Lab : Function approximation

The Problem

In this lab, we will try to approximate the non-linear function

$$f(x) = 1 - x - \sin(-2\pi x^3)\cos(-2\pi x^3)\exp(-x^4). \tag{1}$$

This function is not known a priori and all we have to work with is a noisy data set, $[\boldsymbol{x}^\top, \boldsymbol{y}_{noisy}^\top]^\top$. In order to approximate f, we use a parametric function $\hat{f}_\theta : \mathbb{R} \to \mathbb{R}$ where $\theta \in \mathbb{R}^k$ is a real vector of parameters. We make the assumption that f is non-linear and can be approximated using:

$$\hat{f}(x) = \boldsymbol{\phi}(x)^\top \boldsymbol{\theta} \tag{2}$$

where $\boldsymbol{\phi}(x) = [\phi_1(x), \phi_2(x), \cdots, \phi_k(x)]^\top$ is a vector function commonly referred as *features*. We choose to use Gaussian basis functions throughout the input space:

$$\phi_i(x) = \exp(-\frac{(x - c_i)^2}{\sigma_i^2}) \tag{3}$$

where c_i is a center and σ_i a standard deviation, both constants. The vector $\boldsymbol{\theta}$ is a vector of parameters used to weight each Gaussian.

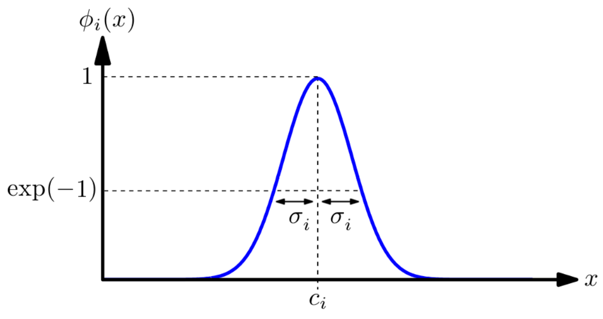

Figure 1: Gaussian basis functions

The goal of this lab is to determine the parameters, $\boldsymbol{\theta}$ which permit $\hat{f}$ to best approximate f using a variety of training techniques; both *Incremental* and *Batch* techniques will be studied.

Incremental vs. Batch Learning

In supervised learning, there are two major classes of learning algorithms: those which estimate $\boldsymbol{\theta}$ iteratively with incoming data samples, or *incremental* methods, and those which take a set of data samples (a batch of data), and estimate $\boldsymbol{\theta}$ in a single calculation, or *batch* methods. Each have their pros and cons and are typically implemented based on the learning context. Here we explore both methods.

Instructions:

- For each learning method, attach a screenshot/image of the results found, to support your responses to the questions.
- You may use any program you wish to write your question responses and attach your results, but you must convert it to a PDF prior to submission. Non PDF submissions will be rejected.
- Do not forget to put the names of all members of your team on each document.

1 Gradient descent (incremental)

The gradient descent approach simply updates $\boldsymbol{\theta}$ in the opposite direction of the gradient of the error function until it cancels out, which means that we have reached a local minimum. Let ∇_θ denote the gradient according to $\boldsymbol{\theta}$, we assume that $\hat{f}$ is differentiable for any $\boldsymbol{\theta}$. The gradient of the error function is:

$$\begin{aligned}\nabla_\theta \hat{\varepsilon}(\theta) &= \nabla_\theta \mathbb{E}[\frac{1}{2}(y - \hat{f}(x))^2] \\ &= \mathbb{E}[(\nabla_\theta(y - \hat{f}(x)))(y - \hat{f}(x))] \\ &= \mathbb{E}[(\nabla_\theta \hat{f}(x))(\hat{f}(x) - y)].\end{aligned}$$

For our linear function approximator with Gaussian basis functions, we have

$$\nabla_\theta \hat{f}(x) = \boldsymbol{\phi}(x). \tag{4}$$

In the gradient descent method, $\boldsymbol{\theta}$ is updated iteratively, that is by using the samples one at a time. An initial guess for $\boldsymbol{\theta}$ must be made in this case (*hint:* `np.random.random()`). Starting from parameters $\boldsymbol{\theta}_0$ (choosen a priori), we apply the following equation, where (x_t, y_t) is the tth sample:

$$\begin{aligned}\theta_{t+1} &= \theta_t - \alpha(\nabla_\theta \hat{f}(x_t))(\hat{f}(x_t) - y_t) \\ \theta_{t+1} &= \theta_t - \alpha\boldsymbol{\phi}(x_t)(\hat{f}(x_t) - y_t), \end{aligned} \tag{5}$$

where α is a constant known as the *learning rate*. It is well known that $\boldsymbol{\theta}_t$ converges asymptotically (as t goes towards infinity) to a local minimum of the error function, provided that α is small enough.

