# INF 552 Assignment 5

## Neural Networks

Author:

Zongdi Xu (USC ID 5900-5757-70, working on neural network implementation),

Wenkai Xu (USC ID 5417-1457-73, working on software familiarization and others).

Date: Mar 29, 2019

## Part 1 Implementation

### 1.1 Single hidden-layer feed-forward neural network

- Neural Network definition

In [7]:
```
def __sigmoid(x):
    if x>=30.0:
        return 1.0
    elif x<=-30.0:
        return 0.0
    else:
        return 1.0 / (1.0 + np.exp(-x))
```

The above is a modified sigmoid function that is more efficient and can avoid overflow of exponential function when the input value is too large.

In [8]:
```
import numpy as np

def __sigmoid_derivative(x):
    return x * (1 - x)

def sigmoid(x):
    vfunc = np.vectorize(__sigmoid)
    return vfunc(x)

def sigmoid_derivative(x):
    vfunc = np.vectorize(__sigmoid_derivative)
    return vfunc(x)

class NeuralNetwork:
    def __init__(self, input_n, hidden_n, output_n):
        self.input_n = input_n + 1
        self.hidden_n = hidden_n
        self.output_n = output_n
        self.input_layer = np.ones((1,self.input_n))
        # init weights
        self.input_weights=np.random.uniform(-0.01,0.01,(self.input_n,self.hidde
n_n))
        self.output_weights=np.random.uniform(-0.01,0.01,(self.hidden_n,self.out
put_n))

    def predict(self, x_train):
        # activate input layer
        for j in range(x_train.shape[0]):
            self.input_layer[:,j]=x_train[j]
        # activate hidden layer
        self.hidden_cells=sigmoid(np.dot(self.input_layer,self.input_weights))
        # activate output layer
        self.output_cells=np.round(sigmoid(np.dot(self.hidden_cells,self.output_
weights)))
        return self.output_cells

    def back_propagate(self, x_train, y_train, learn):
        # feed forward
        self.predict(x_train)
        # get output layer error
        output_deltas=y_train-self.output_cells
        # get hidden layer error
        hidden_deltas=np.dot(output_deltas,self.output_weights.T)*sigmoid_deriva
tive(self.hidden_cells)
        # update output weights
        delta=np.dot(self.hidden_cells.T,output_deltas)
        self.output_weights+=learn*delta
        # update input weights
        delta=np.dot(self.input_layer.T,hidden_deltas)
        self.input_weights+=learn*delta
        # get global error
        # error=(y_train*self.output_cells)**2/len(y_train)
        # return np.sum(error)

    def train(self, x_train, y_train, limit=10000, learn=0.05):
        for j in range(limit):
            for i in range(len(x_train)):
                self.back_propagate(x_train[i], y_train[i], learn)
            if np.sum(np.abs(y_train-self.test(x_train)))<1.0:
                print("Converge after " + str(j) + " epoch(s).")
                return
        print "After " + str(j) + " epoch(s)"
```

```python
    def test(self, x_test):
        y_pred = []
        for case in x_test:
            y_pred.append([np.squeeze(self.predict(case))])
        return np.array(y_pred)
```

- Prepare training data

In [9]:
```python
import re

def read_pgm(filename, byteorder='>'):
    """Return image data from a raw PGM file as numpy array.

    Format specification: http://netpbm.sourceforge.net/doc/pgm.html

    """
    with open(filename, 'rb') as f:
        buffer = f.read()
    try:
        header, width, height, maxval = re.search(
            b"(^P5\s(?:\s*#.*[\r\n])*"
            b"(\d+)\s(?:\s*#.*[\r\n])*"
            b"(\d+)\s(?:\s*#.*[\r\n])*"
            b"(\d+)\s(?:\s*#.*[\r\n]\s)*)", buffer).groups()
    except AttributeError:
        raise ValueError("Not a raw PGM file: '%s'" % filename)
    return np.frombuffer(buffer,
                            dtype='u1' if int(maxval) < 256 else byteorder+'u2',
                            count=int(width)*int(height),
                            offset=len(header)
                            ).reshape((int(height), int(width)))
```

