stocastic_gradient_decent

March 10, 2019

```
In [1]: import numpy as np
        from sklearn.datasets import load_iris
        import matplotlib.pyplot as plt
        from sklearn.model_selection import train_test_split

        iris = load_iris()
        print(iris.DESCR)
```

```
.. _iris_dataset:

Iris plants dataset
--------------------

**Data Set Characteristics:**

    :Number of Instances: 150 (50 in each of three classes)
    :Number of Attributes: 4 numeric, predictive attributes and the class
    :Attribute Information:
        - sepal length in cm
        - sepal width in cm
        - petal length in cm
        - petal width in cm
        - class:
                - Iris-Setosa
                - Iris-Versicolour
                - Iris-Virginica

    :Summary Statistics:

    ============== ==== ==== ======= ===== ====================
                    Min  Max   Mean    SD   Class Correlation
    ============== ==== ==== ======= ===== ====================
    sepal length:   4.3  7.9   5.84   0.83    0.7826
    sepal width:    2.0  4.4   3.05   0.43   -0.4194
    petal length:   1.0  6.9   3.76   1.76    0.9490  (high!)
    petal width:    0.1  2.5   1.20   0.76    0.9565  (high!)
    ============== ==== ==== ======= ===== ====================
```

```
    :Missing Attribute Values: None
    :Class Distribution: 33.3% for each of 3 classes.
    :Creator: R.A. Fisher
    :Donor: Michael Marshall (MARSHALL%PLU@io.arc.nasa.gov)
    :Date: July, 1988

The famous Iris database, first used by Sir R.A. Fisher. The dataset is taken
from Fisher's paper. Note that it's the same as in R, but not as in the UCI
Machine Learning Repository, which has two wrong data points.

This is perhaps the best known database to be found in the
pattern recognition literature.  Fisher's paper is a classic in the field and
is referenced frequently to this day.  (See Duda & Hart, for example.)  The
data set contains 3 classes of 50 instances each, where each class refers to a
type of iris plant.  One class is linearly separable from the other 2; the
latter are NOT linearly separable from each other.

.. topic:: References

   - Fisher, R.A. "The use of multiple measurements in taxonomic problems"
     Annual Eugenics, 7, Part II, 179-188 (1936); also in "Contributions to
     Mathematical Statistics" (John Wiley, NY, 1950).
   - Duda, R.O., & Hart, P.E. (1973) Pattern Classification and Scene Analysis.
     (Q327.D83) John Wiley & Sons.  ISBN 0-471-22361-1.  See page 218.
   - Dasarathy, B.V. (1980) "Nosing Around the Neighborhood: A New System
     Structure and Classification Rule for Recognition in Partially Exposed
     Environments".  IEEE Transactions on Pattern Analysis and Machine
     Intelligence, Vol. PAMI-2, No. 1, 67-71.
   - Gates, G.W. (1972) "The Reduced Nearest Neighbor Rule".  IEEE Transactions
     on Information Theory, May 1972, 431-433.
   - See also: 1988 MLC Proceedings, 54-64.  Cheeseman et al"s AUTOCLASS II
     conceptual clustering system finds 3 classes in the data.
   - Many, many more ...
```

0.1 Loading Data

In [2]:
```
x = iris.data

label = iris.target

y = np.zeros(label.shape + (3,))
y[np.arange(label.shape[0]),label] = 1

print x.shape, y.shape
```

```
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.2)

print x_train.shape, x_test.shape, y_train.shape, y_test.shape
```

```
(150, 4) (150, 3)
(120, 4) (30, 4) (120, 3) (30, 3)
```

0.2 Activation Functions

In [3]:
```
def sigmoid(x):
    return 1/(1 + np.exp(-x))

def relu(x):
    x[x < 0] = 0
    return x

def tanh(x):
    return np.tanh(x)

def softmax(x):
    return np.exp(x) / np.sum(np.exp(x))

def deriv(x, activation = 'relu'):
    if(activation == 'relu'):
        x[x > 0] = 1
        x[x < 0] = 0
        return x
```

0.3 Function to Initialize Weights

In [4]:
```
def xavier_initializer(fan_in,fan_out):
    return np.random.normal(0,np.sqrt(2*1.0/(fan_in+fan_out)),(fan_out,fan_in+1))
```

