Machine Learning Hw2 Report

R05921012

吳宗澤

1.

The input features that I pick are all the 57 kinds of features and also the root of the last 7 kinds of features, and it makes my input matrix X become 4001 x 64. After multiplying my input matrix to weight matrix W (64 x 1) and adding a bias b, it will equals to$(W \times X + b)$. Then, it is the logistic regression part that I make $(W \times X + b)$ into sigmoid function$\frac{1}{1+e^{-(W\times X+b)}}$. At last, I can get a predicted matrix $f_{W,b}(x)$that its values are all between 0 to 1.

Moreover, the lost function I use is the cross entropy$\sum -[\hat{y} \times \ln f_{W,b}(x) + (1 - \hat{y}) \times \ln(1 - f_{W,b}(x))]$. How I find the best function is to do derivative to the lost function just like the first assignment, and we can get$\frac{-\partial \ln L}{\partial \mathrm{W}} = -X^T \cdot \left(\hat{y} - f_{W,b}(x)\right)$ and $\frac{-\partial \ln L}{\partial \mathrm{b}} = -\left(\hat{y} - f_{W,b}(x)\right)$. The derivative will multiple to the transpose of X is because of the derivative of a matrix, and this is what professor didn't mention in the class. In addition, I use adaptive learning rate method adagrad to update my W and b. After finishing learning part, we can use our W and b to predict the testing data's feature and get the predicted value. If the value is bigger than 0.5, I will classify it as 1, and if it is lower than 0.5, I will classify it as 0.

In conclusion, the performance in this method is 0.9333, and it is just a little higher than the baseline.

2.

Because the high dimension input feature may need some feature transformation to do a better classification, I try neural network for another method. I keep the same 64 kinds of input features, then I try several different layers and neurons that are (3,4,3),(3,2),(16),(8),(5) in the hidden layers ((3,4,3)means in the first hidden layer there are 3 neurons,4 neurons in the next layer ,and 3 in the last hidden layer).

However, with the same learning iterations for those different kinds of neural network, the loss I get from them is (3,4,3)<(3,2)<(16)<(8)<(5). What's strange is that the highest performance in the public set is (5), and the reason I guess for this kind of result is maybe overfitting.

Thus, I briefly introduce adding one hidden layer making what kind of difference in the method.

$z1 = (W1 \times X1 + b1)$ (X1 is the input feature)

$$f_{1W,b}(x) = \frac{1}{1+e^{-(z1)}}$$

$$z2 = (W2 \times f_{1W,b}(x) + b2)$$

$$f_{2W,b}(x) = \frac{1}{1+e^{-(z2)}}$$

$$\frac{-\partial \ln L}{\partial \mathrm{W2}} = -f_{1W,b}(x)^T \cdot \left(\hat{y} - f_{2W,b}(x)\right) \, and \, \frac{-\partial \ln L}{\partial \mathrm{b2}} = -\left(\hat{y} - f_{2W,b}(x)\right)$$

$$\frac{-\partial \ln L}{\partial \mathrm{W1}} = -X1^T \cdot \left(\left(\hat{y} - f_{2W,b}(x)\right) \cdot w2^T \times f_{1W,b}(x)\left(1 - f_{1W,b}(x)\right)\right) and$$

$$\frac{-\partial \ln L}{\partial \mathrm{b1}} = -\left(\left(\hat{y} - f_{2W,b}(x)\right) \cdot w2^T \times f_{1W,b}(x)\left(1 - f_{1W,b}(x)\right)\right)$$

The derivatives are all derived by myself. Additionally, I find out there's a rule between every layers, so it easy for me to calculate when there are many layers just like (3,4,3). Also, I still use adagrad to make my learning rate adaptive.

