1. Linear regression function by Gradient Descent

每進入一個 iteration 都有這幾件事情要做,

```
while True:
    t = t + 1
    y_ = np.dot(x[:train_set_size], w)
    L = np.sum((y[:train_set_size] - y_) ** 2) / (2 * train_set_size) + LAMBDA * np.sum(w**2) / 2
    gw = np.dot(-x[:train_set_size].transpose(), (y[:train_set_size] - y_)) / train_set_size
```

1. 算出使用目前的 weight 所預測出來的 y, 也就是我的 y_
2. 定義 L, 並計算出值
3. 算出每一個 weight 的 gradient 也就是 L 對每一個 weight 的偏微分, 如圖上的 gw, 這邊我直接使用 matrix 相乘來一次做完每一筆料的 gradient

```
if Optimizer == "NON":
    w -= Learning_rate * gw
```

```
elif Optimizer == "Adagrad":
    gw_his += (gw)**2
    w -= (Learning_rate / np.sum(gw_his, axis = 0)**0.5) * gw
```

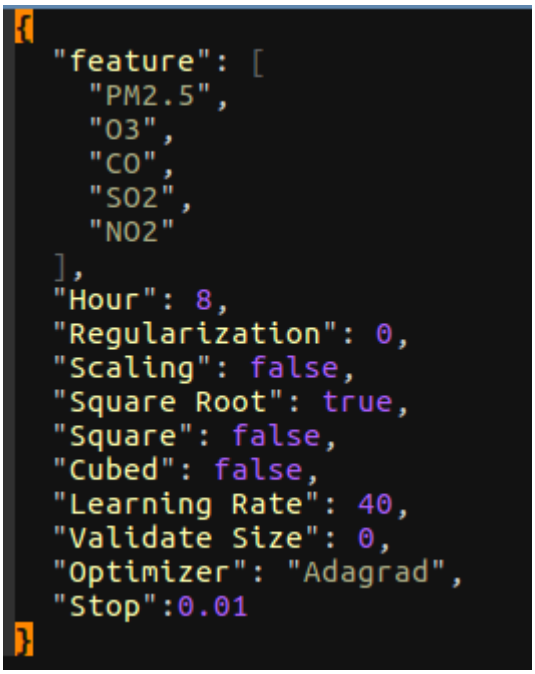
4. 最後一步就是更新 weight, 假設選擇不優化 training 的話, 則更新 weight 就非常簡單, 只要將 gradient * learning rate 減掉即可, 意義上就是往梯度方向移動 gradient*leanrning rate, 若選擇使用 Adagrad, 則必須計算歷史的 gradient, 為了讓 learning rate 能夠在每個方向及不同時間上有更適合的值

2. Describe my method

首先, 在取資料上, 一開始我先將 18 個 feature 先 hash 成 0~17 的數字, 以便我在 numpy 矩陣上 indexing 好操作, 再來, 資料處理上, 我將資料 parse 成一個 18 列的二維大矩陣, 每一列代表一個 feature 的所有資料, 下圖的 train_data 就是我的 18 列的大矩陣

