Project in *First-order methods for large-scale machine learning*

Pierre De Handschutter
534814@umons.ac.be

Faculté Polytechnique
Université de Mons

30th October 2021

Outline

Outline

Context of the project

1. Goal of the project:
   - Big data supervised classification
   - Apply **first order** optimization methods to solve the problem

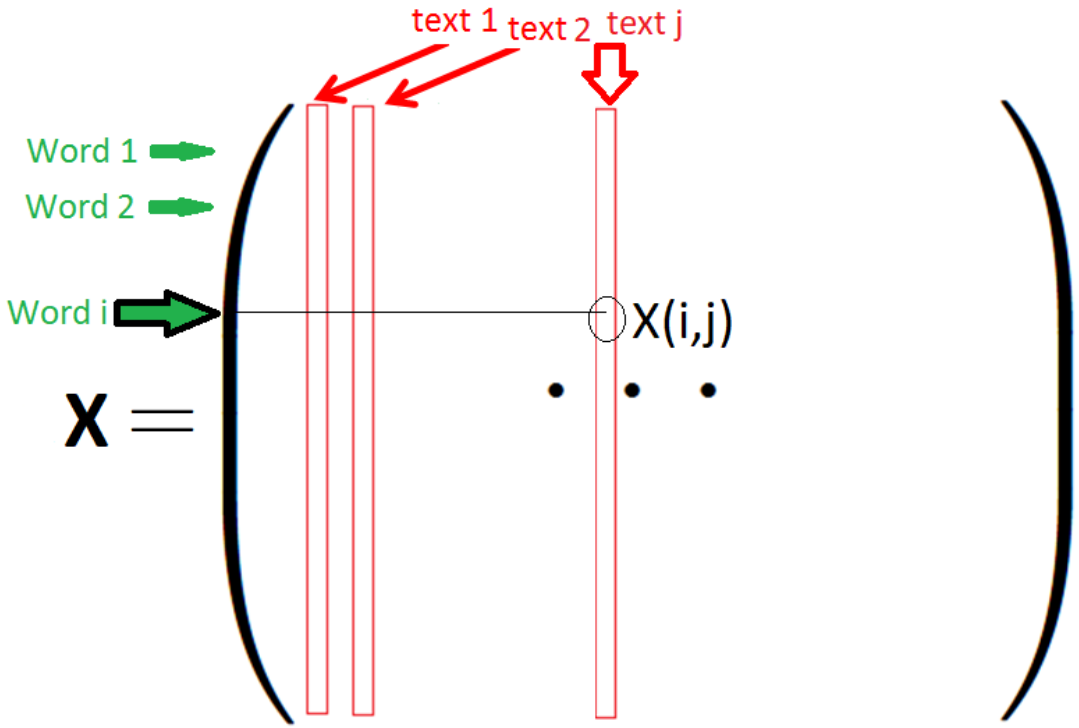
2. Modalities:
   - Evaluation: Matlab (or Python) Code + short report ($\sim$ 10 pages) containing a description of the problem, derivations details, list of tested methods, results, discussion,...
   - The results should be submitted in a Kaggle-challenge-like manner (predictions on the test set)
   - Ideally groups of 2, groups of > 3 are not accepted

Schedule

- Tuesday 2/11 13h30-17h30: presentation of the project, team building and data discovery
- Thursday 4/11 13h30-17h30: work by group, Q&A
- Monday 15/11 08h30-12h30: "free" (work on your own)
- Monday 22/11 08h30-12h30: work by group, Q&A

What's the project ?!

- Data: set of n documents based on a vocabulary of m words
- Each document is represented by a "bag of words" i.e. the number of occurrences of each word in this document
- In summary, in the data matrix $X \in \mathbb{R}^{m \times n}$, each column corresponds to a document (text) and each row to a word. $X(i,j)$ $(i = 1, ..., m,$ $j = 1, ..., n)$ is the number of occurrences of word i in text j



More about the data...

