COMS30007 - Machine Learning

Bayesian Optimisation

Carl Henrik Ek

Week 7

Abstract

This week we are going to make use of our previous knowledge in modelling and use it for black-box optimisation. The idea here is that we have a function that we want to optimise, i.e. find the minima[1], the tricky thing is that we do not know the form of the function but we can evaluate it.

Lets assume that we have a function $f(\mathbf{x})$ that is *explicitly* unknown that we want to find the minima of. We will further assume that it is possible to evaluate the function but that each evaluation is expensive. This means that the problem that we have on our hands is to *search* the input domain for the extreme point but do so in a manner that we minimise the number of evaluation that we make of the function. One approach to address this type of problem is to use a technique called *Bayesian Optimisation* and that is the focus of this lab.

Searching for the extremum of explicitly unknown functions is problem that appears in many applications. The first use of Bayesian methods for approaching this is usually attributed to [1] a Lithuanian mathematician. This important work was slightly overlooked at the time but recently it has gotten the attention that it deserves. The reason for this is that with increasingly complicated models with enormous cost for training being able to efficiently utilise the data have become very important. The use of Bayesian optimisation for learning how to set parameters in complicated unstructured models[2] is often attributed to [2] who really put these types of techniques at the forefront of modern machine learning. Even though the are used everywhere this is often not reported particularly well, as an example it took several years for the authors of AlphaGo to properly publish and discuss the importance of Bayesian optimisation for their task [3].

The main part of a Bayesian optimisation system is a loop where we in an iterative manner decided on new locations to evaluate the objective function. The two components of the loop are a surrogate model of the function which describes how we believe the function looks in every part of the domain and a acquisition function which decides based on our current belief of what the function is where to sample next. Importantly this makes it key to have uncertainty in our system as we need to have a belief about what the function value is everywhere. As we have already looked at Gaussian processes as a rich class of function priors we will use them for our surrogate model. Lets begin by making the problem more concrete.

Lets assume that we want to find the minima of a function $f(x)$,

$$x_* = \text{argmin}_x f(x) \tag{1}$$

we will at each time have observed a set of values of the function at specific function locations, we will refer to this as the data $\mathcal{D} = \{\mathbf{x}, \mathbf{f}\}$. Furthermore at any point we have a current best estimate, we will refer to this location in the data space as x_* and its function value as $f(x_*)$. We will use a surrogate model to describe our beliefs about the function f. We will use a Gaussian process to do so which means that we have access to a distribution $p(f|x, \mathcal{D})$ which is the predictive posterior of the Gaussian process.

[1] we will usually refer to it as the minima, when we want to maximise, we just take the negative

[2] read neural networks

1 Surrogate Model

We will use a Gaussian process as a surrogate model for the function. In Lab 4 we looked at how to work with Gaussian processes. If you feel that you need a bit of a recap of this go back to that lab and make sure that you understand Eq. 10-12 and how Figure 2 was generated. What we need to know from our GP is given a set of data $\mathcal{D}$ what is our belief of what the function is at every other location in the input domain. This is the object that we call the *predictive posterior* of the Gaussian process,

$$p(\mathbf{f}_*|\mathcal{D}, \mathbf{x}_*, \theta) = \mathcal{N}(\mu_{\mathbf{x}_*|\mathbf{x}}, K_{\mathbf{x}_*|\mathbf{x}}) \tag{2}$$

$$\mu_{\mathbf{x}_*|\mathbf{x}} = k(\mathbf{x}_*, \mathbf{x})k(\mathbf{x}, \mathbf{x})^{-1}\mathbf{f} \tag{3}$$

$$K_{\mathbf{x}_*|\mathbf{x}} = k(\mathbf{x}_*, \mathbf{x}_*) - k(\mathbf{x}_*, \mathbf{x})k(\mathbf{x}, \mathbf{x})^{-1}k(\mathbf{x}, \mathbf{x}_*). \tag{4}$$

You will need to implement a function that can return the mean and the variance at a set of locations `x_star` of a Gaussian process parametrised using `theta`.

Code

```
def surrogate_belief(x,f,x_star,theta):

    return mu_star, varSigma_star
```

Now when we have our surrogate model set up it is time to move on to the second component, the acquisition function.

2 Aquisition Function

The idea of the aqusition function is that it encodes the strategy of how we should utilise the knowledge that we currently have in order to decide on where to query the function. The design of this function is where we balance the two important factors, *exploration* where we learn about new things, and *exploitation* where we utilise what we currently know. There are many different acquisition functions to use and we will here only look at one of them but in principle they all describe a *utility-value* across the whole input domain of how much we will "gain" by querying the function in this specific place.

2.1 Expected Improvement

The most commonly used acquisition function is *Expected Improvement*. The idea underlying expected improvement is that the utility of a location in the input domain is relative to how much lower we expect the function value at this point to be. This means that the utility function $u(x)$ can be defined as follows,

$$u(x) = \max(0, f(x_*) - f(x)) \tag{5}$$

This means that we have a reward for every location in the space where the function $f(x)$ is smaller than the current best estimate $f(x_*)$. Now as we do not know $f(x)$ we want to use our knowledge from the surrogate model f. This we can do by taking the expectation of the utility function over our belief in the function as,

$$\alpha(x) = \mathbb{E}\left[u(x)|x, \mathcal{D}\right] = \int_{-\infty}^{f(x_*)} (f(x_*) - f(x))\mathcal{N}(f|\mu(x), k(x, x))\mathrm{d}f. \tag{6}$$

One of the nice things about Expected improvement is that we can evaluate the expectation in closed form resulting in the following acquisition function,

$$\alpha(x) = (f(x_*) - \mu(x))\Psi(f(x_*)|\mu(x), k(x, x)) + k(x, x)\mathcal{N}f(x_*)|\mu(x), k(x, x)) \tag{7}$$

$$\Psi(f(x_*)|\mu(x), k(x, x)) = \int_{-\infty}^{f(x_*)} \mathcal{N}(f|\mu(x), k(x, x))\mathrm{d}f. \tag{8}$$

The function Ψ is the *cumulative density function* or *cdf* of the Gaussian which has the following form,

$$\frac{1}{2}\left(1 + \text{erf}\left(\frac{x-\mu}{\sigma\sqrt{2}}\right)\right), \tag{9}$$

where erf$(\cdot)$ is the *error-function*[3]. Now we want to choose points in the input domain that will maximise the acquisition function. Looking at the function that we have derived we can see that it includes two terms, the first term can be increased by picking an x value such that the difference between $f(x_*) - \mu(x)$ is large. In effect this is *exploiting* the knowledge that we currently have about the function. The second term can be increased by finding a location in the input domain such that $k(x, x)$ is large, i.e. the variance at this location is high. In effect this is *exploration* as we are looking for locations where we are uncertain of what the value is. As you can see these two terms formulates a specific balancing between the two key aspects of search, *exploration* and *exploitation*.

Now we need to write an implementation of the acquisition function we are going to need something looking like this,

Code

```
from scipy.stats import norm
def expected_improvement(f_*, mu, varSigma, x):

    # norm.cdf(x, loc, scale) evaluates the cdf of the normal distribution

    return alpha
```

where `mu` and `varSigma` is the mean and the variance of the predictive posterior of the surrogate model at locations `x` which is the set of candidates for where to pick the next function evaluation from.

We now have all the parts that we need in order to implement our Bayesian optimisation loop, the surrogate model using a Gaussian process and the acquisition function using expected improvement.

3 Experiments

We will now write up the Bayesian optimisation loop that we will iterate through. The first thing we need is a function to evaluate. As we want to be able to play around with the function a bit we will add a set of possible arguments. The functions was taken from this excellent blog post on Bayesian optimisation.

Code

```
def f(x, beta=0, alpha1=1.0, alpha2=1.0):
    return np.sin(3.0*x) - alpha1*x + alpha2*x**2 + beta*np.random.randn(x.shape[0])
```

The next thing that we will do is to decide on a finite set of possible evaluations of the function. The function that we are using is a function in $\mathbb{R}$ what we will do is to divide up this space into a finite set of locations and then our aim is to find at which one of these points we have the minimal value of the function. If we call this set $\mathbf{X}$ we will now start our loop by taking a random set of starting points, compute the predictive posterior over the remaining points, compute the acquisition for all the points not included in the model, pick the location with the highest acquisition and include this into the modelling set. A hand-wavy structure should look something like this.