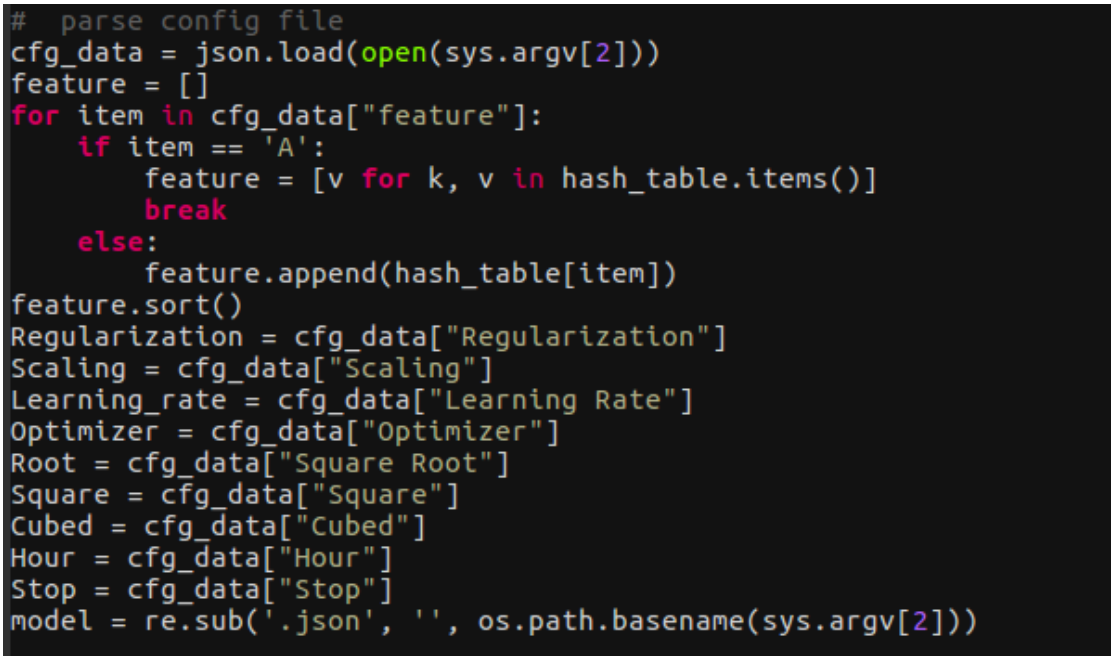
```
hash_table = {
    "AMB_TEMP":0, "CH4":1, "CO":2, "NMHC":3, "NO":4, "NO2":5,
    "NOx":6, "O3":7, "PM10":8, "PM2.5":9, "RAINFALL":10,
    "RH":11, "SO2":12, "THC":13, "WD_HR":14, "WIND_DIREC":15,
    "WIND_SPEED":16, "WS_HR":17
}
inv_hash_table = {v: k for k, v in hash_table.items()}

#  prepare data
train_data = [[] for i in range(18)]
raw_file = open('./data/train.csv')
for idx,row in enumerate(raw_file):
    if idx > 0:
        data = re.sub('\r|\n', '', row).split(',')
        data_float = []
        for x in data[3:]:
            if x != 'NR':
                data_float.append(x)
            else:
                data_float.append(0)
        train_data[hash_table[data[2]]] += data_float
```

資料處理的最後就是產生 X，因為這次作業涉及到相當多的參數，我將 code 寫成吃 model file 的方式來實作，首先我先定義出 model 的 json 檔，如下圖，



```
{
  "feature": [
    "PM2.5",
    "O3",
    "CO",
    "SO2",
    "NO2"
  ],
  "Hour": 8,
  "Regularization": 0,
  "Scaling": false,
  "Square Root": true,
  "Square": false,
  "Cubed": false,
  "Learning Rate": 40,
  "Validate Size": 0,
  "Optimizer": "Adagrad",
  "Stop":0.01
}
```

這個 json 檔首先定義出 model 的 feature 有哪些，再來是從第九小時往回算，要取連續幾小時作為 feature，跟 X 有關的還有 Square Root, Square, Cubed,這邊代表的是我的 model 到的包含哪幾種 X 的 term，舉例來說，若 Square Root 是 true，則表示我的 model 是 y_ = b + w0x0 + w1(x0^0.5)，以此類推，所以以這個 model 當例子，我的 X 有 5 (feature)* 2(x,x^0.5) * 8(hour) + 1(bias) = 81 個 column，下圖為實作吃 model json 檔的 code



```
#  parse config file
cfg_data = json.load(open(sys.argv[2]))
feature = []
for item in cfg_data["feature"]:
    if item == 'A':
        feature = [v for k, v in hash_table.items()]
        break
    else:
        feature.append(hash_table[item])
feature.sort()
Regularization = cfg_data["Regularization"]
Scaling = cfg_data["Scaling"]
Learning_rate = cfg_data["Learning Rate"]
Optimizer = cfg_data["Optimizer"]
Root = cfg_data["Square Root"]
Square = cfg_data["Square"]
Cubed = cfg_data["Cubed"]
Hour = cfg_data["Hour"]
Stop = cfg_data["Stop"]
model = re.sub('.json', '', os.path.basename(sys.argv[2]))
```

接下來就是利用這些從 json 檔讀出來的資料來製作用來 training 的 X 和 Y 了

```
#  prepare x and y
train_data_arr = np.asarray(train_data, dtype=np.float32)
x = []
y = []
for i in range(len(train_data_arr[0]) - 9):
    x.append(train_data_arr[feature, i+9-Hour:i+9])
    y.append(train_data_arr[hash_table["PM2.5"], i+9])
x = np.asarray(x, dtype = np.float32)
y = np.asarray(y, dtype = np.float32)
x = x.reshape(x.shape[0], Hour * len(feature))

#  featrue scaling
if Scaling == True:
    x_mean = np.mean(x, axis = 0)
    x_std = np.std(x, axis = 0)
    x = (x - x_mean) / x_std
bias = np.ones(shape = (x.shape[0], 1))
x_root = (x + 10)**0.5
x_2 = x**2
x_3 = x**3
if Root == True:
    x = np.concatenate((x, x_root), axis = 1)
if Square == True:
    x = np.concatenate((x, x_2), axis = 1)
if Cubed == True:
    x = np.concatenate((x, x_3), axis = 1)
x = np.concatenate((bias, x), axis = 1)
```

