ML_hw1 Report

電機四 B02504086 陳品融

1. Linear regression function by Gradient Descent.

```python
def gradient_descent(x, y_, x_val, y_val, w, lr, numIter, lambda_=0, save_best=True):
    #由於我的input data最後多接一個1給bias, 而regularization不包含bias, 因此需用lambda_arr作mask
    lambda_arr = lambda_ * np.ones(163)
    lambda_arr[-1] = 0
    #設定loss_best的初始值爲無限大, w_best的初始值爲0
    loss_best = float('inf')
    w_best = np.zeros(163)
    for i in range(numIter):
        #calculate_loss爲拿x跟w作內積得y然後最後return y_ - y的norm的平方除以x[0](data的數量)
        train_loss = calculate_loss(x, y_, w)
        val_loss = calculate_loss(x_val, y_val, w)
        #loss包含MSE以及regularization, 其對參數的偏微分計算如反白處, 最後乘上learning rate作update
        w = w - lr*2*(np.dot(x.t, (np.dot(x, w) - y_)) + lambda_arr*w) / len(x)
        #記錄在最低val_loss下的weight, 最後return的w也是這個值, 如此epoch數可調大一點, 不用去擔心overfit
        if save_best is True and val_loss < loss_best:
            loss_best = val_loss
            w_best = w
    return w_best

def calculate_loss(x, y_, w):
    y = np.dot(x, w)
    loss = (np.linalg.norm(y_ - y)**2) / len(x)
    return loss
```

2. Method

(1) 取 training feature: 我的 feature 是一個 163 維的向量，也就是把前 9 小時所有的觀測數據加上 bias 的 1 而得。process 的方式是以 20 天爲單位，把每天的 18*24 的 data 接在前一天之後，形成 18*480 的矩陣，再用 sliding window 的方式把每 9 小時及其對應第十小時的 pm2.5 分別存成 training 以及 label 的 array 並 reshape 成 162 維且加上 bias 的 1，最後 append 到一個 list 上，形成一個 5652*163 的 array，其 code 如下。

```python
train_arr = np.asarray(train_df, dtype=np.float32).reshape(240, 18, 24)
train_concat = train_arr[0]
train = []
label = []
for i in range(1, 241):
    if i % 20 != 0:
        train_concat = np.concatenate((train_concat, train_arr[i]), axis=1)
    else:
        for j in range(471):
            train.append(np.concatenate((train_concat[:, j:j+9].reshape(162), np.array([1]))))
            #train.append(train_concat[:, j:j+9].reshape(162))
            label.append(train_concat[9, j+9])
        if i != 240:
            train_concat = train_arr[i]

train = np.asarray(train).reshape(5652, 163)
```

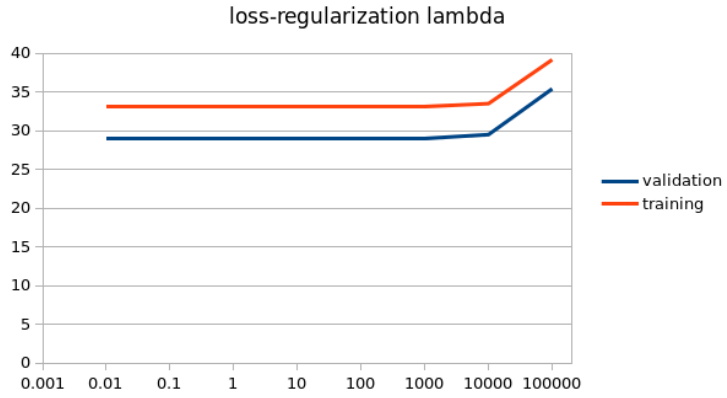
(2) validation_set: 我切了 5%的 data 作爲 validation_set，切的時候會先 random shuffle data。validation_set 可以用來判斷何時該停止 training，如果 training_loss 在下降而 validation_loss 卻不斷上升則可以判定是 overfit。

(3) save_best: 我存了 loss_best 及 w_best 爲到目前爲止最低的 validation_loss 及其對應的 weight。雖然每個 epoch 都多作一點 operation 會有 overhead，但此 save_best 的方法可以讓我調大一點的 epoch 數也不用怕 overfit，省去一些判斷 gradient 小於 threshold 值要終止 training 的麻煩。

(4) batch_size 的選擇: 由於這次我的 training data 只有 5652*0.95=5369 筆，算是沒有很多，而且 loss function 是一個簡單的 convex 二次曲線，因此我選擇每次直接把這 5369 筆 data 丟下去 train。實際跑的結果，速度還在可接受的範圍內，而且 loss 在每個 epoch 之後幾乎都會穩定下降，不會像 mini-batch 那樣可能產生一些震盪。

