Machine Learning HW1 b03901036 陳柏文

**(1.) Linear regression function by Gradient Descent**

My code (based on python):

```
batch = np.array(batch) # batch which contains 40 data
grad_w = np.zeros((data_feature,time)); grad_b = 0.0;   # grad_w and grad_b are gradients of weights and bias
for z in xrange(batch_number):
    label = Dataset[9][rand_[z]+time]
    residual = Dataset[9][rand_[z]+time-1] – label      # I use residual net by using previous hour's PM2.5 value
    temp = 2*(np.multiply(weight,batch[z]).sum() + bias - residual)     # temp stored the sharing value for gradient of w and b
    grad_b = grad_b + temp/batch_number                                 #  temp = 2 * (w · x + b − y)
    for x in xrange(data_feature):                  # data_feature equals to the number feature used
        for y in xrange(time):                      # time equals to the number of time used to train (Here is 9.)
            current_grad = temp*batch[z][x][y]/batch_number    # current_grad used to store the mean value of gradient
            grad_w[x][y] = grad_w[x][y] + current_grad + 2*reg*weight[x][y]      # adding regulation term to the gradient
    bias_adam = math.sqrt(pow(bias_adam,2) + pow(grad_b,2))      # learning is divided by sqrt(δ_{t-1}^2 + ∂θ_t^2)  → adagrad
    bias = bias - learning_rate/bias_adam*(grad_b + 2*reg*bias)  #  θ_t = θ_{t-1} − learning_rate * ∂θ_{t-1}' / sqrt(δ_{t-1}^2 + ∂θ_t^2)
    for x in xrange(data_feature):
        for y in xrange(time):
            weight_adam[x][y] = math.sqrt(pow(weight_adam[x][y],2) + pow(grad_w[x][y],2))
            weight[x][y] = weight[x][y] - learning_rate*grad_w[x][y] / weight_adam[x][y]
```

**(2.) Describe your method**

(a.) *data normalization*: I normalize data first to get actual behavior of each feature

(b.) *special initialization*: I initialize the weights related to PM2.5 with higher value and initialize all the weights which is temporarily closer (more recent) with higher value, since I think that those features are more important.

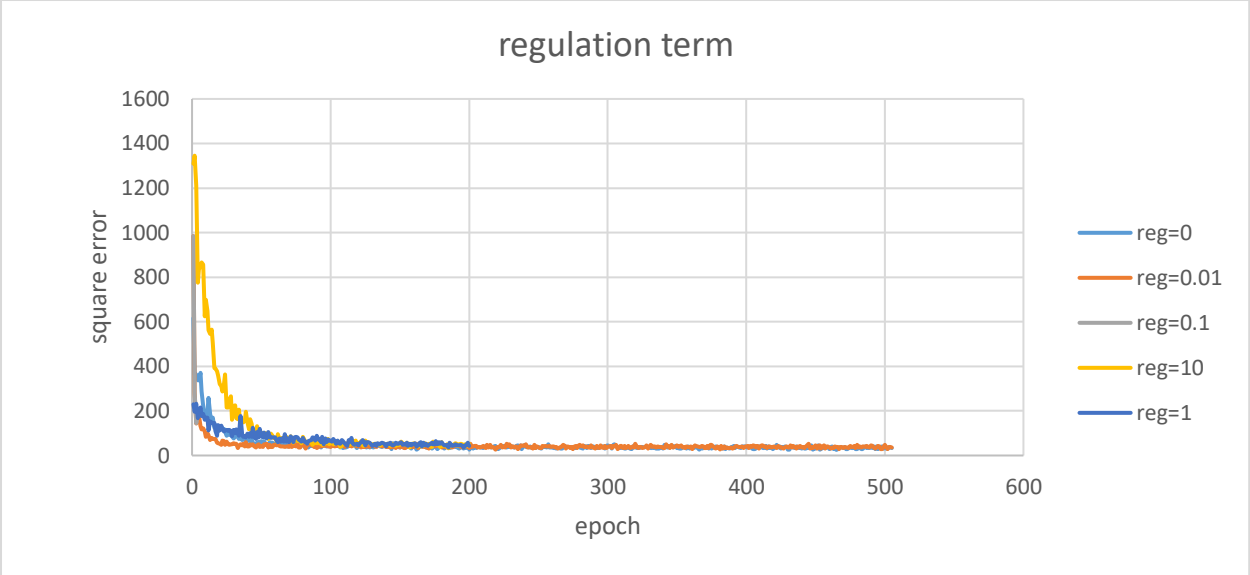
(c.) *residual net*: I use label minus the previous data right before label to get the residual. My trained model is to learn how to predict the residual which has smaller value, so I can converge faster.