- Each document j is associated to a class, denoted y_j ($j = 1, ..., n$). There are 20 classes (numbered from 1 to 20)
- Training data *Xts*, *yts* used to train the model; test data *Xvr* used to evaluate the final performance (*yvr* is hidden to you ☺)
- Some values: $m = 43586$, $n\_train = 13960$, $n\_test = 5989$
- Data are available on Kaggle website: see *https://www.kaggle.com/c/doc-cla/*, read the description above

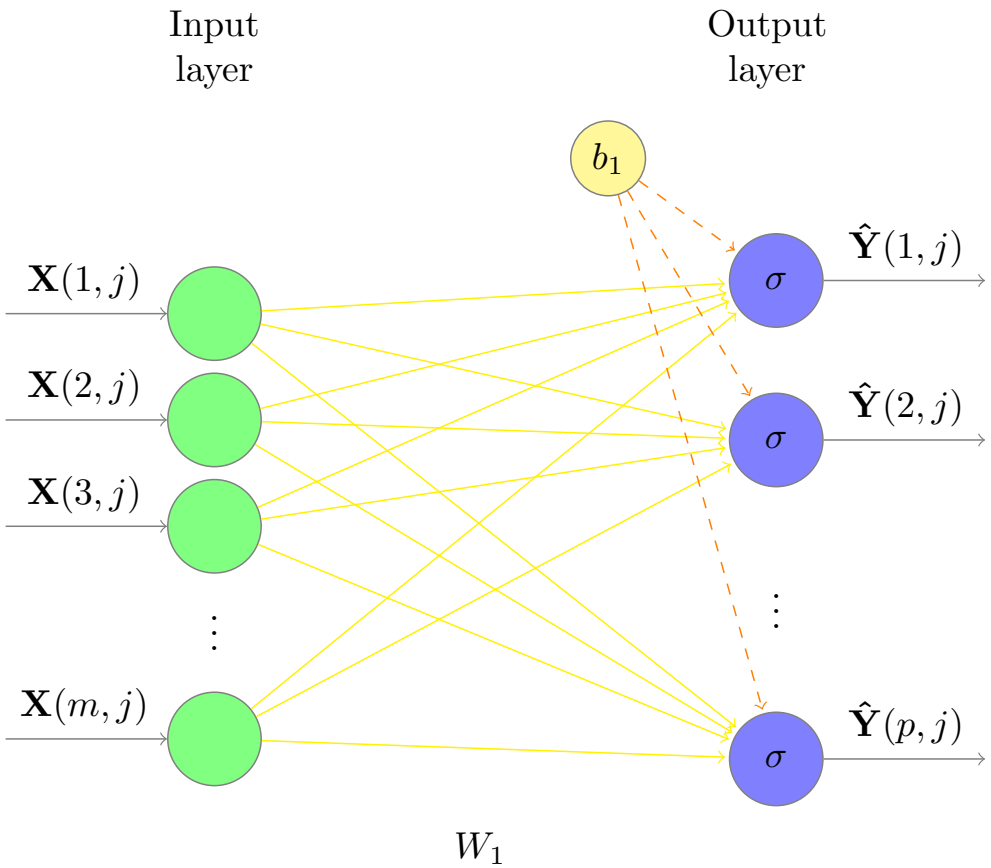
Before going into the details...

First: some preprocessing...

- As for now, the class is a single output corresponding to a category
- It would be odd to consider it as the target for our model, since its numerical value has little interest in itself (e.g. class "3" is not better than class "1", other codings would be possible)
- "One hot encoding" (as in *Hands on AI*): for all j, transform each label y_j into a vector $Y(:, j)$ of length p (p is the number of classes) such that $Y(k, j) = 1$ iff $k = y_j$ and 0 otherwise. Consequently, $Y \in \mathbb{R}^{p \times n}$

Model

- "0-hidden-layer neural network": the output layer directly follows the input one $\rightarrow$ only one weight matrix $W_1 \in \mathbb{R}^{m \times p}$ and a bias vector $b_1 \in \mathbb{R}^p$
- Illustration of the workflow for the j-th data point:



The model in practice

- Predicted one-hot-encoded vector $\hat{Y}_j$ of document j given by $\hat{Y}(k,j) = \sigma(\sum_{i=1}^{m} W_1(i,k)X(i,j) + b_1(k))\ \forall k = 1, ..., p \Rightarrow \hat{Y}(:,j) = \sigma(W_1^T X(:,j) + b_1)$ where the activation σ is taken element-wise [1].

