Machine Learning HW#2 Report

Logistic Regression　　電機四 b02901126 張大方

1. Logistic Regression

```
#define paremeters
Eta = 0.001
beta1, beta2 = 0.9, 0.999
epsilon = 1.0e-08
m_0 = np.zeros((58,1), dtype = np.float)
v_0 = m_0
t = 0
print(train_X.shape,w.shape)
for i in range(0, 10000, 1):
    Xw = np.dot(train_X, w)
    f_wb = sigmoid_arr(Xw)
    gradient = -np.dot(train_X.T, answer - f_wb)/4001.0
    ###ADAM
    t += 1
    lr_t = Eta * math.sqrt(1-(beta2**t)) / (1-(beta1**t))
    m_0  = beta1 * m_0 + (1-beta1) * gradient
    v_0  = beta2 * v_0 + (1-beta2) * (np.square(gradient))
    w = w - lr_t * m_0 / (np.sqrt(v_0)+epsilon)
    ###ADAM
```

參數解釋:

i. Eta: learning rate

ii. Beta1, beta2, epsilon, m_0, v_0, lr_t:皆為 Adam optimizer 會用到的參數

iii. train_X: 就是 training data X 項，是一個 4001*58 的矩陣

iv. w: weight，是一個 58*1 的矩陣

v. sigmoid_arr(x): 回傳$1.0/1.0 + e^{-x}$

作法解釋:

方便起見，我將 bias 項放入 weight 的矩陣中當做第 58 個 feature。我將所有的 training data 當作一個矩陣來運算，所以每一次的 iteration 等於把 4001 個資料都用過了一遍。其中 gradient 的計算方式與作業一相同，利用矩陣運算的技巧處理，結果可以化簡成$\mathbf{\nabla L = 2X^T(Xw - \hat{y})}$。為一的差別在於，這裡的 Xw 換成經過 **sigmoid** 的$\boldsymbol{\sigma}(\mathbf{Xw})$。

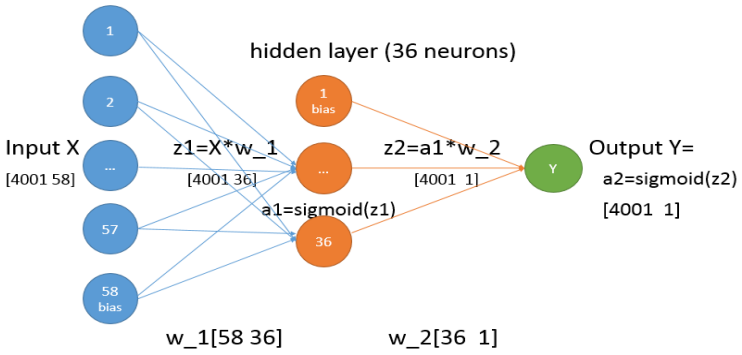
與作業一不同的是，這次選擇的是 **ADAM Optimizer**[1]，跟 Adagrad 不同。我並不了解其背後的數學原理，不過它的效果很顯著，能迅速收斂到理想的結果。我沒有使用 regularization 的方法，因此沒有 lambda 項。

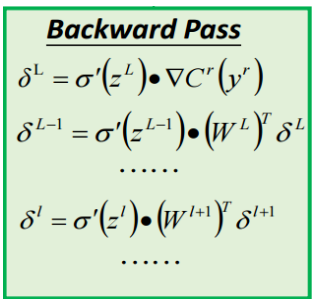
2. Neural Network[2]

```
for i in range(0, 2000, 1):
    #forward propagation
    z1 = np.dot(train_x, w_1)  #(4001, 40)
    a1 = sigmoid(z1)           #(4001, 40)
    a1.T[0].fill(1)            # bias
    z2 = np.dot(a1, w_2)       #(4001,  1)
    a2 = sigmoid(z2)           #(4001,  1)
    #backward propagation
    delta_3 = d_sigmoid(z2) * (-(answer/(a2+1e-30)+(1-answer)/(a2-1+1e-30)))
    delta_2 = np.multiply(d_sigmoid(z1), np.dot(delta_3, w_2.T))
                               #(4001, 40)
    dw2 = np.dot(a1.T, delta_3)          #(40,  1)
    dw1 = np.dot(train_x.T, delta_2)     #(58, 40)
    #regularization
    dw1 += reg * w_1
    dw2 += reg * w_2
    ###ADAM_w1
    t += 1
    lr_t = Eta * np.sqrt(1-(beta2**t))/(1-(beta1**t))
    m_1 = beta1 * m_1 + (1-beta1) * dw1
    v_1 = beta2 * v_1 + (1-beta2) * np.square(dw1)
    w_1 = w_1 - lr_t * m_1 / (np.sqrt(v_1)+e)
    ###ADAM_w2
    m_2 = beta1 * m_2 + (1-beta1) * dw2
    v_2 = beta2 * v_2 + (1-beta2) * np.square(dw2)
    w_2 = w_2 - lr_t * m_2 / (np.sqrt(v_2)+e)
```

參數解釋:

Forward: 以下圖解釋 z1, w_1, a1, z2, w_2, a2





Backward: 根據老師去年 MLDS 的講義[3]，得知 back propagation 的公式其中 **delta3 為 sigmoid 的微分乘上 Cross entropy 的 gradient**，要代入的參數 皆可以由 forward 得到。Delta2 算法類似，將 cross entropy 換成下一層的 weight transpose 乘上 delta3。有了 delta 後，便可將 weight_1, weight_2 的 gradient 算出來，再藉此調整每次 iteration 的 weight。

作法解釋:

與之前相同，我將所有 4001 筆資料放入一個大矩陣(4001*58)中，丟入 neural network 中跑。Gradient 的算法已在上面提過，不過在調整 l earning- rate 方面，也是同樣使用 **Adam Optimize**r，加快收斂速度。

在這方法中我有用 regularization 的方法，避免因為 train 太多次產生 over-fitting 的情形。Hidden layer 中第一個 Node 是為了**保留 bias** 能存在而設的，避免第二層之後沒有 bias 項。

3. Comparison

從兩個方法所得到的結果與 training data 比較 accuracy，可以明顯比較出差別。Logistic regression 的極限大概在 92%，傳上 Kaggle 得到 92.667 分；而對 NN 來說，僅僅 train 了 2000 次，在沒有發生 overfitting 的情況下，就可以得到約 97%的準度，傳上 Kaggle 也有約 94~95 分。

從結構來看也可以說結果本來就該如此，Logistic 是一種 linear 的方法，當 **data 分布無法以直線區隔**的時候，本來就有其極限所在。對 NN 來說，Logistic 就像是只有一個 neuron 的 network，而我所設定的 neuron 數有 36 個，其複雜度與結果想當然會比較好。其 **non-linear 的特性也更能 fit data 的分布情況**，較能明確地將兩個 class 區分出來。但是若過於為了 training data 調整，也會容易 overfitting 使的界線過於為了 training data 而產生，因此我加了 regularization 來避免，也讓 iteration 數不要太高。

4. Reference

[1] https://www.tensorflow.org/versions/r0.11/api_docs/python/train.html#AdamOptimizer

[2] http://www.wildml.com/2015/09/implementing-a-neural-network-from-scratch/

[3] http://speech.ee.ntu.edu.tw/~tlkagk/courses/MLDS_2015_2/Lecture/DNN%20backprop.pdf

討論同學:

電機四 劉致廷 電機四 陳緯哲 電機四 林圓方 電機四 趙佑毅