & k(x_{*M}, x_N) & k(x_{*M}, x_{*1}) & \cdots & k(x_{*M}, x_{*M}) \end{bmatrix} \right) \end{aligned} \quad (8)$$

[3] the plots that you have above should be compared with the plots that you did by sampling from the prior in the previous lab

where $\mu(\cdot)$ and $k(\cdot,\cdot)$ are the mean and covariance function and θ are the parameters of the latter. Now this is just the prior distribution over indexed at the subset of the data and where we are interested in evaluating the function. We are now interested in reaching the posterior distribution over the index set $\mathbf{x}_*$. Maybe this feels a bit daunting at first but lets think of what we actually have and what we want to reach. We have a joint distribution and we want to get the conditional distribution over the index set $\mathbf{x}_*$ given the index set $\mathbf{x}$. If we apply the product rule and factorise the joint distribution we get,

$$p(\mathbf{f}, \mathbf{f}_* | \mathbf{x}, \mathbf{x}_*, \boldsymbol{\theta}) = p(\mathbf{f}_* | \mathbf{f}, \mathbf{x}, \mathbf{x}_*, \theta) p(\mathbf{f} | \mathbf{x}, \theta). \tag{9}$$

Now this is just the factorisation of a Gaussian distribution and its form is something that we proved in the previous works sheet. So we can just use those results and write down what the conditional distribution that we are interested in is.

$$p(\mathbf{f}_* | \mathbf{f}, \mathbf{x}, \mathbf{x}_*, \theta) = \mathcal{N}(\mu_{\mathbf{x}_*|\mathbf{x}}, K_{\mathbf{x}_*|\mathbf{x}}) \tag{10}$$

$$\mu_{\mathbf{x}_*|\mathbf{x}} = k(\mathbf{x}_*, \mathbf{x}) k(\mathbf{x}, \mathbf{x})^{-1} \mathbf{f} \tag{11}$$

$$K_{\mathbf{x}_*|\mathbf{x}} = k(\mathbf{x}_*, \mathbf{x}_*) - k(\mathbf{x}_*, \mathbf{x}) k(\mathbf{x}, \mathbf{x})^{-1} k(\mathbf{x}, \mathbf{x}_*), \tag{12}$$

where we have assumed that the mean function $\mu(\cdot)$ is the constant zero function.

Reflections

Compare this with linear regression can you see the similarity? Compute each of the terms of the posterior covariance individually and plot them using `plt.imshow` do they make sense? One way to consider the different terms is that the posterior covariance is made up of the prior over the new data *corrected* by the information in the data that we have already seen.

Now let us generate some data and sample from this distribution. Lets generate some data from a noisy sine wave.

Code

```
N = 5
x = np.linspace(-3.1,3,N)
y = np.sin(2*np.pi/x) + x*0.1 + 0.3*np.random.randn(x.shape[0])
x = np.reshape(x,(-1,1))
y = np.reshape(y,(-1,1))
x_star = np.linspace(-6, 6, 500)
```

Now we are likely to use compute the posterior quite a few times for different index-sets so its probably worthwhile to dedicate a function for this. Something like the one below would probably be suitable.

Code

```
    def gp_prediction(x1, y1, xstar, lengthScale, varSigma, noise):

        k_starX = rbf_kernel(xstar,x1,lengthScale,varSigma,noise)
        k_xx = rbf_kernel(x1, None, lengthScale, varSigma, noise)
        k_starstar = rbf_kernel(xstar,None,lengthScale,varSigma,noise)

        mu = k_starX.dot(np.linalg.inv(k_xx)).dot(y1)
        var = k_starstar - (k_starX).dot(np.linalg.inv(k_xx)).dot(k_starX.T)

        return mu, var, xstar
```

Once we have the the functionality to compute predictive posterior we can now sample from this.

Code

```
Nsamp = 100
mu_star, var_star, x_star = gp_prediction(x1, y1, x, lengthScale, varSigma, noise)
fstar = np.random.multivariate_normal(mu_star, var_star, Nsamp)
fig = plt.figure()
ax = fig.add_subplot(111)
ax.plot(x_star, f_star.T)
ax.scatter(x1, y1, 200, 'k', '*', zorder=2)
```

If things works out as it should you should achieve a plot looking something similar to the one in Figure 1, As you can all the samples from the function goes through the data.

