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:
Zijie Shu

Supervisor:
Qingyao Wu

Student ID：
201530612712

Grade:
Undergraduate

December 15, 2017

Logistic Regression, Linear Classification and Stochastic Gradient Descent

Abstract—The motivation of this experiment is comparing and understanding the difference between gradient descent and stochastic gradient descent. Compare the differences and relationships between Logistic regression and linear classification, and compare different optimized methods of gradient descent using a9a of LIBSVM Data. Further understand the principles of SVM and practice on larger data.

I. INTRODUCTION

Experimental steps:

(1) Logistic Regression and Stochastic Gradient Descent

a. Load the training set and validation set.
b. Initalize logistic regression model parameters, you can consider initalizing zeros, random numbers or normal distribution.
c. Select the loss function and calculate its derivation, find more detail in PPT.
d. Calculate gradient G toward loss function from partial samples.
e. Update model parameters using different optimized methods(NAG，RMSProp，AdaDelta and Adam).
f. 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 L_{NAG}, $L_{RMSProp}$, $L_{AdaDelta}$ and L_{Adam}.
g. Repeate step 4 to 6 for several times, and drawing graph of L_{NAG}, $L_{RMSProp}$, $L_{AdaDelta}$ and L_{Adam} with the number of iterations.

(2) Linear Classification and Stochastic Gradient Descent

a. Load the training set and validation set.
b. Initalize SVM model parameters, you can consider initalizing zeros, random numbers or normal distribution.
c. Select the loss function and calculate its derivation, find more detail in PPT.
d. Calculate gradient G toward loss function from partial samples.
e. Update model parameters using different optimized methods(NAG，RMSProp，AdaDelta and Adam).
f. 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 L_{NAG}, $L_{RMSProp}$, $L_{AdaDelta}$ and L_{Adam}.
g. Repeate step 4 to 6 for several times, and drawing graph of L_{NAG}, $L_{RMSProp}$, $L_{AdaDelta}$ and L_{Adam} with the number of iterations.

II. METHODS AND THEORY

(1) Logistic Regression

In statistics, logistic regression, or logit regression, or logit model is a regression model where the dependent variable (DV) is categorical. This article covers the case of a binary dependent variable—that is, where the output can take only two values, "0" and "1", which represent outcomes such as pass/fail, win/lose, alive/dead or healthy/sick. Cases where the dependent variable has more than two outcome categories may be analysed in multinomial logistic regression, or, if the multiple categories are ordered, in ordinal logistic regression. In the terminology of economics, logistic regression is an example of a qualitative response/discrete choice model.

Logistic regression was developed by statistician David Cox in 1958. The binary logistic model is used to estimate the probability of a binary response based on one or more predictor (or independent) variables (features). It allows one to say that the presence of a risk factor increases the odds of a given outcome by a specific factor.

Loss Function:

$$\mathrm{L}(w,b) = -\frac{1}{n}\sum_{i=1}^{n}\big[y_i \log h(x_i;w,b) + (1-y_i)\log\big(1-h(x_i;w,b)\big)\big]$$

Gradient:

$$\frac{\partial \mathrm{L}(w,b)}{\partial w} = \frac{1}{n}\sum_{i=1}^{n}(h(x_i;w,b)-y_i)x_i$$

$$\frac{\partial \mathrm{L}(w,b)}{\partial b} = \frac{1}{n}\sum_{i=1}^{n}(h(x_i;w,b)-y_i)$$

(2) Linear Classification

In the field of machine learning, the goal of statistical classification is to use an object's characteristics to identify which class (or group) it belongs to. A linear classifier achieves this by making a classification decision based on the value of a linear combination of the characteristics. An object's characteristics are also known as feature values and are typically presented to the machine in a vector called a feature vector. Such classifiers work well for practical problems such as document classification, and more generally for problems with many variables (features), reaching accuracy levels comparable to non-linear classifiers while taking less time to train and use.

In machine learning, support vector machines (SVMs, also support vector networks) 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 (although methods such as Platt scaling exist to use SVM in a probabilistic classification setting). 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. New examples are then mapped into that same space and predicted to belong to a category based on which side of the gap they fall.

In addition to performing linear classification, SVMs can efficiently perform a non-linear classification using what is called the kernel trick, implicitly mapping their inputs into high-dimensional feature spaces.