3. Discussion on regularization

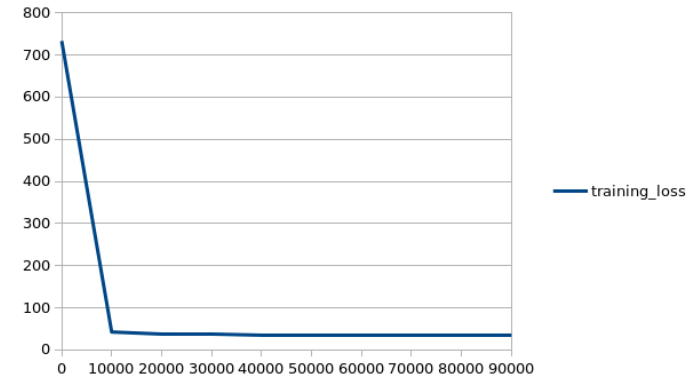
lambda	0.01	0.1	0	1	10	100	1000	10000	100000
val_loss	28.97595	28.97596	28.97595	28.97598	28.97621	28.97858	29.00486	29.49847	35.40807
train_loss	33.09541	33.09541	33.09541	33.09541	33.09545	33.09585	33.10361	33.50168	39.16087



由圖可知，如果 lambda 加的太大的話，會很明顯地影響 gradient decsent 的方向，導致最後收斂在一個不好的地方。然而比較奇怪的是，當 lambda 在一個適合的大小時，似乎對 validation_loss 不會產生什麼效果，甚至還有稍微變大的趨勢，對此，我個人的推測有以下兩點：首先，由於 val_data 只取 5%有點少且剛好取到與 training_data 相去不遠的 distribution，，導致 regularization 沒有發揮其功效；另外一個我覺得比較可能的原因是最後得到的 weight 值本來就都很小，因此加與不加 regularization 沒什麼差別。

4. Discussion on learning rate

learning rate 對這次整個 training 的影響非常顯著，根據觀察，當 learning rate 超過大約 2e-6 時，gradient descent 不但不會讓 loss 下降，甚至一下子就會衝到 infinite。經過一番 trial-and-error 之後，我把 learning rate 設爲 1e-6，並且整個 training 的過程中都固定，沒有使用 adaptive 的方法，原因是我覺得 loss function 是簡單的二次式，調小一點雖然一開始比較慢，但是最後可以收斂到比較好的值。下圖爲 learning rate 爲 1e-6 的 loss 對 epoch 的圖，可以觀察到其實一開始更新的速度還在可接受的範圍。



5. Other Discussion(kaggle_best implementation)

(1) model 架構

我是用一個兩層 hidden layer 的 dnn model，第一層的參數 W1 爲 162*32 的矩陣，第二層則爲 32*8，最後 output layer 輸出一個 scaler。hidden layer 的層數及每層的 size 是經過不斷地 trial-and-error 試出來的結果。根據 training 的觀察結果，我發現如果 hidden layer 的 size 越小，參數越少，則越快收斂到一個極值，只不過 public score 會卡在一個瓶頸無法再進步；hidden layer size 越大則反之，不過值得注意的是參數越多則越容易有 overfit 的情況發生。

(2) initialize

我參數是使用 normal distribution 來 initialize，需注意的是 variance 這個參數要稍微調整一下，在這個 case 下可能不能太大。我一開始是使用 standard normal distribution，結果三不五時 loss 就掉不下去，後來把 variance 調成 1e-4，mean 維持不變，training 的情況就穩定許多。

(3) learning rate & optimizer

learning rate 的選擇對於 dnn 可說是至關重要，如果 training 的過程中皆使用相同的 learning rate 應該 train 不太起來，因此必須採用 adaptive learning rate 的方式去 optimize。我一開始使用的是 Adagrad，其原理就是用一次微分的值去 approximate 二次微分並製造一個反差的效果，實作上就是把每次微分的平方累加起來再取根號放在分母。Adagrad 實際跑起來效果其實還不錯，大概只需 100000 個 epoch 以內即可得到不錯的 loss，只不過後來爲了得到更好的 public score，因此我改採用更複雜的 Adam。Adam 同時計算了微分與微分平方並用參數 beta1 與 beta2 去 weighted average，並用一個 bias-corrected 的 term 1-beta^t，其中 t 爲第幾次 epoch，最後 update 時分子放一次項，分母放平方項的微分取根號並加一個小 epsilon 值避免分母爲 0。Adam 比 Adagrad 多利用了動量的原理，因此較有可能跳脫 local minima，不過也因此 Adam 在更新時比較容易有微小的震盪。實際用 Adam train 會發現速度快許多，大概 2~3000 個 epoch 就會收斂。

(4)gradient

算 gradient 其實就是實際 implement backpropagation 以達到較高的效率。