Questions

1.1. Open the file `functionApproximator.py` and in the function, `train_GD()` complete the missing bits of code, marked "???".

Note: In the annex at the end of this handout you will find a summary of the code provided in the this lab. Please read this for a general understanding of the classes and functions provided. For details on the implementation of the code, documentation has been provided in the git repository you downloaded in the directory Code Documentation. To open the documentation simply open the file `index.html`.

1.2. In `run.py`, tune all the relevant parameters (marked "Tune me !") in order to obtain a reasonable approximation of f. For each of these parameters, explain their impact on the approximation error. To estimate the performance of your algorithm, the `plotFA()` function will provide you with the total error (residual) and execution time statistics [1].

[1] Requires python `psutil` package.

2 Least-Squares (batch)

Let us consider the case where we have a set of Ns samples $[\boldsymbol{x}^\top, \boldsymbol{y}_{noisy}^\top]^\top$. Instead of using them one by one as in the gradient descent method, we can use them all at once to estimate $\nabla_\theta \hat{\varepsilon}(\boldsymbol{\theta})$. This gradient cancels out for any local optimum of the error function, therefore we can find one of these by formulating $\boldsymbol{\theta}$ as follows:

$$\begin{aligned}
& \nabla_\theta \hat{\varepsilon}(\boldsymbol{\theta}) = 0 \\
\Leftrightarrow \quad & \frac{1}{Ns} \sum_{i=1}^{Ns} \boldsymbol{\phi}(x_i)(\boldsymbol{\phi}(x_i)^\top \boldsymbol{\theta} - y_i) \approx 0 \\
\Leftrightarrow \quad & \sum_{i=1}^{Ns} (\boldsymbol{\phi}(x_i)\boldsymbol{\phi}(x_i)^\top)\boldsymbol{\theta} \approx \sum_{i=1}^{Ns} (\boldsymbol{\phi}(x_i) y_i) \\
\Leftrightarrow \quad & A\boldsymbol{\theta} \approx b \\
\Leftrightarrow \quad & \boldsymbol{\theta} \approx A^\sharp b
\end{aligned}$$

with,

$$A = \sum_{i=1}^{Ns} (\boldsymbol{\phi}(x_i)\boldsymbol{\phi}(x_i)^\top) \tag{6}$$

$$b = \sum_{i=1}^{Ns} (\boldsymbol{\phi}(x_i) y_i) \tag{7}$$

where $A^\sharp$ is the pseudo-inverse of matrix A. This is called the Least-Squares (LS) method.

Questions

2.1. Open the file `functionApproximator.py` and in the function, `train_LS()` complete the missing bits of code, marked "???".

2.2. Now that we have seen both a batch and iterative method, what are the advantages and disadvantages of least-squares over gradient descent?

3 Recursive Least-Squares (incremental)

Recursive Least-Squares (RLS) is a variant of the Least-Squares method where the statistics A and b are estimated recursively : since they are computed as a sum over samples, we can add the samples one by one and get a new estimation of the parameters. At time t, we receive a sample (x_t, y_t) and update the statistics:

$$A_{t+1} = A_t + \boldsymbol{\phi}(x_t)\boldsymbol{\phi}(x_t)^\top \tag{8}$$

$$b_{t+1} = b_t + \boldsymbol{\phi}(x_t) y_t \tag{9}$$

where the initial stats A_1 and b_1 are zeros. The corresponding parameters are $\theta_{t+1} = A_{t+1}^\sharp b_{t+1}$. We can also estimate $A_{t+1}^\sharp$ directly using the Sherman-Morrison lemma:

$$A_{t+1}^\sharp = (A_t + uv^\top)^\sharp = A_t^\sharp - \frac{A_t^\sharp u_t v_t^\top A_t^\sharp}{1 + v_t^\top A_t^\sharp u_t} \tag{10}$$

where $u_t = v_t = \boldsymbol{\phi}(x_t)$ here. Notice that we must suppose $1 + v_t^\top A_t^\sharp u_t \neq 0$, but $A_1^\sharp$ must also be non-zero for the update to work. One usually take $A_1^\sharp = \delta\mathbf{I}$ where δ is a positive constant and **I** the identity matrix.

