

## Homework 2 : Momentum accelerated and Normalized GD, Hessians and Linear Regression

You should submit source a python notebook along with a pdf of the notebook for all exercises. If any files are distributed for this assignment they will be available in /home/3G03/HW1 on phys-ugrad

*Due: Thursday February 11 2021, 11:30pm on avenue, with a grace period till the following Monday at 11:30pm*

☛ **Reading:** Parts of the first lectures are based on *The Hundred Page Machine Learning Book* by A. Burkov. Found at <http://the100mlbook.com>. Since then we have gone through Chapters 2,3,4 and parts of App A,B in *Machine Learning Refined*. We're continuing with parts of Chapters 5,6. An early version of *Machine Learning Refined* is available at [https://github.com/jermwatt/machine\\_learning\\_refined](https://github.com/jermwatt/machine_learning_refined)

### Exercise 1 *Momentum Accelerated GD*

In this exercise we you will implement a version of Momentum Accelerated Gradient Descent. Your can base your implementation on the version of gradient descent from last weeks python notebook:

```
import autograd.numpy as np
from autograd import grad

# gradient descent function - inputs: g (input function),
# alpha (steplength parameter),
# max_its (maximum number of iterations), w (initialization)
def gradient_descent(g,alpha,max_its,w):
    # compute gradient module using autograd
    gradient = grad(g)

    # run the gradient descent loop
    weight_history = [w]      # container for weight history
    cost_history = [g(w)]     # container for corresponding cost function history
    for k in range(max_its):
        # evaluate the gradient, store current weights and cost function value
        grad_eval = gradient(w)

        # take gradient descent step
        w = w - alpha*grad_eval

        # record weight and cost
        weight_history.append(w)
        cost_history.append(g(w))
    return weight_history,cost_history
```

Here,  $g$  is the function we're trying to minimize. `alpha` is the step length/learning rate. `max_its` is the maximum number of iterations to be taken and `w` the starting vector. Momentum accelerated GD changes this and instead uses an exponentially averaged descent direction according to:

$$\begin{aligned} \mathbf{d}^{k-1} &= \beta \mathbf{d}^{k-2} + (1 - \beta)(-\nabla g(\mathbf{w}^{k-1})) \\ \mathbf{w}^k &= \mathbf{w}^{k-1} + \alpha \mathbf{d}^{k-1} \end{aligned} \quad (1)$$

with  $\mathbf{d}^0 = -\nabla g(\mathbf{w}^0)$ .

**1.1** Modify the above GD implementation to perform Momentum Accelerated GD use `alpha` as step length.

☛ **Note:** Note that `autograd` has a version of `grad` that returns both the value of the function and it's gradient. They can be used in the following manner:

```
from autograd import value_and_grad
gradient = value_and_grad(g)
cost_eval, grad_eval = gradient(w)
```

**1.2** We now want to study a simple quadratic function:

$$g(\mathbf{w}) = \mathbf{w}^T \mathbf{C} \mathbf{w} \quad (2)$$

with the matrix  $\mathbf{C}$  given by:

$$C = \begin{pmatrix} 0.5 & 0 \\ 0 & 9.75 \end{pmatrix} \quad (3)$$

$$g(\mathbf{w}) = \mathbf{w}^T \mathbf{C} \mathbf{w} \quad (4)$$

in python we can implement this in a straight forward manner as shown below.

```
C = np.array([[0.5,0],[0,9.75]])
g = lambda w: np.dot(np.dot(w.T,C),w)
```

Fix  $\alpha$  at 0.1 and perform 25 steps for 3 values of  $\beta = 0, 0.1$  and  $0.7$  in each case starting from  $\vec{w}^0 = (10, 1)$ . Make a contour plot of  $g(\vec{w})$  and indicate the `weight_history` on the plot for each of the 3 values of  $\beta$ . Also plot the `cost_history`