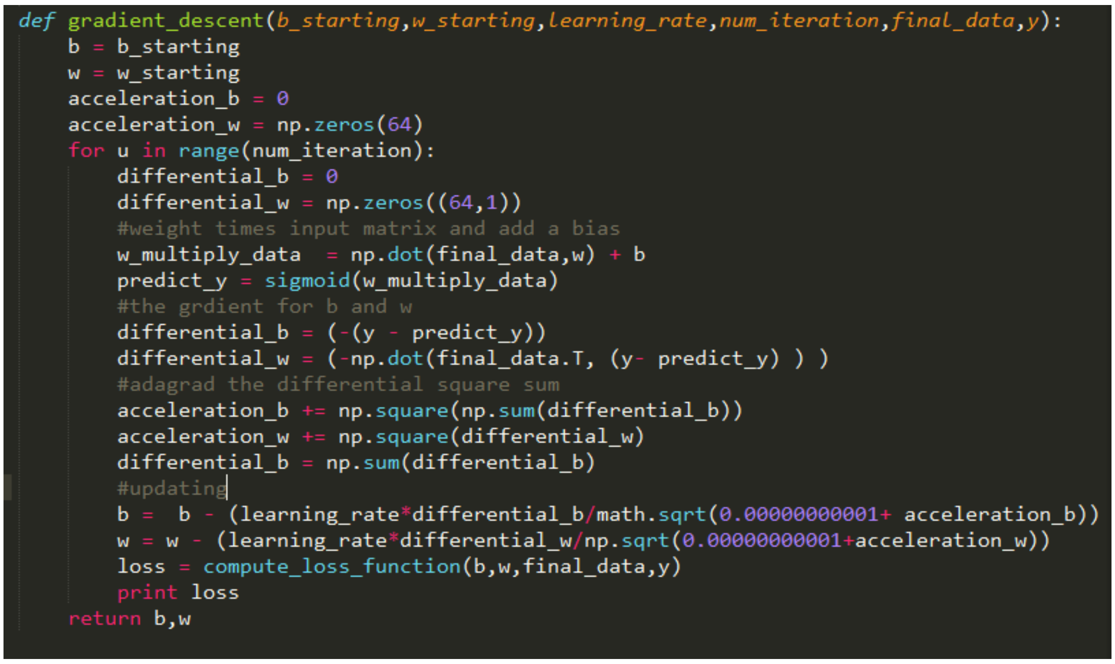
In conclusion, I get 0.94667 accuracy in the public set which pretty higher than the logistic regression.

3. Problems I am thinking about and can't figure it out

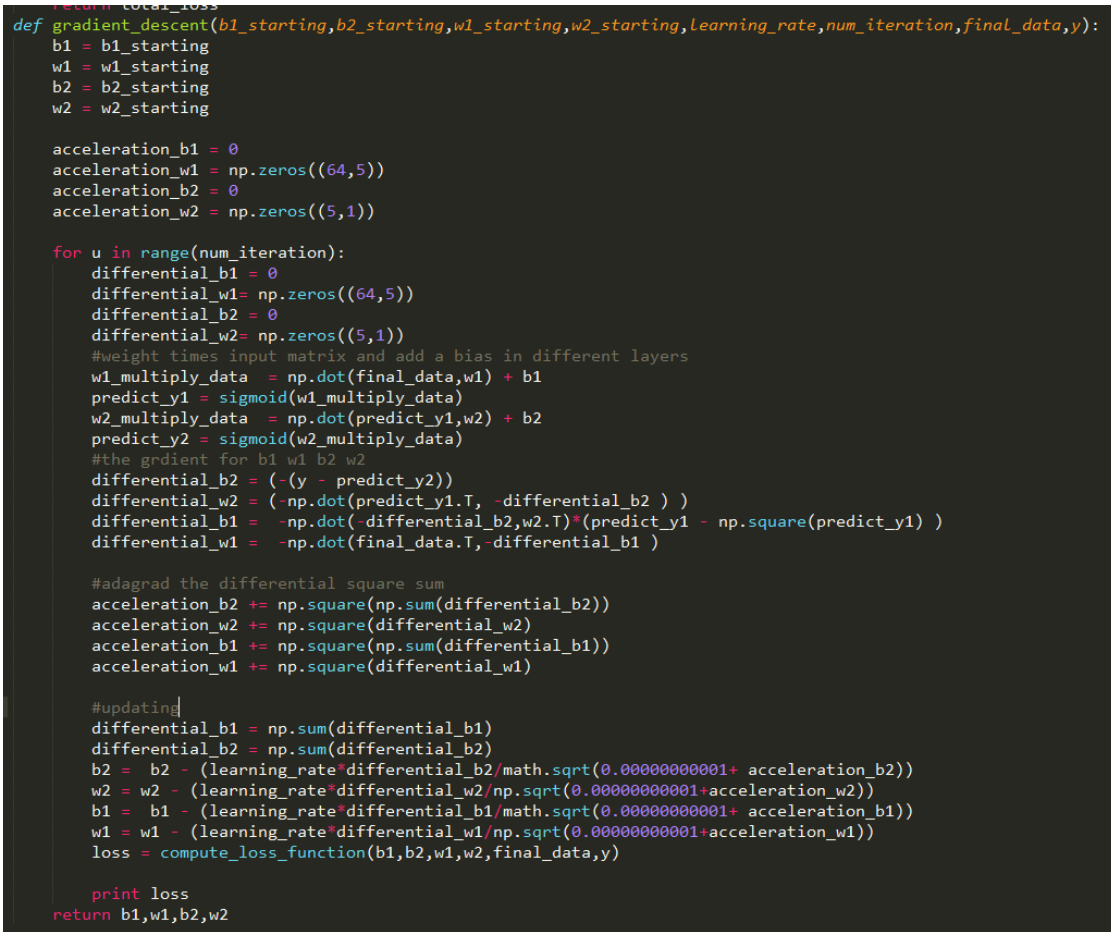
- Is there a specific numbers of layers or neurons I should use to make the performance better? Because in the high dimension space we can't find out the relation between each other, we can't really know what kind of feature transformation it needed.