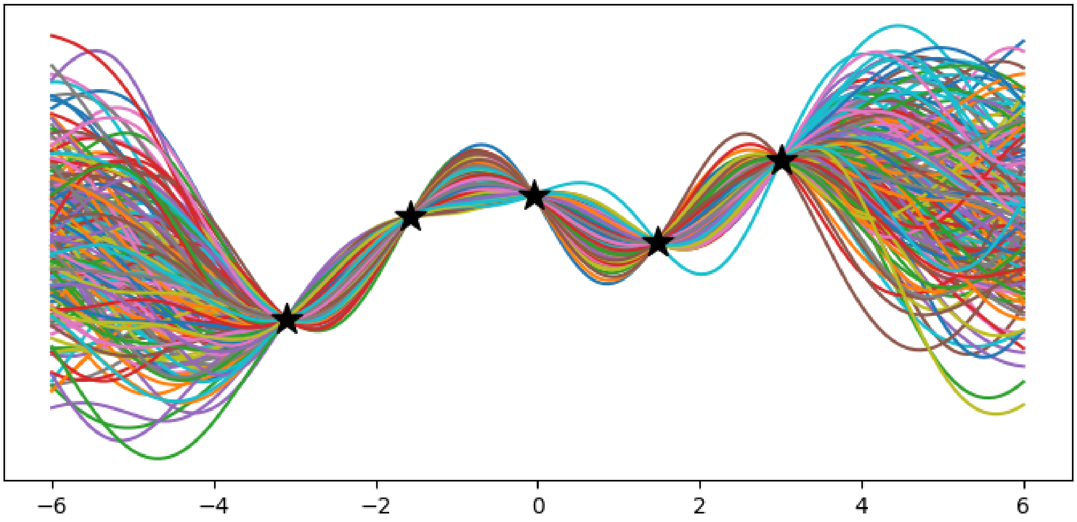

Figure 1: *The above image shows samples from the predictive posterior of a Gaussian process prior specified by a exponentiated quadratic kernel. As you can see all the samples passes exactly through the data.*

Rather than plotting the samples we can also show the mean and the variance around each point in the index set. This should generate a plot looking something similar to the one shown in Figure 2.

$$
\begin{aligned}
&p(y_1,\ldots,y_N,f_{*1}\ldots,f_{*M}|\mathbf{x},\mathbf{x}_*,\boldsymbol{\theta})\\
&=\mathcal{N}\left(\begin{bmatrix}\mu(x_1)\\ \vdots\\ \mu(x_N)\\ \mu(x_{*1})\\ \vdots\\ \mu(x_{*M})\end{bmatrix},\begin{bmatrix}k(x_1,x_1)+\frac{1}{\beta} & \cdots & k(x_1,x_N) & k(x_1,x_{*1}) & \cdots & k(x_1,x_{*M})\\ \vdots & \ddots & \vdots & \vdots & \cdots & \vdots\\ k(x_N,x_1) & \cdots & k(x_N,x_N)+\frac{1}{\beta} & k(x_N,x_{*1}) & \cdots & k(x_N,x_{*M})\\ k(x_{*1},x_1) & \cdots & k(x_{*1},x_N) & k(x_{*1},x_{*1}) & \cdots & k(x_{*1},x_{*M})\\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots\\ k(x_{*M},x_1) & \cdots & k(x_{*M},x_N) & k(x_{*M},x_{*1}) & \cdots & k(x_{*M},x_{*M})\end{bmatrix}\right)
\end{aligned}
\tag{13}
$$

This leads to a tiny change to the predictive posterior in Eq. 12 as we now only need to include the

