Machine learning HW2

B03902071 葉奕廷

1. Logistic regression function

在 Logistic regression 的實作方面，因為其 gradient descent 的更新方式與上次作業的 linear regression 一樣，所以可以輕易地藉由更改上次的 code 實作。

我首先將全部 training data 讀進兩個 numpy array x_train, y_train，x_train 存放 feature y_train 存放 label，array 的每個 row 都是一筆 data。 接著在每次 gradient discent 的時候便可以利用 numpy 提供的矩陣乘法高速的計算出 gradient。 w,b 全部初始值為 1。

```
dw = np.zeros(w.shape,dtype=np.float64) #weight gradient
db = 0. #bias gradient

y = sigmoid(np.dot(x_train,w) + b)
y_err = y_train - y

dw += x_train.T.dot(y_err) #compute gradient
db += y_err.sum() #compute gradient
```

接著我做了簡單的 regularization 並使用 adagrad 去調整 learning rate 來做每次的參數更新，並做了 20000 次更新參數。

```
dw += 0.1 * (np.abs(w)) #regularization
#adagrad
g = g + (dw ** 2)
gb = gb + (db ** 2)
deltaw = (0.1/np.sqrt(g))*dw
deltab = (0.1/np.sqrt(gb))*db
w = w + deltaw
b = b + deltab
```

最後我藉由 `numpy.savez` 去儲存 model。

2. Another mathod -- Neural Network

2.1 Implement detail

我實作了 2 ~ 3 hidden layers 的 Neural Network 做為第二種方法。 選擇 2 ~ 3 hidden layers 是因為查的資料發現以單純的 Neural Network 來講 2 ~ 3 層就很足夠了。

首先我發現如果想要使用 relu 作為 activation function 的話每一層的 w 的初始化必須使用 `numpy.random.randn(# of w) * numpy.sqrt(2.0 / (# of w))` 這個初始化方法否則 train 出來的結果會爛掉，雖然詳細的數學推導我並未細看但是為了在比較 sigmoid 和 relu 時方便我便統一使用了此初始化方法。

```
w0 = np.random.randn(57,neuralnum) * np.sqrt(2.0/(57 * neuralnum))
w1 = np.random.randn(neuralnum,neuralnum) * np.sqrt(2.0/(neuralnum ** 2))
w2 = np.random.randn(neuralnum,5) * np.sqrt(2.0/(neuralnum * 5))
w = np.random.randn(5,1) * np.sqrt(2.0/(5))

b0 = np.zeros((1,neuralnum))
b1 = np.zeros((1,neuralnum))
b2 = np.zeros((1,5))
b = np.zeros(1)
```

Training data 的儲存方面因為與 Logistic regression 一樣便不詳述。而在 neural network 計算 output 的時候也是利用 numpy 的矩陣乘法便可以快速的計算。

```
    x = x_train

    l0 = sigmoid(np.dot(x,w0) + b0)
    l1 = sigmoid(np.dot(l0,w1) + b1)
    l2 = sigmoid(np.dot(l1,w2) + b2)
    y = sigmoid(np.dot(l2,w) + b)
```

計算 gradient 的部分我實作了知名的演算法 back propagation 去加快計算。

```
#back propagation
#sigmoid(deriv=True) means it output the derivative of sigmoid function
y_error = y_train-y
y_delta = y_error * sigmoid(y,deriv=True)

l2_error = y_delta.dot(w.T)
l2_delta = l2_error * sigmoid(l2,deriv=True)

l1_error = l2_delta.dot(w2.T)
l1_delta = l1_error * sigmoid(l1,deriv=True)

l0_error = l1_delta.dot(w1.T)
l0_delta = l0_error * sigmoid(l0,deriv=True)

dw0 = x.T.dot(l0_delta)
dw1 = l0.T.dot(l1_delta)
dw2 = l1.T.dot(l2_delta)
dw = l2.T.dot(y_delta)
```