Part 1 and Part 2 code

I only paste the important part of the derivative in the gradient descent.



```python
def gradient_descent(b_starting,w_starting,learning_rate,num_iteration,final_data,y):
    b = b_starting
    w = w_starting
    acceleration_b = 0
    acceleration_w = np.zeros(64)
    for u in range(num_iteration):
        differential_b = 0
        differential_w = np.zeros((64,1))
        #weight times input matrix and add a bias
        w_multiply_data  = np.dot(final_data,w) + b
        predict_y = sigmoid(w_multiply_data)
        #the grdient for b and w
        differential_b = (-(y - predict_y))
        differential_w = (-np.dot(final_data.T, (y- predict_y) ) )
        #adagrad the differential square sum
        acceleration_b += np.square(np.sum(differential_b))
        acceleration_w += np.square(differential_w)
        differential_b = np.sum(differential_b)
        #updating
        b =  b - (learning_rate*differential_b/math.sqrt(0.00000000001+ acceleration_b))
        w = w - (learning_rate*differential_w/np.sqrt(0.00000000001+acceleration_w))
        loss = compute_loss_function(b,w,final_data,y)
        print loss
    return b,w
```



```python
def gradient_descent(b1_starting,b2_starting,w1_starting,w2_starting,learning_rate,num_iteration,final_data,y):
    b1 = b1_starting
    w1 = w1_starting
    b2 = b2_starting
    w2 = w2_starting

    acceleration_b1 = 0
    acceleration_w1 = np.zeros((64,5))
    acceleration_b2 = 0
    acceleration_w2 = np.zeros((5,1))

    for u in range(num_iteration):
        differential_b1 = 0
        differential_w1= np.zeros((64,5))
        differential_b2 = 0
        differential_w2= np.zeros((5,1))
        #weight times input matrix and add a bias in different layers
        w1_multiply_data  = np.dot(final_data,w1) + b1
        predict_y1 = sigmoid(w1_multiply_data)
        w2_multiply_data  = np.dot(predict_y1,w2) + b2
        predict_y2 = sigmoid(w2_multiply_data)
        #the grdient for b1 w1 b2 w2
        differential_b2 = (-(y - predict_y2))
        differential_w2 = (-np.dot(predict_y1.T, -differential_b2 ) )
        differential_b1 =  -np.dot(-differential_b2,w2.T)*(predict_y1 - np.square(predict_y1) )
        differential_w1 =  -np.dot(final_data.T,-differential_b1 )

        #adagrad the differential square sum
        acceleration_b2 += np.square(np.sum(differential_b2))
        acceleration_w2 += np.square(differential_w2)
        acceleration_b1 += np.square(np.sum(differential_b1))
        acceleration_w1 += np.square(differential_w1)

        #updating
        differential_b1 = np.sum(differential_b1)
        differential_b2 = np.sum(differential_b2)
        b2 =  b2 - (learning_rate*differential_b2/math.sqrt(0.00000000001+ acceleration_b2))
        w2 = w2 - (learning_rate*differential_w2/np.sqrt(0.00000000001+acceleration_w2))
        b1 =  b1 - (learning_rate*differential_b1/math.sqrt(0.00000000001+ acceleration_b1))
        w1 = w1 - (learning_rate*differential_w1/np.sqrt(0.00000000001+acceleration_w1))
        loss = compute_loss_function(b1,b2,w1,w2,final_data,y)

        print loss
    return b1,w1,b2,w2
```