- In matrix form, $\hat{Y} = \sigma(W_1^T X + B_1)$ where again, σ is element-wise and B_1 is a matrix whose each column is b_1 (see *repmat* function in Matlab)

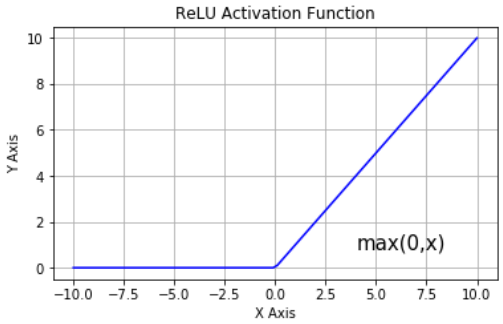
- Loss function: minimize the mean squared error between the prediction and the expected output: $\mathcal{L}(Y, \hat{Y}) = \frac{1}{n}\sum_{j=1}^{n} \| Y_j - \hat{Y}_j \|^2$ or, in matrix form, $\mathcal{L}(Y, \hat{Y}) = \frac{1}{n}\| Y - \hat{Y} \|_F^2$ (Rmq: a factor $\frac{1}{2}$ can be introduced (more convenient for derivation))

[1] $\hat{Y}_j$ and $\hat{Y}(:,j)$ mean the same thing

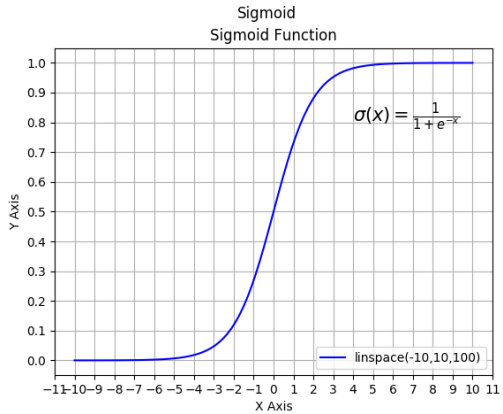
About the non-linear activation functions...

Several possibilities:

- RELU (REctified Linear Unit): $\sigma(x) = \max(0, ax)$, a is a positive scalar



- Sigmoid (logistic function): $\sigma(x) = \frac{1}{1+e^{(-ax)}}$, a is a positive scalar



- Softmax, ELU (exponential linear unit), P-RELU (parametric RELU)

Outline

Idea of the resolution schemes

- Goal: find the matrix W_1 and the vector b_1 that minimize the loss function $\mathcal{L}$
- Idea of resolution: alternatively optimize W_1 and b_1 until some stopping criterion is met (see last slide)

Algorithm 1 Block coordinate descent

1: Initialize somehow W_1 and b_1
2: **for** $k = 0, \ldots$ **do**
3: Optimize W_1 while fixing b_1
4: Optimize b_1 while fixing W_1
5: **end for**

How to solve the optimization problem ?

- First-order (i.e. gradient-based) methods: second-order methods such as Newton's one are computationally too expensive (computing the Hessian is $\mathcal{O}(d^2)$ where d is the number of parameters to update) though they have a better convergence rate of the iterates (quadratic vs linear)

- Several variants exist:
  - Gradient Descent (GD)
  - Stochastic Gradient Descent (SGD)
  - Accelerated Gradient Descent (AGD)
  - ...