Questions

3.1. Open the file `functionApproximator.py` and in the functions, `train_RLS()` and `train_RLS2()` complete the missing bits of code, marked "???".

3.2. Compare the speed of the two variants (estimating A vs. $A^\sharp$ with Sherman-Morrison), using the training and mean iteration durations printed by `plotFA()`. Which one is faster and why?

4 Locally-Weighted Least-Squares (batch)

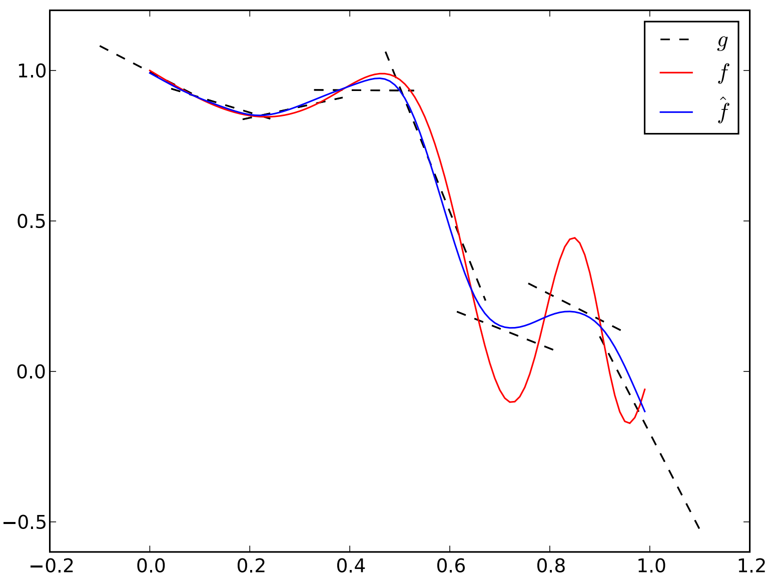

Figure 2: Example of an approximation with Locally-Weighted Least-Squares

Locally-Weighted Least-Squares (LWLS) approximate a function using a weighted sum of local linear models.

In this exercice, we change our Gaussian features for linear features, e.g.,

$$\boldsymbol{\phi}(x) = [x, 1]^\top. \tag{11}$$

Each local model k corresponds to a function approximator

$$g_k(x) = \theta_1^{(k)} x + \theta_2^{(k)} = \boldsymbol{\phi}(x)^\top \boldsymbol{\theta}^{(k)}. \tag{12}$$

Furthermore, it is associated with a weighting function $\boldsymbol{\omega}_k(x)$ which determines how important the sample x is for the local model. We use our previous Gaussian basis functions as the weighting function:

$$\boldsymbol{\omega}_k(x) = \exp(-\frac{(x-c_k)^2}{\sigma_k^2}). \tag{13}$$

Practically speaking, the closer x is to the center c_k, the more "important" it is for the local model k.

The output $\hat{f}$ is a normalized weighted sum of the local models :

$$\hat{f}(x) = \sum_k \frac{\boldsymbol{\omega}_k(x)}{\sum_k \boldsymbol{\omega}_k(x)} g_k(x). \tag{14}$$

To compute each local model parameters $\theta^{(k)}$, we use the Weighted Least-Squares method, which minimizes the weighted Mean Square Error :

$$\varepsilon^{(k)}(\boldsymbol{\theta}^{(k)}) = \mathbb{E}\left[\frac{1}{2}\boldsymbol{\omega}_k(x)(g_k(x) - y)^2\right].$$

