modules/helpers_and_algorithms.py

```python
# -*- coding: utf-8 -*-
"""some helper functions."""
import matplotlib.pyplot as plt
import numpy as np

import numpy as np
from numpy.random import randn
from numpy.random import multivariate_normal
from scipy.linalg import toeplitz
from math import sqrt


def inspector(model, n_iter, verbose=True):
    """A closure called to update metrics after each iteration.
    Don't even look at it, we'll just use it in the solvers."""
    objectives = []
    it = [0] # This is a hack to be able to modify 'it' inside the closure.
    def inspector_cl(w):
        obj = model.loss(w)
        objectives.append(obj)
        if verbose == True:
            if it[0] == 0:
                print(' | '.join([name.center(8) for name in ["it", "obj"]]))
            if it[0] % (n_iter / 5) == 0:
                print(' | '.join([("%d" % it[0]).rjust(8), ("%.6e" % obj)
.rjust(8)]))
            it[0] += 1
    inspector_cl.objectives = objectives
    return inspector_cl

def plot_callbacks(callbacks, names, obj_min, title):

    plt.figure(figsize=(6, 6))
    plt.yscale("log")

    for callback, name in zip(callbacks, names):
        objectives = np.array(callback.objectives)
        objectives_dist = objectives - obj_min
        plt.plot(objectives_dist, label=name, lw=2)

    plt.tight_layout()
    plt.xlim((0, len(objectives_dist)))
    plt.xlabel("Number of passes on the data", fontsize=16)
    plt.ylabel(r"$f(w_k) - f_\star$", fontsize=16)
    plt.legend(loc='lower left')
    plt.title(title, fontsize=16)
    plt.tight_layout()
    return plt

def gd(model, w0, step,  n_iter, callback, verbose=True):
    """Gradient descent
    """
    #step = 1 / model.lip()
    w = w0.copy()
```

```python
    w_new = w0.copy()
    if verbose:
        print("Lauching GD solver...")
    callback(w)
    for k in range(n_iter + 1):
        w_new[:] = w - step * model.grad(w)
        w[:] = w_new
        callback(w)
    return w


def sgd(model, w0, n_iter, step, callback, stepsize_strategy="constant",
        pr_averaging=False, verbose=True):

    """Stochastic gradient descent.

    stepsize_strategy:{"constant", "strongly_convex", "decaying"}
        define your own strategies to update (or not) the step size.
    pr_averaging: True if using polyak-ruppert averaging.
    """

    mu = model.strength
    w = w0.copy()
    w_averaged = w0.copy()
    callback(w)
    n_samples = model.n_samples
    L = model.lip_max()
    it = 0
    for idx in range(n_iter):

        idx_samples = np.random.randint(0, model.n_samples, model.n_samples)
        for i in idx_samples:
            if stepsize_strategy == "constant":
                stepsize = step / (2*L)
            elif stepsize_strategy == "strongly_convex": ##### A enlever si pas traité en cours
                # For strongly-convex (choice in the slides)
                stepsize = step / max(mu*(it + 1), L)
            elif stepsize_strategy == "decaying":
                stepsize = step / (L * np.sqrt(it + 1))
            else:
                raise ValueError('The strategy is not correct')

            w -= stepsize * model.grad_i(i, w)

            if pr_averaging:
                # Polyak-Ruppert averaging
                w_averaged = it/(it+1)*w_averaged + 1/(it+1)*w
            it += 1
        if pr_averaging:
            callback(w_averaged)
        else:
            callback(w)
    if pr_averaging:
        return w_averaged
    return w

```

```python

def agd(model, w0, n_iter, callback, verbose=True, momentum_strategy="constant"):
    """(Nesterov) Accelerated gradient descent.

    momentum_strategy: {"constant","convex","convex_approx","strongly_convex"}
        define your own strategies to update (or not) the momentum coefficient.
    """
    mu = model.strength
    step = 1 / model.lip_max()
    w = w0.copy()
    w_new = w0.copy()
    # An extra variable is required for acceleration
    z = w0.copy() # the auxiliari point at which the gradient is taken
    t = 1. # Usefull for computing the momentum (beta)
    t_new = 1.
    if verbose:
        print("Lauching AGD solver...")
    callback(w)
    for k in range(n_iter + 1):
        w_new[:] = z - step * model.grad(z)

        if momentum_strategy == "constant":
            beta = 0.9
        elif momentum_strategy == "convex":
            # See https://blogs.princeton.edu/imabandit/2018/11/21/a-short-proof-for-nesterovs-momentum/
            # Optimal momentum coefficinet for smooth convex
            t_new = (1. + sqrt(1. + 4. * t * t)) / 2.
            beta = (t - 1) / t_new
        elif momentum_strategy == "convex_approx":
            beta = k/(k+3)
        elif momentum_strategy == "strongly_convex":

            if mu>0: ##### Regularization is used as a lower bound on the strong convexity coefficient
                kappa = (model.lip_max())/(mu)
                beta = (sqrt(kappa) - 1)/(sqrt(kappa) + 1) # For strongly convex
            else:
                beta = k/(k+3)
        else:
            raise ValueError('The momentum strategy is not correct')

        z[:] = w_new + beta * (w_new - w)
        t = t_new
        w[:] = w_new

        callback(w)
    return w

def heavy_ball(model, w0, n_iter, step, momentum, callback, verbose=True):

    w = w0.copy()
    w_previous = w0.copy()
    callback(w)

    for idx in range(n_iter):
        w_next = w - step * model.grad(w) + momentum * (w - w_previous)
```

```python
        w_previous = w
        w = w_next
        callback(w)
    return w

def heavy_ball_optimized(model, w0, n_iter, callback, verbose=True):

    mu = model.strength
    L = model.lip_max()

    gamma = 3.99 / (sqrt(L) + sqrt(mu))**2
    beta = ((sqrt(L) - sqrt(mu)) / (sqrt(L) + sqrt(mu)))**2

    print(gamma, beta)

    return heavy_ball(model=model,
              w0=w0,
              n_iter=n_iter,
              step=gamma,
              momentum=beta,
              callback=callback,
              verbose=verbose,
             )
```