Let's compute the gradient

- Recall of the loss function:
  $\mathcal{L}(Y, \hat{Y}) = \frac{1}{2n}\|(Y - \hat{Y})\|_F^2 = \frac{1}{2n}\|(Y - \sigma(W_1^T X + B_1))\|_F^2$
- Gradient of $\mathcal{L}$ w.r.t to the parameters W_1 and B_1.
- Let's call $Z = W_1^T X + B_1$ Apply chain rule [2] : $\frac{\partial \mathcal{L}}{\partial W_1} = \frac{\partial \mathcal{L}}{\partial \hat{Y}} \frac{\partial \hat{Y}}{\partial Z} \frac{\partial Z}{\partial W_1}$ where
  - $\frac{\partial \mathcal{L}}{\partial \hat{Y}}$ is like the gradient of a basic quadratic function
  - $\frac{\partial \hat{Y}}{\partial Z}$ is like the derivative of the activation function w.r.t. its argument
  - $\frac{\partial Z}{\partial W_1}$ is like the derivative of a linear function
- The same can be applied for the gradient w.r.t. B_1 (only the last factor changes)

[2] There are abuses of notation, should be understood element-wise

Exercise: compute the gradient, given that the activation function is a sigmoid

If it helps, you can derive element-wise, then "matricize".

We have the following:

- $\frac{\partial \mathcal{L}}{\partial \hat{Y}} = \frac{1}{n}(\hat{Y} - Y)$
- $\frac{\partial \hat{Y}}{\partial Z} = \sigma'(Z)$ (simply apply the derivative of the sigmoid): should be applied element-wise
- $\frac{\partial Z}{\partial W_1} = X$

Putting all together and taking care to the dimensions, we have

$$\frac{\partial \mathcal{L}}{\partial W_1} = X\left\{\frac{1}{n}(\sigma(W_1^T X + B_1) - Y) \odot \sigma'(W_1^T X + B_1)\right\}^T$$ and a very similar expression for $\frac{\partial \mathcal{L}}{\partial B_1}$ where $\odot$ is an element-wise multiplication

Recall on gradient descent

To minimize a function, it's better to go along the steepest descent direction, which is the direction opposite to the gradient's one $\Rightarrow$ gradient descent.

Starting from an initial vector x_0, gradient descent makes

Algorithm 2 Basic idea of Gradient Descent

1: **for** $k = 0, \ldots$ **do**
2: $\quad x_{k+1} = x_k - \alpha_k \nabla f(x_k)$
3: **end for**

How to choose the step size α_k ??

How to choose the step size

- Constant step size: not sure to converge ! (non-convex problem)
- Dummy backtracking line search (BLS):

Algorithm 3 Basic Backtracking line search

1: Set initial step α_0 to a huge value (for example $\alpha_0 = 1$ or $\alpha_0 = 0.1\frac{||x_0||}{||\nabla f(x_0)||}$ to ensure that the decay term is the same order as x)
2: **for** $k = 0, \ldots$ **do**
3: $\quad \alpha_k = 1.5\alpha_k$ % Avoid vanishing learning rate
4: $\quad x_{k+1} = x_k - \alpha_k \nabla f(x_k)$
5: $\quad$ **while** $f(x_{k+1}) > f(x_k)$ **do**
6: $\quad\quad \alpha_k = \alpha_k/2$
7: $\quad\quad x_{k+1} = x_k - \alpha_k \nabla f(x_k)$
8: $\quad$ **end while**
9: $\quad \alpha_{k+1} = \alpha_k$
10: **end for**

There exist sharper conditions on the step size...

- You could find the optimal step at each iteration but it would be very costly
- Armijo and Wolfe conditions:
  - Armijo condition: sufficient decrease of the objective function
  - "Mild" Wolfe condition: sufficient increase of the gradient
  - "Sharp" Wolfe condition: sufficient decrease of the gradient in **absolute value**
- (You can test if you want)

What's wrong with gradient descent (GD) ?

- Gradient descent is simple but is still relatively costly: the computation involves all the data points
- Convergence is "slow"

$\Rightarrow$ We will see methods that solve these problems

Outline

How to do best than GD ?