在 X 的準備上我使用剛剛的 feature list 以及 Hour 來讀出 train_data_arr 相對應的資料，每讀出連續 Hour 筆資料就將資料 append 到 X list 中，還有將第十小時的 PM2.5 append 到 Y list 中，最後將 X, Y 轉成 numpy array，並根據 square cubed 等等的值來產生最後的 X 的樣子，做到這邊就產生出 X 和 Y 了，，再來就是 training 的部分，在上面的 json 檔上有定義一些有關 training 的參數, regularization 的值就是 Lambda，還有 Learning rate 跟 training set 的大小，這邊這個 validate_set_size 是為了讓我測試哪一個 model 比較好才需要設定的，扣掉這個 size，就是 training set 的 size 了，另外 optimizer 就是設定更新 weight 的時候的演算法,最後的 stop 就是我利用 mean of gradient of weight 來作為 iteration 停下來的依據. 最後，講一下我選 feature 的依據，因為我只要改 json 檔就能獲得不同的 Model，所以我在選 feature 上就利用不同的 model 算出不同 model 的 e_train 跟 e_test，最後選出一個 e_test 最小的 model 來做為我最終的 model.

3. Discussion on regularization

這次我有實作 regularization，在做測試時我採用後 1000 筆資料作為 validate set，剩下的資料作為 training set，針對不同的 Lambda 值結果如下

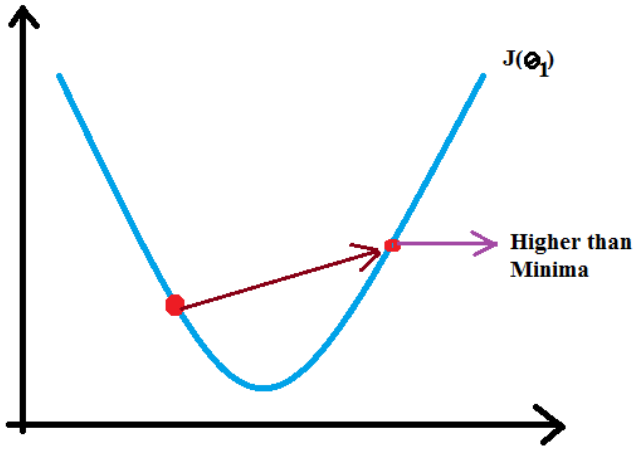
```
17:00:17
jeff0420@ubuntu ~/ML2016/hw1 [±master U:5 ?:6 ✗] python src/model.py vd cfg/model2.json weights/model2_LAMBDA_5_Adagrad_8_.weights
e_train =  4.35814805317 e_test =  4.44932027814

17:00:19
jeff0420@ubuntu ~/ML2016/hw1 [±master U:5 ?:6 ✗] python src/model.py vd cfg/model2.json weights/model2_LAMBDA_10_Adagrad_8_.weights
e_train =  4.34264456447 e_test =  4.46833457769
```

可以看到 Lambda 比較小的時候 e_test 比較低，但是 e_train 比較高，原因就

是因為當設定 Lambda 越高時，會影響到 model 的 bias，也就是讓 model 更平滑，此方法可能造成無法更好的 fit training set，因此 e_train 會比較高，但是卻能避免 overfitting，所以 e_test 比較低

4. Discussion on learning rate

在 training 的過程中，若 learning rate 設的太大可能會造成 w 更新太大步，導致 L 變的更大的情形，就像以下情況，



可以看到 W 一下子更新太大步導致,L 跳過頭的情形

```
iter801648 L: 51.5908647645
mean of gradient: 0.00127803884857 Stop:  1e-05
Learning_rate: 0.0001
Regularization: 5
Scaling: False
Optimizer: NON
feature: [2, 5, 7, 9, 12]
iter801649 L: 51.5908874274
mean of gradient: 0.00127803766561 Stop:  1e-05
Learning_rate: 0.0001
Regularization: 5
Scaling: False
Optimizer: NON
feature: [2, 5, 7, 9, 12]
iter801650 L: 51.5909100902
mean of gradient: 0.00127803648264 Stop:  1e-05
Learning_rate: 0.0001
Regularization: 5
Scaling: False
Optimizer: NON
```

上圖為我實作 learning rate 太大造成 L 網上的情形，可以看到在沒有使用 Adagrad 的情況下 0.0001 對於我的 model 來說已經太大了，導致 L 往上長，衝過頭的情形