同樣我使用了 adagrad 去調整 learning rate.

```
#adagrad
g0 = g0 + (dw0 ** 2)
g1 = g1 + (dw1 ** 2)
g2 = g2 + (dw2 ** 2)
g = g + (dw ** 2)

deltaw0 = (learnrate/np.sqrt(g0))*dw0
deltaw1 = (learnrate/np.sqrt(g1))*dw1
deltaw2 = (learnrate/np.sqrt(g2))*dw2
deltaw = (learnrate/np.sqrt(g))*dw
w0 = w0 + deltaw0
w1 = w1 + deltaw1
w2 = w2 + deltaw2
w = w + deltaw

gb0 += np.mean(l0_delta, axis=0) ** 2
gb1 += np.mean(l1_delta, axis=0) ** 2
gb2 += np.mean(l2_delta, axis=0) ** 2
gb += np.mean(y_delta, axis=0) ** 2
b0 += (learnrate/np.sqrt(gb0)) * np.mean(l0_delta, axis=0)
b1 += (learnrate/np.sqrt(gb1)) * np.mean(l1_delta, axis=0)
b2 += (learnrate/np.sqrt(gb2)) * np.mean(l2_delta, axis=0)
b += (learnrate/np.sqrt(gb)) * np.mean(y_delta, axis=0)
```

2.3 Regularization

我嘗試了兩種 regularization 方法，第一種和 logistic regression 一樣是將 gradient 的值加上 `0.1 * np.abs(w)`，第二種則是 Dropout。

第一種方法(以下稱為m1)根據實驗結果看起來沒有什麼顯著的效果，但是若把 0.1 一口氣調高到 0.5 的話便會讓 training 結果變差，因為時間上的考量我便沒有慢慢的試參數了。

在 Dropout 的部分，我發現 Dropout + sigmoid 會在 train 一段時間過後 error 激增，而 Dropout + relu 則是可以穩定的降低error，並且可以觀察到加上 Dropout 比純用 relu 好了一點，並且在使用 Dropout 時 hidden-layer 的數量是 2 or 3 沒有太大影響。

2.4 Neuron Number

根據查到的資料，在每一層 hidden layer 該有多少 neuron 也是大有學問。 我查到主要有三種算法

1. `# of training sample / (alpha * (# of input neuron)), alpha = 2 ~ 10`
2. `(2/3) * (# of input neuron) + (# of output neuron)`
3. `((# of input neuron) + (# of output neuron))/2 + 1~10`

並且再根據一些實驗，在沒有使用 Dropout 的情況下 3-hidden layer 的 neuron number 我選擇 30,30,5， 2-hidden layer 則是 31,31。

使用 Dropout 的情況下必須要使用較多 neuron 才會有好的表現，所以在有 Dropout 時 3-hidden layer 和 2-hidde layer 的 neuron number 都是全部 45。

2.5 Experiment data

我使用了 5-fold cross validation 進行 model 的好壞比較，以下是 5 種 model。

1. 3-hidden layer NN, activation: sigmoid, regularization: None
2. 3-hidden layer NN, activation: sigmoid, regularization: m1
3. 2-hidden layer NN, activation: sigmoid, regularization: None
4. 2-hidden layer NN, activation: relu, regularization: Dropout

5. logistic regression

model	fold1	fold2	fold3	fold4	fold5	average
3-hidden NN	0.94125	0.93	0.925	0.915	0.93125	0.9285
3-hidden NN, m1	0.94375	0.91375	0.93	0.91125	0.92875	0.9255
2-hidden NN	0.9475	0.91375	0.8875	0.92625	0.9275	0.9205
2-hidden NN, Dropout	0.92875	0.92	0.9275	0.93125	0.92125	0.92575
logistic regression	0.9225	0.92625	0.91375	0.92	0.93625	0.92375

其實可以發現如果以 average 來看 model 間並沒有很大的差距，甚至可以說都在誤差範圍以內。 值得一提的就是 2-hidden layer NN without dropout 算是相對表現比較不穩定的一個，應該是 variance 較大的問題。 然後有沒有 m1 regularization 並沒有什麼大差別。 所以我將 model 的選項縮減到

1. 3-hidden layer NN without regularization
2. 2-hidden layer NN with Dropout
3. logistic regression

最後考量 training time 我刷掉 2-hidden layer NN with Dropout 因為若是加了 Dropout，gradient descent 的次數必須從 600 提升到 3500 才能有較好的結果。 然後再依據 kaggle public scoreboard 的結果我認為 3-hidden layer NN 是最好的。

3. Ensemble

在這次我使用了 ensemble 方法中的 bagging 來降低我的 model 的 variance。 我嘗試了 2 種做法

1. 使用 bootstrap 取 training data 來 train 20 個 model 進行投票。
2. 將 training data 分成 20 份，用第 `1~19`,`1~18,20`,`1~17,19~20` ・・・份的 training data 去 train 20 個 model 進行投票

我發現第二個做法有較好的結果，猜想是因為這樣子 training data 的 diversity 才夠高。 利用 ensemble 我再在 kaggle public scoreboard 上改善了 0.01的準確度。