In [11]:
```python
# get training data
train_filelist = 'downgesture_train.list'
x_train=[]
y_train=[]
with open(train_filelist, 'r') as train_fl:
    for train_fn in train_fl.readlines():

        image = read_pgm(train_fn[:-1], byteorder='<')
        image = image.astype('float32')
        image /= np.max(image)

        x_train.append(np.squeeze(image.reshape(1,-1)))

        y_train.append([0. if re.match(string=train_fn, pattern='.*?down.*?')==N
one else 1.])

x_train = np.array(x_train)
y_train = np.array(y_train)
```

The original raw data loaded from PGM files contains integers ranging roughly from 0 to 255. We convert those into floating-point real numbers ranging from 0.0 to 1.0. What's more, we reshape every 2-dimensional image matrix into 1-dimensional vector to help the neural network better recognize it.

- Prepare testing data

In [13]:
```
# get testing data
test_filelist = 'downgesture_test.list'
x_test=[]
y_test=[]
with open(test_filelist, 'r') as test_fl:
    for test_fn in test_fl.readlines():

        image = read_pgm(test_fn[:-1], byteorder='<')
        image = image.astype('float32')
        image /= np.max(image)

        x_test.append(np.squeeze(image.reshape(1,-1)))

        y_test.append([0. if re.match(string=test_fn, pattern='.*?down.*?')==Non
e else 1.])

x_test = np.array(x_test)
y_test = np.array(y_test)
```

- Training neural network

It usually takes 1-2 minutes to train the all 180+ PGM images.

In [15]:
```
from time import time
# neural network training
NN=NeuralNetwork(x_train.shape[1], 25, y_train.shape[1])

train_start_time = time()
NN.train(x_train, y_train,limit=1000, learn=0.1)
print 'Time elapsed during training: %.3fs' % (time()-train_start_time)
y_pred = np.abs(np.round(NN.test(x_train)))
print y_pred.T
print 'Training accuracy:',1.0-np.sum(np.abs(y_pred-y_train))/len(y_pred)
```

```
Converge after 307 epoch(s).
Time elapsed during training: 75.203s
[[1. 1. 1. 1. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 1. 1.
  0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 1. 0. 0. 0. 0. 0. 0. 1. 1. 0. 0. 0. 0. 0.
  0. 0. 0. 0. 0. 1. 1. 1. 0. 0. 0. 0. 0. 0. 0. 0. 1. 1. 1. 1. 0. 0. 0. 0.
  0. 0. 0. 0. 0. 1. 1. 1. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 1. 1. 1. 0. 0.
  0. 0. 0. 0. 0. 0. 0. 0. 1. 1. 1. 1. 0. 0. 0. 0. 0. 0. 0. 0. 1. 1. 1. 1.
  0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 1. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
  0. 0. 1. 1. 1. 1. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 1. 1. 1. 1.
  1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]]
Training accuracy: 1.0
```

In [17]:
```
y_pred = np.abs(np.round(NN.test(x_test)))
print y_pred.T
print 'testing accuracy: %.3f' % (1.0-np.sum(np.abs(y_pred-y_test))/len(y_pred))
```

```
[[1. 1. 0. 0. 0. 0. 0. 0. 1. 0. 0. 1. 1. 1. 0. 1. 1. 1. 0. 0. 1. 0. 0. 0.
  0. 1. 1. 0. 0. 0. 1. 1. 1. 0. 0. 0. 0. 0. 1. 1. 1. 0. 1. 0. 0. 1. 1. 0.
  0. 0. 1. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 1. 1. 1. 0. 0. 0. 0.
  0. 0. 1. 1. 0. 0. 0. 0. 0. 0. 0.]]
testing accuracy: 0.807
```

## 1.2 Direct solving of Single hidden-layer feed-forward neural network

According to the theory of `Extreme Learning Machine`, such a single hidden-layer neural network can be trained and solved directly, rather than being solved by iteration. The point is to calculate the `pseudo-inversion` of the matrix $H$ where $H = Sigmoid(Wx + b)$.

