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:
Liu Rui

Supervisor:
Qingyao Wu

Student ID：
201721045626

Grade:
PostGraduate

December 9, 2017

Logistic Regression, Linear Classification and Stochastic Gradient Descent

Abstract— In this experiment we aim to compare and understand the difference between gradient descent and stochastic gradient descent. We compare logistic regression and linear classification. We aim to further understand the principles of SVM and practice and larger data.

I. INTRODUCTION

There are three motivations of this experiment: First, compare and understand the difference between gradient descent and stochastic gradient descent. Second, compare and understand the differences and relationships between Logistic regression and linear classification. And last but not least to further understand the principles of SVM and practice on larger data. Experiment uses a9a of LIBSVM Data, including 32561/16281(testing) samples and each sample has 123/123 (testing) features. Please download the training set and validation set.

There are two parts contained in this experiment, logistic regression and linear classification, the experimental steps are here:

1) Load the training set and validation set.
2) Initialize logistic regression /SVM model parameters, you can consider initializing zeros, random numbers or normal distribution.
3) Select the loss function and calculate its derivation, find more detail in PPT.
4) Calculate gradient toward loss function from partial samples.
5) Update model parameters using different optimized methods(NAG，RMSProp，AdaDelta and Adam).
6) Select the appropriate threshold, mark the sample whose predict scores greater than the threshold as positive, on the contrary as negative. Predict under validation set and get the different optimized method loss， ， and .
7) Repeate step 4 to 6 for several times, and drawing graph of， ， and with the number of iterations.

Part II introduce the methods and theory of logistic regression, linear classification and the gradient descent algorithm used in experiment including nesterov accelerated gradient(NAG), RMSProp, AdaDelta and Adam. Part III shows the experimental code and results and finally make some conclusions in part IV.

II. METHODS AND THEORY

A. Gradient Descent and Stochastic gradient descent

Gradient descent is one of the most popular algorithms to perform optimization and by far the most common way to optimize neural networks.

Gradient descent is a way to minimize an objective function $J(\theta)$ parameterized by a model's parameters $\theta \in R^d$ by updating the parameters in the opposite direction of the gradient of the objective function $\nabla_\theta J(\theta)$ w.r.t. to the parameters. The learning rate η determines the size of the steps we take to reach a (local) minimum. In other words, we follow the direction of the slope of the surface created by the objective function downhill until we reach a valley.

Stochastic gradient descent (SGD) in contrast performs a parameter update for each training example $x^{(i)}$ and label $y^{(i)}$:

$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta; x^{(i)}; y^{(i)})$$

SGD also has many shortcomes, such as the converge speed has great relationship with learning rate, which means it is difficult to adjust the learning rate parameter during training the data set. And also, the learning rate has no difference with any attribution parameters, as the matter of fact the converge speed is different to different attribution parameter, SGD doesn't put the sparsity of data matrix. There are any kinds of algorithm was designed to in order to overcoming those difficulties such as NAG, RMSProp, AdaDelta and Adam.

B. NAG

Nesterov accelerated gradient is a way to give our momentum term this kind of prescience. We know that we will use our momentum term γv_{t-1} to move the parameters θ. Computing $\theta - \gamma v_{t-1}$ thus gives us an approximation of the next position of the parameters (the gradient is missing for the full update), a rough idea where our parameters are going to be. We can now effectively look ahead by calculating the gradient not w.r.t. to our current parameters θ but w.r.t. the approximate future position of our parameters:

$$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta - \gamma v_{t-1})$$
$$\theta = \theta - v_t$$

C. RMSProp

RMSprop is an unpublished, adaptive learning rate method proposed by Geoff Hinton in Lecture 6e of his Coursera Class. RMSprop and Adadelta have both been developed independently around the same time stemming from the need to resolve Adagrad's radically diminishing learning rates. RMSprop in fact is identical to the first update vector of Adadelta that we derived above:

$$E[g^2]_t = 0.9E[g^2]_{t-1} + 0.1g_t^2$$
$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t$$

RMSprop as well divides the learning rate by an exponentially decaying average of squared gradients. Hinton suggests γ to be set to 0.9, while a good default value for the learning rate η is 0.001.

D. AdaDelta and Adam

Adadelta seeks to reduce its aggressive, monotonically decreasing learning rate. Instead of accumulating all past squared gradients. The running average $E[g^2]_t$ at time step t then depends (as a fraction γ similarly to the Momentum term) only on the previous average and the current gradient:

$$E[g^2]_t = \gamma E[g^2]_{t-1} + (1-\gamma) g_t^2$$

We set γ to a similar value as the momentum term, around 0.9. For clarity, we now rewrite our vanilla SGD update in terms of the parameter update vector $\Delta_{\theta t}$:

$$\Delta\theta_t = -\eta \cdot g_{t,i}$$
$$\theta_{t+1} = \theta_t + \Delta\theta_t$$