0.4 Compute the Output shapes of each layer

In [5]:
```
def get_model(feed_dict):
    feed_dict['input_shape'] = feed_dict['train_input'].shape[1:]
    inp_shape = feed_dict['input_shape']
    feed_dict['output'] = []
    layers = feed_dict['layers']

    for i in range(len(layers)):
        output_shape = (layers[i]['nodes'],1)
        out_dict = {'layer_number': i , 'type': 'fc', 'output_shape': output_shape}
        feed_dict['output'].append(out_dict)
        inp_shape = output_shape
    return feed_dict
```

0.5 Fully Connected layer

In [6]:
```
def fully_connected(inp, weights,  nodes, activation):
    inp = np.asarray(inp).reshape(len(inp),1)
    inp = np.vstack((np.array(inp),1))
    #initiazing weights
#     weights = np.asmatrix(np.random.rand(nodes, len(inp)))
    output_raw = np.matmul(weights, inp)
    #normalizing the output to ensure no overflow in exp
#     print np.max(output_raw)
    output_raw = output_raw
    #applying activation function
    if(activation == 'sigmoid'):
        output = sigmoid(output_raw)
    elif(activation == 'relu'):
        output = relu(output_raw)
    elif(activation == 'tanh'):
        output = tanh(output_raw)
    elif(activation == 'softmax'):
        output = softmax(output_raw)
    else:
        output = output_raw
    #making the output vector as column matrix
    if(output.shape[0] == 1):
        output = np.moveaxis(output, 0,1)
        output_raw = np.moveaxis(output_raw, 0,1)
    return output, output_raw
```

0.6 Defining the model

In [7]:
```
feed_dict = {}
feed_dict['train_input'] = x_train
feed_dict['train_label'] = y_train
feed_dict['test_input'] = x_test
feed_dict['test_label'] = y_test
feed_dict['learning_rate'] = 1e-3
feed_dict['epochs'] = 1000
feed_dict['batch_size'] = 5
feed_dict['layers'] = [{'type': 'fc',  'nodes': 10, 'activation' : 'relu'},
                       {'type': 'fc',  'nodes': 10, 'activation' : 'relu'},
                       {'type': 'fc',  'nodes': 3, 'activation' : 'softmax'}]
```