### Exercise 2 *Slow-crawling behavior of gradient descent*

In this exercise we will compare the standard and fully normalized gradient descent schemes in minimizing the function

$$g(w_1, w_2) = \tanh(4w_1 + 4w_2) + \max(1, 0.4w_1^2) + 1. \quad (5)$$

The fully normalized gradient descent can for instance be implemented using:

$$\mathbf{w}^k = \mathbf{w}^{k-1} - \frac{\alpha}{\|\nabla g(\mathbf{w}^{k-1})\|_2 + \epsilon} \nabla g(\mathbf{w}^{k-1}) \quad (6)$$

Using an initial vector  $\mathbf{w}^0 = (2, 2)$  make a run of 1000 steps of standard gradient descent as well as with fully normalized gradient descent. In both cases use a step length of  $\alpha = 0.1$ . Plot

the cost function history for both runs and comment on the progress made with each approach. Comment on your observations.

### Exercise 3 *Student Debt - Linear Regression using a single Newton Step*

The data sheet `student_debt_data.csv` contains data for the US student debt as a function of the year. In this exercise we will use linear regression exactly (no gradient descent) to find the best linear fit.

**3.1** Fit a linear model to the data by minimizing the associated linear regression Least Squares problem *using a single Newton step*. The Newton step actually corresponds to solving a set of linear equations of the form  $\mathbf{A}\mathbf{w} = \mathbf{b}$  for the vector  $\mathbf{w}$ . As explained in class and in Watts et al the equations are:

$$\left( \sum_{p=1}^P \dot{\mathbf{x}}_p \dot{\mathbf{x}}_p^T \right) \mathbf{w} = \sum_{p=1}^P \dot{\mathbf{x}}_p y_p \quad (7)$$

For importing the data it is useful to use:

```
import pandas as pd
# import the dataset
csvname = datapath + 'student_debt_data.csv'
data = np.asarray(pd.read_csv(csvname, header = None))
```

Then we can extract the input, turn the array into a column vector and insert a first column of ones:

```
# extract input
x = data[:,0]
x.shape = (len(x),1)

# pad input with ones
o = np.ones((len(x),1))
x_new = np.concatenate((o,x),axis = 1)
```

Then we can extract the input:

```
# extract input
x = data[:,0]
y = data[:,1]
```

What is the dimension of the matrix you need to perform the Newton step ? Once you have the matrix, let's call it  $\mathbf{A}$ , you can calculate the inverse using `np.linalg.pinv(A)`.

**3.2** Make a plot of the data along with the fitted line.

**3.3** If the trend continues, use your model to predict the total student debt in 2050 ?

### Exercise 4 *Linear regression with optimization*

Lets now try to do linear regression on a data set but this time using gradient descent to find the optimal solution. As in the previous exercise we could have found the minimum exactly but it's a nice exercise to use gradient descent. The data set we will be working with is `kleibers_law_data.csv`. After collecting and plotting a considerable amount of data comparing the body mass versus metabolic rate (a measure of at rest energy expenditure) of a variety of animals, early twentieth-century biologist Max Kleiber noted an interesting relation ship between the two values. Denoting by  $x_p$  and  $y_p$  the body mass (in kg) and the metabolic rate (in KJ/day) of a given animal respectively, treating the body mass as the input feature

Kleiber noted (by visual inspection) that the natural log of these two values were linearly related. That is:

$$w_0 + \log(x_p)w_1 \simeq \log(y_p). \quad (8)$$

**4.1** First take the log of  $x$  and  $y$  then define a python function `model` according to the above description. Then define a second python function `least_squares` corresponding to the least squares sum.

**4.2** Use gradient descent to find the minimum of the cost function. Perform 1000 iterations with  $\alpha = 0.01$ , starting from a random point in the two dimensional plane, not too far from the origin.

**4.3** Make a plot of the data along with the fitted linear model. Make another plot of `cost_history` versus iterations.

**4.4** Use the fitted line to determine how many calories an animal weighing 10kg requires (note each calorie is equivalent to 4.18J).