The parameter update vector of Adagrad that we derived previously thus takes the form:

$$\Delta\theta_t = -\frac{\eta}{\sqrt{G_t + \epsilon}} \odot g_t$$

We now simply replace the diagonal matrix Gt with the decaying average over past squared gradients $E[g^2]_t$:

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

As the denominator is just the root mean squared (RMS) error criterion of the gradient, we can replace it with the criterion short-hand:

$$\Delta\theta_t = -\frac{\eta}{RMS[g]_t} g_t$$

The update should have the same hypothetical units as the parameter. To realize this, they first define another exponentially decaying average, this time not of squared gradients but of squared parameter updates:

$$E[\Delta\theta^2]_t = \gamma E[\Delta\theta^2]_{t-1} + (1-\gamma)\Delta\theta_t^2$$

The root mean squared error of parameter updates is thus:

$$RMS[\Delta\theta]_t = \sqrt{E[\Delta\theta^2]_t + \epsilon}$$

Since $RMS[\Delta\theta]_t$ is unknown, we approximate it with the RMS of parameter updates until the previous time step. Replacing the learning rate η in the previous update rule with $RMS[\Delta\theta]_t$−1 finally yields the Adadelta update rule:

$$\Delta\theta_t = -\frac{RMS[\Delta\theta]_{t-1}}{RMS[g]_t} g_t$$
$$\theta_{t+1} = \theta_t + \Delta\theta_t$$

Adaptive Moment Estimation (Adam) is another method that computes adaptive learning rates for each parameter. In addition to storing an exponentially decaying average of past squared gradients v_t like Adadelta and RMSprop, Adam also keeps an exponentially decaying average of past gradients m_t, similar to momentum:

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t$$
$$v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2$$

Model counteract the biases by computing bias-corrected first and second moment estimates:

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}$$
$$\hat{v}_t = \frac{v_t}{1-\beta_2^t}$$

Then use these to update the parameters just as we have seen in Adadelta and RMSprop, which yields the Adam update rule:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t + \epsilon}} \hat{m}_t$$

III. Experiment

A. Logistic Regression and Stochastic Gradient Descent Experimental Code

```
train_file_path = './a9a.txt'
validation_file_path = './a9a.t'
X_train, y_train = get_data(train_file_path)
X_validation, y_validation = get_data(validation_file_path)

X_train = X_train.toarray()
X_validation = X_validation.toarray()

col = np.zeros(( X_validation.shape[0]))
col_train = np.ones(( X_train.shape[0]))
col_validation = np.ones(( X_validation.shape[0]))

X_validation = np.column_stack((X_validation,col))
X_train = np.column_stack((X_train,col_train))
X_validation = np.column_stack((X_validation,col_validation))

N = X_train.shape[1]
learning_rate = 0.005
iter_i = 5
batch_size = 128
iter_j = math.ceil(X_train.shape[0] / float(batch_size))
max_iteration = iter_i * iter_j
randomlist = np.arange(X_train.shape[0])
lambdal = 0.005
gamma = 0.9
epsilon = np.e**(-8)
```

Load data and initial:

```
Loss = np.sum(np.log(1 + np.exp(-y * np.dot(X,W)))) / X.shape[0] + lambdal / 2 * np.dot(W,W.T)
```

Compute loss

```
Gradient = np.dot(((-y)/ (1+ np.exp(y * np.dot(X,W)))),X ) / X.shape[0] + W * lambdal
```

Compute Gradient

```
W_t = W - v_t * gamma
 Loss_train_NAG[j*runs+l] = Loss(X_batch , W,y_batch,lambdal)
Loss_validation_NAG[j*runs+l] = Loss(X_validation,W,y_validation,lambdal)
Gradient_1 = Gradient(X_batch,W_t,y_batch,lambdal)
v_t = v_t * gamma + Gradient_1 * learning_rate
W = W - v_t
```

NAG

```
Loss_train_RMSprop[j*runs+l] = Loss(X_batch,W,y_batch,lambdal)
Gradient_1 = Gradient(X_batch,W,y_batch,lambdal)
Loss_validation_RMSprop[j*runs+l] = Loss(X_validation,W,y_validation,lambdal)
Gradient_2 = Gradient_2 * 0.9 + np.dot(Gradient_1,Gradient_1.T) * 0.1
W = W - Gradient_1 *(learning_rate / math.sqrt(Gradient_2 + epsilon))
```

