华南理工大学

South China University of Technology

The Experiment Report of Machine Learning

SCHOOL: SCHOOL OF SOFTWARE ENGINEERING

SUBJECT: SOFTWARE ENGINEERING

Author:
SunXingxing

Supervisor:
Qingyao Wu

Student ID：
201721045350

Grade:
Undergraduate

December15, 2017

Logistic Regression, Linear Classification and Stochastic Gradient Descent

Abstract—

Batch methods, such as BGD, which use the full training set to compute the next update to parameters at each iteration . They can work well because they have very few hyper-parameters to tune. However, often in practice computing the cost and gradient for the entire training set can be very slow. Sometimes dataset is too big to read them into memeory at one time.Another issue with batch optimization methods is that they don't give an easy way to incorporate new data in an online setting. Stochastic Gradient Descent (SGD) addresses both of these issues . We take a short look at algorithms and architectures to optimize gradient descent in a parallel and distributed setting.

I. Introduction

Gradient descent is one of the most popular algorithms to perform optimization and by far the most common way to optimize all kinds machine learing method. Try to implementations of various algorithms to optimize gradient descent .

II. Methods and Theory

We have many methods to optimize gradient descent,such as nag, Adam and so on.Here we descibe it at detail.

momentum helps SGD to navigate along the relevant directions and softens the oscillations in the irrelevant. It simply adds a fraction of the direction of the previous step to a current step. This achieves amplification of speed in the correct direction and softens oscillation in wrong directions. This fraction is usually in the (0, 1) range.

$$v = \gamma v + \alpha \nabla_\theta J(\theta; x^{(i)}, y^{(i)})$$
$$\theta = \theta - v$$

In the above equation vv is the current velocity vector which is of the same dimension as the parameter vector θ θ . The learning rate α α is as described above, although when using momentum α α may need to be smaller since the magnitude of the gradient will be larger. Finally $\gamma \in (0,1]$ $\gamma \in (0,1]$ determines for how many iterations the previous gradients are incorporated into the current update. Generally γ γ is set to 0.5 until the initial learning stabilizes and then is increased to 0.9 or higher.

AdaGrad or adaptive gradient allows the learning rate to adapt based on parameters. It performs larger updates for infrequent parameters and smaller updates for frequent one. Because of this it is well suited for sparse data (NLP or image recognition). Another advantage is that it basically eliminates the need to tune the learning rate. Each parameter has its own learning rate and due to the peculiarities of the algorithm the learning rate is monotonically decreasing. This causes the biggest problem: at some point of time the learning rate is so small that the system stops learning

$$g_{t,i} = \nabla_\theta J(\theta_i).$$
$$\theta_{t+1,i} = \theta_{t,i} - \eta \cdot g_{t,i}.$$

updates each parameter based on past gradients (cumultion of sum of gradient squares)

Gt—diagonal matrix; each element is sum of squares of past gradients for each parameter.

ϵ—smoothing term (avoids "divide by zero" case, usually order of 1e-8)

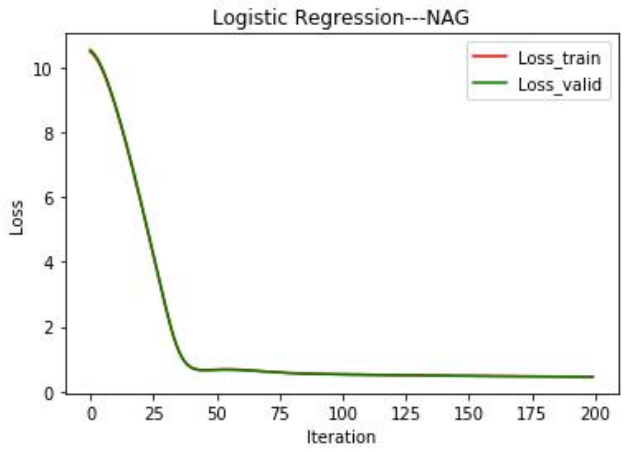
AdaDelta resolves the problem of monotonically decreasing learning rate in AdaGrad. In AdaGrad the learning rate was calculated approximately as one divided by the sum of square roots. At each stage you add another square root to the sum, which causes denominator to constantly decrease. In AdaDelta instead of summing all past square roots it uses sliding window which allows the sum to decrease. RMSprop is very similar to AdaDelta

$$\Delta\theta_t = -\frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t.$$
$$\Delta\theta_t = -\frac{\eta}{RMS[g]_t} g_t.$$

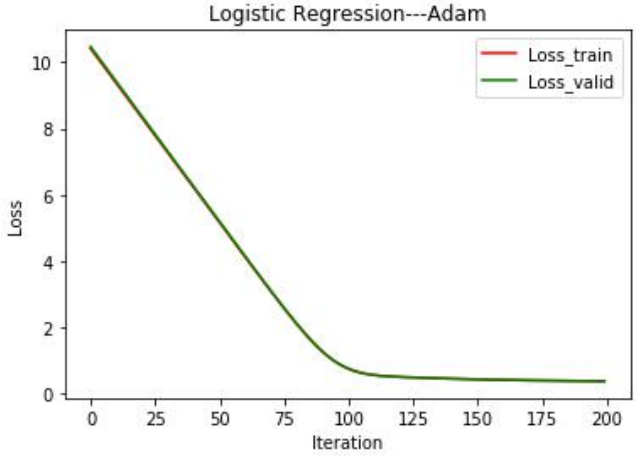
Adam or adaptive momentum is an algorithm similar to AdaDelta. But in addition to storing learning rates for each of the parameters it also stores momentum changes for each of them separately

III. Experiment

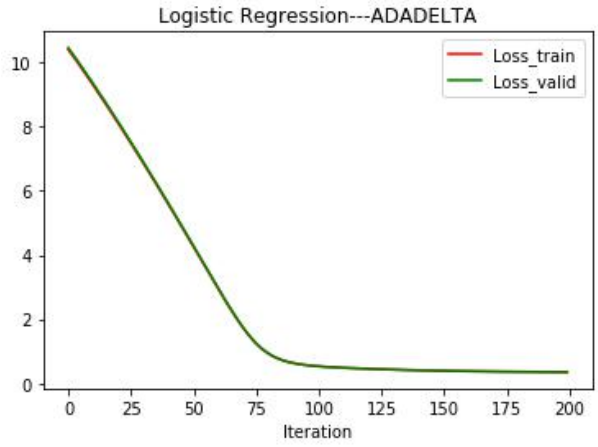
```python
def Adam(iterate_number, W, X_train, y_train, x_test, y_test, learning_rate):
    beta_1 = 0.9
    beta_2 = 0.99
    ep = 0.00000001

    m = 0
    v = 0
    t = 0

    loss_train = []
    loss_valid = []

    N = X_train.shape[0]

    for i in range(iterate_number):
        h = output(X_train, W)
        error = h - y_train
        gradient = (X_train.T * error) / N
        t = t + 1
        m = beta_1 * m + (1 - beta_1) * gradient
        v = beta_2 * v + (1 - beta_2) * (np.power(gradient, 2))
        mt = m / (1 - beta_1**t)
        vt = v /(1 - (beta_2**t))

        W = W - learning_rate * mt / (np.sqrt(vt) + ep)
        loss_train.append(compute_loss(X_train, W, y_train))
        loss_valid.append(compute_loss(x_test, W, y_test))
    return loss_train, loss_valid
```



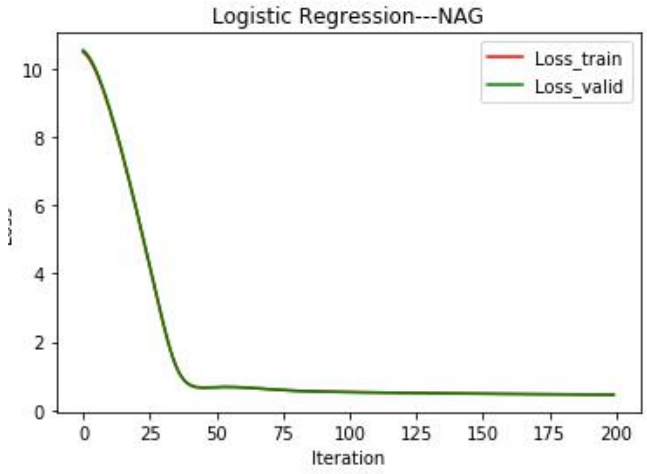

```python
def RMSProp(iterate_number, W, X_train, y_train, x_test, y_test, learning_rate):
    N = X_train.shape[0]

    d = 0.9

    Egt=0
    Edt = 0
    delta = 0
    ep = 0.00000001

    loss_train = []
    loss_valid = []

    for i in range(iterate_number):
        h = output(X_train, W)
        error = h - y_train
        gradient = (X_train.T * error) / N
        Egt = d * Egt + (1 - d)*(np.power(gradient, 2))

        W = W - learning_rate * gradient / (np.sqrt(Egt) + ep)
        loss_train.append(compute_loss(X_train, W, y_train))
        loss_valid.append(compute_loss(x_test, W, y_test))
    return loss_train, loss_valid
```




```python
def ADADELTA(iterate_number, W, X_train, y_train, x_test, y_test):
    N = X_train.shape[0]

    ep = 0.00001

    d = 0.9

    Egt = 0
    Edt = 0
    sumDelta = 0

    loss_train = []
    loss_valid = []

    for i in range(iterate_number):
        h = output(X_train, W)
        error = h - y_train
        gradient = (X_train.T * error) / N

        Egt = d * Egt + (1 - d) * (np.power(gradient, 2))
        delta =  np.multiply(np.sqrt(Edt + ep), gradient) / np.sqrt(Egt + ep)
        Edt = d * Edt + (1 - d) * (np.power(delta, 2))

        W = W - delta
        loss_train.append(compute_loss(X_train, W, y_train))
        loss_valid.append(compute_loss(x_test, W, y_test))
    return loss_train, loss_valid
```



Resutl:



IV. CONCLUSION

we have initially looked at the three variants of gradient descent, among which SGD is the most popular. We have then investigated algorithms that are most commonly used for optimizing SGD: Momentum, Nesterov accelerated gradient, Adagrad, Adadelta,RMSprop, Adam, AdaMax, Nadam, as well as different algorithms to optimize asynchronous SGD. Finally,we do experiments to find the best way.