As in the Least-Squares method, we cancel its gradient

$$
\begin{aligned}
& \nabla_{\theta^{(k)}} \hat{\varepsilon}^{(k)}(\theta^{(k)}) = 0 \\
\Leftrightarrow\ & \frac{1}{Ns} \sum_{i=1}^{Ns} \boldsymbol{\omega}_k(x_i)(\nabla_{\theta^{(k)}} g_k(x_i))(g_k(x_i) - y_i) \approx 0 \\
\Leftrightarrow\ & \frac{1}{Ns} \sum_{i=1}^{Ns} \boldsymbol{\omega}_k(x_i)\boldsymbol{\phi}(x_i)(\boldsymbol{\phi}(x_i)^\top \boldsymbol{\theta}^{(k)} - y_i) \approx 0 \\
\Leftrightarrow\ & A_k \boldsymbol{\theta}^{(k)} \approx b_k \\
\Leftrightarrow\ & \boldsymbol{\theta}^{(k)} \approx A_k^\sharp b_k
\end{aligned}
$$

with,

$$
A_k = \sum_{i=1}^{N} (\boldsymbol{\omega}_k(x_i)\boldsymbol{\phi}(x_i)\boldsymbol{\phi}(x_i)^\top) \tag{15}
$$

$$
b_k = \sum_{i=1}^{N} (\boldsymbol{\omega}_k(x_i)\boldsymbol{\phi}(x_i) y_i). \tag{16}
$$

Questions

4.1. Open the file `functionApproximator_LW.py` and in the function, `train_LWLS()`, complete the missing bits of code, marked "???".

4.2. How does this method compare with the Least-Squares method in Section 2? Which one provides a better approximation of f given the same meta-parameters? Which one is faster? Compare Eqns. (6)-(7) and (15)-(16), illustrate their similarities and differences.

Concluding Questions

4.1. Between batch and iterative methods when would you use each method and why?

4.2. Other than the algorithms' meta-parameters, what could we change to decrease the error even more?

Bonus: You are given a robotic arm with multiple electric motor driven articulations, and you wish to identify models for each of the motors.The input to each motor is a current, i, in $Amps$ and the output is a torque, τ, in Nm. Sensors provide you with noisy sample data, $[i, \tau]$ at a rate of 20 kHz. How would you go about learning the underlying control models for each of these motors? Justify your response based on the results you have found in this lab.

Submission

Please submit ...

Annex: Code Structure

In the git repertoire you have downloaded there are 5 files:

4.1. `functionApproximator.py`

In this file we define the `fa` or function approximator class. When the class is instantiated 4 arguments are needed: number of features, learning rate, minimum delta, and maximum iterations. Details on these parameters can be found in the docstring (comments) below the `fa` class `__init__` member function.

Once a `fa` object has been created (example: `fa_object = fa(param1, param2 ...)`) then the object can be trained to approximate the "`noisyDataSet`" using any one of the training member functions, `train_XXX(data)`, where `XXX` is the training method acronym (i.e. Gradient Descent = GD). In this lab we will study 4 supervised learning algorithms using Gaussian basis functions as features. These methods are detailed in Sections 1 - 3.

The member functions `featureOutput()` and `functionApproximatorOutput()` have been provided in vectorized form for convenience, but please read their documentation carefully.

4.2. `functionApproximator_LW.py`

In addition to the `fa` class, a locally weighted function approximator class, `fa_lw`, is provided in the file `functionApproximator_LW.py`. The bulk of the class code is the same as that of the `fa` class but here our features are linear models and they are locally weighted by Gaussians. Therefore, the member function `getWeights()` is introduced. Details on this are shown in Section 4.

4.3. `run.py`

To test your implementation of the training algorithms, the script `run.py` has been provided. In the script we first retrieve the data to learn from the `noisyDataSet.txt` file and put it into a $[2 \times Ns]$ numpy array, where Ns is the number of samples with inputs in the first row and their outputs in the second row. This data will be used for batch learning. Incremental methods will use randomly generated data inside of the learning function which are genenerated by the function `generateDataSample()`. Then a function approximator (FA) object is created using the user specified parameters. The FA is then trained on the data using the algorithms that you will complete in this lab. With the trained FA object we can then plot the output after training and compare it to the real function to obtain various information about the efficacy of the training.

4.4. `functionApproximator_PlottingTools.py`

The functionApproximator_PlottingTools.py file provides two functions with which to visualize output. The `plotFA()` function can be used to get all relevant output for all training methods. The `animPlotFA()` function can be used to visualize FA output over each iteration for incremental methods.

4.5. `noisyDataSet.txt`

This file contains the raw data used for batch learning methods.