0.7 Computing the outputs and initializng the weights

In [8]:
```
feed_dict = get_model(feed_dict)

print 'output shapes:'
for i in range(len(feed_dict['layers'])):
```

```
            print feed_dict['layers'][i]['type']+str(i) , ': ', feed_dict['output'][i]['output_

        print("\n")

        feed_dict['output'][0]['weights'] = xavier_initializer(feed_dict['input_shape'][0], fee
        feed_dict['output'][1]['weights'] = xavier_initializer(feed_dict['output'][0]['output_s
        feed_dict['output'][2]['weights'] = xavier_initializer(feed_dict['output'][1]['output_s

        print 'weight matrices shapes (with biases):'
        print feed_dict['layers'][0]['type']+str(0),feed_dict['output'][0]['weights'].shape
        print feed_dict['layers'][1]['type']+str(1),feed_dict['output'][1]['weights'].shape
        print feed_dict['layers'][2]['type']+str(2),feed_dict['output'][2]['weights'].shape
```

```
output shapes:
fc0 :  (10, 1)
fc1 :  (10, 1)
fc2 :  (3, 1)


weight matrices shapes (with biases):
fc0 (10, 5)
fc1 (10, 11)
fc2 (3, 11)
```

```
In [9]: epochs = feed_dict['epochs']
        no_samples = len(x_train)
        batch_size = feed_dict['batch_size']
        no_batches = no_samples/batch_size
```

```
In [10]: layers = feed_dict['layers']
         train_losses = []
         for epoch in range(epochs):
             cost_per_epoch = 0
             #shuffling the data
             s = np.arange(feed_dict['train_input'].shape[0])
             np.random.shuffle(s)
             feed_dict['train_input'] = feed_dict['train_input'][s]
             feed_dict['train_label'] = feed_dict['train_label'][s]
             for batch in range(no_batches):
                 # weight matrices for sum of updates of batch
                 weights_fc_0 = np.zeros(feed_dict['output'][0]['weights'].shape)
                 weights_fc_1 = np.zeros(feed_dict['output'][1]['weights'].shape)
                 weights_fc_2 = np.zeros(feed_dict['output'][2]['weights'].shape)
                 for i in range(batch_size):
                     #feeding forward
                     feed_dict['output'][0]['output'], feed_dict['output'][0]['output_raw'] = :
                     feed_dict['output'][1]['output'], feed_dict['output'][1]['output_raw'] = :
```

```
                feed_dict['output'][2]['output'], feed_dict['output'][2]['output_raw'] = 

                #cost calculation
                cost_per_epoch = cost_per_epoch - np.log(feed_dict['output'][2]['output']

                #calculating the gradients
                feed_dict['output'][2]['semi_update'] = feed_dict['output'][2]['output'] 
                feed_dict['output'][2]['update'] = np.matmul(feed_dict['output'][2]['semi_

                temp = feed_dict['output'][2]['weights'][:,0:feed_dict['output'][2]['weig
                feed_dict['output'][1]['semi_update'] = np.matmul(np.transpose(temp), fee
                feed_dict['output'][1]['update'] = np.matmul(feed_dict['output'][1]['semi_

                temp = feed_dict['output'][1]['weights'][:,0:feed_dict['output'][1]['weig
                feed_dict['output'][0]['semi_update'] = np.matmul(np.transpose(temp), fee
                feed_dict['output'][0]['update'] = np.matmul(feed_dict['output'][0]['semi_

                weights_fc_0 += feed_dict['output'][0]['update']
                weights_fc_1 += feed_dict['output'][1]['update']
                weights_fc_2 += feed_dict['output'][2]['update']

            #updating the gradient after each batch
            feed_dict['output'][0]['weights'] -= feed_dict['learning_rate'] * weights_fc_
            feed_dict['output'][1]['weights'] -= feed_dict['learning_rate'] * weights_fc_
            feed_dict['output'][2]['weights'] -= feed_dict['learning_rate'] * weights_fc_

        #printing the Average Loss after each epoch
        if((epoch+1)%50 == 0):
            print("Epoch: " + str(epoch+1) + " Loss: " + str(cost_per_epoch/no_samples))
        train_losses.append(cost_per_epoch/no_samples)
```

```
Epoch: 50 Loss: [0.38323718]
Epoch: 100 Loss: [0.21047086]
Epoch: 150 Loss: [0.13648092]
Epoch: 200 Loss: [0.13210621]
Epoch: 250 Loss: [0.10390012]
Epoch: 300 Loss: [0.10173987]
Epoch: 350 Loss: [0.09617796]
Epoch: 400 Loss: [0.10876418]
Epoch: 450 Loss: [0.09167417]
Epoch: 500 Loss: [0.0968994]
Epoch: 550 Loss: [0.08105786]
Epoch: 600 Loss: [0.08307529]
Epoch: 650 Loss: [0.07520889]
Epoch: 700 Loss: [0.08902226]
Epoch: 750 Loss: [0.081143]
Epoch: 800 Loss: [0.09291695]
Epoch: 850 Loss: [0.09225247]
```