- How to reduce the computational cost ?
  - Stochastic gradient descent (SGD): compute the gradient with only one data point at the time
  - Mini-batch gradient descent (MBGD): compute the gradient with only some data points (=mini-batch) at the time (compromise between the expensiveness/"wholeness" of GD and the quickness/"stochasticness" of SGD)
- How to decrease the objective function faster ? Accelerated Gradient Descent (AGD): objective function decreases in $\mathcal{O}(\frac{1}{k^2})$ vs $\mathcal{O}(\frac{1}{k})$ for GD (in the convex case) ! (the best you can achieve with first-order methods)

Cheap variants of GD

- How to decrease the computational complexity ($\sim$ time by iteration) ?
- Recall, the objective function of our problem is $\mathcal{L}(Y, \hat{Y}) = \frac{1}{n}\sum_{j=1}^{n}(Y_j - \hat{Y}_j)^2 = \frac{1}{n}\sum_{j=1}^{n}\ell(Y_j, \hat{Y}_j)$, it is just the sum of n "separable" quadratic functions
- Idea of the following methods: avoid computing the whole $\mathcal{L}$ and the whole gradient, just limit to a few data points (i.e. a few functions ℓ). In the following, $f_j = \ell(Y_j, \hat{Y}_j)$.

Stochastic gradient descent

- Take one data point at the time and compute the corresponding gradient
- Algorithm:

Algorithm 4 Stochastic gradient descent (SGD)

1: **for** $k = 0, ...$ **do**
2: Pick up randomly an index u_k of $\{1, ..., n\}$
3: $x_{k+1} = x_k - \alpha_k \cdot \nabla f_{u_k}(x_k)$
4: **end for**

- Some remarks:
  - The selection procedure of the index u_k can either be "totally random" or you may ensure that each point has been picked once before any point has been picked a second time (for example, you initialize a pool equal to $\{1, ..., n\}$ at the beginning, then remove u_k from it once it has been picked and reset the pool to $\{1, ..., n\}$ all n iterations)
  - The step size can no longer be chosen through backtracking as the direction at each iteration may not lead to a decrease of the overall cost function (at most, you could do a BLS on a given f_{u_k} but it would be costly and without any guarantee) $\rightarrow$ fixed (small) step size or diminishing step size $\alpha_k = \frac{\beta}{\gamma + k}$ for some β, γ.

Mini-batch gradient descent

Algorithm 5 Mini-batch gradient descent (MBGD)

1: **for** $k = 0, \ldots$ **do**
2: Choose a MBGD size m_k and pick up randomly a subset B_k of size m_k in $\{1, \ldots, n\}$
3: $x_{k+1} = x_k - \frac{\alpha_k}{m_k} \sum_{b=1}^{m_k} \nabla f_{B_k(b)}(x_k)$
4: **end for**

- m_k can be chosen either constant or increasing
- Special cases: $m_k = 1$ comes to SGD while $m_k = n$ comes to GD
- Do NOT compute the whole gradient of $\mathcal{L}$ (nor the whole $\mathcal{L}$ itself), just compute the terms of the function and the gradient corresponding to the current subset B_k of data points (otherwise, you lose the interest)
- Pay attention ! When you will compare the results of GD and SGD, *of course* one iteration of GD will decrease more the objective function than one iteration of SGD (though being $\sim n$ times more expensive). To be fair, you need to compare the results of one iteration of GD with the results of n iterations of SGD (and similarly for any mini-batch size)

Still some variants...

- Averaged SGD:
  - Goal: reduce the variance of the iterates generated with SGD by averaging them
  - Rather than considering the x_{k+1} as in slide 24, consider $z_{k+1} = \frac{1}{k+2} \sum_{l=0}^{k+1} x_l$ where $z_0 = x_0$
  - Remark: rather than computing the whole sum, you should notice that $z_{k+1} = z_k + \frac{1}{k+2}(x_{k+1} - z_k)$. Besides, if you want to give more weights to recent iterates, you can replace $\frac{1}{k+2}$ by some $\gamma > \frac{1}{k+2}$
- Stochastic approximation with gradient aggregation (SAGA):
  - Rather than averaging the iterates, average the gradients: at iteration k, consider the mean of the stochastic gradients computed so far $g(k+1) = g(k) + \gamma(\nabla f_{u_k}(x_k) - g(k))$ with $g(0) = 0$ (set $\gamma = \frac{1}{k+1}$ if you want same weights for all the iterates, other if not)