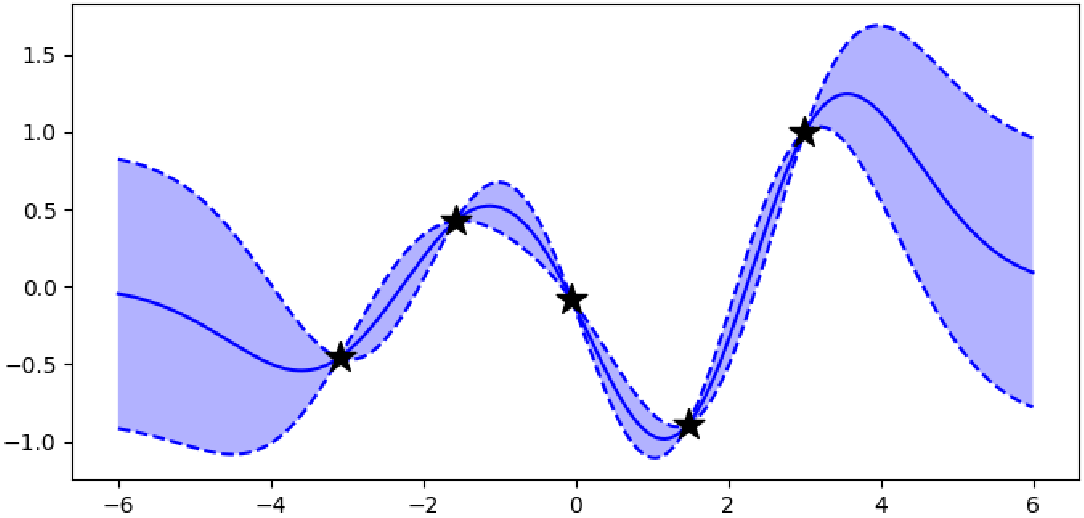

Figure 2: *The above image shows the mean prediction and one standard deviation around the mean. You can generate the plot by using the nice command* ***`np.fill_between()`****.*

independent noise term as,

$$p(\mathbf{f}_*|\mathbf{y},\mathbf{x},\mathbf{x}_*,\theta) = \mathcal{N}(\mu_{\mathbf{x}_*|\mathbf{x}}, K_{\mathbf{x}_*|\mathbf{x}}) \tag{14}$$

$$\mu_{\mathbf{x}_*|\mathbf{x}} = k(\mathbf{x}_*,\mathbf{x})\left(k(\mathbf{x},\mathbf{x}) + \frac{1}{\beta}\mathbf{I}\right)^{-1}\mathbf{y} \tag{15}$$

$$K_{\mathbf{x}_*|\mathbf{x}} = k(\mathbf{x}_*,\mathbf{x}_*) - k(\mathbf{x}_*,\mathbf{x})\left(k(\mathbf{x},\mathbf{x}) + \frac{1}{\beta}\mathbf{I}\right)^{-1}k(\mathbf{x},\mathbf{x}_*). \tag{16}$$

Think about this and justify to yourself that this makes sense. If you now generate samples from the above distribution you will see that the samples no longer necessarily goes through the data. We are now allowing some of the variations in the data to be explained by noise instead of by signal. What you can do now is to play around with different covariance functions and different parameters etc. and get an intuitive feeling for what we have done.

4 Summary

Hopefully you have gotten to the end of this worksheet and realised how simple Gaussian processes actually are. They do really cool stuff and they are called all-sorts of fancy names but really at the core the only thing it really is is one big Gaussian where the index set have been associated with and interesting interpretation as an ordered set. If this hasn't become clear just yet, take a step back and play around with them a bit more, alter the covariance function, draw some samples write up the derivation. Hopefully after some time you will feel that this all makes sense. So even though the result looked quite different, what new stuff have we actually done? Very little. Think about it, we have walked through exactly the same procedure as in linear regression. The process and the language is all the same, and that is the real benefit of thinking about machine learning in a Bayesian perspective, its a consistent story. If you look at the equations they are actually the same due to both methods using Gaussian priors and likelihoods. However, I hope that you feel how much richer this prior is and that is because we place the prior over completely different things.

Now we have the tools that will allow us to really start looking at the core of this unit, how we can build machine learning that are connected to real physical systems. In next weeks lab we will use Gaussian processes as a building block in optimisation. In specific we will have underlying explicitly unknown function that we want to find the extremum of. What we will do is to use a Gaussian process as a surrogate to the real unknown function. Given the probablistic formulation this will allow us to input our knowledge in the formulation of the prior distribution.

References

Bishop, Christopher M. (2006). *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Secaucus, NJ, USA: Springer-Verlag New York, Inc. ISBN: 0387310738.

Candela, Joaquin Quiñonero and Carl Edward Rasmussen (2005). "A Unifying View of Sparse Approximate Gaussian Process Regression." In: *Journal of Machine Learning Research* 6, pp. 1939–1959.

Chen, Yutian et al. (2018). "Bayesian Optimization in Alphago." In: *CoRR*. arXiv: `1812.06855 [cs.LG]`.

Deisenroth, Marc Peter, Dieter Fox, and Carl Edward Rasmussen (Feb. 2015). "Gaussian Processes for Data-Efficient Learning in Robotics and Control." In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* stat.ML.2, pp. 408–423.

Hensman, James, N Fusi, and Neil D Lawrence (2013). "Gaussian Processes for Big Data." In: *Uncertainty in Artificial Intelligence*.

Kaiser, M., C. Otte, T. Runkler, and C. H. Ek (2018). "Bayesian Alignments of Warped Multi-Output Gaussian Processes." In: *Advances in Neural Information Processing Systems 32, [NIPS Conference, Montreal, Quebec, Canada, December 3 - December 8, 2018]*.