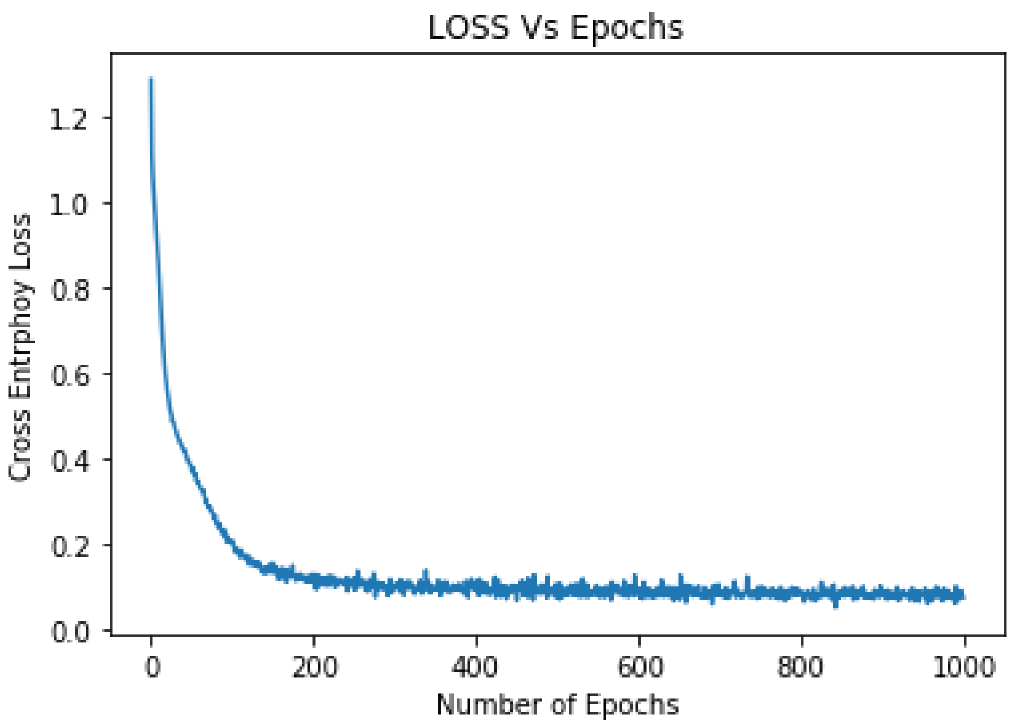
```
Epoch: 900 Loss: [0.08941543]
Epoch: 950 Loss: [0.08457685]
Epoch: 1000 Loss: [0.0767278]
```

0.8 Saving the Model

```
In [11]: np.save("stocastic.npy", feed_dict)
```

0.9 Plotting the training loss

```
In [12]: train_losses = np.array(train_losses)
         plt.plot(train_losses)
         plt.title('LOSS Vs Epochs')
         plt.ylabel('Cross Entrphoy Loss')
         plt.xlabel('Number of Epochs')
         plt.show()
```



0.10 Prediction on one sample

```
In [13]: feed_dict['output'][0]['output'], feed_dict['output'][0]['output_raw'] = fully_connec
         feed_dict['output'][1]['output'], feed_dict['output'][1]['output_raw'] = fully_connec
         feed_dict['output'][2]['output'], feed_dict['output'][2]['output_raw'] = fully_connec
```

```
        print 'output of softmax for one sample:'
        print feed_dict['output'][2]['output']

        print '\nGround Truth of the same sample above:'
        print feed_dict['train_label'][0]
```

```
output of softmax for one sample:
[[4.36283671e-03]
 [9.95600637e-01]
 [3.65260328e-05]]

Ground Truth of the same sample above:
[0. 1. 0.]
```

0.11 Predicting on Test Data

```
In [14]: test_predicted = []
         gt = []
         # print(np.argmax(out))
         for i in range(feed_dict['test_input'].shape[0]):
             feed_dict['output'][0]['output'], feed_dict['output'][0]['output_raw'] = fully_co
             feed_dict['output'][1]['output'], feed_dict['output'][1]['output_raw'] = fully_co
             feed_dict['output'][2]['output'], feed_dict['output'][2]['output_raw'] = fully_co

             test_predicted.append(np.argmax(feed_dict['output'][2]['output']))
             gt.append(np.argmax(feed_dict['test_label'][i]))
```

0.12 Outputs and the respective Groucd Truths

```
In [15]: print 'predicted: ',test_predicted
         print 'Actual    : ', gt
```

```
predicted:  [0, 1, 1, 2, 1, 0, 0, 1, 2, 0, 2, 2, 0, 1, 1, 2, 1, 1, 2, 2, 2, 0, 2, 2, 1, 1, 1, 2
Actual    :  [0, 1, 1, 2, 2, 0, 0, 1, 2, 0, 2, 2, 0, 1, 1, 2, 1, 1, 2, 2, 2, 0, 2, 2, 1, 1, 1, 2
```

0.13 Accuracy on the Test Dataset

```
In [16]: a = np.array(test_predicted) - np.array(gt)
         test_accuracy = (len(a) - np.count_nonzero(a))/float(len(a))

         print 'accuracy: ', str(test_accuracy)
```

```
accuracy:  0.966666666667
```