Loss Function:

$$\mathrm{L}(w,b) = C\sum_{i=1}^{n}\max\left(0, 1-y_i(w^T x_i + b)\right)$$

Gradient:

$$\frac{\partial \mathrm{L}(w,b)}{\partial w} = \|w\| - \frac{C}{n}\sum_{i=1}^{n} y_i x_i$$

$$\frac{\partial \mathrm{L}(w,b)}{\partial b} = -\frac{C}{n}\sum_{i=1}^{n} y_i$$

(3) Stochastic Gradient Descent

Stochastic gradient descent (often shortened to SGD), also known as incremental gradient descent, is a stochastic approximation of the gradient descent optimization and iterative method for minimizing an objective function that is written as a sum of differentiable functions. In other words, SGD tries to find minima or maxima by iteration.

SGD works similar as GD, but more quickly by estimating gradient from a few examples at a time.

Gradient Descent :

```
while True :
    oss = f(params)
    dw = . . . compute gradient
    params -=learning rate ·dw
    if (stopping condition is met):
        return params
```

Stochastic Gradient Descent :

```
for (xi, yi) in training set:
    loss = f(params, xi, yi)
    dw = . . . compute gradient
    params -=learning rate ·dw
    if (stopping condition is met):
        return params
```

The Benefits of SGD:

a. Gradient is easy to calculate (instantaneous)
b. Less prone to local minima
c. Small memory footprint
d. Get to a reasonable solution quickly
e. Works for non-stationary environments as well as online settings
f. Can be used for more complex models and error surfaces

III. Experiment

Datasets: The experiment uses a9a of LIBSVM Data, including 32561/16281(testing) samples and each sample has 123/123 (testing) features.

Code:

(1) Logistic Regression and Stochastic Gradient Descent

```
# coding=utf-8
import numpy as np
import math
from matplotlib import pyplot as plt
from sklearn.datasets import load_svmlight_file

def loadDataSet():
    X_train,y_train=load_svmlight_file("a9a.txt")
    X_validation, y_validation = load_svmlight_file("a9a.t")
    X_train = X_train.todense()
    X_validation = X_validation.todense()
    X_train = np.column_stack((X_train, np.ones(y_train.shape[0])))
    X_validation = np.column_stack((X_validation, np.zeros(y_validation.shape[0])))
    X_validation = np.column_stack((X_validation, np.ones(y_validation.shape[0])))
    y_train = np.array(list(map((lambda x: 0 if x <= 0 else 1), y_train)))
    y_validation = np.array(list(map((lambda x: 0 if x <= 0 else 1), y_validation)))
    print(X_train.shape,y_train.shape)
    print(X_validation.shape,y_validation.shape)
    return X_train, X_validation, y_train, y_validation

def sigmoid(inX):
    return 1.0/(1+math.exp(-inX))

def loss_function(X_data,y_data,w):
    loss = 0.0
    num = y_data.shape[0]
    for i in range(num):
        y = sigmoid(np.dot(X_data[i][0].getA()[0], w))
        loss += - (y_data[i] * math.log(y) + (1 - y_data[i]) * math.log(1 - y)) / num
    return loss

def stocGradDescent(epoch,X_train,X_validation,y_train,y_validation,opt):
    num = y_train.shape[0]
    batch = int(5000/epoch)
    w = np.zeros(X_train.shape[1])
    v = np.zeros(X_train.shape[1],dtype=np.float)
    G = np.zeros(X_train.shape[1], dtype=np.float)
    dx = np.zeros(X_train.shape[1], dtype=np.float)
    m = np.zeros(X_train.shape[1],dtype=np.float)
    t = 0
```

```
    losss = []

    for n in range(epoch):
      grad_w = np.zeros(X_train.shape[1])
      loss = loss_function(X_validation, y_validation, w)
      for i in range(batch):
        index = np.random.randint(0,num-1)
        y = sigmoid(np.dot( X_train[index][0].getA()[0], w))
        grad_w += ( y - y_train[index] ) * X_train[index][0].getA()[0] / batch

      if ( opt == 'SGD'):
        updates = SGD(w, grad_w)
      elif ( opt == 'NAG'):
        updates, v = NAG(w, grad_w, v)
      elif ( opt == 'RMSProp'):
        updates, G = RMSProp(w, grad_w, G)
      elif ( opt == 'AdaDelta'):
        updates, G, dx= AdaDelta(w, grad_w, G, dx)
      elif ( opt == 'Adam'):
        updates, G, m, t = Adam(w, grad_w, G, m, t)

      for i in range(len(w)):
        w[i] = updates[i][1]
      losss.append(loss)
      print("%s_loss = %f" % (opt, loss))
    return losss

  def SGD(parameters, gradients, eta=0.1):
    """
    Basic Mini-batch Stochastic Gradient Descent
    """
    updates = [(parameters[i], parameters[i] - eta * gradients[i])
           for i in range(len(parameters))]
    return updates

  def NAG(parameters, gradients, v, eta=0.05, gamma=.9):
    """
    Nesterov accelerated gradient
    """
    para_num = len(parameters)
    v_prev = v
    v = np.array([gamma * v[i] + eta * gradients[i] for i in range(para_num)])
    updates = [(parameters[i], parameters[i] - ( - gamma * v_prev[i] + ( 1 + gamma ) * v[i]))
               for i in range(para_num)]

    return updates, v

  def RMSProp(parameters, gradients, G, eta=.01, gamma=0.9, epsilon=1e-8):
    """
    RMSProp: Divide the gradient by a running average of its recent magnitude
    """
    para_num = len(parameters)
    G = np.array([gamma * G[i] + (1 - gamma) * (gradients[i])**2 for i in range(para_num)])
    updates = [(parameters[i], parameters[i] - eta * gradients[i] / math.sqrt(G[i] + epsilon))
           for i in range(para_num)]
    return updates, G

  def AdaDelta(parameters, gradients, G, dx, gamma=0.95, epsilon=1e-5):
    """
    AdaDelta: An Adaptive Learning Rate
    """
    para_num = len(parameters)
    G = np.array([gamma * G[i] + (1 - gamma) * (gradients[i]**2) for i in range(para_num)])
    dw = [math.sqrt(dx[i] + epsilon) / math.sqrt(G[i] + epsilon) * gradients[i] for i in range(para_num)]
    updates = [(parameters[i], parameters[i] - dw[i])
           for i in range(para_num)]
    dx = np.array([gamma * dx[i] + (1 - gamma) * (dw[i]**2) for i in range(para_num)])
    return updates, G, dx

  def Adam(parameters, gradients, G, m, t, eta=0.01, gamma=0.999, beta=0.9, epsilon=1e-8):
    """
    Adam: adaptive estimates of lower-order moments
    """
    t += 1
    para_num = len(parameters)
    m = np.array([beta * m[i] + (1 - beta) * gradients[i] for i in range(para_num)])
    G = np.array([gamma * G[i] + (1 - gamma) * (gradients[i]**2) for i in range(para_num)])
    updates = [(parameters[i], parameters[i] - eta * math.sqrt(1 - gamma**t) / (1 - beta**t) * m[i] / math.sqrt(G[i] + epsilon))
           for i in range(para_num)]
    return updates, G, m, t

  def plotLossPerTime(epoch, sgd_losss, nag_losss, rms_losss, adad_losss, adam_losss):
    plt.xlabel('iteration times')
    plt.ylabel('loss')
    plt.title('Logistic Regression & SGD')
    n_cycles = range(1,epoch+1)
    plt.plot(n_cycles, sgd_losss, label = "Loss of SGD", linewidth=2)
```

```
    plt.plot(n_cycles, nag_losss, label="Loss of NAG", linewidth=2)
    plt.plot(n_cycles, rms_losss, label="Loss of RMSProp", linewidth=2)
    plt.plot(n_cycles, adad_losss, label="Loss of AdaDelta", linewidth=2)
    plt.plot(n_cycles, adam_losss, label="Loss of Adam", linewidth=2)
    plt.legend(loc=0)
    plt.grid()
    plt.show()

  # main
  X_train, X_validation, y_train, y_validation = loadDataSet()
  epoch = 200
  sgd_losss = stocGradDescent(epoch, X_train, X_validation, y_train, y_validation, 'SGD')
  nag_losss = stocGradDescent(epoch, X_train, X_validation, y_train, y_validation, 'NAG')
  rms_losss = stocGradDescent(epoch, X_train, X_validation, y_train, y_validation, 'RMSProp')
  adad_losss = stocGradDescent(epoch, X_train, X_validation, y_train, y_validation, 'AdaDelta')
  adam_losss = stocGradDescent(epoch, X_train, X_validation, y_train, y_validation, 'Adam')
plotLossPerTime(epoch, sgd_losss, nag_losss, rms_losss, adad_losss, adam_losss)
```