Snoek, Jasper, Hugo Larochelle, and Ryan P Adams (2012). "Practical Bayesian Optimization of Machine Learning Algorithms." In: *Advances in Neural Information Processing Systems 25*. Ed. by F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger. Curran Associates, Inc., pp. 2951–2959.

Titsias, Michalis (2008). *Variational Model Selection for Sparse Gaussian Process Regression*. Tech. rep. University of Manchester.

5 Practical Gaussian processes

Now you have seen the basics of Gaussian processes. In some ways you have seen the "pretty" side of Gaussian processes, they also have a dark side, the side that you have to work on if you would want to apply these to larger data-sets and things that you have to do to make the computations better conditioned. This information is just here for those of you who are interested and want to know what the inner-workings of how to implement GPs. There exists many packages for GPs and it for the project you are much more likely to use one of those rather than implement everything on your own. But at least I find it important and interesting to know the things that you have to do in order to make these methods work.

5.1 Numerical Issues

While Gaussian processes are mathematically nice we quickly run into numerical issues when we implement them on finite precision hardware. In order to make the learning more stable we often have to think quite carefully about how we do the computations. What we try to exploit in doing so is that the problem that we work on is actually quite structured. Most things that we do deals with manipulating the covariance matrix. The covariance matrix is in a class of matrices that are called positive-definite matrices meaning the are symmetric, full rank and with all eigen-values positive. This we can exploit to make computations better conditioned.

We often use what is called the Cholesky decompostion to make better conditioned computations with positive definite matrices. The Cholesky decomposition allows us to write a positive definite matrix as a product of a lower-triangular matrix and its transpose,

$$\mathbf{K} = \mathbf{L}\mathbf{L}^{\mathrm{T}}$$

The easiest example of this is the computation of the `log determinant` in the marginal log-likelihood. Lets look at an example of this,

$$\log\det\mathbf{K} = \log\det\left(\mathbf{L}\mathbf{L}^{\mathrm{T}}\right) = \log\left(\det\mathbf{L}\right)^2$$

Now we have to compute the determinant of the Cholesky factor $\mathbf{L}$, this turns out to be very easy as the determinant of a upper/lower diagonal matrix is the product of the values on the diagonal,

$$\det\mathbf{L} = \prod_i^N \ell_{ii}.$$

If we put everything together we get this,

$$\log\det\mathbf{K} = \log\left(\prod_i^N \ell_{ii}\right)^2 = 2\sum_i^N \ell_{ii}.$$

So the `log determinant` of the covariance matrix is simply the sum of the diagonal elements of the Cholesky factor and from our first classes in Computer science we know that taking the product of lots of values of different scale is an operation that really destroys precision.

You can also use the Cholesky factors in order to do address the term that includes the inverse and making this better conditioned. What we will do is to solve the inverse by solving two systems of linear equations, both who have already been made into upper and lower triangular form therefore being trivial to solve. This might sounds a bit confusing but when you see the math hopefully it becomes obvious. Lets write down a system of linear equations,

$$\mathbf{A}\mathbf{x} = \mathbf{b}. \tag{17}$$

The brute-force way to solve this would be as,

$$\mathbf{x} = \mathbf{A}^{-1}\mathbf{b},$$

but if we know that A is positive-semi definite we can exploit this using the Cholesky factors. Lets re-write the system using this knowledge,

$$\mathbf{L}\mathbf{L}^{\mathrm{T}}\mathbf{x} = \mathbf{b}. \tag{18}$$

The matrix $\mathbf{L}$ is now a lower triangular matrix so we can instead look at two system of equations that we can solve, one using forward substitution and then finally one using backward substition. First lets re-write the right-hand side of the equation,

$$\mathbf{b} = \mathbf{L}\mathbf{y}.$$

We can now easily solve for $\mathbf{y}$ by exploiting the lower-triangular structure,

$$\begin{array}{ccccccccc} \ell_{1,1}y_1 & & & & & & & = & b_1 \\ \ell_{2,1}y_1 & + & \ell_{2,2}y_2 & & & & & = & b_2 \\ \vdots & & & & & & & & \\ \ell_{n,1}y_1 & + & \ell_{n,2}y_2 & + & \dots & + & \ell_{n,n}y_n & = & b_n, \end{array} \tag{19}$$

by using $y_1 = \frac{b_1}{\ell_{1,1}}$ and then use this results to solve,

$$y_2 = \frac{b_2 - \ell_{2,1}y_1}{\ell_{2,2}},$$

we can then continue to solve for all of $\mathbf{y}$ from the top to the bottom, this process is called *forward substitution.* This is an intermediate result that we can use to find $\mathbf{x}$. To see how to do this we bring back the left-hand side of the equation,