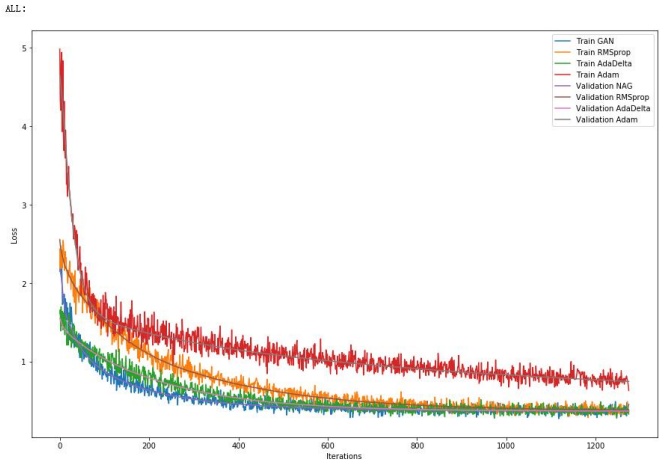
RMSprop

```
Loss_train_AdaDelta[j*runs+l] = Loss(X_batch,W,y_batch,lambdal)
Gradient_1 = Gradient(X_batch,W,y_batch,lambdal)
Loss_validation_AdaDelta[j*runs+l] = Loss(X_validation,W,y_validation,lambdal)
Gradient_2 = Gradient_2 * gamma + np.dot(Gradient_1, Gradient_1.T) * (1 - gamma)
RMS_g = math.sqrt(Gradient_2 + epsilon)
W = W - Gradient_1 *(RMS_W / RMS_g)
W_delta = Gradient_1 *(- learning_rate / RMS_g)
W_2 = W_2 * gamma + np.dot(W_delta, W_delta.T) * (1 - gamma)
RMS_W = math.sqrt(W_2 + epsilon)
```

AdaDelta

```
Loss_train_Adam[j*runs+l] = Loss(X_batch,W,y_batch,lambdal)
Gradient_1 = Gradient(X_batch,W,y_batch,lambdal)
Loss_validation_Adam[j*runs+l] = Loss(X_validation,W,y_validation,lambdal)
m_t = m_t * beta1 + Gradient_1 * (1-beta1)
n_t = n_t * beta2 + np.dot(Gradient_1, Gradient_1.T) * (1-beta2)
hat_m = m_t * (1 / (1-beta1))
hat_n = n_t * (1 / (1-beta2))
W = W - hat_m * (learning_rate / (math.sqrt(hat_n)+epsilon))
```

Adam



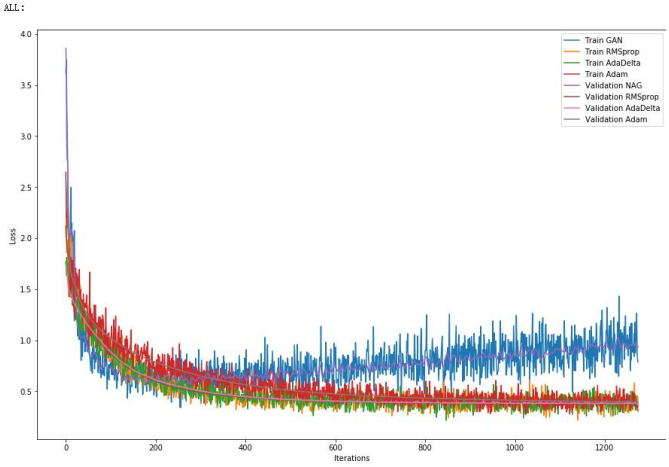
B. Linear Classification and Stochastic Gradient Descent Experimental Code

```
def Loss(X, W, y, lambdal, W_0):
    diif = np.ones(y.shape[0]) - y * np.dot(X, W)
    diif[diif < 0] =0
     Loss =np.sum(diif) / X.shape[0] + np.dot(W_0, W_0.T) / 2 * lambdal
      return Loss

def Gradient(X, W, y, lambdal, W_0):
    diif = np.ones(y.shape[0]) - y * np.dot(X, W)
    y_copy = y.copy()
    y_copy[diif <= 0] =0
    Gradient = - np.dot(y_copy,X) / X.shape[0] + W_0 * lambdal
  return Gradient
```

Loss and Graident

Result



IV. CONCLUSION

A. The difference between gradient descent and SGD

In SGD we using a random input data to update our model and make the loss function arrive the local minimum. In normal gradient descent the value of goal function will become a function of parameter, gradient descent update one dimension of the parameter every iteration but SGD update all dimension of the parameter every iteration, only need to make the total loss become smaller and smaller.

B. The Difference and relation between logistic regression and linear classification

Both of the models training a set of data using a function, and find a group of parameters in function to make the loss between output value and real value smaller.

Logical regression is not a truly regression model but a classifier using the sigmoid function as regression object, and reject input data into a float between 0 to 1. People usually considerate it as a probability.

Linear Classification is a classifier using a hyperplane model as the decision boundary.

C. The principles of SVM

support vector machines are supervised learning models with associated learning algorithms that analyze data used for classification and regression analysis.

Given a set of training examples, each marked as belonging to one or the other of two categories, an SVM training algorithm builds a model that assigns new examples to one category or the other, making it a non-probabilistic binary linear classifier. An SVM model is a representation of the examples as points in space, mapped so that the examples of the separate categories are divided by a clear gap that is as wide as possible.