(2) Linear Classification and Stochastic Gradient Descent

```
  # coding=utf-8
  import numpy as np
  from matplotlib import pyplot as plt
  from sklearn.datasets import load_svmlight_file
  import random

  def loadDataSet():
    train_data = load_svmlight_file('a9a.txt')
    X_train = np.reshape(train_data[0].todense().data, (train_data[0].shape[0], train_data[0].shape[1]))
    y_train = np.reshape(train_data[1].data, (train_data[1].shape[0], 1))

    validation_data = load_svmlight_file('a9a.t')
    zeros = np.zeros(validation_data[0].shape[0])
    X_validation = np.reshape(validation_data[0].todense().data, (validation_data[0].shape[0], validation_data[0].shape[1]))
    X_validation = np.column_stack((X_validation, zeros))
    y_validation = np.reshape(validation_data[1].data, (validation_data[1].shape[0], 1))

    print(X_train.shape,y_train.shape)
    print(X_validation.shape,y_validation.shape)
    return X_train, X_validation, y_train, y_validation

  def loss_function(X_data, y_data, w, C):
    hinge_loss = 0
    losses = (1 - y_data * np.dot(X_data, w))
    for one_loss in losses:
      hinge_loss += C * max(0, one_loss)
    return hinge_loss / len(X_data)


  def compute_gradient(X_data, y_data, w, C):
    gradient = np.zeros((1, X_data.shape[1]))
    losses = (1 - y_data * np.dot(X_data, w))
    for i, loss in enumerate(losses):
      if loss <= 0:
        gradient += w.T
      else:
        gradient += w.T - C * y_data[i] * X_data[i]
    return gradient / len(X_data)


  def NAG(w, gradient, v, mu=0.9, eta=0.0003):
    v_prev = v
    v = mu * v + eta * gradient
    w += ( mu * v_prev - (1 + mu) * v).reshape((123, 1))
    return w, v

  def RMSProp(w, gradient, cache, decay_rate=0.9, eps=1e-8, eta=0.0005):
    cache = decay_rate * cache + (1 - decay_rate) * (gradient ** 2)
    w += (- eta * gradient / (np.sqrt(cache + eps))).reshape((123, 1))
    return w, cache

  def AdaDelta(w, gradient, cache, delta_t, r=0.95, eps=1e-8):
    cache = r * cache + (1 - r) * (gradient ** 2)
    delta_theta = - np.sqrt(delta_t + eps) / np.sqrt(cache + eps) * gradient
    w = w + delta_theta.reshape((123, 1))
    delta_t = r * delta_t + (1 - r) * (delta_theta ** 2)
    return w, cache, delta_t

  def Adam(w, gradient, m, i, t, beta1=0.9, beta2=0.999, eta=0.0005, eps=1e-8):
    m = beta1 * m + (1 - beta1) * gradient
    mt = m / (1 - beta1 ** i)
    t = beta2 * t + (1 - beta2) * (gradient ** 2)
    vt = t / (1 - beta2 ** i)
    w += (-eta * mt / (np.sqrt(vt + eps))).reshape((123, 1))
    return w, m, t
```

```
def plotLossPerTime(epoch, nag_losses, rms_losses, adad_losses, adam_losses):
    plt.xlabel('iteration times')
    plt.ylabel('loss')
    plt.title('Linear Classification & SGD')
    n_cycles = range(1,epoch+1)
    plt.plot(n_cycles, nag_losses, label="Loss of NAG", linewidth=2)
    plt.plot(n_cycles, rms_losses, label="Loss of RMSProp", linewidth=2)
    plt.plot(n_cycles, adad_losses, label="Loss of AdaDelta", linewidth=2)
    plt.plot(n_cycles, adam_losses, label="Loss of Adam", linewidth=2)
    plt.legend(loc=0)
    plt.grid()
    plt.show()

X_train, X_validation, y_train, y_validation = loadDataSet()
nag_w = np.zeros((X_train.shape[1], 1))
rms_w = np.zeros((X_train.shape[1], 1))
adad_w = np.zeros((X_train.shape[1], 1))
adam_w = np.zeros((X_train.shape[1], 1))

v = np.zeros(X_train.shape[1])
cache = np.zeros(X_train.shape[1])
adad_cache = np.zeros(X_train.shape[1])
delta_t = np.zeros(X_train.shape[1])
m = np.zeros(X_train.shape[1])
t = np.zeros(X_train.shape[1])

batch_size = 5000
epoch = 200
C = 1
nag_losses = []
rms_losses = []
adad_losses = []
adam_losses = []

for i in range(epoch):
    index = list(range(len(X_train)))
    random.shuffle(index)

    # NAG
    nag_gradient = compute_gradient(X_train[index][:batch_size], y_train[index][:batch_size], nag_w, C)
    nag_w, v = NAG(nag_w, nag_gradient, v)
    nag_loss = loss_function(X_validation, y_validation, nag_w, C)
    nag_losses.append(nag_loss)
    print("NAG_loss = %f" % nag_loss)

    # RMSProp
    rms_gradient = compute_gradient(X_train[index][:batch_size], y_train[index][:batch_size], rms_w, C)
    rms_w, cache = RMSProp(rms_w, rms_gradient, cache)
    rms_loss = loss_function(X_validation, y_validation, rms_w, C)
    rms_losses.append(rms_loss)
    print("RMSProp_loss = %f" % rms_loss)

    # AdaDelta
    adad_gradient = compute_gradient(X_train[index][:batch_size], y_train[index][:batch_size], adad_w, C)
    adad_w, adad_cache, delta_t = AdaDelta(adad_w, adad_gradient, adad_cache, delta_t)
    adad_loss = loss_function(X_validation, y_validation, adad_w, C)
    adad_losses.append(adad_loss)
    print("AdaDelta_loss = %f" % adad_loss)

    # Adam
    adam_gradient = compute_gradient(X_train[index][:batch_size], y_train[index][:batch_size], adam_w, C)
    adam_w, m, t = Adam(adam_w, adam_gradient, m, i+1, t)
    adam_loss = loss_function(X_validation, y_validation, adam_w, C)
    adam_losses.append(adam_loss)
    print("Adam_loss = %f" % adam_loss)

    print("---------------------------------")

plotLossPerTime(epoch, nag_losses, rms_losses, adad_losses, adam_losses)
```