Accelerated gradient descent

- Idea: add momentum ($\approx$ inertia) in the update scheme
- x_0 chosen, x_1 computed with classical GD, start acceleration for x_2
- Polyak's Heavy ball method: add momentum to the GD iterate
- Nesterov method: add momentum to the GD iterate **and in the gradient**

Polyak's Heavy ball method

- Replace gradient iteration by [3] $x_{k+1} = x_k - t_k \nabla f(x_k) + \beta_k (x_k - x_{k-1})$

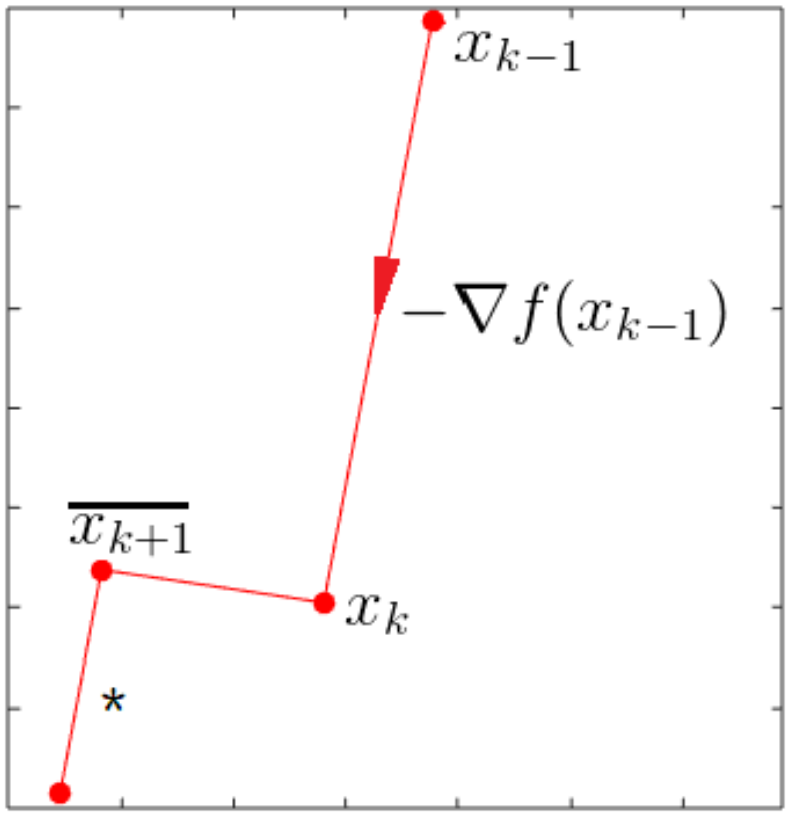


Figure taken from Andersen's notes `https://angms.science/notes.html`

[3] We use t_k rather than α_k as learning rate to avoid confusion (will understand after)

Nesterov acceleration method

- Replace gradient iteration by
  $x_{k+1} = x_k - t_k \nabla f(x_k + \beta_k(x_k - x_{k-1})) + \beta_k(x_k - x_{k-1})$
- Other way to see it: build a sequence y_k's such that
  $y_k = x_k + \beta_k(x_k - x_{k-1})$, we have $x_{k+1} = y_k - t_k \nabla f(y_k)$

