

## 1. Linear regression function by Gradient Descent.

This is my core linear regression code (using momentum).

```

1. while True:
2.     # one training epoch
3.
4.     # reset temp variables
5.     loss = 0
6.     b_grad = 0
7.     w_grad = np.zeros((x_row_num, x_col_num))
8.
9.     # calculate loss and gradient over all training data
10.    for month in range(12):
11.        for hour in range(470):
12.            # extract x and y from training data
13.            # train_array_months is a list of 12 arrays
14.            # x will be an ?x9 numpy array. '?' depends on how we process training
            data
15.            x = train_array_months_processed[month][:, hour:hour+9]
16.            y_data = train_array_months[month][9, hour+9]
17.
18.            # calculate y from x, w, b
19.            y_pred= einsum('ij,ij', x, w) + b
20.
21.            # update loss
22.            diff = y_data - y_pred
23.            loss += diff * diff
24.
25.            # update gradient of b and w
26.            mult = 2 * diff * (-1)
27.            b_grad += mult
28.            w_grad += mult * x # 2 * diff * (-1) * x
29.
30.        # calculate rms error
31.        loss = sqrt(loss/(12*470))
32.
33.        # save model
34.        if total_epoch % 100 == 0:
35.            model_file_info = "epo" + str(total_epoch) + "_los" + str(loss)[:6]
36.            print(model_file_info)
37.            np.savez(MODEL_FOLDER + model_file_info, b=b, w=w, vb=vb, vw=vw, lr=lr, total_epoch=total_epoch)
38.
39.            # stop training in some cases
40.            if loss < 6:
41.                break
42.
43.        # update parameters
44.        vb = lr * b_grad + gamma * vb
45.        vw = lr * w_grad + gamma * vw
46.        b = b - vb
47.        w = w - vw
48.        total_epoch += 1
49.
50. print('end of training')

```

## 2. Describe your method.

- Pre-process training data
  - 我嘗試過四種方式來構成  $x$ ：
    1. 只使用 PM2.5 的資料，成為  $1 \times 9$  的陣列。
    2. 使用全部 data，成為  $18 \times 9$  的矩陣。
    3. 從 18 列中抽出與 PM2.5 較相關的 11 列，成為  $11 \times 9$  的矩陣。  
(判斷相關性的方法是把每一列跟 PM2.5 那列算相關係數 (Pearson r) 的絕對值，再取出前 11 高的。)
    4. 承 2，再將 PM2.5 取平方形成新的列，成為  $12 \times 9$  的矩陣。
  - 我的最佳結果出自第 4 種方法。
- Gradient Descent Optimization
  - 我嘗試過以下幾種方式來更新參數
    1. 單純的根據  $\text{learning\_rate} * \text{gradient}$
    2. 使用  $\text{momentum} = 0.9$
    3. 使用 Adagrad
  - 但因為 Adagrad 比較晚實現，train 的時間比 momentum train 的時間短，所以最後最佳結果出自使用 momentum 的。

## 3. Discussion on regularization.

我另外寫了一個程式 (hw\_linear-regression\_part-data\_try-regularization) 來測試 regularization 的效果。使用相同的初始參數。使用的 data 是前面所述的抽出 11 列的版本。使用的 learning rate 是  $4 * 10^{-9}$

測試的 lambda 的值有 0, 1, 10, ...,  $10^6$ ，共 8 個。

我使用不同的 Lambda 值跑 1000 個 iterations 後得到以下結果

lambda	0	1	10	$10^2$	$10^3$	$10^4$	$10^5$	$10^6$
Test error	6.5504	6.5502	6.5459	6.5354	6.4380	5.9578	6.2418	7.8717

可見在相同的 iteration 次數之下，加上 regularization 以及適當的 lambda ( $10^4$ ) 後，可以加快收斂速度，得到更低的 test error。大於或小於那個適當的 lambda 值，都會使 error 增加。

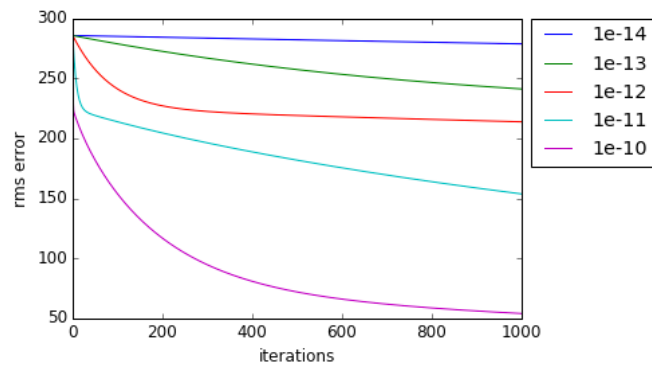
## 4. Discussion on learning rate.

我另外寫了一個程式 (hw\_linear-regression\_all-data\_try-lr.ipynb) 來測試適合的 Learning Rate， $x$  使用  $18 \times 9$  的 matrix，沒有做額外處理。

測試的 Learning Rate 有 1,  $10^{-1}$ , ...,  $10^{-14}$ ，共 15 個。

測試結果是：

- 當 Learning Rate  $\geq 10^{-9}$ ，每更新一次  $w$ ,  $b$ ，error 就會不斷擴大，無法收斂。  
這部分沒有畫在下圖中
- 當 Learning Rate  $\leq 10^{-10}$ ，error 可以收斂且  $10^{-10}$  的收斂速度最快，且 Learning Rate 越小，收斂的越慢，如下圖



可見對於原始資料而言，最適合的起始 Learning Rate 在  $10^{-10}$  附近。  
值得注意的是，Learning Rate 只要低於無法收斂的臨界值 ( $10^{-9}$ )，就會直接到接近最佳的 Learning Rate ( $10^{-10}$  左右)。