(d.) *validation set*: I sliced the whole training data into two slice. One is the dataset for training, and the other is the validation set. So I can view the loss using validation set which might come close to actual loss on private test set on Kaggle, and then adjust my training policy (ex: hyper parameter).

(e.) *stochastic gradient descent*: Instead of computing the average of gradient from whole sets of training data, I only compute random average partial gradient from partial dataset.

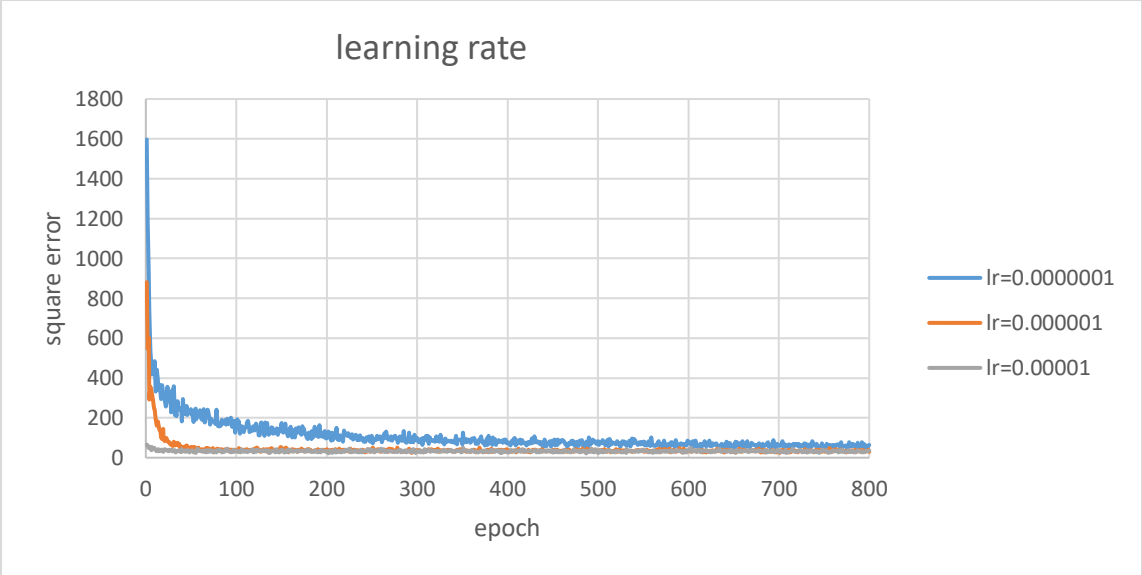
**(3.) Discussion on regularization.**

I think regulation actually doesn't give much help to my training, since that for linear regression model, it is over-fitting is hard to happen, except that adding weights which associated with higher power of data **x** (such as $\boldsymbol{w} \cdot \boldsymbol{x}^2$, ). But I don't use this kind of term, I preferred to regularization term in my loss function. From the following diagram, we can see that, by picking proper regularization parameter ($\epsilon = \mathrm{reg}$), we can get faster convergence speed.




**(4.) Discussion on learning rate.**

Learning rate is used to determine the convergence speed. And we can see from the diagram below shows that higher learning rate will have more observable oscillation. And if we use Adagrad, we can have larger learning rate.




**(5.) Other discussion and detail**

(a.) partial data: I try to use only partial data to the model, picking only partial days in the given nine-day to train. Since there is some nonlinear term with respect to day. But I didn't get better result.

(b.) PyPy: using PyPy interpreter can save a large amount of your time, since that python runs really slow. (But latest PyPy doesn't supply numpy.)

(c.) Storing the trained model: I stored the weights ,biases , information associated with Adagrad and the history value of loss function in .npy file. So I can reloaded these data as initialization of next time's training for better performance.

(d.) Discussion: After discussing with teacher, I know that to get lower testing error, we have to use high order term of data x, so that we can mimic the nonlinear part of oscillation of PM2.5.