Experimental results and curve:

(1) Logistic Regression and Stochastic Gradient Descent

a. Hyper-parameter selection:

epoch = 200

optimized methods	Hyper-parameter
NAG	η = 0.05，gamma = 0.9
RMSProp	η = 0.01，gamma = 0.9，epsilon=1e-8
AdaDelta	gamma = 0.95，epsilon=1e-5
Adam	η = 0.01，gamma=0.999，beta=0.9，epsilon=1e-8

b. Predicted Results (Best Results):

L_{NAG} = 0.336550
$L_{RMSProp}$ = 0.339171
$L_{AdaDelta}$ = 0.337936
L_{Adam} = 0.338454

c. Loss curve:

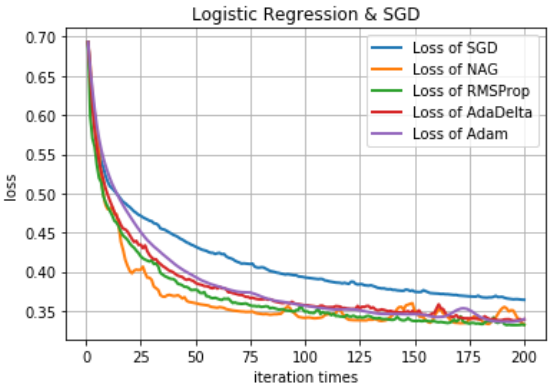

Fig1 loss of logistic regression

(2) Linear Classification and Stochastic Gradient Descent

a. Hyper-parameter selection:

epoch = 200

optimized methods	Hyper-parameter
NAG	η = 0.0003， gamma = 0.9
RMSProp	η = 0.0005， gamma = 0.9， epsilon=1e-8
AdaDelta	gamma = 0.95， epsilon=1e-8
Adam	η = 0.0005， gamma=0.999， beta=0.9， epsilon=1e-8

b. Predicted Results (Best Results):

L_{NAG} = 0.480252
$L_{RMSProp}$ = 0.481588
$L_{AdaDelta}$ = 0.487978
L_{Adam} = 0.483233

c. Loss curve:

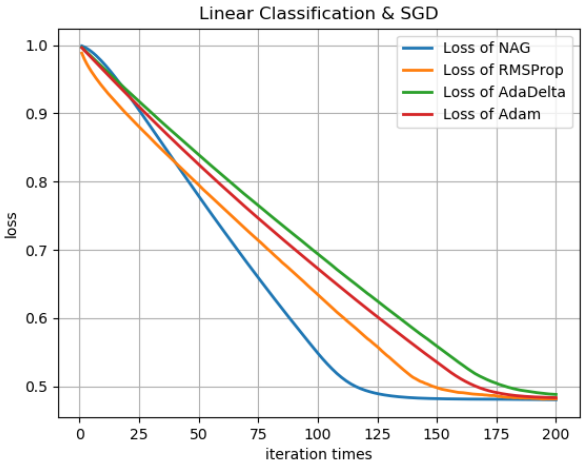

Fig2 loss of Linear Classification

IV. CONCLUSION

Through this experiment, I realize the difference between regression and classification. They are both related to prediction, where regression predicts a value from a continuous set, whereas classification predicts the 'belonging' to the class. For example, the price of a house depending on the 'size' (in some unit) and say 'location' of the house, can be some 'numerical value' (which can be continuous): this relates to regression. Similarly, the prediction of price can be in words, viz., 'very costly', 'costly', 'affordable', 'cheap', and 'very cheap': this relates to classification. Each class may correspond to some range of values.

In addition, I learned about the differences of GD and SGD. Stochastic Gradient Descent is often used in solving optimization problems related to model fitting on huge training data set, searching for the best possible parameters that have minimal cost. Usually, the cost function is continuous and smooth. When dealing with large data set and performing online learning, high order gradient descent methods is not applicable. Even conjugate gradient descent (CG) is not good because it requires accurate computation of the conjugate descent direction, hence the full data set is necessarily involved in each gradient computation.
On the other hand, using SGD, each time the gradient direction is computed based on only part of the data set (or even a single data point), so that the computational cost of each gradient step is constant, doesn't scale up with the size of the data set. In addition, it is reported that SGD converges much faster than regular gradient descent (GD).

In terms of the learning rate, it has a large impact on convergence. If the learning rate is too small, the rate of gradient descent is too slow. If it is too large, it will lead to oscillatory and may even diverge. So we use the different optimized methods, such as NAG, RMSProp, AdaDelta and Adam in this lab. The relation of these algorithms is shown in the following figure:

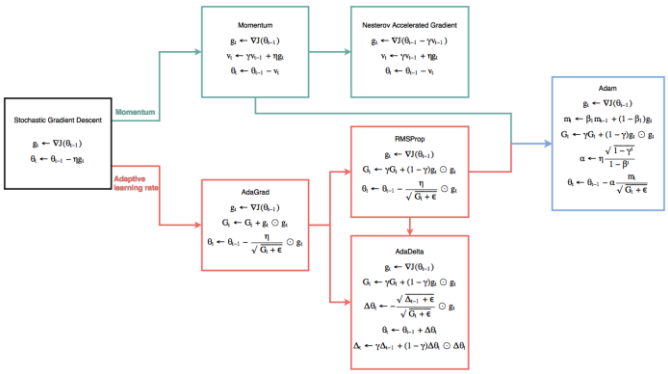

Fig3 the relation of different optimized methods

Nesterov Momentum is a slightly different version of the momentum update that has recently been gaining popularity. It enjoys stronger theoretical converge guarantees for convex functions and in practice it also consistenly works slightly better than standard momentum. RMSprop is a very effective, but currently unpublished adaptive learning rate method. Amusingly, everyone who uses this method in their work currently cites slide 29 of Lecture 6 of Geoff Hinton's Coursera class. The RMSProp update adjusts the Adagrad method in a very simple way in an attempt to reduce its aggressive, monotonically decreasing learning rate. AdaDelta can also solve the problem of AdaGrad, although it is often seen as similar to RMSProp, I feel AdaDelta is more advanced because it doesn't even have to set the initial learning rate, and AdaDelta

is sometimes relatively slow. Adam is a recently proposed update that looks a bit like RMSProp with momentum.

Through this experiment, I found out different optimization algorithms should be tried for different tasks.