One useful way to code this things is to keep two sets of points, you first start with an array with all locations that you can evaluate the function at, then you pick a random subset from this and move them to another set. Then for each evaluation you keep removing points from the initial set. This can easily be done with numpy arrays like this,

[3] https://en.wikipedia.org/wiki/Error_function

Algorithm 1 Bayesian Optimisation

procedure BO($f(x), \alpha(x), \mathbf{X}$)
  $x_{\text{start}} \subset \mathbf{X}$ ▷ Pick a random set of start-points
  $x \leftarrow x_{\text{start}}$
  $f' \leftarrow \text{argmin}_{x' \in x} f(x')$
  while iter **do** ▷ Loop until we have reached max number of iterations
    Evaluate $\mu_{\mathbf{X}|\mathbf{x}}$ and $K_{\mathbf{X}|\mathbf{x}}$ ▷ Predictive Posterior of Surrogate
    Evaluate $\alpha(\mathbf{X})$ ▷ Aquisition Function
    $x' = \text{argmax}_{\hat{x} \in \mathbf{X}} \alpha(\hat{x})$ ▷ Pick "best" candidate to evaluation set
    $x = x \cup x'$ ▷ Add element x' to the set
    if $f(x') < f'$ **then** ▷ Update current minima
      $f' = f(x')$
  return f'

Code

```
# remove points from an array
x_2 = np.arange(10)
index = np.random.permutation(10)
x_1 = x_2[index[0:3]]
x_2 = np.delete(x_2, index[0:3])

# remove largest element
ind = np.argmax(x_2)
x_1 = np.append(x_1, x_2[ind])
x_2 = np.delete(x_2, ind)
```

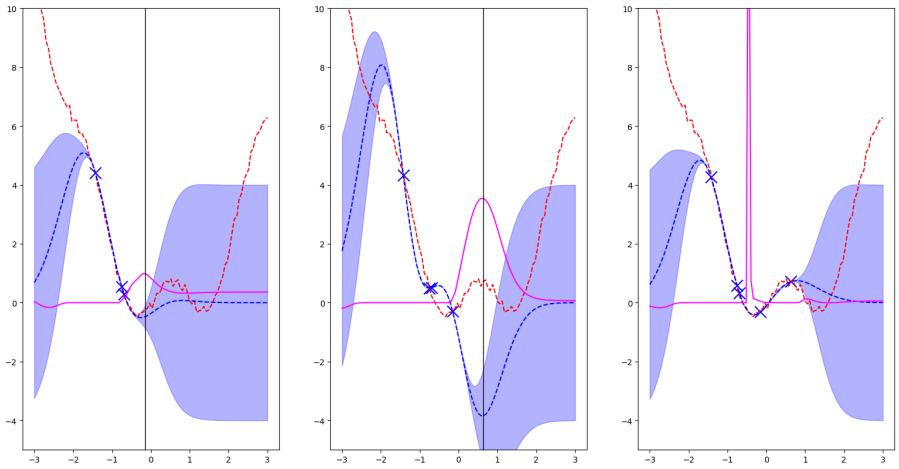

Figure 1: The image below shows the aqcuisition function in magenta, the true function in red and the surrogate models belief in blue. The left-most pane is the first iteration, at each iteration we add in the location of the highest acuisition and update the surrogate model. The image on the far right shows the third iteration.

When you have gotten the loop implemented you can try and see how good result you generally get in a fixed number of iterations. Then you compare this result with taking the same number of locations uniformly at random from the index set and evaluating them. If you compare the runs how often do you get a better value with the Bayesian optimisation approach compared to the random search? Now we can alter this question slightly, given that you have a current best estimate using BO, how many random samples do you need in order to get an equally good result?

4 Summary

Hopefully you have seen that having the concept of uncertainty can be really useful in order to direct a sequential search strategy as in Bayesian optimisation. The example that we evaluated here was in one dimension for us to be able to visualise the results. In general the improvements that you achieve will only improve with increasing dimension[4].

References

[1] J. Močkus. On bayesian methods for seeking the extremum. In G. I. Marchuk, editor, *Optimization Techniques IFIP Technical Conference Novosibirsk, July 1–7, 1974*, pages 400–404, Berlin, Heidelberg, 1975. Springer Berlin Heidelberg.

[2] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. Practical bayesian optimization of machine learning algorithms. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 2951–2959. Curran Associates, Inc., 2012.

[3] Yutian Chen, Aja Huang, Ziyu Wang, Ioannis Antonoglou, Julian Schrittwieser, David Silver, and Nando de Freitas. Bayesian optimization in alphago. *CoRR*, 2018.

[4] after a while they will start to do the opposite but thats a different question