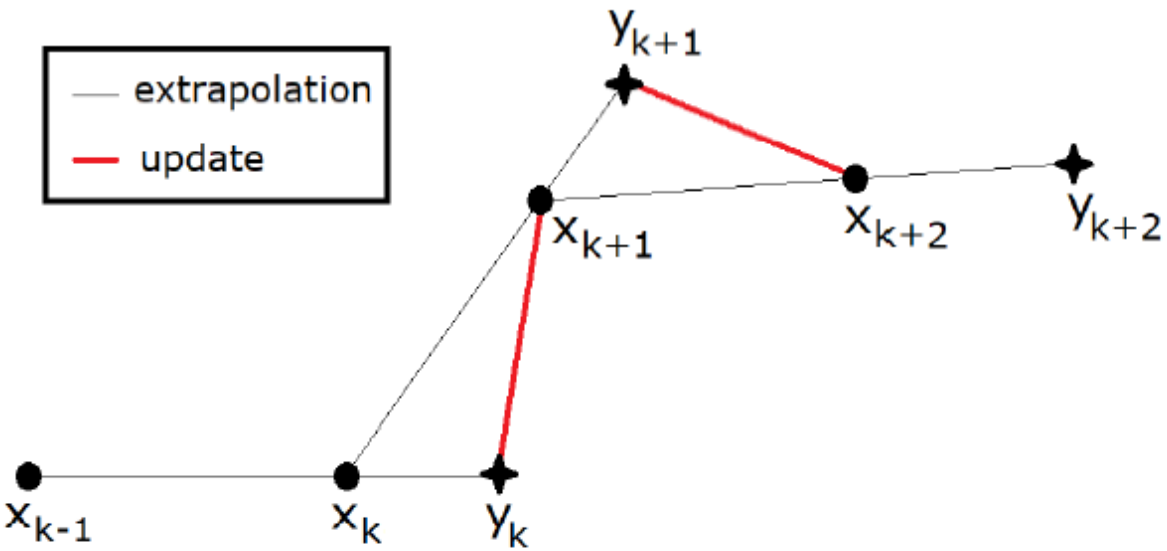


Figure taken from *ANG et GILLIS Accelerating nonnegative matrix factorization algorithms using extrapolation.*

How to choose the parameter β_k ?

There are several ways ! (test and see what works ☺)

- Constant (lazy way)
- Nesterov scheme: $\beta_k = \frac{\alpha_k \cdot (1-\alpha_k)}{\alpha_k^2 + \alpha_{k+1}}$ with $\alpha_{k+1} = \frac{\sqrt{\alpha_k^4 + 4\alpha_k^2} - \alpha_k^2}{2}$, α_1 should be chosen between 0 and 1
- Paul Tseng scheme: $\beta_k = \frac{k-1}{k+2}$

Adaptive restart in acceleration schemes

- Classical GD guarantees descent at each iteration, AGD not necessarily (due to the momentum) !
- If the error increases, you should come back to classical GD for one step (then return to AGD as it's faster ☺)
- Restart procedure in the case of Nesterov's acceleration:

Algorithm 6 Adaptive restart

Suppose you have the 2 first iterates x_0, x_1 (see above) and therefore y_1
for $k = 1, \dots$ **do**
    Compute somehow β_{k+1}
    $x_{k+1} = y_k - t_k \nabla f(y_k)$
    $y_{k+1} = x_{k+1} + \beta_{k+1}(x_{k+1} - x_k)$
    if $f(x_{k+1}) > f(x_k)$ **then**
        $y_{k+1} = x_{k+1}$ % at the next iteration, we will have classical GD
    end if
end for

Outline

Test phase

Once you have trained your model on the training data, you need to predict the class of the test data X_test.

- For the test document of index j, we have $\hat{Y\_test}(:,j) = \sigma(W_1^T X\_test(:,j) + b_1)$
- The class of $X\_test(:,j)$ is given by looking at the index of the maximum entry in the output one-hot-encoded vector (which is similar to probabilities) $\hat{y\_test}_j = \underset{k=1,\dots,p}{\operatorname{argmax}} \hat{Y\_test}(k,j)$
- Typical accuracy measure: percentage of the well classified data

$$ACC = \frac{\sum_{j=1}^{n} \mathbb{1}(\hat{y\_test}_j, y\_test_j)}{n}$$

where $\mathbb{1}(a,b) = 1$ if $a = b$ and 0 otherwise (indicator function)

Some remarks

- Stopping criterion:
  - Maximum number of iterations
  - (Relative) difference between two successive values of the objective function inferior to a tolerance
- W_1 and b_1 can be initialized by random entries between -0.5 and 0.5
- Example of expected result: