Accelerating optimization via machine learning with different surrogate models

Ludovico Bessi

23 March 2019

Introduction

A surrogate model is an approximation method that mimics the behavior of a computationally expensive simulation.
The exact working of the simulation is not assumed to be known, solely the input-output behavior is important.
For this reason, a model is constructed based on the response of the simulator to a limited number of chosen data points.

In more mathematical terms: suppose we are attempting to optimize a function $f(p)$, but each calculation of f is very expensive. It may be the case we need to solve a PDE for each point or use advanced numerical linear algebra machinery which is usually costly. The idea is then to develop a surrogate model g which approximates f by training on previous data collected from evaluations of f.

There are many ways for building a surrogate model. Just to name a few, we can use: response surfaces, neural networks, random forests, support vector machines and Gaussian processes. It is worth noting that the nature of the function is not known a priori so usually it is not clear *which* surrogate model will be the most accurate.

The construction of a surrogate model can be seen as a three steps process:

1 **Sample selection**

2 **Construction of the surrogate model and optimizing the parameters**

3 **Accuracy appraisal of the surrogate**

The accuracy of the surrogate depends on the number and location of samples in the design space.

Example 1 (Non linear ODE): Lotka-Volterra

Let's consider the parametrized system of Lotka-Volterra equations:

$$\begin{cases} \frac{dx}{dt} = (A - By)x, \\ \frac{dy}{dt} = (Cx - D)y \end{cases} \tag{1}$$

This system is made up of two non linear differential equations.
It can be solved numerically, but let's try to build the simplest possible surrogate model: a *linear model*.
Before diving into the overview of Julia implementation, it is worth noting that it cannot be expected that such a surrogate will work well, simply because it will try to model the system linearly, but (1) is non-linear.

Our surrogate should work like this: with $[A, B, C, D]$ as input, we want to be able to calculate the solution (x, y) at a time t^*.
We proceed as follows:

1 Solve the system in an interval (t_{min}, t_{max}) for n times with random input $[A, B, C, D]$.

2 Use the library GLM to build a linear model out of the samples.

3 Use model to find solution at time t^*.

You can find the full implementation here: https://github.com/ludoro/Julia_surrogate

Example 2 (Stiff non linear ODE): Air pollution

$$
\begin{aligned}
\frac{du_1}{dt} =& - p_1u_1 - p_{10}u_11u_1 - p_{14}u_1u_6 - p_{23}u_1u_4 - p_{23}u_{19}u_1 + \\
& p_2u_2u_4 + p_3u_5u_2 + p_9u_{11}u_2 + p_{11}u_{13} + p_{12}u_{10}u_2 + \\
& + p_{22}u_{19} + p_{25} + u_{20} \\
\frac{du_2}{dt} =& - p_2u_2u_4 - p_3u_5u_2 - p_9u_{11}u_2 - p_{12}u_{10}u_2 + p_1u_1 + p_{21}u_{19} \\
\frac{du_3}{dt} =& - p_{15}u_3 + p_1u_1 + p_{17}u_4 + p_{19}u_{16} + p_{22}u_{19} \\
\frac{du_4}{dt} =& - p_2u_2u_4 - p_16u_4 - p_17u_4 - p_{23}u_1u_4 + p_{15}u_3 \\
\frac{du_5}{dt} =& - p_3u_5u_2 + p_4u_7 + p_4u_7 + p_6u_7u_6 + p_7u_9 + p_{13}u_{14} + p_{20}u_{17}u_6 \\
\frac{du_6}{dt} =& - p_6u_7u_6 - p_8u_9u_6 - p_{14}u_1u_6 - p_{20}u_{17}u_6 + p_3u_5u_2 + p_{18}u_{16} + p_{18}u_{16} \\
\frac{du_7}{dt} =& - p_4u_7 + p_5u_7 - p_6u_7u_6 + p_{13}u_{14} \\
\frac{du_8}{dt} =& + p_4u_7 + p_5u_7 + p_6u_7u_6 + p_7u_9 \\
\frac{du_9}{dt} =& - p_7u_9 - p_8u_9u_6 \\
\frac{du_{10}}{dt} =& - p_{12}u_{10}u_2 + p_7u_9 + p_9u_{11}u_2 \\
\frac{du_{11}}{dt} =& - p_9u_{11}u_2 - p_{10}u_{11}u_1 + p_8u_9u_6 + p_{11}u_{13} \\
\frac{du_{12}}{dt} =& + p_9u_{11}u_2 \\
\frac{du_{13}}{dt} =& - p_{11}u_{13} + p_{10}u_{11}u_1 \\
\frac{du_{14}}{dt} =& - p_{13}u_{14} + p_{12}u_{10}u_2 \\
\frac{du_{15}}{dt} =& + p_{14}u_1u_6 \\
\frac{du_{16}}{dt} =& - p_{18}u_{16} - p_{19}u_{16} + p_{16}u_4 \\
\frac{du_{17}}{dt} =& - p_{20}u_{17}u_6 \\
\frac{du_{18}}{dt} =& + p_{20}u_{17}u_6 \\
\frac{du_{19}}{dt} =& - p_{21}u_{19} - p_{22}u_{19} - p_{24}u_{19}u_1 + p_{23}u_1u_4 + p_{25}u_{20} \\
\frac{du_{20}}{dt} =& - p_{25}u_{20} + p_{24} + u_{19}u_1
\end{aligned}
$$

The second example is a system of 20 non linear differential equation that simulates air pollution, it is a common benchmark problem.
The word *stiff* means that certain methods for solving the equations are numerically unstable, unless the step size is extremely small.
Building a surrogate that approximates the solution is then very useful.

Example 3 (PDE): 2D Brusselator semilinear Heat equation

$$\begin{cases} \dfrac{\partial u}{\partial t} = 1 + u^2 v - Bu + \alpha \Delta u + f(x, y, z) \\ \dfrac{\partial v}{\partial dt} = Au - u^2 v + \alpha \Delta v \end{cases} \tag{2}$$

The Brusselator is a theoretical model for a type of autocatalytic reaction.
It can be transformed into a system of NxNx2 ordinary differential equations.
The fully optimized numerical solution takes about 6 seconds when $N = 32$.
However, if N grows the computational time will grow to hours or even days.

This example shows that a well optimized surrogate model is needed to drastically reduce computational time.

Proposal

My goal for this project is to develop a library for performing optimization with different surrogates model fully tested and with clear documentation. In the documentation, there will be a section containing lots of examples. I feel this is the best way to help new users get acquainted with the library itself.

The library will be extremely useful for a wide range of STEM professionals because in every field of science there are models that are too computational expensive that need to be approximated. I am sure this tool will make this problem easier to tackle.

I am going to mainly focus on the following surrogates: **Response surfaces**, **Machine learning based surrogates** and **Gaussian Processes**.

Response surfaces

Suppose we have n points x_i, for which we can calculate $y(x_i) = y_i$. It might be the case that x_i multi dimensional with dimension d, so we could have: $x_i = (x_{i1}, \ldots, x_{id})$.
Let $\{\pi_k(x)|k = 1, \ldots, m\}$ a basis of polynomials of degree G. If we just tried to interpolate we would have a normal polynomial interpolation. Instead, we add n basis functions terms, each centered around each sampled term. Then our predictor becomes:

$$\hat{y}(x^*) = \sum_{k=1}^{m} a_k \pi_k(x^*) + \sum_{j=1}^{n} b_j \phi(x^* - x_j)$$

Generally, basis functions are:

1) $\phi(z) = \|z\|$ (linear)

2) $\phi(z) = \|z\|^3$ (cubic)

3) $\phi(z) = \|z\|^2 \log \|z\|$ (thin plate spline)

4) $\phi(z) = \sqrt{\|z\|^2 + \lambda^2}$ (multi quadric)

5) $\phi(z) = e^{-\sum_{l=1}^{d} \theta_l |z_l|^{p_l}}$ (Kriging)

To interpolate the data, we would need:

$$y_i = \sum_{k=1}^{m} a_k \pi_k(x_i) + \sum_{j=1}^{n} b_j \phi(x_i - x_j) \; i = 1, \ldots, n$$

However, we have $n + m$ unknowns, so we need to add the following constraints as well:

$$\sum_{j=1}^{n} b_k \pi_k(x_j) \; k = 1, \ldots, m$$

Adding these constraints and mixing polynomials with basis functions give the interpolator desirable properties.

The algorithm would work as follows: we find the coefficient that interpolates the sampled points. Then, we minimize the response surface and we add that point to the set of sampled points. We keep going until the difference from one iteration to the other is non negligible. However, if we use the basis functions from 1 to 4, we may incur in problems because they might fail to find the global minimum, but just a local minimum as seen in: *A Taxonomy of Global Optimization Methods Based on Response Surfaces* by Donald Jones.

Kriging stands out because it has a statistical interpretation: we can compute an interpolator as well as a measure of its possible error. We can then develop search methods that put some emphasis on sampling where this error is high.
The kriging predictor is shown to be the predictor that minimizes the expected square prediction error, subject to being unbiased and being a linear function of predicted y_i's.
Suppose we want to make a prediction at a point x. Before we have sampled any points, we would be uncertain about its value. We model the uncertainty by saying that the value of the function at x is like the realization of a random variable $Y(x)$ that is normally distributed with mean μ and variance σ^2. If we consider two points x_i and x_j, assuming the sampled function is continuous, we can say:

$$Corr[Y(x_i), Y(x_j)] = exp\Big(- \sum_{l=1}^{d} \theta_l |x_{il} - x_{jl}|^{pl} \Big) \tag{3}$$

The θ_l parameter determines how fast the correlation drops off as one moves in the l^{th} coordinate. Large values of θ_l serve to model functions that are

highly active in the l^{th} variable. The pl determines the smoothness of the function in the l^{th} coordinate direction.

We can then represent the uncertainty at the n points using the random vector $\mathbf{Y} = (Y(x_1), ..., Y(x_n))$. It has mean $\mathbf{1}\mu$ and $Cov(\mathbf{Y}) = \sigma^2\mathbf{R}$, where each element of $\mathbf{R}$ is given by (1).
To estimate the values of μ, σ^2, θ_l and pl, we choose the parameters to maximize the likelihood of observed data.
After some calculations, we get:

$$\hat{\mu} = \frac{\mathbf{1}'R^{-1}\mathbf{y}}{\mathbf{1}'R^{-1}\mathbf{1}} \tag{4}$$

$$\hat{\sigma}^2 = \frac{(\mathbf{y} - 1\hat{\mu})'\mathbf{R}^{-1}(\mathbf{y} - 1\hat{\mu})}{n} \tag{5}$$

$$-\frac{n}{2}log(\hat{\sigma}^2) - \frac{1}{2}\log|\mathbf{R}| \tag{6}$$

In practice, we use the last equation to get estimates $\hat{\theta}_l$ and $\hat{pl}$, then we use equations (2) and (3) to compute $\hat{\mu}$ and $\hat{\sigma}^2$ Suppose we now want to predict a new point x^*. After some calculations, we get that:

$$\hat{y}(x^*) = \hat{\mu} + \mathbf{r}'\mathbf{R}^{-1}(\mathbf{y} - \mathbf{1}\hat{\mu}) \tag{7}$$

with $\mathbf{r} = (Corr[(Y(x^*), Y(x_1)], \ldots, Corr[(Y(x^*), Y(x_n)])$

Machine learning based surrogates

There are a number of different algorithms that can be applied to learn nonlinear relationships. I will focus on Random forests regression algorithms and neural networks, mainly radial basis neural networks. For the first one, I plan on using the library "XGBoost.jl" which already has many functionalities implemented. It is the Julia interface with XGBoost. It is an efficient and scalable implementation of gradient boosting framework, parallelized using OpenMP.
I used this library for regression involving Lotka Volterra, as seen here: LINK DA AGGIUNGERE

The main work here will be the integration between this library and the package Diffeq.

Below, a mathematical explanation of what I plan to implement regarding NN's.

Radial Basis Neural Network

Radial basis functions were introduced as a tool for interpolating multivariate functions. I will introduce the radial basis function for interpolation problems, then we see how they can be employed in neural networks. Given a set of sampled points, we can use the following as an interpolation function:

$$f(x) = \sum_{i=1}^{l} w_i \phi(\|x - x_i\|)$$

To calculate the weights w_i, the following system of equations needs to be solved:

$$\Phi w = y$$

In order to employ radial basis functions in neural networks, techniques from regularization theory are employed. Regularization techniques search for the approximating function f by minimizing a functional formed by two terms:

$$\Gamma(x, y) = \frac{1}{2} \sum_{i=1}^{l} [y_i - y(x_i)]^2 + \frac{1}{2} \lambda \|\Psi g\|^2 \tag{8}$$

where $\lambda > 0$ is a regularization parameter, Ψ is a differentiable operator. The first term measures the distance between the approximating function and the terms in the training set, while the second term is comprised by a functional the penalizes the violation on some regularization terms on the function $g(\cdot)$.

It can be proved that the function that solves (6) has the form:

$$f(x) = \sum_{i=1}^{l} w_i \phi(\|x - x_i\|) \tag{9}$$

With w solution of:

$$(\Phi + \lambda I) w = y$$

Equation (7) can be seen as the output of a feedforward neural network with an hidden layer. The neurons in the hidden layers will have as activation function ϕ, which argument will be the distance between the input vector and the center associated with that neuron. In the output layer of this network there will be a single neuron that operates a weighted sum of the outputs of the hidden layers neurons.

The training problem of a generalized RBF neural network can be formulated in a way similar to that of a 2-layer ANN. The strategy will be to train

with supervised weights and non-supervised centers. That is, the centers are chosen among the the sampled points. The training consists in solving:

$$min_w E(w) = \frac{1}{2}\sum_{i=1}^{l}(\sum_{j=1}^{N} w_j \phi(\|x_i - c_j\|) - y_i)^2 + \frac{\gamma}{2}\|w\|^2 \tag{10}$$

where γ is the regularization term for the weights. It is luckily possible to reformulate this problem:

$$min_w E(w) = \left\| \begin{bmatrix} \Phi(c) \\ \gamma^{\frac{1}{2}}\mathbf{I} \end{bmatrix} \mathbf{w} - \begin{bmatrix} y \\ 0 \end{bmatrix} \right\|^2$$

Which can be solved with least squares.

Gaussian processes

A gaussian process (GP) is a collection of indexed random variables such that every finite collection of those random variables has a multivariate normal distribution. The distribution if fully characterized by its mean function and its covariance function (kernel). They are useful for regression because not only we get a predicted value in a unknown region but also the confidence level of such a prediction. There is a number of kernels we can choose from depending on the data we need to fit. The choice drastically influences the result.

I will now give a brief overview on how they work, supposing I am just interested in a single output GP. For convenience, we let the mean function be 0. If that is the case, the distribution of the model $f^{GP}(\theta)$ is characterized by its covariance function. Let's choose the squared exponential:

$$cov(f^{GP}(\theta), f^{GP}(\theta') = c(\theta, \theta') := \sigma_c^2 exp\Big[-\sum_{i=1}^{p} \frac{(\theta_i - \theta_i')^2}{l_i^2} \Big]$$

The vector of the parameters of the covariance function is denoted by ψ:

$$\psi = (\sigma_c, l_1, ..., l_p)$$

They are usually called *hyperparameters*.
Given a training set $\mathcal{D}$, the hyperparameters are chosen by maximizing the logarithm of the marginal likelihood of the training values:

$$\psi^* = arg\,max\, logL(\psi|D)$$

My work will be based on the *Stheno.jl* and *GaussianProcess.jl* libraries, which already have an implementation for GP's.

Using the *GaussianProcesses* library, I built a GP to approximate the solution of Lotka Volterra. Normally the system deals with 4 parameters, but it can be transformed in such a way that only a parameter is needed. In this way, I can employ a single input single output gaussian process, as seen here: https://github.com/ludoro/Julia_surrogate/blob/master/GP_trial.jl

Timeline

6th May - 27th May – Bonding time

- Study in depth the theory required to understand the algorithms to be employed.
- Find and study in depth the libraries I will depend on for the project: mainly Flux and XGBoost.jl.
- Start working on my proposal ahead of the three months period.

28th May - 28th June – Response surfaces

- Work on surrogate based on Response surfaces.
- Start working on Machine learning based surrogates.

29th June - 26th July – ML based surrogates

From 28th of June to approximately 3th of July I will be under exams, therefore my output will be reduced.

- Finish work on ML based surrogates. This will take the majority of time because there are many of possible algorithms to implement.

27th July - 26th August – Finish neural networks, documentation, examples.

I have left more time than necessary in this last month to account for problems that might have occurred along the way. I will be on vacation for a few days with my girlfriend as well.

- Finish ML based surrogates.
- Write documentation and examples that serve as tutorials.
- **Bonus:** Start developing surrogate models using Gaussian processes.

About me

I am a third year applied mathematics student at Politecnico di Torino, Italy. I know how to code in Python, C, R, Matlab and Julia. I have a strong background in **Probability theory** and **statistics**, thanks to a *measure theory* based course.

I am working as a Data scientist for "Policumbent", a team at my University that is building the fastest bike on earth.

I won an Hackaton with a machine learning project based on Keras. You can find my CV and more information about on my personal webite: https://ludoro.github.io

Contact information:

- Mobile phone: +39 3467172332
- Email: ludovicobessi@gmail.com
- GitHub: @ludoro
- Slack: @ludoro

Contact information in case of emergency: +39 360882130 (Father).