$$\mathbf{L}\mathbf{L}^{\mathrm{T}}\mathbf{x} = L\mathbf{y} \tag{20}$$

$$\mathbf{L}^{\mathrm{T}}\mathbf{x} = \mathbf{y}. \tag{21}$$

Now we have the same type of system as we had in the previous case where we have a matrix of coefficients that is now upper-triangular being the transpose of the lower-triangular Cholesky factor,

$$\begin{array}{ccccccccc} \ell_{n,1}x_1 & + & \ell_{n,2}x_2 & + & \dots & + & \ell_{n,n}y_n & = & y_1 \\ \ell_{2,1}x_1 & + & \ell_{2,2}x_2 & & & & & = & y_2 \\ \vdots & & & & & & & & \\ \ell_{1,1}x_n & & & & & & & = & y_n, \end{array} \tag{22}$$

so we can now use the same approach to solve for $\mathbf{x}$ just going from the bottom row upwards instead which is called *back-substitution.* So by doing this approach we have avoided solving the inverse of a matrix and instead found the solution by solving two simpler systems of linear equations by using the Cholesky factors. Obviously we still need to find the decomposition itself and in total the two paths to the solution have the same complexity it is just that the latter is much more computationally stable because we are exploiting the structure of the problem.

5.1.1 Tricks of the trade

Sometimes, whatever you are going to have issues with the co-variance matrix being massively ill-conditioned. This happens when the range of eigen-values are really large. What you can do in these scenarios is to add a small value to the diagonal to the co-variance matrix before you perform anything involving the inverse. This will in effect just make the function values more independent. This is fine to do as, and will if it has any effect at all, be recovered from as the marginal log-likelihood has the `log-determinant` term which penalises this effect. What you can do is to put whatever code that you have involving the co-variance matrix in a `try catch` statement and then keep adding a small diagonal matrix till the computation works. If you look into most GP libraries you will see some variant of this.

5.2 Computational Issues

Looking at the marginal likelihood of the Gaussian process it requires inverting a matrix that is the size of the data. As we have already seen this numerically tricky due to the structure of the co-variance matrix. It is also a very expensive process of cubic complexity which severely limits the size of data-sets that we can use. There has been a lot of work over the last few decades focusing on this, way too much to outline in this report. A good paper to start with is this one Candela et al., 2005. The main idea behind most work is to reduce the degrees of freedom of the problem and say that rather than using all the data-points that you see as a degree of freedom assume that there is a smaller set of points, usually referred to as *inducing points* that are sufficient to describe all the other points. In practice what this means is that we have a set of N function values f that we want to describe and we are going to use a set of M *inducing* function u to describe them. Due to the marginal property of the Gaussian process this means that we can now write the joint distribution as[4],

$$p(\mathbf{f}, \mathbf{u}) = \mathcal{N}\left(\begin{bmatrix} f \\ \mu_u \end{bmatrix}, \begin{bmatrix} K_{ff} & K_{fu} \\ K_{uf} & K_{uu} \end{bmatrix}\right).$$

We can then use the chain rule and write the joint such as,

$$p(\mathbf{f}, \mathbf{u}) = p(\mathbf{f} \mid \mathbf{u})p(\mathbf{u}).$$

The first term on the left-hand side of the equation is the normal predictive Gaussian distribution which will have the form,

$$p(\mathbf{f} \mid \mathbf{u}) = \mathcal{N}\left(K_{fu}K_{uu}^{-1}\mathbf{u}, K_{ff} - K_{fu}K_{uu}^{-1}K_{uf}\right).$$

In the expression above we are only inverting an M matrix rather than the N that we did previously. So far we haven't done any approximation it is just following the rules of probability. What we will do is that rather than seeing $\mathbf{u}$ as just another set of locations, we will see them as the parameters that describe the GP prior and we will use the marginal log-likelihood to directly optimise them. There are several different approaches to doing this, one very popular is to doing this using a variational bound Titsias, 2008 which can be extended to allow for very large data-set Hensman et al., 2013.

This was just a small little hint towards the very active field of Gaussian process approximations and the tricks that you can do to apply these to large volumes of data. This is a very active research field in itself and way beyond what we can cover in this course. We added this here so that you can have an idea to how things are done and if you are interested in looking at these models beyond the course this might provide a starting point for this.

[4] where I have omitted the input locations for clarity.