In [27]:
```
# definition of neural network
class SingeHiddenLayer(object):
    def __init__(self, X, y, num_hidden):
        self.data_x = np.atleast_2d(X)  #
        self.data_y = np.array(y).flatten()
        self.num_data = len(self.data_x)
        self.num_feature = self.data_x.shape[1]
        self.num_hidden = num_hidden

        self.w = np.random.uniform(-0.01, 0.01, (self.num_feature, self.num_hidd
en))

        for i in range(self.num_hidden):
            b = np.random.uniform(-0.01, 0.01, (1, self.num_hidden))
            self.first_b = b

        for i in range(self.num_data - 1):
            b = np.row_stack((b, self.first_b))
        self.b = b

    def sigmoid(self, x):
        return 1.0 / (1 + np.exp(-x))

    def train(self, x_train, y_train, classes=1):
        mul = np.dot(self.data_x, self.w)
        add = mul + self.b
        H = self.sigmoid(add)

        H_ = np.linalg.pinv(H)

        self.train_y = y_train

        self.out_w = np.dot(H_, self.train_y)

    def predict(self, x_test):
        self.t_data = np.atleast_2d(x_test)
        self.num_tdata = len(self.t_data)
        self.pred_Y = np.zeros((x_test.shape[0]))

        b = self.first_b

        for i in range(self.num_tdata - 1):
            b = np.row_stack((b, self.first_b))

        self.pred_Y = np.dot(self.sigmoid(
            np.dot(self.t_data, self.w) + b), self.out_w)

        return(self.pred_Y)
```

In [29]:
```
# neural network training
from time import time

NN = SingeHiddenLayer(x_train, y_train, 25)
train_start_time = time()
NN.train(x_train, y_train)
print 'Time elapsed during training: %.3fs' % (time()-train_start_time)
y_pred = np.abs(np.round(NN.predict(x_train)))
print y_pred.T
print 'training accuracy:',1.0-np.sum(np.abs(y_pred-y_train))/len(y_pred)
```

```
Time elapsed during training: 0.001s
[[1. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1.
  0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 1. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0.
  1. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0.
  0. 0. 0. 1. 0. 1. 1. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 1. 0. 1. 0. 0.
  0. 0. 0. 0. 0. 0. 0. 0. 1. 1. 1. 0. 0. 0. 0. 0. 0. 0. 1. 0. 1. 1. 1. 0.
  0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
  0. 0. 1. 1. 1. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 1. 1. 1.
  1. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 1. 0. 0.]]
training accuracy: 0.8478260869565217
```

In [30]:
```
# neural network testing
y_pred = np.abs(np.round(NN.predict(x_test)))
print y_pred.T
print 'testing accuracy:',1.0-np.sum(np.abs(y_pred-y_test))/len(y_pred)
```

```
[[1. 1. 0. 0. 0. 0. 1. 0. 0. 1. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
  0. 0. 0. 1. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 1. 0. 0. 0. 0. 1. 1. 0.
  1. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 1. 1. 0. 0. 0. 0.
  0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0.]]
testing accuracy: 0.8313253012048193
```

It is much faster, without noticeable loss of accuracy.

## Part 2 Software Familiarization

We can use packages from sklearn to help with the implementation.

Class MLPClassifier implements a multi-layer perceptron (MLP) algorithm that trains using Backpropagation.

In [ ]:
```
from sklearn.neural_network import MLPClassifier
X = [[0., 0.], [1., 1.]]
y = [0, 1]
clf = MLPClassifier(solver='lbfgs', alpha=1e-5,hidden_layer_sizes=(5, 2), random
_state=1)
clf.fit(X, y)
```

In [ ]:
```
After the training process, we can use it to predict new data:
```

In [32]:
```
clf.predict([[2., 2.], [-1., -2.]])
clf.predict_proba([[2., 2.], [1., 2.]])
```

Out[32]:
```
array([[1.96718015e-04, 9.99803282e-01],
       [1.96718015e-04, 9.99803282e-01]])
```

## Part 3 Application:

Here are some popular applications of neural network:

1. Image processing
2. Character recognition
3. Credit card fraud detection

# Reference

Wikipedia - Extreme learning machine